Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Clasificación automática de sonidos utilizando aprendizaje máquina

Autor: Patricio Rodríguez Ramírez
Tutor: Francisco José Simois Tirado

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Clasificación automática de sonidos utilizando aprendizaje máquina

Autor:
Patricio Rodríguez Ramírez

Tutor:
Francisco José Simois Tirado
Profesor Contratado Doctor

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Clasificación automática de sonidos utilizando aprendizaje máquina

Autor: Patricio Rodríguez Ramírez

Tutor: Francisco José Simois Tirado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis amigos

Agradecimientos

Me gustaría agradecer a tantísimas personas, pero este apartado se haría eterno. En primer lugar, me gustaría darle las gracias a mi tutor Francisco José Simois por haberme abierto las puertas de un campo tan apasionante como es el *Machine Learning* y por los continuos consejos para que el desarrollo de este trabajo fuese lo más perfecto posible. En segundo lugar, gracias a mi madre Martina, a mi padre Manuel y a mis hermanas Martina y Lourdes por cuidarme, aconsejarme y por enseñarme a ser la persona que soy hoy.

Durante estos cuatro años me he rodeado de bellísimas personas que han conseguido que todo este tiempo se haya pasado volando. A Igna, a Rodri, a Franito, a Chelo, a Elena, a Ana y a Juanje, gracias por haber sido los mejores compañeros de clase que jamás haya podido imaginar. Para siempre quedan los momentos previos a un examen difícil, las noches de fiesta y los viajes que hicimos (o que no hicimos). A Juan Carlos, a Daniel y a Pablo, gracias no solo por haber sido unos excelentes compañeros de clase, sino también unos excelentes compañeros de piso. Nunca olvidaré los piques por el fútbol, las conversaciones profundas a altas horas de la noche y los trayectos entre Sevilla y Huelva llenos de charlas y música. Gracias, en definitiva, por haberme hecho sentir como en casa.

No quiero dejar pasar esta oportunidad para agradecer a dos personas que saben ver lo mejor de mí y siempre me han mostrado su confianza. A Alfonso, gracias por ser mi mejor amigo desde el instituto y el hermano que nunca tuve, por los partidos del Recre y por contar conmigo cada finde que regresas al pueblo. A Rocío, gracias porque desde que llegaste a mi vida hace ya bastantes años te has convertido en una hermana (otra más) que me escucha, me comprende y que siempre tiene las puertas abiertas para mí, sea en Trigueros, Jerez o en Polonia. Ambos sois parte de esto y lo menos que puedo hacer por vosotros es daros las gracias.

Por último, agradecer a mis abuelos Rafael e Isabel. Sé el orgullo tan grande que sentían por cada uno de sus nietos. Ojalá pudieseis estar aquí para verme y abrazaros muy fuerte una vez más. También quiero darle las gracias a las personas que hace poco llegaron a mi vida y a las personas que se fueron de ella, pero sé que en algún momento volverán. Todos vosotros me habéis enseñado algo valioso que me ha ayudado en mi camino. Espero que por mi parte también os haya enseñado algo que os haya hecho crecer en mayor o menor medida.

Muchísimas gracias por todo.

Patricio Rodríguez Ramírez

San Juan del Puerto, 2020

Resumen

En los últimos años, el aprendizaje máquina se ha venido utilizando intensamente para el reconocimiento de sonidos. Algunos son fácilmente distinguibles, como una risa, pero otros en cambio pueden ser muy similares entre sí, como una batidora y una motosierra. Además, la variabilidad inherente a estos audios hace que este problema sea bastante complicado de resolver mediante técnicas de procesado clásicas, pero supone un desafío apropiado para los altos niveles de abstracción que se pueden conseguir con las técnicas de aprendizaje máquina. En este trabajo se presentan dos modelos de red neuronal convolucional (CNN) para resolver un problema de clasificación de sonidos ambientales en siete categorías distintas. Los extractos de audio usados son los proporcionados por la base de datos UrbanSound8K. El rendimiento de ambos modelos llega a alcanzar el 90% de precisión en la clasificación de estos sonidos.

Abstract

Machine learning has been used intensively for sound recognition in recent years. Some sounds are easily distinguishable, like a laugh, but others can be very similar to each other, like a blender and a chainsaw. Furthermore, the inherent variability in these audios makes this problem quite difficult to solve using classical processing techniques, but it is an appropriate challenge for the high levels of abstraction that can be achieved with machine learning techniques. In this work, two convolutional neural network (CNN) models are presented to solve a problem of environmental sound classification in seven different labels. The audio excerpts used are those provided by the UrbanSound8K database. The performance of both models reaches 90% accuracy in the classification of these sounds.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xv**

Índice de Tablas **xvii**

Índice de Figuras **xix**

Índice de Códigos **xxi**

Índice de Ecuaciones **xxiii**

1 Introducción **1**
1.1. Motivación del trabajo *1*
1.2. Objetivos *1*
1.3. Estructura de la memoria *1*

2 Estado del arte **3**
2.1. Según el tipo de problema *3*
2.2. Según la arquitectura de la red *4*
2.3. Según las características del sonido *5*

3 Deep Learning para análisis de audio **7**
3.1. Análisis de audio *7*
3.1.1 Conversión analógico-digital 7
3.1.2 Dominio frecuencial 8
3.1.3 Espectrograma 9
3.2. Deep Learning *11*
3.2.1 Mapa conceptual 12
3.2.2 De la neurona a la red 13
3.2.3 Prueba y error 15
3.2.4 Resultado 16
3.2.5 Redes Neuronales Convolucionales 17

4 Método propuesto **23**
4.1. Material *23*
4.1.1 Hardware 23
4.1.2 Software 24
4.2. Base de datos *26*
4.3. Implementación del primer modelo *26*
4.3.1 Primeros pasos 26
4.3.2 Preprocesamiento de los datos 27

4.3.3 Arquitectura propuesta 30
4.4. Implementación del segundo modelo *32*
5 Entrenamiento y resultados **35**
5.1. Resultados del primer modelo *35*
5.1.1 Configuración del entrenamiento 35
5.1.2 Matriz de confusión 37
5.2. Resultados del segundo modelo *39*
5.2.1 Configuración del entrenamiento 39
5.2.2 Matriz de confusión 40
5.3. Validación cruzada *41*
5.3.1 Resultados del primer modelo 42
5.3.2 Resultados del segundo modelo 42
6 Conclusiones y líneas futuras **45**
6.1. Conclusiones *45*
6.2. Líneas futuras *46*
Referencias **47**
Glosario **53**

ÍNDICE DE TABLAS

Tabla 3-1. Matriz de confusión de un clasificador binario 20
Tabla 4-1. Especificaciones del equipo 24
Tabla 4-2. Identificadores de clases de la base de datos 26
Tabla 5-1. Matriz de confusión del primer modelo 38
Tabla 5-2. Matriz de confusión normalizada del primer modelo 38
Tabla 5-3. Matriz de confusión del segundo modelo 41
Tabla 5-4. Matriz de confusión normalizada del segundo modelo 41

ÍNDICE DE FIGURAS

Figura 2-1. Esquemas de los distintos sistemas de análisis de audio 3
Figura 2-2. Resultados de precisión de los distintos modelos de Salamon et al. 6
Figura 3-1. Esquema de conversión analógico-digital 7
Figura 3-2. Distintos ejemplos de espectros 8
Figura 3-3. Audio con su espectrograma 9
Figura 3-4. Banco de filtros de Mel 10
Figura 3-5. Banco de filtros *Gammatone* 10
Figura 3-6. Distintos ejemplos de espectrogramas 11
Figura 3-7. Mapa conceptual 13
Figura 3-8. Esquema completo de una neurona 13
Figura 3-9. Ejemplos de funciones de activación 14
Figura 3-10. Esquema de una red neuronal artificial 14
Figura 3-11. Algoritmo de descenso del gradiente 15
Figura 3-12. Ejemplos de diferentes *learning rates* 16
Figura 3-13. Ejemplos de resultados de una red 16
Figura 3-14. Error de entrenamiento y validación 17
Figura 3-15. Operación de convolución 18
Figura 3-16. Operación de *max pooling* 18
Figura 3-17. Funcionamiento del *Dropout* 19
Figura 3-18. Ejemplo de validación cruzada para cuatro divisiones 20
Figura 3-19. Esquema completo de una red neuronal convolucional 21
Figura 4-1. Puntuación de los diferentes *frameworks* en 2018 25
Figura 4-2. Esquema del primer modelo 31
Figura 4-3. Esquema del segundo modelo 33
Figura 5-1. Evolución de la precisión del primer modelo 36
Figura 5-2. Evolución del error del primer modelo 37
Figura 5-3. Evolución de la precisión del segundo modelo 39
Figura 5-4. Evolución del error del segundo modelo 40
Figura 5-5. Resultado de la validación cruzada para el primer modelo 42
Figura 5-6. Resultado de la validación cruzada para el segundo modelo 43

ÍNDICE DE CÓDIGOS

Código 4-1. Librerías del primer modelo 27
Código 4-2. Preprocesamiento de los datos en el primer modelo 28
Código 4-3. Obtención de datos en el primer modelo 29
Código 4-4. Normalización de los datos en el primer modelo 30
Código 4-5. Arquitectura del primer modelo 32
Código 4-6. Arquitectura del segundo modelo 34
Código 5-1. Entrenamiento del primer modelo 36
Código 5-2. Evaluación del primer modelo 37
Código 5-3. Entrenamiento del segundo modelo 39
Código 5-4. Evaluación del segundo modelo 40

ÍNDICE DE ECUACIONES

Ecuación 3-1. Transformada Discreta de Fourier 8
Ecuación 3-2. Transformada de Fourier de Tiempo Reducido 8
Ecuación 4-1. Función de pérdida *cross-entropy* 31
Ecuación 4-2. Precisión para un clasificador binario 32

1 INTRODUCCIÓN

1.1. Motivación del trabajo

Nuestro sentido del oído nos permite distinguir lo que está sucediendo a nuestro alrededor en todo momento. Podemos cerrar los ojos estando en la calle y apreciaríamos el sonido de coches, niños jugando y animales. Podemos saber en todo momento qué sonido predomina sobre otro, qué duración ha tenido y desde qué posición lo hemos escuchado. Esta habilidad es natural para los seres humanos y la pasamos por alto. Sin embargo, la clasificación de sonidos ambientales (*Environmental Sound Classification*, ESC) para un ordenador es una tarea difícil y abre un nuevo problema.

El reconocimiento automático de sonidos permite desarrollar una extensa variedad de aplicaciones donde el sentido de la audición está presente. Por ejemplo, un sistema de reconocimiento de sonidos se podría instalar en el sistema de seguridad de una vivienda. Así podríamos distinguir cuando se ha roto un cristal, cuando ha habido un disparo, un golpe de un coche o si existe un barullo de gente hablando. Este sistema bien podría implementarse en un vehículo para evitar posibles accidentes o en un móvil, el cual podría cambiar automáticamente de tono de llamada si estamos en una sala vacía o llena de personas.

Existen otros campos relacionados con el ESC y de los cuales se han trasladados muchos métodos para ser implementados en nuestro problema, ya que muchas tareas son similares en cuanto a procedimiento. El reconocimiento automático de voz (*Automatic Speech Recognition*, ASR) es un campo extenso que tiene como objetivo permitir la comunicación hablada entre el ser humano y el ordenador. Entre sus tareas se encuentran el reconocimiento de las palabras, la identificación de la persona que está hablando o el número de personas que lo están haciendo. La recuperación de información musical (*Music Information Retrieval*, MIR) es una disciplina que tiene como tareas el reconocimiento de notas musicales, identificación del género musical o de los instrumentos que suenan.

Existen numerosos retos, como los propuestos por el DCASE, para que la comunidad científica experimente y desarrolle nuevos métodos. La motivación de este trabajo vino a raíz de la tarea número 2 del ***DCASE Challenge 2018*** llamada ***General-purpose audio tagging of Freesound content with AudioSet labels***. Los datos de la competición los vimos en la plataforma *Kaggle* y los resultados están publicados en una sección de la página web del DCASE.

1.2. Objetivos

El objetivo de este trabajo consiste en la realización de un sistema capaz de clasificar sonidos ambientales con una precisión aceptable. Para ello estudiaremos los distintos métodos de extracción de características de audio e investigaremos en profundidad sobre la inteligencia artificial para elegir el mejor modelo que sea capaz de resolver nuestro problema.

Finalmente implantaremos dicho modelo y calcularemos los distintos parámetros de precisión. Debido a nuestra inexperiencia en el campo del aprendizaje máquina y a la falta de recursos computacionales no pretendemos llegar a las altas cotas de precisión que los expertos en la materia logran alcanzar. Nuestro trabajo va a ser una ligera aproximación al estudio de las redes neuronales, reduciendo la dimensión de nuestro problema para que este nos sirva como introducción a un campo en continuo crecimiento y evolución.

1.3. Estructura de la memoria

Este documento se divide en los siguientes apartados:

- Capítulo 1. Introducción: Es el capítulo actual. Exponemos la motivación de nuestro proyecto,

introducimos el trabajo que vamos a desarrollar en los próximos capítulos y fijamos los objetivos que buscamos alcanzar.

- Capítulo 2. Estado del arte: En este capítulo se realiza un estudio de investigación sobre los distintos problemas de análisis de audio y los métodos de resolución hasta ahora elaborados, clasificándolos según las características del sonido que emplean o según el tipo de arquitectura del clasificador.
- Capítulo 3. Deep Learning para análisis de audio: Tras haber comprobado las distintas posibilidades que existen elegimos trabajar con aprendizaje profundo. En este capítulo nos centramos en la teoría detrás del análisis de audio y de los modelos de redes neuronales.
- Capítulo 4. Método propuesto: Es en este capítulo cuando exponemos todo el código desarrollado para la creación de dos arquitecturas de redes neuronales que van a trabajar en la tarea de clasificación de sonidos con una base de datos previamente seleccionada.
- Capítulo 5. Entrenamiento y resultados: Tras programar todo el código toca entrenar nuestros dos modelos. En este capítulo se muestran los resultados obtenidos y se introduce un nuevo método para mejorar el rendimiento de nuestras redes neuronales.
- Capítulo 6. Conclusiones y líneas futuras: En este capítulo final concluimos nuestro trabajo con nuestra opinión y sugerimos una serie de futuras líneas de investigación que no se desarrollan en este proyecto.

2 ESTADO DEL ARTE

En el capítulo anterior hemos hablado sobre distintas aplicaciones de análisis de audio, como pueden ser la detección del llanto de un bebé o si nos encontramos en un ambiente silencioso o ruidoso gracias al micrófono de nuestro móvil. A pesar de lo distintas que puedan parecer estas aplicaciones, los procesos implementados por el ordenador suelen ser similares. Todas estas aplicaciones siguen el mismo flujo de trabajo.

2.1. Según el tipo de problema

Podemos dividir el análisis de audio en varios tipos de problemas. Se distinguen dos grandes grupos, dependiendo si el resultado del audio a analizar va a tener información temporal o no. Si el resultado va a tener información de las etiquetas o clases que se van dando en el tiempo, entonces estamos ante un problema de detección de audio (uno de los muchos nombres en inglés es *Sound Event Detection*).

En el otro caso, tenemos que hacer distinción de otros dos problemas. Si la salida nos va a mostrar la clase o el tipo de sonido que se le ha introducido, estamos ante un problema de clasificación de audio (*Sound Scene Classification*). Por último, si la salida va a ser varias etiquetas del sonido, el problema será de etiquetado de audio (*Audio Tagging*). En la Figura 2-1 mostramos un esquema visual de los distintos problemas de análisis de audio.

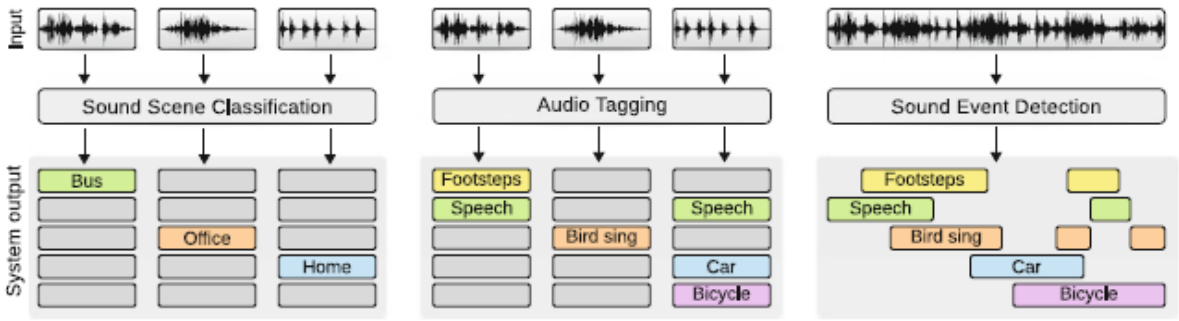

Figura 2-1. Esquemas de los distintos sistemas de análisis de audio

La clasificación de audio se puede dividir en clasificación de una sola clase o multiclase, dependiendo de para qué necesitamos la aplicación. La clasificación multiclase es lo que hemos denominado anteriormente un problema de etiquetado de audio.

Para la clasificación de audio de una sola clase los datos de entrada al sistema son extractos bien diferenciados de sonido. La salida será la etiqueta más parecida que el sistema ha podido seleccionar. Existen numerosos trabajos, algunos ejemplos son los siguientes.

Para la clasificación de sonidos ambientales Bae et al. [1] (2016) emplearon una arquitectura con distintas capas neuronales unidas. Petetin et al. [2] (2015) desarrollaron una arquitectura que compararon con clasificadores no basados en redes neuronales, consiguiendo un aumento de la precisión de más de un 5%. Valenti et al. [3] (2016) contribuyeron con su trabajo a la tarea presentada por DCASE de ese año, consiguiendo con su red neuronal convolucional un aumento de la precisión del 8.8% más que el sistema base presentado por el concurso. Lim et al. [4] (2016) utilizaron en su trabajo distintas bases de datos para entrenar mejor su modelo de reconocimiento de sonidos eventuales.

Para el problema de etiquetado de audio la entrada al sistema es un extracto de audio con uno o múltiples sonidos,

o incluso los mismos extractos de audios del problema anterior. El sistema dará a su salida las etiquetas en las que ha clasificado el sonido, pudiendo ser una o varias. De ahí el nombre de clasificación multiclase, ya que el problema anterior la salida del sistema solo podía ser la etiqueta con mayor probabilidad.

Çakir et al. [5] (2016) desarrollaron su trabajo para la tarea 4 del DCASE de ese año, siendo su resultado una red neuronal convolucional que aumentaba en 23.8% el modelo base de la competición, que no estaba basado en redes neuronales. Piczak [6] (2015) desarrolló un extenso estudio sobre su modelo de red neuronal, aplicándole distintos conjuntos de datos e implementando distintos métodos, consiguiendo excelentes resultados. Xu et al. [7] (2016) implementaron una red neuronal profunda y la compararon con distintas arquitecturas no basadas en redes neuronales, también en el marco de la competición DCASE 2016.

El problema de detección de audio presenta una complejidad mayor. En esencia es la misma idea que el problema de etiquetado, pero el sistema tiene que ser capaz de conocer en el tiempo cuando la etiqueta ha aparecido en el audio y cuando ha desaparecido. Los datos de entrada son extractos de audio de mayor duración y la salida son las etiquetas que han ido apareciendo en estos audios en el tiempo. Este tipo de sistemas están pensados para capturar el sonido en tiempo real a la vez que van detectando qué sonido o sonidos está escuchando.

Adavanne et al. [8] (2017) emplean características espaciales y armónicas de los sonidos para implementarlas en su modelo. Çakir et al. [9] (2015) proponen un sistema que aumenta en un 19% la precisión de los trabajos previos en esta materia. Espi et al. [10] (2015) emplearon dos recorridos para mejorar el problema de detección de audio. En el primero mejoraban la resolución de los espectrogramas de los audios de entrenamiento del modelo. En el segundo creaban una nueva arquitectura. En ambos casos mejoraban notablemente la precisión en este problema.

2.2. Según la arquitectura de la red

Para desarrollar todas estas tareas de clasificación y detección de audio la amplia mayoría de sistemas empleados están basados en redes neuronales. Como es tanta su aplicación en este campo, desarrollaremos la teoría de estas redes en el capítulo posterior. Ahora vamos a introducir algunos tipos de arquitecturas y a citar trabajos anteriores que las han empleado.

En primer lugar, nos encontramos con las redes neuronales convolucionales en una dimensión. Hinton et al. [11] (2012) emplearon este tipo de arquitectura para un problema de reconocimiento de voz. Como mencionamos anteriormente, los problemas relacionados con el reconocimiento de voz o de música emplean las mismas técnicas que los problemas de reconocimiento de sonidos ambientales, por lo que en nuestra investigación encontraremos indistintamente trabajos en estas áreas. Lee et al. [12] (2009) hacen un estudio de una red neuronal convolucional para distintas tareas, ya sea de voz o música. Van der Oord et al. [13] (2013) enfocan su trabajo en la recomendación automática de música según tus gustos, como podemos disfrutar en aplicaciones como *Spotify, Apple Music, Amazon Music, Google Play Music, Deezer, Tidal*, etc.

La evolución natural de estas redes son las redes neuronales convolucionales en dos dimensiones. Su uso es bastante estandarizado, ya que no solo se puede emplear para resolver nuestras tareas de audio, sino que originalmente se desarrollaron para solventar problemas de clasificación de imágenes. La entrada a este tipo de redes tendría que ser una representación en el tiempo y en la frecuencia del audio. Humphrey et al. [14] (2012) emplearon esta arquitectura para el reconocimiento automático de acordes musicales. Schlüter et al. [15] (2015) trabajaron en el problema de detección de voz cantada.

Otro tipo de arquitectura son las redes neuronales recurrentes (RNN). Se implementan en el mismo tipo de problemas anteriores y nos proporciona otra posibilidad de abordar estas tareas. Boulanger-Lewandowski et al. desarrollaron en 2013 un sistema de reconocimiento de acordes [16] y en 2014 un sistema de reconocimiento de voz [17]. En ambos sistemas se utilizaron redes neuronales recurrentes.

Al igual que las redes neuronales recurrentes, las redes bidireccionales nos abre otra puerta a la hora de resolver los problemas. Böck et al. [18] (2011) desarrolló su red para obtener la localización de los pulsos musicales en una serie de audios en el contexto de la competición sobre recuperación de información musical MIREX 2010. Graves et al. [19] (2014) crearon un sistema de reconocimiento de voz que transcribía directamente el audio a texto. Mesnil et al. [20] (2013) se centraron en el problema del reconocimiento del lenguaje hablado. Parascandolo et al. [21] (2016) emplearon una red neuronal bidireccional para la detección de sonidos.

Estos modelos anteriormente presentados son arquitecturas individuales. Las posibilidades se multiplican si combinamos varias arquitecturas en una. Uno de los modelos más implementados en el estado del arte une la red neuronal convolucional con una red densamente conectada. Se ha demostrado que este modelo proporciona buenos resultados en una amplia variedad de tareas. Dieleman et al. [22] (2014) emplearon esta arquitectura híbrida para la tarea de recuperación de información musical. Piczak [6] (2015) lo empleó para la tarea de clasificación de sonido ambiental. Schüler et al. [15] (2015) la utilizaron para la detección de voz cantada. Van den Oord et al. [13] (2013) para la recomendación de música.

Finalmente, otro tipo de arquitectura híbrida es aquella que une la red neuronal convolucional con la red neuronal recurrente. También es empleada en una gran cantidad de problemas. Amodei et al. [23] (2016) la utilizaron para el reconocimiento de voz tanto en el idioma inglés como en el chino mandarín. Zuo et al. [24] (2015) presentaron esta red para el problema de reconocimiento de imágenes, consiguiendo buenos resultados en la competición ILSVRC de 2012. Tang et al. [25] (2015) desarrollaron un sistema de análisis de texto que clasificaba los sentimientos, es decir, a partir de las frases escritas clasificaba si la persona le gustaba o no el tema del que estaba hablando. Sigtia et al. [26] (2016) trabajaron en el problema de transcripción automática de música de piano.

2.3. Según las características del sonido

Todas estas citas anteriores tienen como punto en común su arquitectura basada en redes neuronales. A continuación, vamos a introducir otros trabajos centrándonos en las características del audio que le pasamos al clasificador. Este punto también lo desarrollaremos en el apartado siguiente.

Uno de los primeros trabajos en el área del reconocimiento de audio fue el de Aucouturier et al. [27] (2007). En él los autores extraían los Coeficientes Cepstrales en la Frecuencia de Mel (MFCCs) a los audios y utilizaban un modelo mixto gaussiano (GMM) para clasificar. Su trabajo dio muy buenos resultados. Sin embargo, el trabajo de Lagrange et al. [28] (2015) demostró que el resultado estaba sesgado a un set de datos con muy poca variabilidad. Lagrange et al. aplicaron nuevos *datasets* con mayor variabilidad al modelo y los resultados obtenidos no fueron tan excepcionales.

Una opción es calcular una variedad de características al sonido. Eronen et al. [29] (2006) calcularon los MFCCs, la tasa de cruces por cero, el ancho de banda, etcétera. Para la clasificación emplearon el modelo oculto de Márkov (HMM). Geiger et al. [30] (2013) emplearon 13 características cepstrales, 35 características espectrales, 6 características de energía y 3 características relacionadas con la voz. Para la clasificación usaron una máquina de soporte de vectores (SVM).

Phan et al. [31] (2016) emplearon como características los Coeficientes Cepstrales en la Frecuencia Gammatone (GFCCs) y como clasificador una máquina de soporte de vectores. Por su parte, Roma et al. [32] (2013) emplean como características los coeficientes cepstrales en la frecuencia de mel más el cálculo de parámetros de recurrencia en los audios.

Una característica de entrada al sistema de clasificación puede ser el histograma de gradientes orientados (HOG). Bisot et al. [33] (2015) obtienen el histograma de gradientes orientados del espectrograma logarítmico de frecuencia del extracto de audio. Utilizan un clasificador basado en una máquina de soporte de vectores. Rakotomamonjya et al. [34] (2015) obtienen el histograma de gradientes orientados del espectrograma de la transformada de Q constante (CQT). Emplean varios sets de datos para su experimentación, con distintos clasificadores y resultados.

Cauchi et al. [35] (2013) desarrollaron un modelo en base a la factorización no negativa de matrices (NMF) para establecer las similitudes de los audios grabados en estaciones de tren. Bisot et al. [36] (2016) también emplearon la factorización no negativa de matrices para su problema de clasificación de sonidos ambientales.

Benetos et al. [37] (2012) usaron una variación del análisis de componentes principales (PCA) para extraer las características de los audios y entrenar su modelo. Lee et al. [38] (2013) utilizan una máquina estricta de Boltzmann dispersa (RBM) para obtener las características de entrada. Salamon et al. [39] (2015) calcularon los espectrogramas de mel, después aplicaron análisis de componentes principales y finalmente emplearon el algoritmo *k-means* esférico (SKM) para extraer las características. También estudiaron este procedimiento de manera no supervisada. [40] (2015).

Todos los trabajos citados nos han demostrado que existen muchas maneras de abordar las tareas de clasificación de audio ambiental. Para finalizar este extenso capítulo sobre el estado del arte nos vamos a centrar en el trabajo de Salamon et al., que han experimentado diversos métodos con un set de datos de sonidos ambientales.

En el trabajo [41] (2014) los autores presentan la taxonomía de los sonidos urbanos y crean el *dataset* con el que van a trabajar. Las características de entrada al sistema son los coeficientes cepstrales a la frecuencia de mel y el clasificador se basa en una máquina de soporte de vectores. En el trabajo [40] (2015) visto con anterioridad emplean como entrada al sistema el resultado del algoritmo *k-means* esférico aplicado a los espectrogramas. El clasificador se basa en el algoritmo de árboles aleatorios. En el trabajo [39] (2015) también emplean como entrada la salida del algoritmo *k-means* esférico aplicado a los espectrogramas. El clasificador se basa en una máquina de soporte de vectores. Finalmente en el trabajo [42] (2017) emplean como entrada al sistema los espectrogramas de los extractos de audio y el clasificador es una red neuronal convolucional profunda.

Los resultados de sus trabajos van en progresiva mejora. Mientras que el primer trabajo se considera la base y alcanza una precisión de más del 68%, los dos siguientes con el algoritmo *k-means* esférico alcanzan precisiones del 74% y 75%. La primera implementación con la red neuronal convolucional les deparó un resultado del 73% de precisión. Sin embargo, al aplicar un algoritmo de aumento de datos la precisión alcanzó cerca del 79%. Finalmente implementaron el modelo de Piczak [6] (2015) con la misma base de datos. El resultado de este modelo fue del 73%.

En la Figura 2-2 el punto rojo indica el resultado medio del modelo y las barras nos indican la desviación estándar de la medida. Fijándonos en la arquitectura basada en red neuronal convolucional vemos como, en el caso de no aplicar el algoritmo de aumento de datos (zona a la izquierda de la línea de puntos), la precisión media es menor que en los modelos que emplean *k-means* esférico. Sin embargo, la desviación estándar de la precisión es menor en el caso de la CNN ya que sus barras tienen un menor tamaño en comparación con los modelos que emplean SKM.

En el caso de aplicar un algoritmo de aumento de datos (zona a la derecha de la línea de puntos) vemos como la precisión media de la CNN aumenta con respecto al modelo SKM. A pesar de esto, la desviación estándar de la medida es mayor en el caso de la red neuronal convolucional, ya que sus barras ocupan un mayor tamaño en comparación con el modelo que emplea *k-means* esférico.

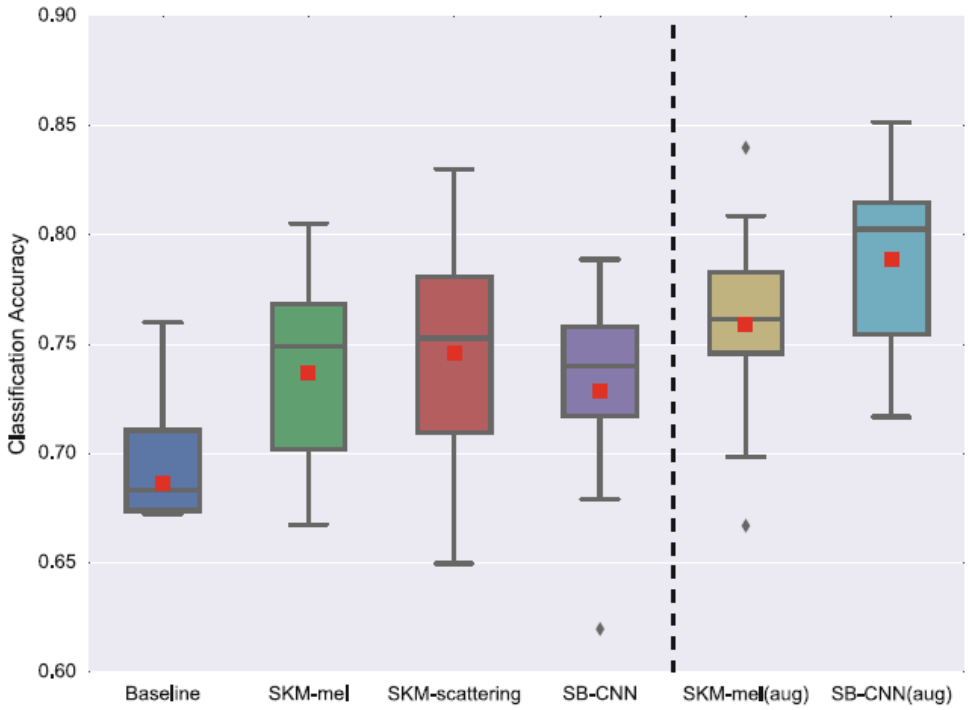

Figura 2-2. Resultados de precisión de los distintos modelos de Salamon et al.

3 DEEP LEARNING PARA ANÁLISIS DE AUDIO

En este capítulo vamos a explicar toda la teoría que constituye este trabajo. Primero explicaremos el tratamiento de las muestras de audio para después indagar sobre el modelo de red neuronal que vamos a implementar en los siguientes capítulos.

3.1. Análisis de audio

El sonido es una onda de presión longitudinal formada por las compresiones y rarefacciones de las moléculas de aire, que se propaga en dirección paralela a la aplicación de energía. (Huang et al. [43] (2001)). La representación de esta onda en el dominio del tiempo no es fácil de interpretar directamente. En muchas ocasiones resulta imposible localizar sonidos en una forma de onda e identificar a simple vista de qué clase de sonido se trata.

Es por ello por lo que las representaciones en el dominio de la frecuencia o las representaciones tiempo-frecuencia cobran mayor importancia. En este capítulo vamos a explicar cómo se calculan estas representaciones y qué utilidad tiene para nuestro proyecto. Además, introduciremos y desarrollaremos las redes neuronales para nuestro problema de clasificación.

3.1.1 Conversión analógico-digital

El sonido lo podemos obtener grabándolo con un sistema analógico, lo que obtendríamos una señal dependiente en el tiempo $x(t)$. Esta señal, si la representamos en el tiempo, veríamos su forma de onda. Esta señal no puede ser almacenada en un ordenador para su análisis, ya que es una señal continua y posee infinitos valores. Es por eso por lo que aplicamos una conversión analógico-digital (Figura 3-1), que consta de las siguientes fases:

- Filtrado paso bajo: la señal $x(t)$ se hace pasar por un filtro cuya frecuencia máxima la llamaremos B. Normalmente B se encuentra en torno a los 20 kHz, ya que esa es la frecuencia máxima de audición del ser humano. Los valores de la señal por encima de esa frecuencia no los vamos a tener en cuenta, ya que serían inaudibles para nosotros y una carga computacional innecesaria.
- Muestreo temporal: de la señal $x(t)$ obtenemos muestras en el dominio del tiempo con una frecuencia de $2*B$. Es decir, cada $1/2B$ segundos guardamos el valor de la señal $x(t)$ en ese momento. Se elige este valor de frecuencia para evitar el fenómeno de *aliasing*. La frecuencia de muestreo en un CD convencional es de 44.1 kHz, algo más del doble de la frecuencia máxima de audición humana.
- Cuantificación: los valores de amplitud de cada muestra también pueden tener valores infinitos. Es por ello por lo que se limitan los valores que pueden tener las muestras a unos predefinidos para preservar la capacidad de memoria del ordenador.

Figura 3-1. Esquema de conversión analógico-digital

Tras esta conversión analógica-digital ya tenemos nuestro audio almacenado en el ordenador y podemos trabajar con él. El siguiente paso para la obtención de una representación tiempo-frecuencia es convertir la señal de audio

en el dominio de la frecuencia.

3.1.2 Dominio frecuencial

Para convertir la señal temporal al dominio de la frecuencia se emplea la transformada discreta de Fourier (DFT) cuya fórmula se muestra en la Ecuación 3-1.

$$X[f] = \sum_{n=-\infty}^{+\infty} x[n] * e^{-i2\pi fn}$$

Ecuación 3-1. Transformada Discreta de Fourier

Para cada muestra de la señal temporal se convierte al dominio frecuencial lineal. Normalmente el espectro de $X[f]$ se aproxima al cálculo de la DFT aplicando una ventana de tamaño N a la señal $x[n]$. Así resulta la transformada de Fourier de tiempo reducido (STFT). La fórmula para calcular la componente f en la muestra t de la señal $x[n]$ se muestra en la Ecuación 3-2.

$$X[t, f] = \sum_{k=0}^{N-1} w[k] * x[tN + k] * e^{-\frac{i2\pi kf}{N}}$$

Ecuación 3-2. Transformada de Fourier de Tiempo Reducido

En la Ecuación 3-2 $w[k]$ es una ventana que puede adquirir diversas formas: rectangular, *Hamming*, *Blackman*, etcétera. El cómputo de la STFT se realiza mediante un algoritmo llamado transformada rápida de Fourier (FFT). Tras calcularlo tenemos entonces un conjunto de grupos de muestras de $x[n]$ enventanados y pasados al dominio de la frecuencia. De cada uno de estos grupos podemos adquirir una representación frecuencial, poniendo en el eje de abscisas las frecuencias y en el eje de ordenadas los niveles que alcanzan dichas frecuencias en la ventana. Se suele pasar a unidades logarítmicas, por lo que el eje de ordenadas estaría en decibelios.

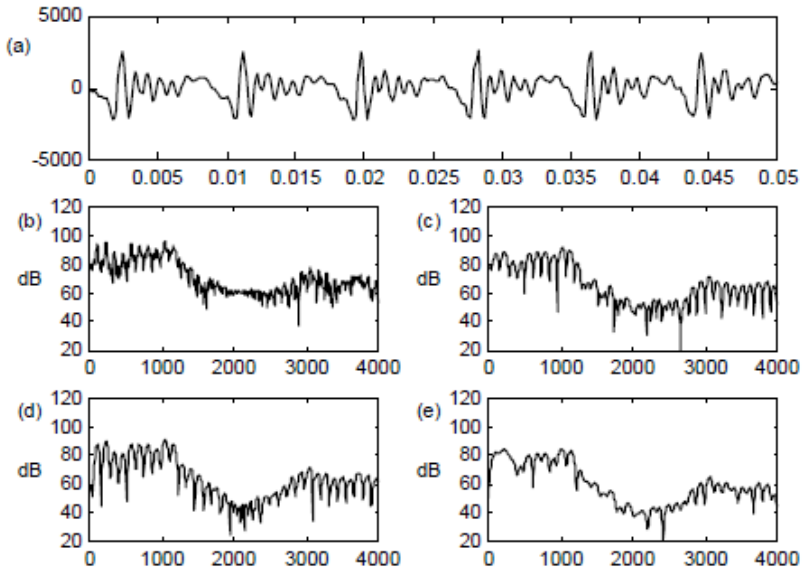

Figura 3-2. Distintos ejemplos de espectros

En la Figura 3-2 se muestra un ejemplo donde se representa (a) una señal de voz y su espectro con (b) una ventana rectangular de 30 milisegundos, (c) una ventana rectangular de 15 milisegundos, (d) una ventana *Hamming* de 30 milisegundos y (e) una ventana *Hamming* de 15 milisegundos.

3.1.3 Espectrograma

Para obtener una representación tiempo-frecuencia tenemos que hacer la DFT de la señal cada cierto tiempo. En el eje de abscisas representamos el tiempo transcurrido y en el eje de ordenadas las frecuencias. El nivel de intensidad de esa frecuencia se representa en el espectrograma, donde las zonas más oscuras indican mayor nivel de decibelios y las zonas más claras menor nivel. En la Figura 3-3 tenemos un ejemplo de espectrograma.

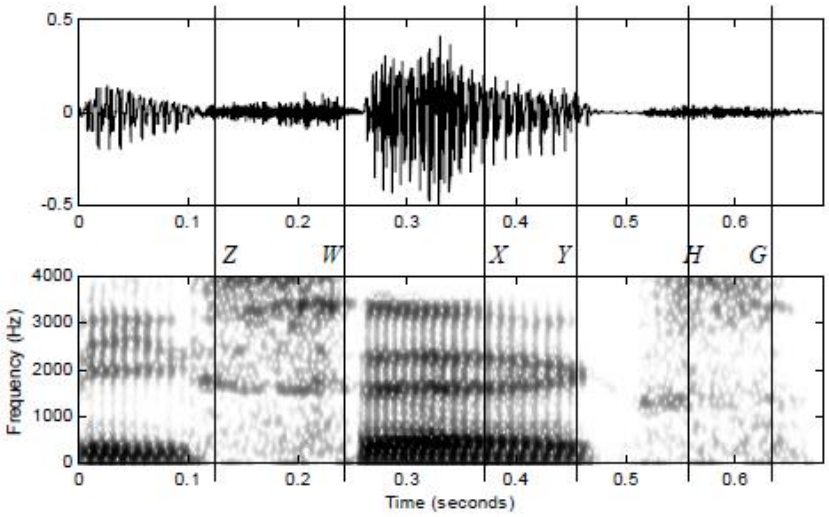

Figura 3-3. Audio con su espectrograma

Este espectrograma representa el eje de ordenadas, es decir, las frecuencias, de manera lineal. Sin embargo, a veces queremos obtener información en ciertas bandas de frecuencia. Para ello hacemos pasar el resultado de la FFT por un banco de filtros. Estos filtros pueden estar distribuidos de diversas maneras.

La escala de Mel está relacionada con la percepción auditiva humana. Estudios como el de Stevens et al. [44] (1937) observan que el oído humano actúa como un filtro y que concentra su actividad en ciertas partes del espectro de frecuencia. Esta característica se aproxima por el banco de filtros de Mel (Figura 3-4), donde existen más filtros y con bandas más estrechas en las zonas de baja frecuencia que en las zonas de alta frecuencia, donde los filtros son pocos y de banda ancha.

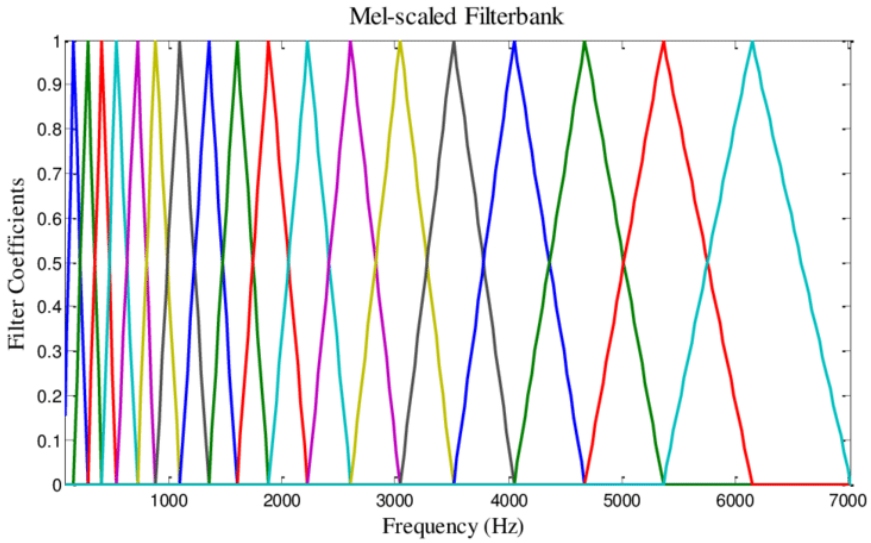

Figura 3-4. Banco de filtros de Mel

Este procedimiento lo siguen distintos tipos de escala, como las bandas críticas (Zwicker et al. [45] (1980)) o la transformada *Constant-Q* (Brown et al. [46] (1991)). En este último caso la escala de frecuencias no está uniformemente espaciada, sino que su distribución es geométrica. Otro ejemplo son los filtros *Gammatone* (Patterson et al. [47] (1992)). Estos filtros son más anchos que los de Mel, pero están distribuidos de manera parecida: en las frecuencias bajas tenemos más filtros que en las frecuencias altas. Se muestran en la Figura 3-5.

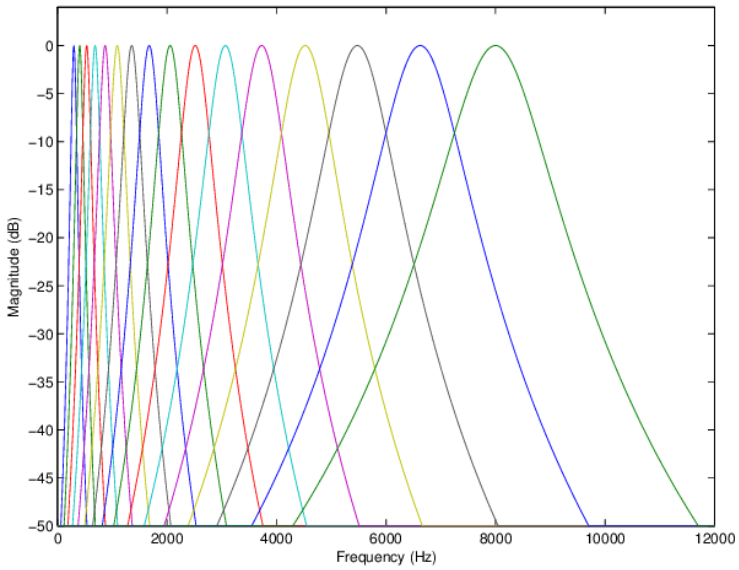

Figura 3-5. Banco de filtros *Gammatone*

Una vez elegido el filtro tendremos un resultado distinto en el espectrograma. En la Figura 3-6 se muestra (a) la forma de onda del audio, (b) el espectrograma de frecuencia lineal, (c) el espectrograma de frecuencia de Mel y (d) el espectrograma de la transformada *Constant-Q*.

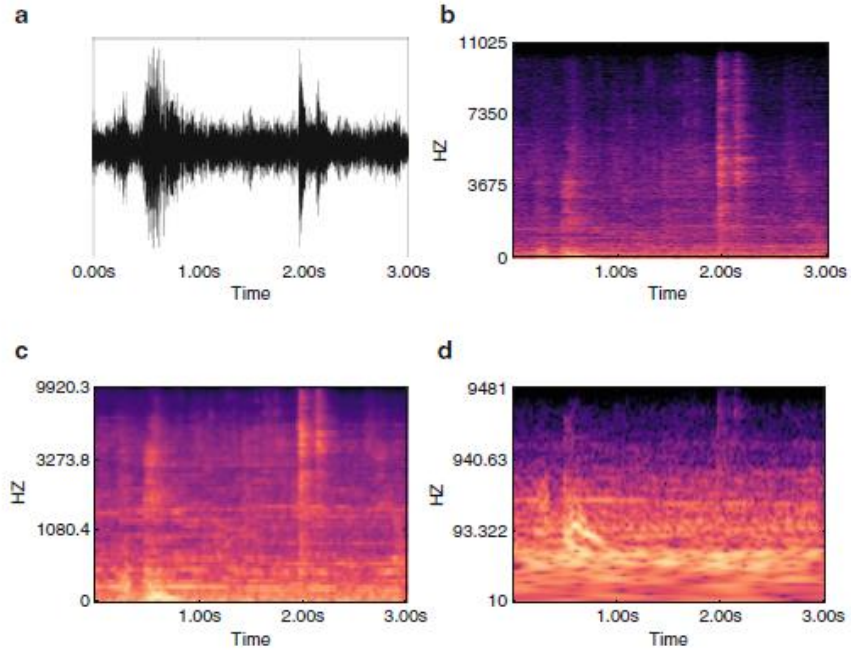

Figura 3-6. Distintos ejemplos de espectrogramas

Estas representaciones visuales del audio pueden ser utilizadas como datos de entrada a nuestro clasificador. Sin embargo, como vimos en el capítulo anterior se suele aplicar un paso más y se calculan unos valores numéricos, llamados coeficientes, para que sean estos los que se pasen como entrada al sistema y no la imagen del espectrograma completo.

Los coeficientes más frecuentes a la hora de trabajar con audio son los coeficientes cepstrales a la frecuencia de Mel (MFCC) (Peltonen et al. [48] (2002), Valero et al. [49] (2012)). Para computar estos coeficientes tenemos que calcular el logaritmo de cada ventana de espectrograma a las que le hicimos la DFT, como vimos anteriormente. Seguidamente a cada ventana le aplicamos la Transformada Discreta del Coseno (Discrete Cosine Transform, DCT). Los valores de las amplitudes resultantes son los denominados coeficientes cepstrales a la frecuencia de Mel.

Normalmente se almacenan los 13 primeros MFCC de cada extracto de espectrograma, ya que los siguientes no contienen información relevante sobre el audio. El conjunto de vectores de MFCC de todas las ventanas del audio es el que se pasa como parámetro de entrada al modelo. En este caso los datos de entrada del modelo serían menos pesados si los comparamos computacionalmente con un espectrograma completo de un audio. A su vez existen más alternativas en cuanto al cálculo de coeficientes se trata, como por ejemplo los coeficientes cepstrales *Gammatone* (GFCC) (Phan et al. [50] (2016), Valero et al. [51] (2012)).

3.2. Deep Learning

Una vez adquiridos los espectrogramas o los coeficientes de entrada, necesitamos un sistema para procesarlos. En las últimas décadas los algoritmos de aprendizaje automático han evolucionado hasta tal punto que conviven en nuestro día a día. Nuestros smartphones, televisores y electrodomésticos llevan insertados algoritmos de inteligencia artificial. Una consecuencia de esto es la cantidad de definiciones que han surgido alrededor de este campo y que pueden dificultar la comprensión de este. Es por ello por lo que necesitamos un mapa conceptual (Figura 3-7) para adentrarnos en este mundo.

3.2.1 Mapa conceptual

Inteligencia Artificial

En primer lugar, definiremos la Inteligencia Artificial (Russell [52] (2014)). Una de las definiciones que podemos proporcionar es que la Inteligencia Artificial es aquella disciplina que busca la creación de máquinas que puedan imitar comportamientos inteligentes. Esta disciplina engloba todos estos sistemas, desde un brazo mecánico en una línea de fabricación hasta el algoritmo que controla los movimientos de un juego de ajedrez por ordenador. Dentro de esta disciplina podemos diferenciar entre Inteligencia Artificial débil o fuerte.

Una Inteligencia Artificial débil es aquella que está programada para realizar una tarea en concreto, y si ese mismo sistema se dedica a realizar otra tarea, fallará. Todos los sistemas en la actualidad son débiles. Por el contrario, una Inteligencia Artificial fuerte es aquella que es capaz de realizar varias tareas, adaptándose al entorno. No existe ningún ejemplo de este tipo, pero es un campo del conocimiento apasionante en un futuro.

En definitiva, toda Inteligencia Artificial está programada para realizar un comportamiento inteligente. Existen subdisciplinas dentro de la inteligencia artificial para cada uno de los comportamientos que se quiere imitar. Por ejemplo, la robótica persigue que los sistemas sean capaces de realizar movimientos. Existe otro campo relacionado con el procesamiento del lenguaje o la voz. Otra capacidad del ser humano es la visión y existe otro campo para resolver problemas de este estilo. Estos sistemas realizarán sus tareas según se les haya programado, pero existe una capacidad humana que permite adaptar todos nuestros comportamientos según el tipo de problema al que nos enfrentemos. Es la capacidad de aprendizaje.

Aprendizaje Automático

El Aprendizaje Automático (Alpaydin [53] (2014)) es una rama de la Inteligencia Artificial que persigue cómo dotar a las máquinas de la capacidad de aprendizaje. Esta subdisciplina no programa un sistema para que se mueva, programa el sistema para que aprenda a moverse. Lo mismo podemos decir para los demás campos anteriormente mencionados. Cuando escuchamos *Machine Learning* estamos ante un problema en el cual el sistema utilizado va a aprender a realizar la tarea. Existen varias subramas, como si de un árbol se tratase, en las que se divide el Aprendizaje Automático dependiendo de cómo va a realizarse esta tarea de aprender.

Aprendizaje Supervisado

En primer lugar, nos encontramos con el aprendizaje supervisado. En este tipo de aprendizaje se le proporciona al sistema dos cosas: un conjunto de datos de entradas y sus correspondientes categorías. Estas categorías son las salidas que debería dar nuestro sistema y que las ha puesto un ser humano, de ahí el nombre de aprendizaje supervisado. El sistema ajustará sus parámetros para que su salida se parezca mucho a las etiquetas proporcionadas y así, cuando se le proporcione un nuevo dato de entrada sea capaz de clasificarlo con una probabilidad alta de acierto. Algunos de los sistemas empleados para aprendizaje supervisado son los sistemas de regresión lineal, las máquinas de soporte de vectores o las redes neuronales.

Aprendizaje No Supervisado

En segundo lugar, nos encontramos con el aprendizaje no supervisado. En este caso solo se le introduce al sistema datos de entrada sin etiquetar. Es el propio sistema que se encarga de aprender similitudes entre ellos y alcanza abstracciones interesantes. Todavía es un campo el cual se está avanzando, pero todo indica que la verdadera evolución del aprendizaje automático pasa por esta disciplina, ya que evitaría la acción humana al estar etiquetando datos de entrada. Algunos de los sistemas empleados para aprendizaje automático son el análisis de componentes principales (PCA) o las técnicas de *clustering* como el *K-means*.

Aprendizaje Reforzado

Por último, tenemos un tercer paradigma dentro del aprendizaje automático que se llama el aprendizaje reforzado. En él se persigue que el sistema tome acciones dentro de un ambiente que maximice la señal de recompensa. Es un sistema muy extendido en el campo de los videojuegos, creando sistemas que son capaces de resolver problemas muy diversos.

Para concluir este apartado de mapa conceptual podemos decir que las redes neuronales, que explicaremos en sucesivos apartados, no son más que un tipo de sistema de aprendizaje supervisado para dotar a una máquina con la capacidad de aprender a realizar una tarea. En nuestro caso, la tarea a realizar por nuestra red neuronal

será la de clasificar tipos de sonidos ambientales.

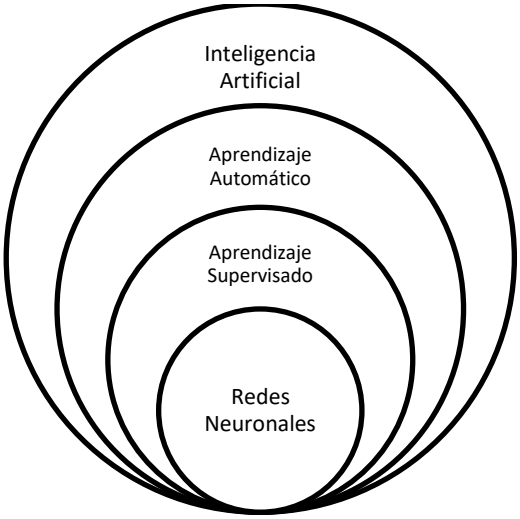

Figura 3-7. Mapa conceptual

3.2.2 De la neurona a la red

Para entender las redes neuronales (Müller [54] (1995)) vamos a adentrarnos en el sistema más sencillo y que fue el primero en desarrollarse: el perceptrón.

El perceptrón

El perceptrón o neurona (Figura 3-8) es la unidad básica de procesamiento de una red. Realiza una operación de suma ponderada. Es decir, asigna un peso a las entradas de esta neurona y las suma entre sí. Existe un parámetro llamado sesgo o *bias* que poseen todas las neuronas y es una entrada siempre a uno que se multiplica por un valor. Este valor evita un resultado en la suma ponderada de cero en el caso de que todas las entradas sean cero.

El perceptrón soluciona problemas sencillos, como aquellos resueltos por una recta de regresión. Sin embargo, se evidenció sus limitaciones a la hora de lidiar con problemas de clasificación más complejos, como el problema de la puerta XOR (Bapu Ahire [55] (2017)). La solución fue añadir más neuronas y más capas de neuronas, creando así el perceptrón multicapa o también conocido como las redes neuronales artificiales (ANN).

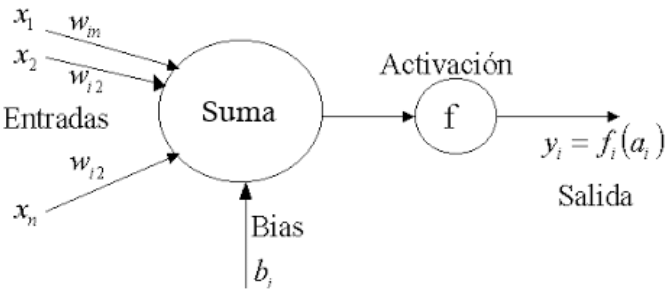

Figura 3-8. Esquema completo de una neurona

Sin embargo, una red de neuronas que solo realiza sumas ponderadas colapsaría en una única neurona capaz de realizar todas las sumas ponderadas en una. Así, para evitar este colapso de la red y seguir manteniendo la estructura de múltiples neuronas y múltiples capas, cada neurona realiza dos tipos de operaciones. La primera es la suma ponderada que hemos hablado, una función lineal ya que es un sumatorio, y la segunda es pasar dicha

suma ponderada por una función no lineal, la función de activación. Algunos ejemplos de funciones de activación las mostramos en la Figura 3-9.

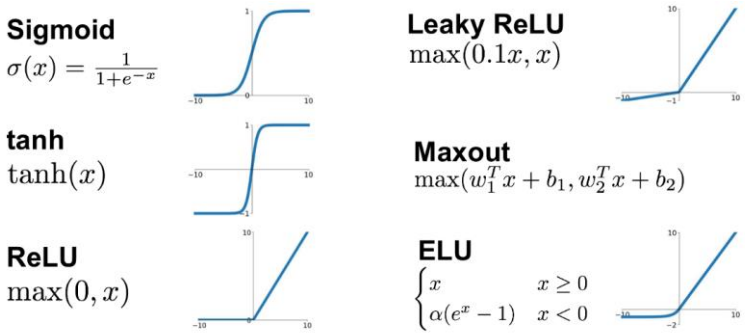

Figura 3-9. Ejemplos de funciones de activación

Red neuronal

Así tenemos el esquema completo de una neurona. Para conformar la red necesitamos una capa de entrada, que recibirá los datos de entrada del modelo, una serie de capas intermedias que ajustarán sus pesos para adaptarse a dichos datos, y una capa de salida, que proveerá el resultado de la red (Figura 3-10).

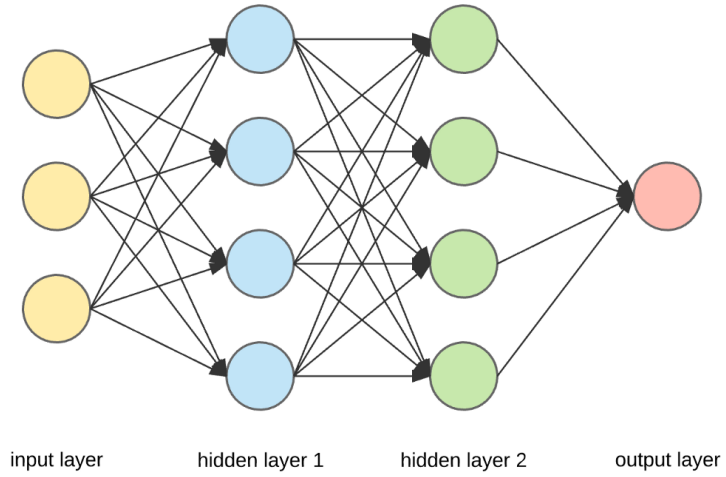

Figura 3-10. Esquema de una red neuronal artificial

Existen distintos tipos de capas de salida para clasificar los datos de entrada. Por ejemplo, la capa *softmax* muestra como salida una sola clase, la más probable que se corresponda con el dato de entrada. Se utiliza para los problemas de clasificación de audio, como el nuestro. La capa *sigmoid* se emplea cuando hay más de una clase en el dato de entrada, ya que devuelve la probabilidad de que el audio pertenezca a cada una de las clases de salida. Se utiliza para problemas de etiquetado o detección de audio.

Cuantas más capas intermedias tenga nuestra red mayor capacidad de detectar los detalles de nuestros datos tendrá. Es por eso por lo que el nombre de *Deep Learning* (Goodfellow [56] (2016)) se refiere a este tipo de aprendizaje en el que las redes neuronales tienen numerosas capas intermedias. A su vez, estas capas están totalmente conectadas entre sí. Es decir, todas las salidas de las neuronas de una capa están conectadas a todas las neuronas de la capa siguiente. Este tipo de capas reciben el nombre de capas densamente conectadas.

3.2.3 Prueba y error

Función de coste

Una vez tenemos nuestra red es hora de entrenarla con nuestros datos y que sea capaz de resolver nuestro problema. Como el título indica este entrenamiento se hace a base de prueba y error. Al iniciar nuestra red los pesos y sesgos tendrán un valor aleatorio. Al pasar un dato de entrenamiento por nuestra red y al llegar a la capa de salida, lo más seguro es que el resultado sea erróneo. Para cuantificar este error existen numerosas funciones de pérdida (*loss function*), por ejemplo, el error cuadrático medio. Ahora queremos mejorar nuestra red para que el error disminuya. Entonces entra en juego el algoritmo de *Backpropagation*.

Backpropagation

El algoritmo de *Backpropagation* (Rumelhart et al. [57] (1986)) calcula todas las derivadas parciales de la función de pérdidas con respecto a todos los pesos y sesgos del modelo. A priori se antoja un cálculo eterno, ya que existen miles de pesos y sesgos en una red. Sin embargo, este algoritmo se vuelve eficiente ya que va calculando las derivadas parciales de atrás hacia delante. Es decir, empieza calculando las derivadas de la última capa oculta, después con estos datos calcula los de la capa anterior, y así sucesivamente. Antes de que el algoritmo de *Backpropagation* se crease el cálculo de las derivadas se hacía por fuerza bruta, calculando todos los posibles caminos desde la capa de entrada hasta la capa de salida, lo que se convertía en una tarea ineficiente.

Función de reducción del coste

Una vez tenemos todas las derivadas parciales necesitamos un algoritmo de reducción de error. Existen varios, pero el más conocido es el algoritmo del descenso del gradiente (Cui [58] (2018)). Estos tipos de algoritmos se nutren de las derivadas parciales calculadas en el paso anterior, es decir, el gradiente, y calcula los valores que tienen que poseer los pesos y sesgos para que en la próxima iteración el error disminuya. Mostramos un ejemplo en la Figura 3-11.

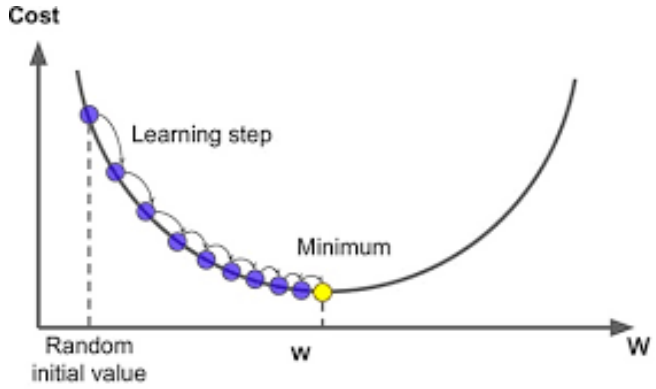

Figura 3-11. Algoritmo de descenso del gradiente

Así se habría completado una iteración. Sin embargo, los datos de entrenamiento no se constituyen de una sola muestra. Consiste en un gran set de datos, que se van introduciendo en lotes de datos más pequeños (*batches*) a lo largo de más de una iteración (*epoch*). Así el algoritmo de reducción del error irá mejorando hasta alcanzar el mínimo global, aquel conjunto de valores de pesos y sesgos que hayan reducido al mínimo el error.

Para llegar a él es necesario ajustar un valor llamado *learning rate* o *step size*. Cuanto mayor es este valor, más grande van a ser los pasos que se dé y más rápido iterará, pero puede que nunca llegue al mínimo global. Cuanto menor es este valor, más pequeños serán los pasos y con mayor seguridad se alcanzará el mínimo global, pero se necesitaría de mucho tiempo para alcanzarlo. Hay que alcanzar un compromiso entre un valor mayor o menor. Ambos casos los podemos apreciar en la Figura 3-12.

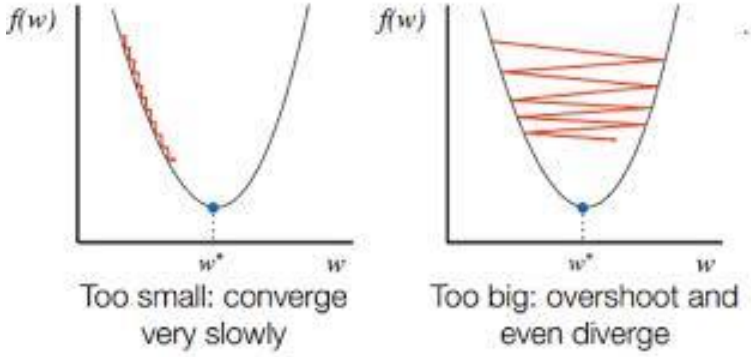

Figura 3-12. Ejemplos de diferentes *learning rates*

3.2.4 Resultado

Una vez entrenada la red con todos nuestros datos podemos tener varios resultados. El primero de ellos es que la red no converge y no sea capaz de proporcionarnos un resultado. El siguiente caso es que la red sí converge y nos proporcione un resultado. Sin embargo, el resultado puede ser satisfactorio o no según tres tipos de situaciones, que pasamos a explicarlas a continuación (Figura 3-13).

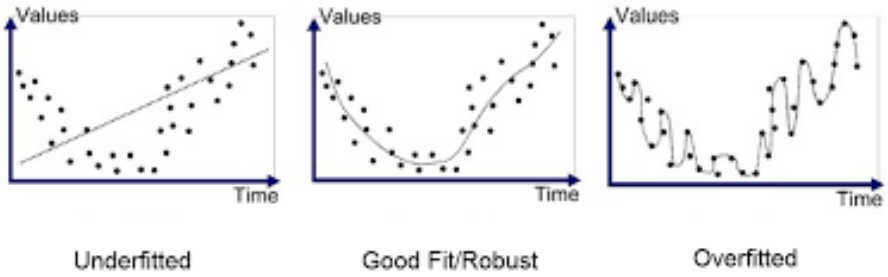

Figura 3-13. Ejemplos de resultados de una red

Subajuste

El subajuste es el fenómeno por el cual la red no ha sido capaz de ajustarse a los datos de entrenamiento y no ejerce la clasificación de la manera adecuada. Uno de los principales motivos del subajuste se debe a la rigidez del modelo. Es decir, el modelo es poco complejo para los tipos de datos que queremos clasificar y una solución sería incluir más capas de neuronas para que el modelo gane en complejidad y abstracción.

Buen ajuste

Un buen ajuste se produce cuando el modelo ha sido capaz de adaptarse a los datos de entrenamiento y realiza una buena clasificación. Además, el modelo ha conseguido la capacidad de generalizar, por lo tanto, cuando se le introduzca un nuevo dato nunca visto la red lo clasificará con una alta probabilidad de acierto.

Sobreajuste

Este es el objetivo que se persigue a la hora de entrenar una red, que alcance un buen ajuste y sea capaz de generalizar. Eso se consigue iterando en un gran número de veces. Sin embargo, al entrenar mucho una red esta puede llegar a memorizar los datos de entrada, perdiendo la capacidad de generalización. En este caso, si se le da un nuevo dato nunca visto por la red, esta lo clasificará de manera errónea. Este fenómeno se conoce como sobreajuste y es muy común que suceda en el campo del aprendizaje automático.

Entrenamiento, validación y test

Para evitar el problema del sobreajuste se aplican distintas técnicas a los datos. Una de las más comunes es dividir el *dataset* completo de entrenamiento en dos subconjuntos: entrenamiento y validación. Normalmente se emplea las proporciones aproximadas de 80% de los datos para el conjunto de entrenamiento y el 20% restante para el conjunto de validación. En una iteración, el conjunto de entrenamiento servirá para calcular el error de la red y modificar los pesos y sesgos de nuestro modelo. Después se le pasará el conjunto de validación para calcular el error del modelo, pero a diferencia con el error de entrenamiento, el error de validación no servirá para modificar pesos y sesgos.

A lo largo del entrenamiento veremos como el error de entrenamiento y el de validación irán disminuyendo paulatinamente. Sin embargo, llegará un punto en el que el error de entrenamiento seguirá disminuyendo, pero el error de validación comenzará a crecer. Es en este punto cuando la red está empezando a memorizar los datos de entrenamiento y está perdiendo capacidad de generalización. Por lo tanto, el punto para que la red esté bien ajustada es aquel con el mínimo error de validación (Figura 3-14).

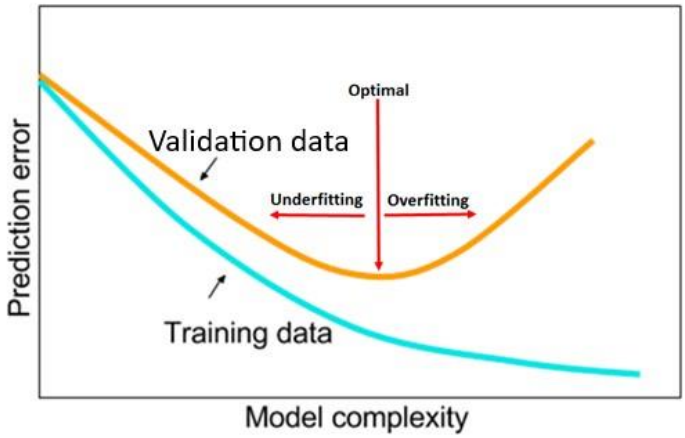

Figura 3-14. Error de entrenamiento y validación

Normalmente existe otro conjunto de datos que no se emplean ni en el entrenamiento ni en la validación. Una vez que el modelo ha sido entrenado se le pasa este conjunto de datos para hacer pruebas, llamado conjunto de test, y calcular parámetros como la precisión del modelo o el error. Este conjunto debe poseer igual número de datos para todas las clases y así al realizar las medidas de la forma más equitativa posible.

A veces no existen *datasets* con un número elevado de muestras para cada clase, o tienen un número desigual de datos para cada clase. En este caso se emplean técnicas como la de *Data Augmentation* (Taylor et al. [59] (2017)) para incrementar el número de datos y así generar un buen *dataset*. En el caso de un conjunto de datos de imágenes, aplicar *Data Augmentation* consiste en aplicar rotaciones, incrementos o decrementos de tamaños para así poseer más variedad de datos con los que trabajar.

Habríamos completado así el entrenamiento de una red neuronal, explicando todos los elementos.

3.2.5 Redes Neuronales Convolucionales

En los apartados anteriores nos hemos centrado en explicar toda la teoría detrás de las redes neuronales basándonos en una red neuronal artificial que tenía capas densamente conectadas. Sin embargo, existe otra arquitectura de red neuronal que ha avanzado a pasos agigantados en los últimos años y que son capaces de resolver todo tipo de problemas, como ya vimos en el capítulo anterior. Estas son las redes neuronales convolucionales (CNN) (Koushik [60] (2016)). A continuación, vamos a explicar los tipos de capas que poseen estas redes y que realizan distintas operaciones.

Convolución

Las capas de convolución, como su nombre indican, realizan una operación de convolución a los datos de entrada. Normalmente los datos de entrada suelen ser una representación en dos dimensiones, como una imagen o un espectrograma. La capa de entrada de una red neuronal convolucional tiene el mismo tamaño que los datos de entrada. Es por eso por lo que todas las imágenes de entrada tienen que poseer el mismo tamaño.

El siguiente paso es realizar la operación de convolución (Figura 3-15). Para ello la imagen será recorrida por un filtro (*kernel*) de izquierda a derecha y de arriba abajo. Este filtro constituye la matriz de pesos y sesgos de la capa. El paso de píxel a píxel en horizontal y en vertical se puede decidir con un parámetro llamado *stride*. El tamaño del *kernel* puede ser variable. La operación de convolución consiste en multiplicar los valores de la imagen por los del *kernel* y sumarlos. Este resultado se pasa a la siguiente capa de la red.

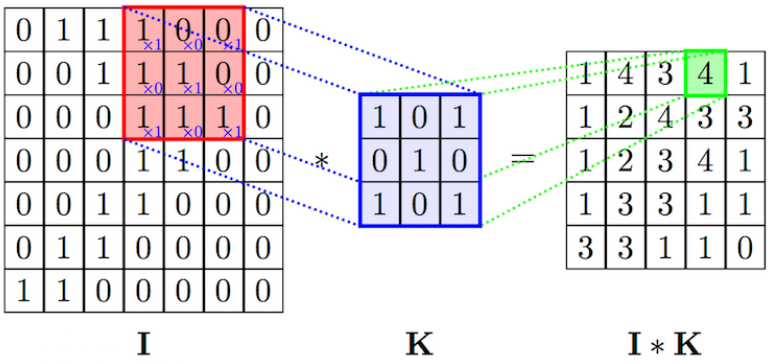

Figura 3-15. Operación de convolución

Esta operación que hemos descrito sería para una sola convolución en la capa, lo que equivale a una sola neurona en una capa. Es por ello por lo que las capas poseen más de un filtro de pesos y sesgos, y por lo tanto realizan más de una operación de convolución.

Pooling

Otra de las capas empleadas en una red neuronal convolucional es la capa de *pooling*. Para detectar patrones más complejos en las imágenes necesitamos implementar múltiples capas de convolución una detrás de otra, lo que repercute en una alta capacidad de computación y una gran cantidad de tiempo a la hora de entrenar. Para reducir el número de parámetros se utilizan las capas de *pooling*.

Estas capas actúan de manera similar a las capas de convolución. Hacen pasar un filtro a la imagen y devuelven un resultado. En este caso cogen los píxeles dentro de una vecindad y devuelven un único valor. La operación más común es la de *max pooling* (Figura 3-16). En ella se cogen píxeles en una vecindad, por ejemplo, 2x2, y se pasa a la capa siguiente el valor del píxel más alto.

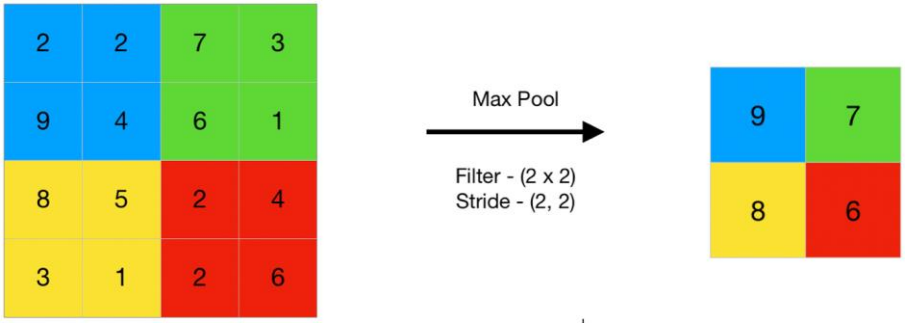

Figura 3-16. Operación de *max pooling*

Podemos modificar el tamaño de la matriz del filtro y el parámetro *stride*. Gracias a esta operación podemos ir acoplando capas convolucionales con sus respectivas capas de *pooling* para aumentar el grado de profundidad y abstracción de nuestra red.

Flatten

Por último, como hemos visto en este capítulo, el final de una red neuronal era una capa de salida densamente conectada. Es por eso por lo que en algún momento tenemos que transformar los datos bidimensionales en una capa de neuronas con la que podamos conectar a las siguientes capas densas o a la capa de salida.

La capa *flatten* realiza esa función, transforma la anterior capa de convolución o de *pooling* en una capa de neuronas para conectarla densamente a la siguiente. Es una capa esencial a la hora de mezclar dos tipos de arquitecturas, como vimos en el capítulo anterior con las redes híbridas.

Dropout y BatchNormalization

Existen distintos métodos para evitar el sobreajuste en una red neuronal. En este caso se trata de dos capas que se insertan en el modelo y realizan distintas funciones, ambas con el mismo objetivo. Hablamos de los algoritmos de *Dropout* y de *BatchNormalization*.

El algoritmo de *Dropout* (Srivastava [61] (2014)) consiste en que, para cada iteración del modelo, un número aleatorio de neuronas no participarán en el entrenamiento, poniendo sus entradas a cero. Un ejemplo del funcionamiento lo encontramos en la Figura 3-17. Un valor habitualmente utilizado es 0.5, lo que significa que el 50% de las neuronas seleccionadas aleatoriamente no participarán en el entrenamiento. Este valor lo utilizaremos en nuestro modelo.

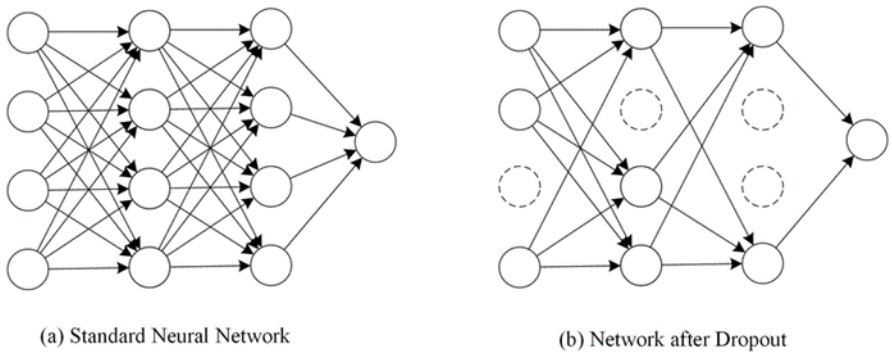

Figura 3-17. Funcionamiento del *Dropout*

Otro algoritmo para evitar el sobreajuste que emplearemos será el *BatchNormalization*. Este método consiste en normalizar la activación de la capa anterior en cada iteración (Ioffe et al. [63] (2015)). Nosotros emplearemos estas capas en sustitución de las capas *Dropout* para el segundo modelo.

Cross-validation

La validación cruzada tiene el mismo objetivo que las capas de *Dropout* y *BatchNormalization*, evitar el sobreajuste. Métodos de *cross-validation* hay muchos, pero todos se basan en la misma idea. Hacen particiones del *dataset* en entrenamiento y test y entrena el modelo, todas las veces necesarias hasta que todas las muestras de la base de datos hayan sido tomadas como test. Así nos aseguramos de que la precisión del modelo no depende de la partición aleatoria de datos que hayamos hecho, ya que todos los datos van a formar parte del entrenamiento y del test.

Uno de los algoritmos más comunes para implementar validación cruzada es el llamado *k-fold* (Krishni [65] (2018)). Consiste en dividir la base de datos en *k* lotes de datos. El entrenamiento se realizará con *k*-1 lotes de datos y el test se hará con un lote de datos solo. Se guardan los resultados de precisión y error de esa evaluación

y se repite el entrenamiento con un lote de test distinto y los restantes para entrenamiento.

El algoritmo *k-fold* no tiene en cuenta el número de muestras de cada clase que existan en la base de datos total. Nosotros buscamos que la separación en datos de test y entrenamiento se haga de la manera más homogénea posible. Para ello se implementa el algoritmo *stratified k-fold*, que usaremos en nuestros experimentos. Este método sí tiene en cuenta el número de muestras totales de cada clase y las separa según el número de subdivisiones que le hayamos especificado. Un ejemplo visual lo podemos encontrar en la Figura 3-18.

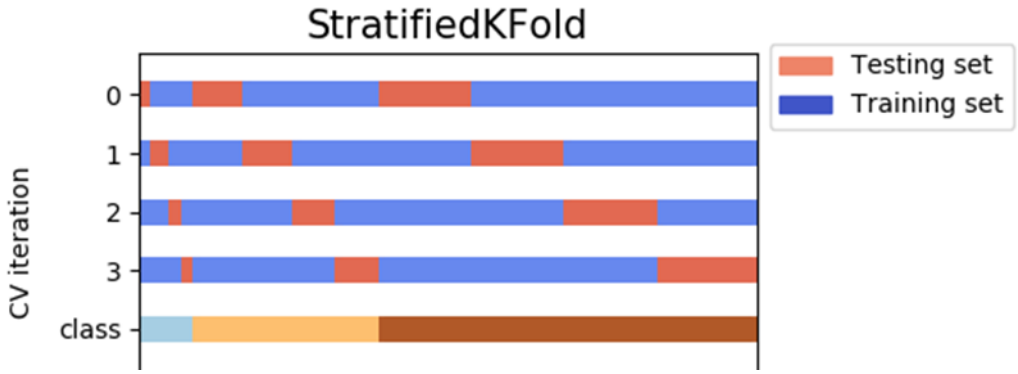

Figura 3-18. Ejemplo de validación cruzada para cuatro divisiones

Matriz de confusión

La matriz de confusión (Narkhede [64] (2018)) es un método de visualización de las predicciones de un modelo, dividiéndolo entre las clases que participan en la clasificación. En las filas de la tabla se sitúan las clases observadas y en las columnas se sitúan las clases predichas por la red neuronal. Para entender mejor la distribución de datos en una matriz de confusión vamos a ejemplificarlo con un clasificador binario (Tabla 3-1).

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

Tabla 3-1. Matriz de confusión de un clasificador binario

En este tipo de matriz se pueden dar cuatro tipos de resultados:

- *TN* (*True Negative*): Muestras cuyo valor real observado es 0 y el clasificador ha evaluado correctamente como valor 0.
- *FP* (*False Positive*): Muestras cuyo valor real observado es 0 y el clasificador ha evaluado incorrectamente como valor 1.
- *FN* (*False Negative*): Muestras cuyo valor real observado es 1 y el clasificador ha evaluado incorrectamente como valor 0.

- *TP* (*True Positive*): Muestras cuyo valor real observado es 1 y el clasificador ha evaluado correctamente como valor 1.

Esta matriz de confusión se puede extender *N*-dimensionalmente, tantas filas y columnas como clases tenga el modelo. Gracias a este método podemos apreciar qué clases evalúa mejor nuestra red neuronal y qué clases son más propensas a confusión. Por último, es importante destacar que para tener un resultado homogéneo en la matriz de confusión todas las clases deben tener aproximadamente el mismo número de muestras de test. Por el contrario, los resultados mostrados podrían verse sesgados.

Tras este extenso apartado tenemos nuestra red neuronal convolucional completa, como podemos ver en la Figura 3-19. Las técnicas de entrenamiento y los resultados que nos puede dar esta red son idénticos si se tratara de una red neuronal artificial, que hemos explicado con anterioridad. En el capítulo siguiente introduciremos nuestro modelo propuesto para resolver nuestro problema de clasificación de sonido ambiental.

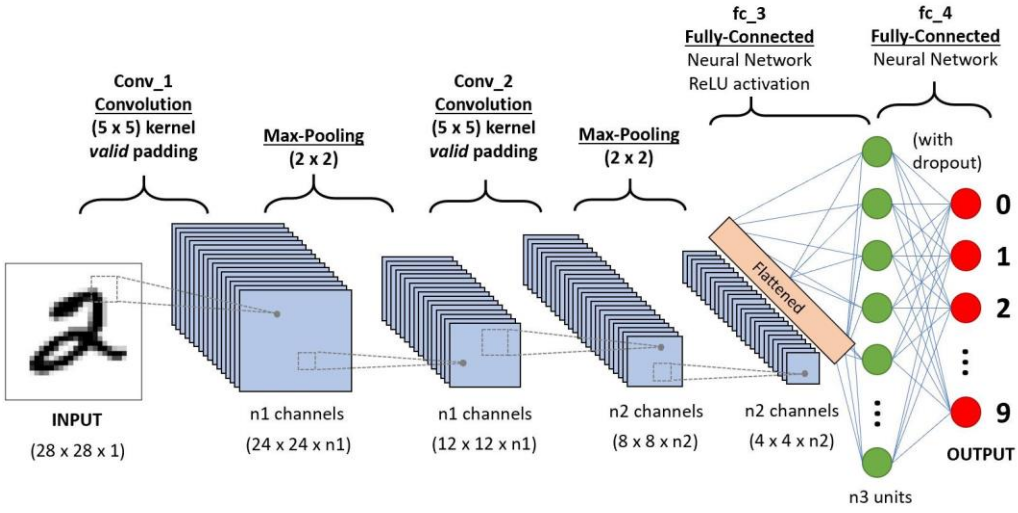

Figura 3-19. Esquema completo de una red neuronal convolucional

4 MÉTODO PROPUESTO

Una vez sentadas las bases teóricas del análisis de sonidos y los modelos de *Deep Learning* en los dos anteriores capítulos, en este vamos a desarrollar nuestro método propuesto para la resolución del problema inicialmente planteado. Iremos desgranando todas las herramientas que hemos empleado para la consecución de dicho objetivo.

4.1. Material

4.1.1 Hardware

Las tareas de *Deep Learning* son de una elevada carga computacional. Como hemos visto, un modelo puede contar con miles de parámetros. Cada uno de ellos se puede modificar y por cada ligera modificación el resultado final es distinto. Es por ello por lo que los requisitos en cuanto a hardware son más exigentes.

Tanto una Unidad Central de Procesamiento (CPU) como una Unidad de Procesamiento Gráfico (GPU) realizan cálculos matemáticos en un ordenador. La principal diferencia es que la CPU está destinada a tareas de propósito general mientras que la GPU realiza tareas muy específicas. Es por eso por lo que una CPU tiene pocos núcleos de procesamiento en comparación con una GPU.

Las tareas específicas de una GPU son diversas: visualización de videojuegos en gran calidad, tareas de renderizado 3D, posproducción audiovisual, y como no, inteligencia artificial. Tener una GPU instalada en el equipo reducirá considerablemente el tiempo de entrenamiento de nuestra red, igual que podrá soportar redes más extensas y profundas.

En los últimos años han surgido nuevas propuestas por parte de *Amazon Web Services* (AWS) y de las *Tensor Processing Unit* (TPU) de *Google Cloud Plataform*. Consisten en proveer de servicios de computación en la nube, pudiendo personalizar las características que vamos a necesitar para nuestra tarea, el procesador, la memoria o la GPU.

Para nuestro trabajo empleamos un portátil de uso personal con las siguientes características, mostradas en la Tabla 4-1. Ante todas estas posibilidades anteriormente mencionadas, nuestra estación de trabajo no disponía de una tarjeta gráfica dedicada. Podemos considerar como GPU la unidad de procesamiento gráfico que viene por defecto en el portátil. Sin embargo, esta GPU es de bajo rendimiento, y es por eso por lo que los tiempos de ejecución, de procesado y de entrenamiento de nuestro modelo se van a ver mermados.

Otra característica de nuestra estación de trabajo es que no dispone de memoria del tipo SSD, toda la capacidad de memoria de almacenamiento que tiene es del tipo HDD. Las memorias SSD son más rápidas a la hora de acceder a los datos en comparación con las memorias HDD. Este es otro motivo por el cual nuestro modelo tardará más en ejecutarse.

A pesar de lo comentado, decidimos escoger nuestro portátil como estación de trabajo ya que no disponíamos de otra opción más potente. Además, al ser nuestro ordenador personal podemos disponer de él durante todo el tiempo que queramos, sin ningún coste adicional y sin ninguna interrupción.

Componente	Modelo
CPU	Intel Core i7-6500U @ 2.50 GHz
GPU	Intel HD Graphics 520
Memoria RAM	8 GB
Memoria HDD	1 TB
Sistema Operativo	Windows 10

Tabla 4-1. Especificaciones del equipo

4.1.2 Software

Lenguaje de programación

En este trabajo hemos elegido *Python* como lenguaje de programación para programar nuestro modelo. En la actualidad, *Python* es uno de los lenguajes de programación más usados en todo el mundo. En el campo de la Inteligencia Artificial, el *Machine Learning* y el *Deep Learning* la mayoría de código desarrollado es en este lenguaje, compartiendo protagonismo con el lenguaje de programación *R*.

Si bien es cierto que durante el grado no hemos dado una asignatura de programación en *Python* como tal, es un lenguaje de programación muy fácil de comprender y el aprendizaje se puede realizar de manera autodidacta gracias al conocimiento en otros lenguajes como *C* y *Java* que sí hemos aprendido en la carrera. Es por esta facilidad a la hora de programar y comprender el lenguaje por lo que *Python* es la primera opción.

Entorno de desarrollo y librerías

Existen multitud de opciones para descargar y empezar a programar en *Python*. En nuestro caso decidimos descargar la distribución de *Python* llamada *Anaconda* y desarrollada por *Continuum Analytics*. La descargamos y la optimizamos para programar en la versión 3.6 de *Python*. Para la escritura de código, ejecución y depuración de este utilizamos el entorno de programación *Spyder*. *Anaconda* viene con multitud de librerías preinstaladas, de las cuales empleamos las siguientes para nuestro trabajo:

- *Numpy*: principal librería para la computación científica. Posee funciones para trabajar en el campo del álgebra, transformadas de Fourier, matrices *N*-dimensionales y el campo de la estadística.
- *Pandas*: librería que posee herramientas para el análisis de datos de manera rápida, potente y fácil de usar.
- *Matplotlib*: librería para la representación y visualización de datos de distintas formas, ya sea estática, animada, interactiva, en 2D, en 3D, etcétera.
- *Random*: librería que aporta funciones para la generación de números aleatorios.

Además de las librerías anteriormente comentadas necesitamos disponer de herramientas para el procesamiento de señales de audio. Existen multitud de posibilidades, pero en nuestro caso optamos por la librería *libROSA*. Este paquete posee funciones para el análisis de sonidos y música. Sus posibilidades son extensas, como por ejemplo extraer características del sonido, aplicar efectos en él, descomponerlo en su parte temporal y frecuencial, visualizar su espectrograma, etcétera.

Con todos estos paquetes podemos comenzar a realizar el preprocesamiento de nuestras señales de audio. En la documentación proporcionada por las páginas web de cada una de las librerías buscamos la información que necesitamos y aplicamos las funciones adecuadas, al igual que solucionamos los posibles errores que nos

surgieron.

Framework empleado para Deep Learning

Existen multitud de *frameworks* o entornos de trabajo para trabajar en el campo del aprendizaje automático. Los más utilizados son *TensorFlow*, *Keras*, *PyTorch*, *Theano* y *Caffe* (Figura 4-1). Para nuestro trabajo vamos a utilizar *Keras* para programar nuestros modelos de *Deep Learning*.

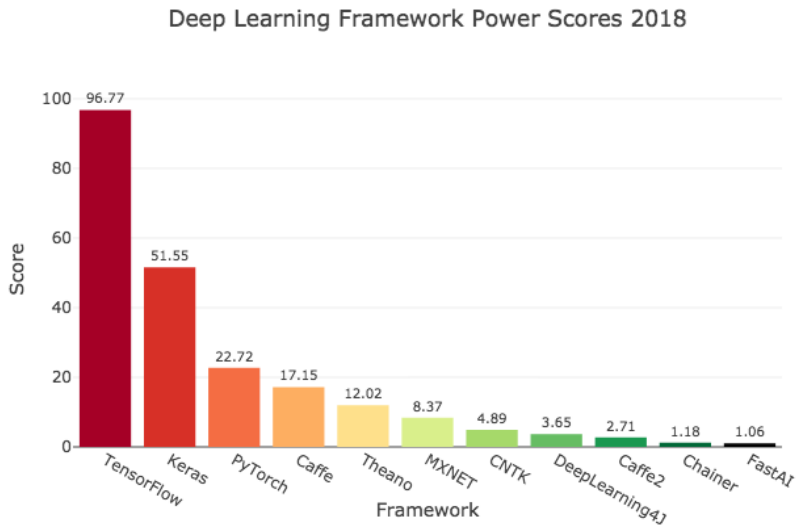

Figura 4-1. Puntuación de los diferentes *frameworks* en 2018

Esta elección viene motivada por la facilidad que ofrece *Keras* a la hora de programar redes neuronales. Se puede importar en nuestro código como si de una librería se tratase y con unas líneas ya hemos desarrollado una arquitectura completa que podemos entrenar y ejecutar. Para ofrecer esta facilidad a la hora de programar, *Keras* se sostiene de un *framework* de menor nivel de intuición como puede ser *Theano* o *TensorFlow*. Para nuestro trabajo elegimos este último como nuestro *backend*.

Como hemos comentado, *Keras* es un entorno de alto nivel de intuición. Para nuestro trabajo importamos las siguientes herramientas principales para la creación de una red neuronal y que describimos en el anterior capítulo:

- *Sequential*: función que se emplea para la creación de un modelo secuencial. Es decir, nuestro modelo seguirá una estructura de capas apiladas, donde las salidas de una capa irán conectadas a las entradas de la siguiente.
- *Dense*: crea una capa de neuronas densamente conectadas.
- *Conv2D*: crea una capa de convolución en dos dimensiones.
- *MaxPooling2D*: crea una capa de *max pooling* en dos dimensiones.
- *Flatten*: capa que transforma los datos multidimensionales de su entrada en datos unidimensionales a su salida.
- *Activation*: este método aplica la función de activación que queramos a la salida de nuestra capa.

Por último, otra librería que hemos empleado para trabajar con modelos de *Deep Learning* ha sido *scikit-learn*. Esta librería aporta infinidad de herramientas para el aprendizaje automático, como por ejemplo modelos de regresión, de *clustering*, visualización de resultados. Para nuestro trabajo empleamos esta potente librería para

visualizar la matriz de confusión y para implementar la validación cruzada.

4.2. Base de datos

Para nuestro proyecto empleamos la base de datos proporcionada por Salamon et al. [41] llamada *UrbanSound8K*. Este *dataset* está compuesto por 8732 extractos de audio en formato WAV, con una duración no superior a 4 segundos. Estos extractos de audio están sacados de la página web de *FreeSound* y clasificados en 10 categorías: aire acondicionado, claxon de coche, niños jugando, ladrido de perro, máquina taladradora, motor al ralentí, disparo de arma, martillo neumático, sirena y música sonando en la calle.

Estos sonidos se encuentran mezclados en diez carpetas. Además, el *dataset* viene con un fichero *comma-separated values* (CSV) donde se encuentra la información de cada extracto de audio, como por ejemplo su nombre, el tiempo de comienzo y de final del audio, la carpeta en la que está situado, el identificador de clase, etcétera. En la Tabla 4-2 se muestra cada número de identificador y su clase correspondiente.

Número identificador	Clase	Número identificador	Clase
0	*air_conditioner*	5	*engine_idling*
1	*car_horn*	6	*gun_shot*
2	*children_playing*	7	*jackhammer*
3	*dog_bark*	8	*siren*
4	*drilling*	9	*street_music*

Tabla 4-2. Identificadores de clases de la base de datos

La principal razón por la que hemos elegido este *dataset* y no otro es por el tamaño. Esta base de datos nos proporcionaba más de 800 muestras de audio para cada una de las diez clases con un tamaño total de 6.60 GB. Además, en este *dataset* todas las muestras de audio están manualmente etiquetadas, lo que hace que la tarea de clasificación sea más sencilla.

Por poner otro ejemplo de base de datos tenemos la propuesta en la competición de *Kaggle* mencionada en la introducción. Este *dataset* está formado por más de 9500 muestras de audio distribuidas desigualmente en 41 categorías distintas. Además, no todas las muestras de este *dataset* se encuentran manualmente verificadas. Tratar con una base de datos tan extensa iba a resultar en alargar los tiempos de ejecución del trabajo.

Por eso decidimos trabajar con el *dataset UrbanSound8K*, ya que con este podemos aproximarnos a la tarea de clasificación de audio de una manera simplificada. Las investigaciones en este campo usan bases de datos más extensas al poseer estaciones de trabajo más potentes.

4.3. Implementación del primer modelo

Este primer modelo supuso nuestro acercamiento a la codificación y ejecución de nuestra primera red neuronal convolucional. Para ello nos basamos en el modelo propuesto por Salamon et al. [42] en el cual utilizan el *dataset UrbanSound8K* y crean una arquitectura de red neuronal convolucional que describiremos más adelante.

4.3.1 Primeros pasos

Tras leer el documento descargamos la base de datos, la descomprimimos y la situamos en la carpeta en la cual íbamos a trabajar. En *Anaconda* descargamos los paquetes que nos hacían falta, como *Keras*, *libROSA* y *scikit-*

learn. También lo configuramos para trabajar en *Python 3.6*. Comenzamos abriendo un nuevo documento .py en *Spyder* e importando las librerías que vamos a usar (Código 4-1).

```
#%% Importamos las librerías que vamos a utilizar
import keras
from keras.layers import Activation, Dense, Dropout, Conv2D, Flatten,
MaxPooling2D, BatchNormalization
from keras.models import Sequential
from keras.optimizers import Adam
from keras.models import load_model
from keras.callbacks import EarlyStopping, ModelCheckpoint

import h5py

import librosa
import librosa.display

import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt

from sklearn.metrics import confusion_matrix
from keras.utils import plot_model
```

Código 4-1. Librerías del primer modelo

Además de estas librerías definimos una función llamada `plot_confusion_matrix()` que nos servirá para visualizar correctamente la matriz de confusión. La librería `h5py` nos ayudará a guardar los parámetros del modelo, al igual que las funciones `load_model`, `EarlyStopping` y `ModelCheckpoint`.

4.3.2 Preprocesamiento de los datos

Una vez definidas las librerías comenzamos con el preprocesamiento de los datos. Esta es sin lugar a duda la parte más importante de todo el trabajo, ya que sin unos datos bien procesados el modelo entrenaría y clasificaría de manera errónea.

Nuestra estación de trabajo no es tan potente como ya describimos anteriormente, por lo tanto, nuestro problema de clasificación lo reducimos de 10 clases a 7. Nuestro objetivo era calcular el espectrograma de los extractos de audio y que estos fuesen los datos de entrada de nuestro modelo. Para ello todos los extractos de audio debían tener la misma longitud, por eso nuestra primera decisión fue eliminar todas aquellas muestras que tuviesen menos de 2.95 segundos.

Tras esto nuestra base de datos quedó reducida a las siete siguientes clases con sus muestras: 0-*air_conditioner* (997 muestras), 2-*children_playing* (981 muestras), 4-*drilling* (831 muestras), 5-*engine_idling* (973 muestras), 7-*jackhammer* (842 muestras), 8-*siren* (901 muestras) y 9-*street_music* (1000 muestras). Las clases excluidas fueron: 1-*car_horn* (218 muestras), 3-*dog_bark* (697 muestras), 6-*gun_shot* (34 muestras).

Además, para mayor facilidad a la hora de programar cambiamos los identificadores de clase para que todos estuviesen entre el rango de 0 a 6. Entonces la clase *jackhammer* pasó de 7 a 1, la clase *siren* pasó de 8 a 3 y la clase *street_music* pasó de 9 a 6. Todos estos pasos se muestran en el Código 4-2.

```
#%% Leemos los datos del archivo csv que viene en el paquete de datos
UrbanSound8K
data = pd.read_csv('UrbanSound8K/metadata/UrbanSound8K.csv')

v_data=pd.DataFrame(data=None,columns=['slice_file_name','fold',
                                       'classID', 'class'])

for row in data.itertuples():
    if (row.end - row.start) >= 2.95:
        if (row.classID != 1) and (row.classID != 3) and
                (row.classID != 6):

            fila=pd.DataFrame([[row.slice_file_name,row.fold,
                                row.classID,row._8]],
                              columns=['slice_file_name','fold',
                                'classID', 'class'])
            v_data=v_data.append(fila,ignore_index=True)

data['path']='fold'+data['fold'].astype('str')+'/'+
                data['slice_file_name'].astype('str')
v_data['path']='fold'+v_data['fold'].astype('str')+'/'+
                v_data['slice_file_name'].astype('str')

for row in v_data.itertuples():
    if row.classID == 7:
        columna=pd.Series(int(1), name='classID', index=[row.Index])
        v_data.update(columna)
    elif row.classID == 8:
        columna=pd.Series(int(3), name='classID', index=[row.Index])
        v_data.update(columna)
    elif row.classID == 9:
        columna=pd.Series(int(6), name='classID', index=[row.Index])
        v_data.update(columna)
```

Código 4-2. Preprocesamiento de los datos en el primer modelo

A continuación, nos disponemos a calcular el espectrograma de todas las muestras. En este paso ya comprobamos lo que será el mayor hándicap en nuestro trabajo, los largos tiempos de ejecución. La decisión de quedarnos con las muestras que tuviesen una duración mayor o igual a 2.95 segundos fue porque, al ejecutar la función `melspectrogram()` con un audio de esta duración, la función te devolvía una matriz cuadrada 128x128. Es decir, 128 muestras en el eje temporal y 128 muestras en el eje frecuencial.

Si el audio fuese más largo el eje temporal del espectrograma contaría con más muestras, y viceversa. Aquí tomamos una decisión de compromiso y optamos por mantener el tamaño de todos los espectrogramas en 128x128. Los siguientes pasos fueron convertir el espectrograma en decibelios y transformar los datos de la matriz en `float32`, ya que el resultado de todas estas operaciones te daba una matriz de datos en `float64`, así reduciríamos tiempo de computación en las siguientes actividades.

Tras esto mezclamos aleatoriamente la variable con todas las muestras y creamos tres variables vacías, las cuales vamos a rellenar con nuestras muestras para entrenamiento, validación y test. En nuestra primera aproximación separamos manualmente las muestras. Creamos unos bucles para que fuesen recorriendo la variable de muestras totales y dividir en el 70% de las muestras para entrenamiento, el 15% para validación y el 15% para test. Esta

aproximación la realizamos para todas las clases y así conseguimos la misma distribución de datos de entrenamiento, validación y test (Código 4-3).

```
#%% Calculamos el espectrograma de los archivos y los vamos añadiendo
a la variable D
D=[]

for row in v_data.itertuples():
    y,sr=librosa.load('UrbanSound8K/audio/' + row.path,duration=2.95 )
    s=librosa.feature.melspectrogram(y=y,sr=sr)
    ps=librosa.power_to_db(s)
    ps=ps.astype(np.float32)
    # Si no tienen tamaño uniforme no se incluye en la variable D
    if ps.shape != (128,128):
        continue
    D.append((ps,row.classID))

random.shuffle(D)

D_train=[]
D_valid=[]
D_test=[]

for classID in range(7):
    c=0
    for i in range(len(D)):
        if D[i][1] == classID:
            if (c < round(0.7*cont_clase[classID])):
                D_train.append(D[i])
                c=c+1
            elif (c >= round(0.7*cont_clase[classID])) and
                        (c < round(0.85*cont_clase[classID])):
                D_valid.append(D[i])
                c=c+1
            else:
                D_test.append(D[i])
                c=c+1
```

Código 4-3. Obtención de datos en el primer modelo

Los últimos pasos del procesamiento de los datos son mezclar aleatoriamente los datos de entrenamiento, validación y test y crear las matrices con el tamaño y formato que admite la red neuronal. Los datos de entrada tendrán dimensión 128x128x1. Cada uno de los datos tendrá una etiqueta del tipo categórico. Esta etiqueta no es más que un vector numerado con siete posiciones, tantas como clases queramos clasificar. En todas las posiciones hay un 0 como valor excepto en la posición que corresponda el identificador de clase, en la que habrá un 1. Ambos datos, tanto el espectrograma como su etiqueta categórica, se pasan como entrada al modelo.

Para finalizar el preprocesamiento, una fase que no se puede pasar por alto es la normalización de los datos. Es fundamental antes de proceder al entrenamiento de la red neuronal, ya que además de homogeneizar los valores de los datos, estos se vuelven computacionalmente más manejable. Así nuestro modelo no tardará tanto en finalizar su entrenamiento. Estos pasos los mostramos en el Código 4-4.

```
#%% Preparamos el dataset, desordenando las muestras y separándolas en
muestras para entrenar, para evaluar y para testear
dataset_train=D_train
dataset_valid=D_valid
dataset_test=D_test

random.shuffle(dataset_train)
random.shuffle(dataset_valid)
random.shuffle(dataset_test)

train=dataset_train
valid=dataset_valid
test=dataset_test

X_train, y_train = zip(*train)
X_valid, y_valid = zip(*valid)
X_test, y_test = zip(*test)

X_train = np.array([x.reshape( (128, 128, 1) ) for x in X_train])
X_valid = np.array([x.reshape( (128, 128, 1) ) for x in X_valid])
X_test = np.array([x.reshape( (128, 128, 1) ) for x in X_test])

# Normalización
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)

X_train = (X_train - mean)/std
X_valid = (X_valid - mean)/std
X_test = (X_test - mean)/std

y_train = np.array(keras.utils.to_categorical(y_train, 7))
y_valid = np.array(keras.utils.to_categorical(y_valid, 7))
y_test = np.array(keras.utils.to_categorical(y_test, 7))
```

Código 4-4. Normalización de los datos en el primer modelo

4.3.3 Arquitectura propuesta

Nuestro modelo cuenta con tres capas convolucionales 2D, todas con la función de activación *ReLu* y las dos primeras cuentan con una capa de *Max Pooling* 2D a su salida. Los filtros de estas capas son de tamaño 5x5, la primera capa convolucional cuenta con 24 filtros y las dos últimas con 48 filtros. Las capas de *Max Pooling* tienen un filtro de 4x2.

Tras esta primera mitad de arquitectura le sigue una capa *Flatten* seguida de una capa de *Dropout*. Continúa el modelo con una capa densamente conectada de 64 neuronas, también seguida de una capa de *Dropout* y con la función de activación *ReLu*. Finalmente se encuentra la capa de salida con 7 neuronas y función de activación *softmax*. En la Figura 4-2 mostramos un esquema visual de nuestro primer modelo.

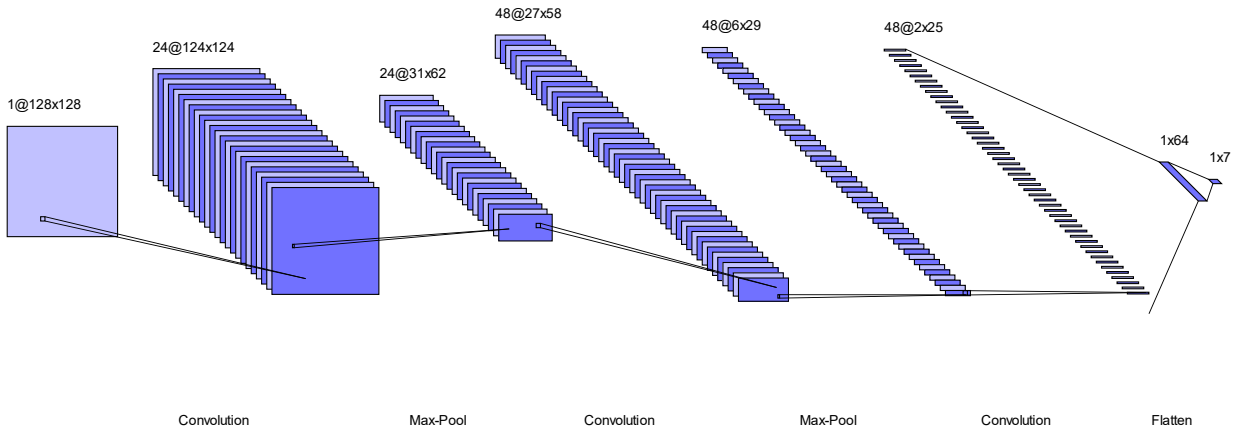

Figura 4-2. Esquema del primer modelo

Finalmente decidimos los parámetros para compilar el modelo. Como función de pérdida, también llamada función de coste, elegimos `categorical_crossentropy` ya que estamos utilizando la función de pérdida *cross-entropy* para nuestro problema de clasificación con varias clases. La fórmula para calcular la pérdida en un clasificador con *C* clases la mostramos en la Ecuación 4-1.

$$CE = -\sum_{i}^{C} y_i * \log y_i^{pred}$$

Ecuación 4-1. Función de pérdida *cross-entropy*

Para calcular la pérdida *cross-entropy* o *CE* tenemos: el valor de la clase real que le pasamos al modelo, y_i, que será 1 si el audio pertenece a esa clase y 0 en los demás casos; y el valor de la clase predicha por el modelo, y_i^{pred}, que nos lo proporcionará la capa *softmax*.

Esta capa devuelve a su salida la probabilidad de que el audio pertenezca a las distintas clases del modelo. En nuestro caso la capa *softmax* posee siete neuronas, una por cada clase (*C*=7), y para cada audio a clasificar esta capa mostrará en cada neurona la probabilidad de que pertenezca a esa clase. Todas las probabilidades mostradas por la capa final suman la unidad. La clase que el modelo intuya como correcta tendrá un valor cercano a 1 en la neurona correspondiente y las demás un valor cercano a 0.

Volviendo al cálculo del error, solo nos interesa averiguar *CE* para la clase a la que pertenece el audio, ya que el valor de y_i en la multiplicación será 1 y el resto 0. Se puede comprobar que cuando el valor de y_i^{pred} es cercano a 1, *CE* es cercano a 0.

Como función de reducción del coste, también llamado optimizador, elegimos Adam con un *learning rate* de 0.0001. Por último, decidimos que se mida la precisión del modelo. Esto nos devolverá en cada iteración del entrenamiento la precisión del entrenamiento y validación y la pérdida en el entrenamiento y la validación. Mostramos la arquitectura en el Código 4-5.

Para la definición matemática de precisión podemos tomar como ejemplo el clasificador binario descrito en el capítulo anterior, cuya matriz de confusión la mostramos en la Tabla 3-1. La precisión es la relación entre los datos correctamente clasificados por el modelo y el total de datos, tanto correctos como incorrectos (Ecuación 4-2).

Se trata de un porcentaje que puede resultar engañoso si no se tiene en cuenta otras medidas, como es el caso de la pérdida. La pérdida, como hemos visto antes, no se trata de un porcentaje. Su valor es la suma de los errores que el modelo ha cometido al clasificar y es la medida que automáticamente el modelo intenta minimizar con la función reducción de coste y el algoritmo de *Backpropagation*.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Ecuación 4-2. Precisión para un clasificador binario

```
#%% Construimos el modelo propuesto y lo compilamos
model = Sequential()
input_shape=(128, 128, 1)

model.add(Conv2D(24, (5, 5), strides=(1, 1), input_shape=input_shape))
model.add(MaxPooling2D((4, 2), strides=(4, 2)))
model.add(Activation('relu'))

model.add(Conv2D(48, (5, 5), padding="valid"))
model.add(MaxPooling2D((4, 2), strides=(4, 2)))
model.add(Activation('relu'))

model.add(Conv2D(48, (5, 5), padding="valid"))
model.add(Activation('relu'))

model.add(Flatten())
model.add(Dropout(rate=0.5))

model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(rate=0.5))

model.add(Dense(7))
model.add(Activation('softmax'))

adam = Adam(0.0001)

model.compile(
    optimizer=adam,
    loss="categorical_crossentropy",
    metrics=['accuracy'])
```

Código 4-5. Arquitectura del primer modelo

4.4. Implementación del segundo modelo

Durante nuestra investigación para realizar el capítulo del estado del arte en esta memoria pudimos comprobar como muchas de las arquitecturas que se proponían en las investigaciones tenían cuatro capas convolucionales en vez de tres. También nos dimos cuenta de este detalle en los modelos propuestos para la competición de *Kaggle*, de los cuales muchos de ellos consiguieron buena puntuación en el reto.

Por eso decidimos implementar un segundo modelo con una arquitectura nueva. Nos basamos en el trabajo del

usuario de *Kaggle* Razar [62] (2018) para desarrollar esta arquitectura. Los pasos del preprocesamiento de los datos son los mismos que para el primer modelo, lo único que alteramos es la arquitectura.

Como hemos comentado, este modelo tiene cuatro capas convolucionales 2D, cada una con 32 filtros de tamaño 4x10 y seguidas una capa de *Batch Normalization*, una función de activación *ReLu* y una capa de *Max Pooling* 2D con filtros 2x2. Continúa con una capa *Flatten*, otra densamente conectada de 64 neuronas, otra capa de *Batch Normalization* y la función de activación *ReLu*. Finalmente tenemos la capa de salida como en el modelo anterior, con 7 neuronas y la función de activación *softmax*. Como en el caso del modelo anterior mostramos un esquema visual de esta arquitectura en la Figura 4-3. Finalmente empleamos los mismos parámetros del modelo anterior para la compilación de este modelo (Código 4-6).

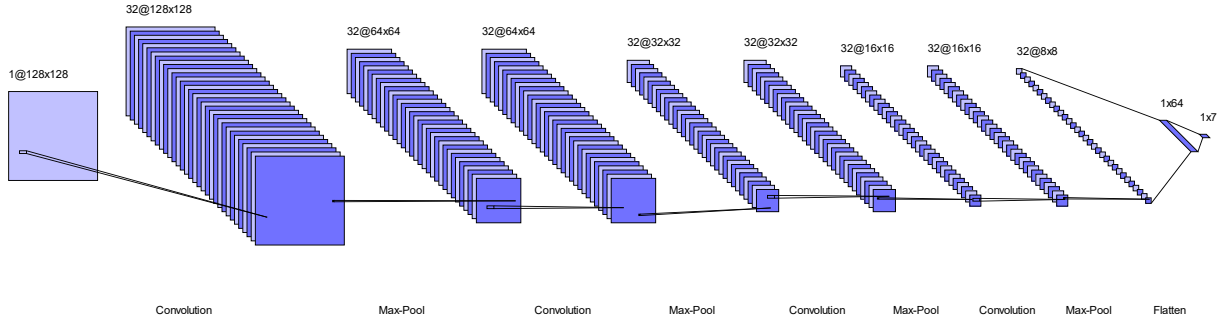

Figura 4-3. Esquema del segundo modelo

```
#%% Construimos el modelo propuesto y lo compilamos
model = Sequential()
input_shape=(128, 128, 1)

model.add(Conv2D(32, (4, 10), padding="same",
input_shape=input_shape))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D())

model.add(Conv2D(32, (4, 10), padding="same"))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D())

model.add(Conv2D(32, (4, 10), padding="same"))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D())

model.add(Conv2D(32, (4, 10), padding="same"))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D())

model.add(Flatten())
model.add(Dense(64))
model.add(BatchNormalization())
model.add(Activation("relu"))

model.add(Dense(7))
model.add(Activation("softmax"))

adam = Adam(0.0001)

model.compile(
    optimizer=adam,
    loss="categorical_crossentropy",
    metrics=['accuracy'])
```

Código 4-6. Arquitectura del segundo modelo

5 ENTRENAMIENTO Y RESULTADOS

Tras haber descrito el código que implementa nuestros modelos es el momento de entrenarlos con nuestros datos de entrenamiento, validación y test. En este capítulo pasaremos a evaluar el rendimiento de ambos modelos. Utilizaremos en primer lugar la división manual de la base de datos descrita anteriormente y pasaremos a introducir el método de la validación cruzada en nuestro entrenamiento.

5.1. Resultados del primer modelo

5.1.1 Configuración del entrenamiento

Este primer modelo está formado por tres capas convolucionales como describimos en el capítulo anterior. Es por eso por lo que decidimos entrenar el modelo a lo largo de 100 *epochs*, ya que el entrenamiento de una iteración se computaba rápidamente. Utilizamos un *batch size* de 64, lo que significa que los datos entran al modelo en lotes de 64 muestras.

Estuvimos buscando en la documentación que nos proporciona la página web de *Keras* la forma de guardar el modelo cuyos parámetros ajustados nos diesen el menor error de validación. La solución es pasar al modelo un conjunto de funciones que personalizan el entrenamiento, llamadas *callbacks*. Para nuestro modelo definimos dos funciones entre el conjunto que ofrece la librería de *Keras*:

- *ModelCheckpoint*: esta función guarda el modelo tras realizar una iteración si se cumple unas condiciones determinadas. En nuestro caso lo modificamos para que en cada iteración comprobase si el error de validación había disminuido. Si se cumplía entonces se hacía una copia de seguridad del modelo entero, con su arquitectura y los valores de los pesos y sesgos.
- *EarlyStopping*: esta función detiene por completo el entrenamiento del modelo si se cumplen unas condiciones. En nuestro caso quisimos que si en diez iteraciones el error de validación no había disminuido detuviese el entrenamiento.

Con todos estos parámetros ajustados el modelo estaba listo para ser entrenado. Le pasamos nuestros datos de entrenamiento y las funciones *callbacks*. También le pasamos los datos de validación, que como mencionamos en el capítulo sobre *Deep Learning* para análisis de audio no influían en la modificación de pesos y sesgos.

```
#%% Entrenamos el modelo
callbacks = [ModelCheckpoint(filepath='best_model.h5', verbose=1,
                             monitor='val_loss', save_best_only=True,
                             mode='min'),
             EarlyStopping(monitor='val_loss', patience=10, verbose=1,
                             mode='min')]

history = model.fit(
    x=X_train,
    y=y_train,
    epochs=100,
    batch_size=64,
    validation_data= (X_valid, y_valid),
    callbacks=callbacks)
```

Código 5-1. Entrenamiento del primer modelo

El entrenamiento de este primer modelo (Código 5-1) duró aproximadamente una hora y media. Cada *epoch* duraba en torno a un minuto en completarse y el modelo se entrenó a lo largo de 83 iteraciones. A pesar de que habíamos configurado el modelo para que iterase en cien ocasiones, la función `EarlyStopping` entró en juego y detuvo el entrenamiento ya que el error de validación no había disminuido más.

Como mencionamos en anteriores capítulos, si el entrenamiento se hubiese alargado más el error de validación habría comenzado a crecer, lo que sería el claro indicio de que nuestro modelo estaba sufriendo sobreajuste. Tras este entrenamiento visualizamos unas gráficas donde podemos apreciar la evolución de nuestro modelo, mostradas en Figura 5-1 y Figura 5-2.

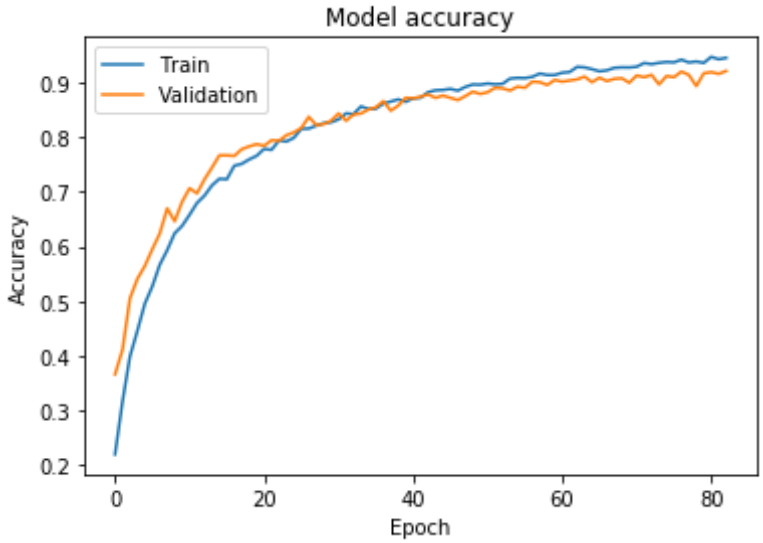

Figura 5-1. Evolución de la precisión del primer modelo

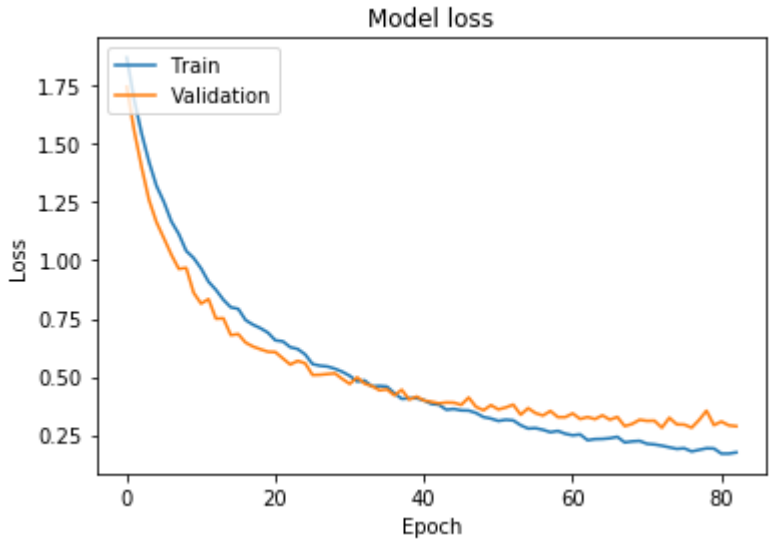

Figura 5-2. Evolución del error del primer modelo

5.1.2 Matriz de confusión

El entrenamiento nos dejó una copia de seguridad del modelo con los parámetros que hacían el error de validación mínimo. Concretamente la mejor combinación de valores de pesos y sesgos se dio en la *epoch* número 73. Se consiguió un error de validación de **0.28129** y una precisión en la validación de **0.9141**.

Estos resultados en el entrenamiento nos dejaron una buena impresión. En primer lugar, la curva del error de validación no se separa apenas de la curva del error de entrenamiento, lo cual significa que se está realizando un buen ajuste. En segundo lugar, el error de validación alcanza un valor mínimo cercano a los valores de pérdida de validación y test de los diversos estudios vistos en el capítulo del estado del arte, alrededor de 0.2.

A continuación, nos dispusimos a cargar el mejor modelo guardado para evaluarlo con los datos de test (Código 5-2). Estos datos han estado apartados durante todo el entrenamiento, por lo tanto, va a ser la primera vez que nuestra red neuronal clasifique estos datos.

```
#%% Mejor modelo
model_best = load_model('best_model.h5')

score = model_best.evaluate(x=X_test,
                            y=y_test,
                            batch_size=64)
```

Código 5-2. Evaluación del primer modelo

Esta evaluación de los datos de test nos proporcionó un resultado en la precisión del **0.9172** y un error del **0.2471**. Como vemos el valor de pérdida del test se acerca aún más al valor del estado del arte. Tras estos resultados en la evaluación nos disponemos a visualizar la matriz de confusión del modelo, empleando para ello la librería importada de *scikit-learn* y la función que previamente definimos.

La precisión del modelo, anteriormente calculada, nos indica el porcentaje de evaluaciones correctas sobre el total de evaluaciones hechas por el modelo. Esta métrica es bastante significativa a la hora de visualizar globalmente el rendimiento del modelo, pero a veces es interesante investigar sobre las predicciones que realiza el modelo en cada clase.

Para ello calculamos la matriz de confusión de nuestro modelo. En primer lugar, se visualiza la matriz de

confusión con todas las muestras de test que se les han pasado al modelo (Tabla 5-1). En segundo lugar, normalizamos los resultados al número de muestras de test verdaderas por cada clase (Tabla 5-2). Así conseguimos visualizar un resultado en tanto por uno dependiendo de las muestras de test observadas en cada clase.

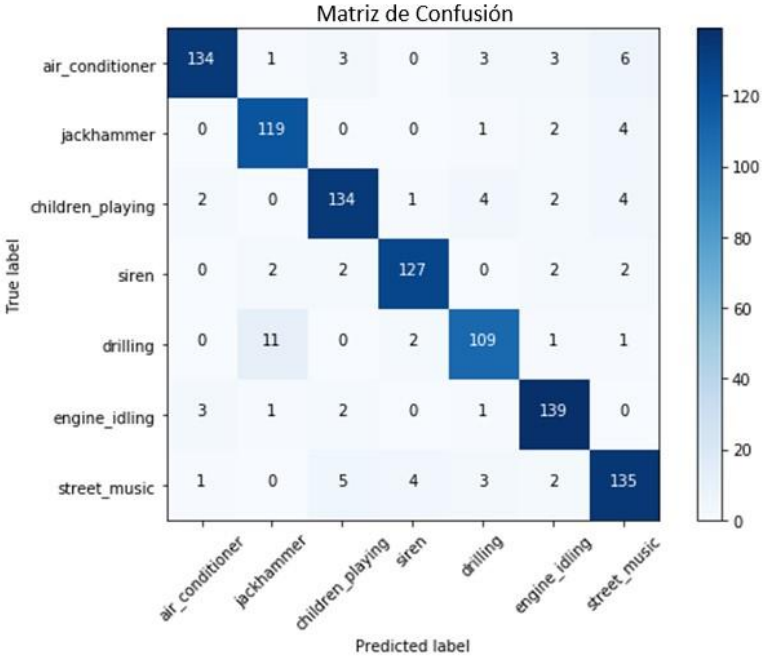

Tabla 5-1. Matriz de confusión del primer modelo

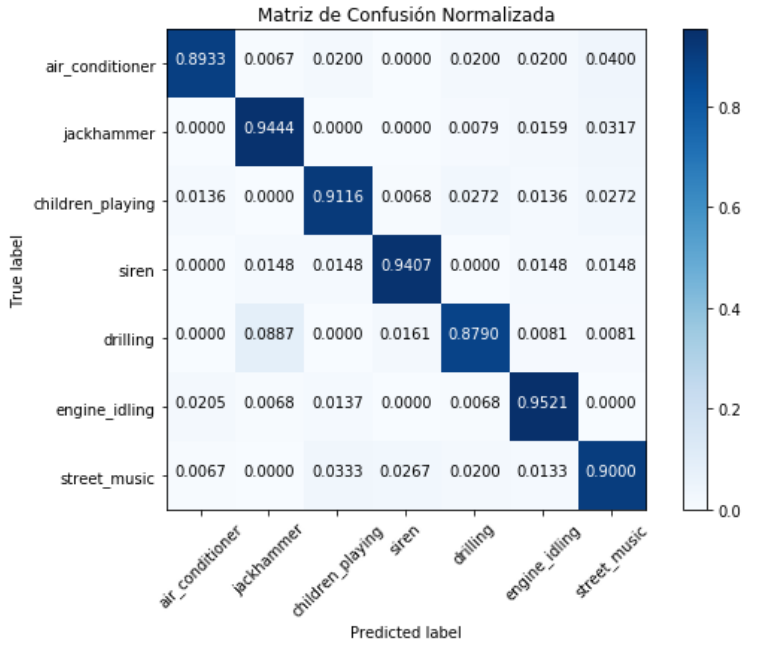

Tabla 5-2. Matriz de confusión normalizada del primer modelo

Como vemos nuestro modelo ha clasificado de manera idónea un 90% de las muestras de cada clase. Podemos destacar las 11 muestras de la clase *drilling* que han sido clasificadas por nuestro modelo como la clase *jackhammer*. Ambos sonidos son parecidos, ruidos agudos de maquinaria, por lo que encontramos lógico la clasificación incorrecta de estas muestras. Sin embargo, apenas suponen el 9% de las muestras de la clase

drilling.

5.2. Resultados del segundo modelo

5.2.1 Configuración del entrenamiento

Este modelo posee cuatro capas convolucionales, lo que se traduce en un mayor número de parámetros a ajustar. Es por esa razón por la que decidimos reducir el número de *epochs* a 50 para realizar el entrenamiento. Esto reduciría considerablemente el tiempo de entrenamiento. Mantuvimos el mismo tamaño del *batch* y las mismas funciones *callbacks*. Le pasamos como entrada los mismos datos de entrenamiento y validación (Código 5-3).

```
#%% Entrenamos el modelo
callbacks = [ModelCheckpoint(filepath='best_model.h5', verbose=1,
                             monitor='val_loss', save_best_only=True,
                             mode='min'),
             EarlyStopping(monitor='val_loss', patience=10, verbose=1,
                             mode='min')]

history = model.fit(
    x=X_train,
    y=y_train,
    epochs=50,
    batch_size=64,
    validation_data= (X_valid, y_valid),
    callbacks=callbacks)
```

Código 5-3. Entrenamiento del segundo modelo

En este caso el entrenamiento, como era de esperar, tardó más tiempo en ejecutarse. Cada iteración se completaba en algo más de 6 minutos. En este entrenamiento la función `EarlyStopping` no entró en juego, por lo que se completaron 50 *epochs*. El tiempo total empleado para el entrenamiento de este modelo fueron cinco horas. Como en el anterior caso visualizamos la gráfica de evolución del modelo, mostradas en Figura 5-3 y Figura 5-4.

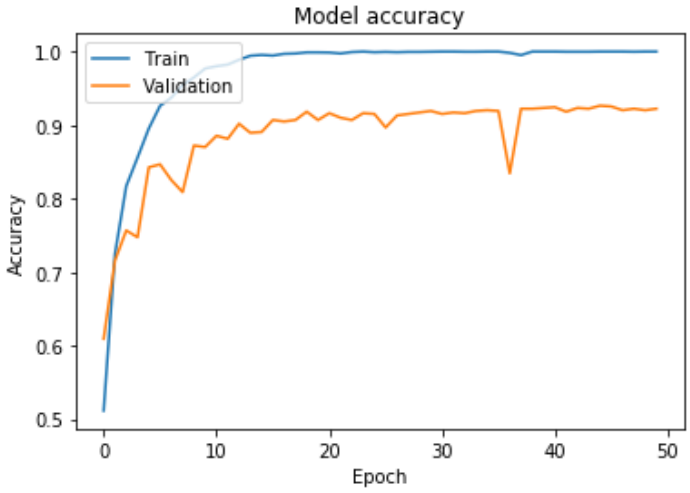

Figura 5-3. Evolución de la precisión del segundo modelo

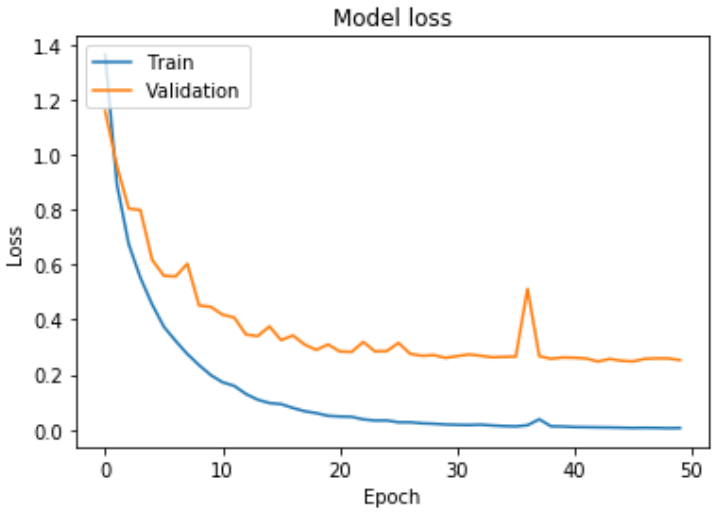

Figura 5-4. Evolución del error del segundo modelo

Vemos como la precisión de validación del modelo llega a estabilizarse en el 90%. Mirando la gráfica del error de validación sucede lo mismo para el valor cercano a 0.25. Apreciamos el repunte del error de validación en la iteración 37, donde la anterior había tenido un valor de 0.2659 y en esa alcanza un valor de 0.5121. No le damos mucha importancia puesto que en las siguientes iteraciones el modelo vuelve a situarse en torno a los valores anteriores.

5.2.2 Matriz de confusión

Realizamos el mismo procedimiento que el seguido para el modelo anterior. La arquitectura guardada en el entrenamiento por la función `ModelCheckpoint` fue la conseguida en la iteración número 43, donde se alcanzó un error de validación del **0.24796** y una precisión de validación del **0.9233**. Nos disponemos a cargar el mejor modelo y a realizar la evaluación para los mismos datos de test (Código 5-4).

```
#%% Mejor modelo
model_best = load_model('best_model.h5')

score = model_best.evaluate(x=X_test,
                            y=y_test,
                            batch_size=64)
```

Código 5-4. Evaluación del segundo modelo

Esta evaluación nos arrojó unos resultados similares a los vistos en el modelo anterior. Un error del **0.2595** y una precisión del **0.9162**. Aunque el error es ligeramente mayor en la evaluación de este modelo, solo lo entrenamos a lo largo de 50 *epochs*. En el siguiente experimento igualamos este número al del primer modelo para intentar disminuir el error. También queremos desgranar este resultado en las distintas clases que compone el clasificador, por lo que visualizamos la matriz de confusión para este caso y mostradas en Tabla 5-3 y Tabla 5-4.

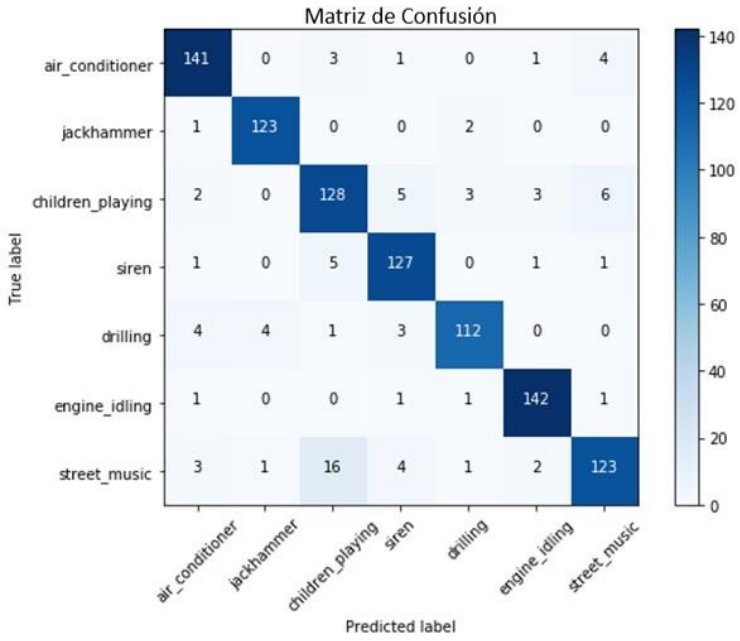

Tabla 5-3. Matriz de confusión del segundo modelo

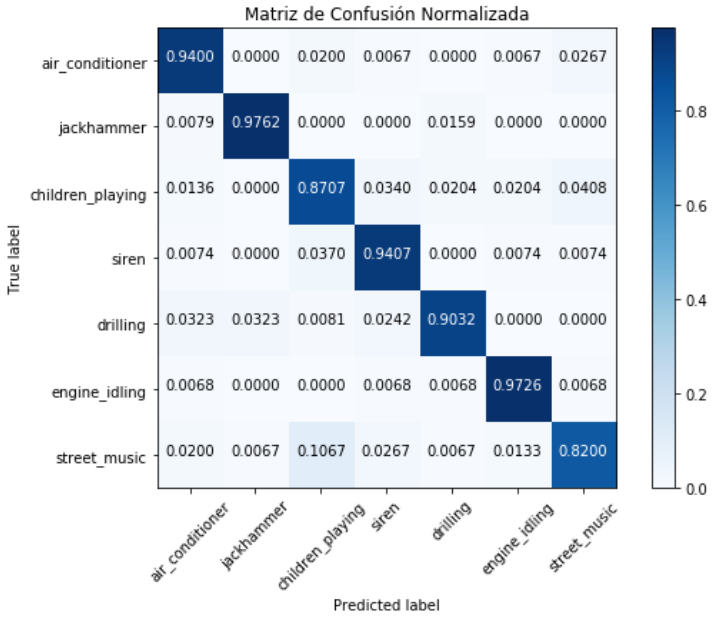

Tabla 5-4. Matriz de confusión normalizada del segundo modelo

Vemos como la amplia mayoría de las muestras se encuentran correctamente clasificadas. Podemos resaltar en este modelo las 16 muestras de la clase *street_music* incorrectamente clasificadas en la clase *children_playing*. Ambos sonidos están grabados en la calle, probablemente con un sistema de grabación similar como puede ser un móvil. Incluso suponemos que en los extractos de música callejera también había personas hablando o niños jugando de fondo. Sin embargo, este error no supone más del 10% con relación a la clase *street_music*.

5.3. Validación cruzada

Los resultados mostrados en los apartados anteriores son muy esperanzadores. Podríamos decir incluso que nuestro clasificador tiene un buen rendimiento. Sin embargo, no hemos explorado todas las posibilidades para

concluir aquí nuestro trabajo. Es posible que nuestra división manual de los datos en entrenamiento, validación y test estuviese sesgada. Es decir, puede que los datos de test divididos manualmente fuesen bien clasificados, pero otros datos no. Es por eso por lo que decidimos implementar el método de la validación cruzada para asegurarnos que las arquitecturas de nuestros modelos funcionaban con todos los datos de nuestro *dataset*.

Además de aplicar validación cruzada decidimos mostrar al final del entrenamiento el error medio del modelo, la precisión media del modelo y la desviación estándar de la precisión. Los anteriores experimentos nos sirvieron para determinar las *epochs* necesarias en el entrenamiento, el *learning rate* del optimizador y el tamaño de *batch*. En este experimento, aprovechando la implementación de la validación cruzada, buscamos alcanzar un resultado de cada modelo lo más completo posible para poder compararlos entre sí.

5.3.1 Resultados del primer modelo

Para nuestro modelo optamos por aplicar una validación cruzada con 5 divisiones. Buscamos información en la documentación de *scikit-learn* para aplicarlo. Tuvimos que importar la librería `StratifiedKFold` para ello. Ahora no existía división manual, todos los espectrogramas quedaban almacenados en una única variable. Normalizamos esta variable y comenzamos el entrenamiento.

Este entrenamiento se repitió en un bucle durante cinco veces. Eliminamos la función de *callback* `EarlyStopping` ya que queríamos que entrenase a lo largo de las 100 *epochs*. Mantuvimos la función de *callback* `ModelCheckpoint` ya que evaluamos con los datos de test el mejor modelo. Finalmente guardamos en variables los resultados de precisión y error de los cinco modelos y los representamos.

Ejecutar este código nos llevó un total de más de siete horas y media, ya que para cada división se creaba una nueva arquitectura y se iteraba durante 100 *epochs*. Tras este tiempo se visualizaron los resultados para cada subdivisión y se calculó la media de la precisión y el error de los cinco modelos.

```
------------------------------------------------------------------------
Resultados por fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.234340780807866 - Accuracy: 93.026819882265%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.20399938450690888 - Accuracy: 94.02298846464048%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.2337646386004499 - Accuracy: 92.72030647230332%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.2121072199038619 - Accuracy: 92.72030647230332%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.20319618737222228 - Accuracy: 92.7147238898131%
------------------------------------------------------------------------
Resultado medio:
> Accuracy: 93.04102903626504 (+- 0.5053013345919821)
> Loss: 0.2174816422382618
------------------------------------------------------------------------
```

Figura 5-5. Resultado de la validación cruzada para el primer modelo

Vemos en la Figura 5-5 como la precisión del modelo asciende a un **93.04%** con una desviación del 0.51%. El error medio es de tan solo el **0.21**. Podemos deducir tras aplicar validación cruzada que nuestros resultados no están sesgados por la división de los datos. La arquitectura del primer modelo clasifica las muestras con un sorprendente resultado de acierto.

5.3.2 Resultados del segundo modelo

Para entrenar el segundo modelo implementando validación cruzada eliminamos las funciones *callbacks* `EarlyStopping` y `ModelCheckpoint`. Nuestro objetivo consiste en iterar cada modelo 100 veces a lo largo de las cinco divisiones del algoritmo *k-fold*. Al ser una arquitectura más compleja el entrenamiento total llevó más de 48 horas en completarse, otro ejemplo de los largos tiempos de ejecución que debemos tener en cuenta

en todo nuestro proyecto. Los resultados de la validación cruzada para el segundo modelo fueron los mostrados en la Figura 5-6.

```
------------------------------------------------------------------------
Resultados por fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.2502050715159639 - Accuracy: 93.63984669305356%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.2635766193327776 - Accuracy: 94.0229884874775%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.23942366817901875 - Accuracy: 94.7126436598913%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.2651268898984025 - Accuracy: 93.56321838166978%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.2916025318251066 - Accuracy: 92.33128834355828%
------------------------------------------------------------------------
Resultado medio:
> Accuracy: 93.65399711313009 (+- 0.7765161488196198)
> Loss: 0.26198695615025386
------------------------------------------------------------------------
```

Figura 5-6. Resultado de la validación cruzada para el segundo modelo

Apreciamos que la precisión media del modelo clasificando de manera correcta es del **93.65%**, con una desviación en el resultado del 0.78%. La pérdida de nuestra arquitectura alcanza un valor del **0.26**, similar al proporcionado en los apartados anteriores, cuando la división de los datos la hicimos de manera manual y no se aplicaba validación cruzada. Podemos concluir que los resultados del rendimiento de esta arquitectura no se encuentran sesgados por la elección de los datos.

Para terminar este capítulo de experimentación con ambos modelos, vamos a comparar sus resultados. El segundo modelo, con una capa de convolución adicional, estaba diseñado para un problema de clasificación de sonidos con una base de datos más extensa que la que le hemos proporcionado, tanto en número de muestras como en número de categorías.

En nuestro problema de clasificación hemos empleado 6525 muestras para 7 categorías y el problema para el que estaba diseñado el segundo modelo contaba con una base de datos de más de 9500 muestras para 41 categorías. La diferencia de tamaño entre ambos problemas es notable. Sin embargo, el primer modelo empleado con tres capas convolucionales estaba diseñado para un problema similar al nuestro, ya que usaba la base de datos *UrbanSound8K*. Aquí la diferencia de tamaño no es tan considerable.

Es cierto que la segunda arquitectura de red neuronal clasifica las muestras de nuestro problema de manera excepcional y la adición de una capa de convolución dota al sistema de un grado de abstracción mayor. Sin embargo, podemos decir que para nuestro problema la segunda red neuronal está sobredimensionada, no se necesita un modelo tan complejo para la clasificación en siete categorías.

En nuestra opinión el primer modelo ejecutado está perfectamente dimensionado al problema. El tiempo de entrenamiento de este ha sido considerablemente menor que el tiempo de entrenamiento del segundo y, a pesar de tener una capa de convolución menos, los resultados de rendimiento son prácticamente idénticos al segundo. En un supuesto caso real de implementación de un sistema de clasificación de sonidos con similar número de categorías a clasificar optaríamos por adoptar la primera red neuronal, y a medida que el problema se fuese haciendo más complejo optar entonces por aplicar la segunda red neuronal.

6 CONCLUSIONES Y LÍNEAS FUTURAS

A lo largo de este trabajo hemos aprendido a programar una red neuronal desde cero. Gracias al punto de partida proporcionado por la competición del DCASE sobre etiquetado de audio hemos desarrollado e implementado dos arquitecturas para la clasificación de sonidos ambientales, con una base de datos reducida debido a las limitaciones de hardware de nuestro equipo. A continuación, exponemos en este capítulo final las conclusiones de nuestro trabajo y las posibles líneas futuras de investigación.

6.1. Conclusiones

Cuando realizamos la investigación previa para redactar el capítulo del estado del arte nos dimos cuenta de lo extenso que es el campo de la Inteligencia Artificial y del *Machine Learning*. Existen diversas maneras de afrontar un mismo problema, con algoritmos tanto de aprendizaje supervisado como no supervisado. Estos últimos algoritmos todavía se encuentran en proceso de estudio, pero todo parece indicar que serán la clave para una nueva evolución en el campo del aprendizaje máquina.

Nos vimos sorprendidos por la cantidad de problemas que a priori un humano puede resolver sin mayor misterio y un ordenador necesita de algoritmos como los vistos en este trabajo. En el campo del audio, por ejemplo, la distinción de fuentes de sonidos distintas, cuál es el coche y cuál es el perro, cuánto tiempo duran cada una, etcétera.

Adentrándonos en el mundo de las redes neuronales nos encontramos con que estas no son nada novedoso de hoy en día. El estudio de estos modelos se remonta hasta los años 50, cuando se inventó el perceptrón. Desde aquellos años hasta el día de hoy la investigación en las redes neuronales ha sufrido idas y venidas. Sobre 1970 se abandonó la investigación en redes neuronales por el tiempo que se tardaba en el entrenamiento. Diez años después, con el desarrollo de algoritmos más rápidos como *Backpropagation* se continuó investigando el poder de las redes neuronales. Se volvió a dejar de lado por la enorme carga computacional que alcanzaban estas redes hasta que finalmente, con el comienzo del siglo XXI y la introducción de las GPUs, surgieron multitud de modelos que hemos revisado en este trabajo.

Como hemos visto existe una gran variedad de arquitecturas, dependiendo del tipo de capas que queramos usar, el número de neuronas de cada capa, la profundidad de la red, la función de activación, etcétera. Además, una vez tengamos nuestra arquitectura diseñada podemos elegir entre diversos parámetros como el número de iteraciones, el tamaño del *batch*, el optimizador del modelo, la función de coste, etcétera. Cada una de estas elecciones modifica profundamente el resultado de nuestra red. No es nada fácil diseñar un modelo, se necesita un nivel de intuición muy elevado para ello.

Otro punto importante es la elección de una buena base de datos. A menudo cuando se trata con modelos de *Deep Learning* el punto más importante es tanto la elección de una buena base de datos como el preprocesamiento de los datos antes de ser tratados por la red. La base de datos tiene que ser lo suficientemente representativa de la realidad, contando con un gran número de muestras igualmente distribuida entre todas las clases. El preprocesamiento incluye tareas como la elección del tamaño de las muestras, métodos de *Data Augmentation*, normalización, etcétera. Con unos buenos datos podemos llegar a alcanzar buenos resultados.

Un aspecto fundamental a la hora de ejecutar estos modelos es el tiempo. Trabajar en un equipo con una gran capacidad en cuanto a hardware es casi una obligación. Entrenar un modelo con mayor rapidez y ver los resultados al instante para así poder avanzar en tu investigación sin que pasen horas muertas esperando. No solo a la hora de entrenar, también para poder trabajar con modelos mucho más complejos o con datos de entrada en más de una dimensión, como imágenes en los tres planos de color RGB.

Finalmente, nos sentimos muy satisfechos con el trabajo expuesto en esta memoria y los resultados obtenidos, a pesar de no contar con una estación de trabajo potente y de no tener muchos conocimientos tanto en *Python* como en *Deep Learning*. Investigando y a base de prueba y error hemos desarrollado un modelo completamente

funcional para la clasificación de sonidos ambientales con un rendimiento sorprendente. No solo hemos cumplido el objetivo principal de este trabajo, también hemos adquirido conocimientos en el lenguaje de programación *Python* y en el campo de las redes neuronales. Este trabajo es una aproximación a un campo de conocimiento que está ligado desde su concepción al campo del análisis de señales y, por tanto, al campo de las telecomunicaciones.

6.2. Líneas futuras

La primera línea de investigación sería probar ambos modelos propuestos para todas las clases que contiene el *dataset UrbanSound8K*. A la hora de incrementar el número de muestras se podría aplicar *Data Augmentation* en aquellas clases que posean menor número de datos. Además, se pueden extraer más muestras de diversas bases de datos para rellenar las clases. Por ejemplo, de la base de datos proporcionada por *Google* llamada *AudioSet*. Para hacer estas tareas se necesitaría de un equipo con mayor potencia que desafortunadamente nosotros no tuvimos.

La segunda línea que se nos ocurre es transformar nuestro problema de clasificación de audio en un problema de etiquetado de sonidos o de detección de sonidos. Pudimos observar en la matriz de confusión del segundo modelo que 16 muestras de la clase *street_music* fueron clasificadas como si fueran de la clase *children_playing*. Estaría interesante ver el resultado en un modelo que pudiese etiquetar un sonido con dos o más clases a la vez. También se podría evolucionar a un modelo que tras ser entrenado recibiese audio en tiempo real y detectase qué clases está escuchando en ese momento. Para ello se necesitaría de otros modelos más complejos y profundos.

Por último, nuestro trabajo se ha centrado en desarrollar dos modelos de red neuronal convolucional, un tipo de algoritmo del aprendizaje supervisado. La última línea de investigación que proponemos consiste en desarrollar el mismo problema descrito en este trabajo con otros modelos de aprendizaje supervisado y comparar los resultados con los vistos en esta memoria. Sería también muy interesante afrontar este problema desde el punto de vista del aprendizaje no supervisado, qué algoritmos se utilizarían, qué tipo de abstracciones sería capaz de extraer el modelo, etcétera.

REFERENCIAS

[1] S. H. Bae, I. Choi, and N. S. Kim, "Acoustic Scene Classification Using Parallel Combination of LSTM and CNN," *Proc. Detect. Classif. Acoust. Scenes Events 2016 Work.*, no. September, pp. 11–15, 2016.

[2] Y. Petetin, C. Laroche, and A. Mayoue, "Deep neural networks for audio scene recognition," in *2015 23rd European Signal Processing Conference, EUSIPCO 2015*, 2015, pp. 125–129, doi: 10.1109/EUSIPCO.2015.7362358.

[3] M. Valenti, A. Diment, G. Parascandolo, S. Squartini, and T. Virtanen, "DCASE 2016 acoustic scene classification using convolutional neural networks," 2016, doi: 10.1007/978-981-15-2475-2_28.

[4] H. Lim, M. J. Kim, and H. Kim, "Cross-acoustic transfer learning for sound event classification," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2016, vol. 2016-May, pp. 2504–2508, doi: 10.1109/ICASSP.2016.7472128.

[5] E. Çakır, T. Heittola, and T. Virtanen, "Domestic audio tagging with convolutional neural networks," 2016.

[6] K. J. Piczak, "Environmental sound classification with convolutional neural networks," in *IEEE International Workshop on Machine Learning for Signal Processing, MLSP*, 2015, vol. 2015-Novem, doi: 10.1109/MLSP.2015.7324337.

[7] Y. Xu, Q. Huang, W. Wang, P. J. B. Jackson, and M. D. Plumbley, "Fully DNN-based multi-label regression for audio tagging," no. September, 2016.

[8] S. Adavanne, G. Parascandolo, P. Pertilä, T. Heittola, and T. Virtanen, "Sound Event Detection in Multichannel Audio Using Spatial and Harmonic Features," Jun. 2017, [Online]. Available: http://arxiv.org/abs/1706.02293.

[9] E. Cakir, T. Heittola, H. Huttunen, and T. Virtanen, "Polyphonic sound event detection using multi label deep neural networks," in *Proceedings of the International Joint Conference on Neural Networks*, 2015, vol. 2015-Septe, doi: 10.1109/IJCNN.2015.7280624.

[10] M. Espi, M. Fujimoto, K. Kinoshita, and T. Nakatani, "Exploiting spectro-temporal locality in deep learning based acoustic event detection," *Eurasip J. Audio, Speech, Music Process.*, vol. 2015, no. 1, Dec. 2015, doi: 10.1186/s13636-015-0069-2.

[11] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012, doi: 10.1109/MSP.2012.2205597.

[12] H. Lee, Y. Largman, P. Pham, and A. Y. Ng, "Unsupervised feature learning for audio classification using convolutional deep belief networks," 2009, doi: 10.1109/SCET.2012.6342000.

[13] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," 2013.

[14] E. J. Humphrey and J. P. Bello, "Rethinking automatic chord recognition with convolutional neural networks," in *Proceedings - 2012 11th International Conference on Machine Learning and Applications, ICMLA 2012*, 2012, vol. 2, pp. 357–362, doi: 10.1109/ICMLA.2012.220.

[15] J. Schlüter and T. Grill, "Exploring data augmentation for improved singing voice detection with neural networks," in *Proceedings of the 16th International Society for Music Information Retrieval Conference,*

ISMIR 2015, 2015, pp. 121–126.

[16] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Audio chord recognition with recurrent neural networks," in *Proceedings of the 14th International Society for Music Information Retrieval Conference, ISMIR 2013*, 2013, pp. 335–340.

[17] N. Boulanger-Lewandowski, J. Droppo, M. Seltzer, and D. Yu, "Phone sequence modeling with recurrent neural networks," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2014, pp. 5417–5421, doi: 10.1109/ICASSP.2014.6854638.

[18] S. Böck and M. Schedl, "Enhanced beat tracking with context-aware neural networks," pp. 1–5, 2011, [Online]. Available: http://www.music-ir.org/mirex/wiki/2006:Audio_Beat_Tracking.

[19] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *31st International Conference on Machine Learning, ICML 2014*, 2014, vol. 5, pp. 3771–3779.

[20] G. Mesnil, X. He, L. Deng, and Y. Bengio, "Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2013, pp. 3771–3775.

[21] G. Parascandolo, H. Huttunen, and T. Virtanen, "Recurrent neural networks for polyphonic sound event detection in real life recordings," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, Apr. 2016, vol. 2016-May, pp. 6440–6444, doi: 10.1109/ICASSP.2016.7472917.

[22] S. Dieleman and B. Schrauwen, "End-to-end learning for music audio," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2014, pp. 6964–6968, doi: 10.1109/ICASSP.2014.6854950.

[23] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in English and Mandarin," in *33rd International Conference on Machine Learning, ICML 2016*, Dec. 2016, vol. 1, pp. 312–321, [Online]. Available: http://arxiv.org/abs/1512.02595.

[24] Z. Zuo *et al.*, "Convolutional recurrent neural networks: Learning spatial dependencies for image representation," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2015, vol. 2015-Octob, pp. 18–26, doi: 10.1109/CVPRW.2015.7301268.

[25] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Conference Proceedings - EMNLP 2015: Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1422–1432, doi: 10.18653/v1/d15-1167.

[26] S. Sigtia, E. Benetos, and S. DIxon, "An end-to-end neural network for polyphonic piano music transcription," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 24, no. 5, pp. 927–939, Aug. 2016, doi: 10.1109/TASLP.2016.2533858.

[27] J.-J. Aucouturier, B. Defreville, and F. Pachet, "The bag-of-frames approach to audio pattern recognition: A sufficient model for urban soundscapes but not for polyphonic music," *J. Acoust. Soc. Am.*, vol. 122, no. 2, pp. 881–891, 2007, doi: 10.1121/1.2750160.

[28] M. Lagrange, G. Lafay, B. Défréville, and J.-J. Aucouturier, "The bag-of-frames approach: A not so sufficient model for urban soundscapes," *J. Acoust. Soc. Am.*, vol. 138, no. 5, pp. EL487–EL492, Dec. 2015, doi: 10.1121/1.4935350.

[29] A. J. Eronen *et al.*, "Audio-based context recognition," in *IEEE Transactions on Audio, Speech and Language Processing*, Jan. 2006, vol. 14, no. 1, pp. 321–329, doi: 10.1109/TSA.2005.854103.

[30] J. T. Geiger, B. Schuller, and G. Rigoll, "Large-scale audio feature extraction and SVM for acoustic

scene classification," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, 2013, doi: 10.1109/WASPAA.2013.6701857.

[31] H. Phan, L. Hertel, M. Maass, P. Koch, and A. Mertins, "Label tree embeddings for acoustic scene classification," in *MM 2016 - Proceedings of the 2016 ACM Multimedia Conference*, Oct. 2016, pp. 486–490, doi: 10.1145/2964284.2967268.

[32] G. Roma, W. Nogueira, and P. Herrera, "Recurrence quantification analysis features for environmental sound recognition," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, 2013, doi: 10.1109/WASPAA.2013.6701890.

[33] V. Bisot, S. Essid, and G. Richard, "HOG and subband power distribution image features for acoustic scene classification," in *2015 23rd European Signal Processing Conference, EUSIPCO 2015*, 2015, pp. 719–723, doi: 10.1109/EUSIPCO.2015.7362477.

[34] A. Rakotomamonjy and G. Gasso, "Histogram of gradients of time-frequency representations for audio scene classification," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 23, no. 1, pp. 142–153, Jan. 2015, doi: 10.1109/TASLP.2014.2375575.

[35] B. Cauchi, M. Lagrange, N. Misdariis, and A. Cont, "Saliency-based modeling of acoustic scenes using sparse non-negative matrix factorization," in *International Workshop on Image Analysis for Multimedia Interactive Services*, 2013, doi: 10.1109/WIAMIS.2013.6616131.

[36] V. Bisot, R. Serizel, S. Essid, and G. Richard, "Acoustic scene classification with matrix factorization for unsupervised feature learning," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2016, vol. 2016-May, pp. 6445–6449, doi: 10.1109/ICASSP.2016.7472918.

[37] E. Benetos, M. Lagrange, and S. Dixon, "Characterisation of acoustic scenes using a temporally-constrained shift-invariant model," in *15th International Conference on Digital Audio Effects, DAFx 2012 Proceedings*, 2012, [Online]. Available: https://hal.archives-ouvertes.fr/hal-01126770.

[38] K. Lee, Z. Hyung, and J. Nam, "Acoustic scene classification using sparse feature learning and event-based pooling," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, 2013, doi: 10.1109/WASPAA.2013.6701893.

[39] J. Salamon and J. P. Bello, "Feature learning with deep scattering for urban sound analysis," in *2015 23rd European Signal Processing Conference, EUSIPCO 2015*, 2015, pp. 724–728, doi: 10.1109/EUSIPCO.2015.7362478.

[40] J. Salamon and J. P. Bello, "Unsupervised feature learning for urban sound classification," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2015, vol. 2015-Augus, pp. 171–175, doi: 10.1109/ICASSP.2015.7177954.

[41] J. Salamon, C. Jacoby, and J. P. Bello, "A dataset and taxonomy for urban sound research," in *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*, Nov. 2014, pp. 1041–1044, doi: 10.1145/2647868.2655045.

[42] J. Salamon and J. P. Bello, "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification," *IEEE Signal Process. Lett.*, vol. 24, no. 3, pp. 279–283, Mar. 2017, doi: 10.1109/LSP.2017.2657381.

[43] X. Huang, A. Acero, and H.-W. Hon, *Spoken language processing: A guide to theory, algorithm, and system development*. 2001.

[44] S. S. Stevens, J. Volkmann, and E. B. Newman, "A Scale for the Measurement of the Psychological Magnitude Pitch," *J. Acoust. Soc. Am.*, vol. 8, no. 3, pp. 185–190, Jan. 1937, doi: 10.1121/1.1915893.

[45] E. Zwicker and I. E. Terhardt, “Analytical expressions for critical-band rate and critical bandwidth as a function of frequency,” *Journal of the Acoustical Society of America*, vol. 68, no. 5. Acoustical Society of America, pp. 1523–1525, Nov. 26, 1980, doi: 10.1121/1.385079.

[46] J. C. Brown, “Calculation of a constant Q spectral transform,” *Journal of the Acoustical Society of America*, vol. 89, no. 1. pp. 425–434, 1991.

[47] R. D. Patterson, K. Robinson, J. Holdsworth, D. McKeown, C. Zhang, and M. Allerhand, “Complex Sounds and Auditory Images,” in *Auditory Physiology and Perception*, Pergamon, 1992, pp. 429–446.

[48] V. Peltonen, J. Tuomi, A. Klapuri, J. Huopaniemi, and T. Sorsa, “Computational auditory scene recognition,” *ICASSP, IEEE Int. Conf. Acoust. Speech Signal Process. - Proc.*, vol. 2, pp. 1941–1944, 2002, doi: 10.1109/ICASSP.2002.5745009.

[49] X. Valero and F. Alías, “Análisis de la señal acústica mediante coeficientes cepstrales bio-inspirados y su aplicación al reconocimiento de paisajes sonoros,” *VIII Congr. Ibero-americano Acústica*, pp. 1–9, 2012, [Online]. Available: http://www.sea-acustica.es/fileadmin/Evora12/227.pdf.

[50] H. Phan, L. Hertel, M. Maass, P. Koch, and A. Mertins, “CaR-FOREST: Joint Classification-Regression Decision Forests for Overlapping Audio Event Detection,” Jul. 2016, [Online]. Available: http://arxiv.org/abs/1607.02306.

[51] X. Valero and F. Alias, “Gammatone cepstral coefficients: Biologically inspired features for non-speech audio classification,” *IEEE Trans. Multimed.*, vol. 14, no. 6, pp. 1684–1689, 2012, doi: 10.1109/TMM.2012.2199972.

[52] S. J. Russell, *Artificial intelligence : a modern approach* , 3rd ed., i. Upper Saddle River (New Jersey): Pearson, 2014.

[53] E. Alpaydin, *Introduction to machine learning* , 3rd ed. Cambridge, Mass. ; The MIT Press, 2014.

[54] B. Müller, *Neural networks : An introduction* , 2nd update. Berlin [etc: Springer-Verlag, 1995.

[55] J. Bapu Ahire, “The XOR Problem in Neural Networks. – Jayesh Bapu Ahire – Medium,” 2017. https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b (accessed Apr. 21, 2020).

[56] I. Goodfellow, *Deep learning* . Cambridge, Massachusetts ; MIT Press, 2016.

[57] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986, doi: https://doi.org/10.1038/323533a0.

[58] N. Cui, “Applying Gradient Descent in Convolutional Neural Networks,” vol. 1004, no. 1, 2018, doi: 10.1088/1742-6596/1004/1/012027.

[59] L. Taylor and G. Nitschke, “Improving Deep Learning using Generic Data Augmentation,” 2017, [Online]. Available: http://arxiv.org/abs/1708.06020.

[60] J. Koushik, “Understanding Convolutional Neural Networks,” 2016, [Online]. Available: http://arxiv.org/abs/1605.09081.

[61] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfittin,” *J. Mach. Learn. Res.*, no. 15, pp. 1929–1958, 2014, doi: 10.1016/0370-2693(93)90272-J.

[62] Zafar, “Beginner’s Guide to Audio Data,” 2018. https://www.kaggle.com/fizzbuzz/beginner-s-guide-to-audio-data/.

[63] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal

covariate shift," *32nd Int. Conf. Mach. Learn. ICML 2015*, vol. 1, pp. 448–456, 2015.

[64] S. Narkhede, "Understanding Confusion Matrix," 2018. https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

[65] Krishni, "K-Fold Cross Validation," 2018. https://medium.com/datadriveninvestor/k-fold-cross-validation-6b8518070833.

GLOSARIO

ANN	Artificial Neural Network
ASR	Automatic Speech Recognition
AWS	Amazon Web Services
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CQT	Constant-Q Transform
CSV	Comma-Separated Values
DCASE	Detection and Classification of Acoustic Scenes and Events
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
ESC	Enviromental Sound Classification
FFT	Fast Fourier Transform
GFCC	Gammatone Frequency Cepstral Coefficients
GMM	Gaussian Mixture Model
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HMM	Hidden Markov Model
HOG	Histogram of Oriented Gradients
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
MFCC	Mel Frequency Cepstral Coefficients
MIR	Music Information Retrieval
MIREX	Music Information Retrieval Evaluation Exchange
NMF	Non-negative Matrix Factorization
PCA	Principal Component Analysis
RAM	Random Access Memory
RBM	Restricted Boltzmann Machine
RNN	Recurrent Neural Network
SKM	Spherical K-Means
SSD	Solid-State Drive
STFT	Short-Time Fourier Transform
SVM	Support Vector Machine
TPU	Tensor Processing Unit
WAV	Waveform Audio Format