PROYECTO FIN DE GRADO

TÍTULO: Sistema de conversión Audio-MIDI basado en redes neuronales

AUTOR/A: Nicolás Uriz Roldán

TITULACIÓN: Grado en Ingeniería de Sonido e Imagen

TUTOR/A: Lino Pedro García Morales

DEPARTAMENTO: Ingeniería Audiovisual y Comunicaciones

VºBº TUTOR

Miembros del Tribunal Calificador:

PRESIDENTE/A: María Pilar Velasco Cebrián

TUTOR/A: Lino Pedro García Morales

SECRETARIO/A: Jorge Grundman Isla

Fecha de lectura:

Calificación:

El Secretario,

Resumen

La inteligencia artificial y en especial el aprendizaje automático se presentan como una revolución tecnológica, que en los últimos tiempos ha cambiado por completo el paradigma de la computación. Gracias a su enfoque se han logrado resolver problemas que antes resultaban impensables y mejorar el rendimiento y precisión de muchas tecnologías ya existentes. El propósito de este proyecto es aplicar el aprendizaje automático para proponer y probar la viabilidad de un sistema de conversión de audio a MIDI, especialmente pensado para guitarra y bajo eléctricos.

Para llevar a cabo el proyecto, se han estudiado las distintas metodologías seguidas en el campo del aprendizaje automático aplicado a audio y se ha elegido la óptima para el problema a resolver.

Se ha planteado un sistema basado en redes neuronales enfocadas a la clasificación, cuya información de entrada es una señal de audio enventanada que contiene el transitorio inicial de la nota. De esta forma, se propone una alternativa para mejorar la latencia impuesta por otros algoritmos de detección de tono, presentes en las aplicaciones ya existentes de conversión de audio a MIDI en tiempo real para guitarra y bajo.

En este proyecto se propone una arquitectura de redes neuronales basada en capas convolucionales unidimensionales y capas densas. La arquitectura se ha codificado y entrenado en Python empleando un conjunto de datos propio. Durante el entrenamiento se ha evaluado la precisión de la red y como de sólido ha sido su aprendizaje.

Se han propuesto varios casos de estudio con los que analizar el comportamiento de las redes neuronales para distintos números de notas, tamaños del conjunto de datos con los que se entrenan y para los dos instrumentos contemplados.

A partir de los resultados obtenidos, se puede afirmar que las bases del sistema propuesto cumplen de forma sobresaliente con su cometido, aunque el factor de la latencia se debe estudiar en futuras investigaciones.

Abstract

Artificial intelligence and especially machine learning are presented as a technological revolution, which in recent times has completely changed the paradigm of computing. Thanks to its basis, previously unthinkable problems have been solved and the performance and accuracy of many existing technologies have been improved. The purpose of this project is to apply autonomous learning to propose and test the feasibility of an audio-to-MIDI conversion system, especially designed for electric guitar and bass.

To carry out the project, the different methodologies followed in the field of machine learning applied to audio have been studied and the optimal one for the problem to be solved has been chosen.

A system based on neural networks focused on classification has been proposed, whose input information is an enveloped audio signal containing the initial transient of the note. In this way, an alternative is proposed to improve the latency imposed by other pitch detection algorithms present in already existing real-time audio-to-MIDI conversion applications for guitar and bass.

In this project, a neural network architecture based on one-dimensional convolutional layers and dense layers is proposed. The architecture has been coded and trained in Python using a proprietary dataset. During training, the accuracy and learning capability of the network has been evaluated.

Several case studies have been proposed to analyze the behavior of the neural networks for different numbers of notes, sizes of the dataset with which they are trained and for the two instruments considered.

From the results obtained, it can be affirmed that the basis of the proposed system performs outstandingly well, although the latency factor should be studied in future research.

Índice de figuras

Lista de acrónimos

ML (*Machine Learning*)

IA (*Inteligencia Artificial*)

ASR (*Automatic Speech Recognition*)

MIDI (*Musical Instrument Digital Interface*)

FFT (*Fast Fourier Transform*)

IFFT (*Inverse Fast Fourier Transform*)

VST (*Virtual Studio Technology*)

DAW (*Digital Audio Workstation*)

ANN (*Artificial Neural Networks*)

CNN (*Convolutional Neural Network*)

MFCC (*Mel-frequency Cepstral Coefficients*)

ReLU (Rectified Linear Unit)

FNN (*Feedforward Neural Network*)

MLP (Multi-Layer Perceptron)

CEL (*Cross Entropy Loss*)

MSE (*Mean Squared Error*)

CSV (*Comma-separated values*)

Índice de contenidos

1. Introducción

1.1 Marco y motivación del proyecto

La conversión de audio a MIDI (*Musical Instrument Digital Interface*) [1] ha resultado siempre una tecnología de gran interés en la industria del audio. Esta técnica permite extraer la información musical de una señal de audio digital, diferenciando distintas notas musicales, su colocación temporal e intensidad, que una vez detectadas se codifican por medio de mensajes MIDI. Esta tecnología brinda a los músicos un gran abanico de posibilidades creativas y funcionales. Su aplicación en guitarra y bajo eléctricos otorga utilidades muy interesantes al usuario, entre ellas la transcripción automática para generar partituras y tablaturas; o su uso en tiempo real para controlar sintetizadores e instrumentos virtuales por medio de un instrumento tradicional.

A la hora de abordar el problema, existen distintos métodos que ya han sido ampliamente explorados por empresas que han desarrollado productos para cubrir la demanda de los consumidores, además de la comunidad investigadora. El método convencional más popular para solventar este problema se basa en la utilización de algoritmos de detección de tono (*pitch detection*), que se basan en la estimación de la frecuencia fundamental de cada nota (F0). Dentro de los algoritmos de detección de tono se diferencian dos enfoques: basados en el dominio de la frecuencia y basados en espectro/tiempo. El libro "Signal Processing Methods for Music Transcription" [2] proporciona una visión detallada de varios algoritmos tradicionales y técnicas más novedosas.

Cuando se aplican estas tecnologías en tiempo real la latencia que produzca el algoritmo es un factor clave, ya que esto afecta de manera dramática la experiencia del músico. Como se menciona en el paper "The Effects of Latency on Live Sound Monitoring" [3], la latencia en la monitorización de los músicos de una grabación empieza a ser perceptible entre los 5 y 10 milisegundos, y sus efectos empiezan a afectar a la interpretación del músico con tan solo 7 ms. Por lo general, los algoritmos de detección de tono acarrean una latencia equivalente a la duración de dos periodos de la frecuencia fundamental. Esto se debe al propio procesado que llevan a cabo los propios algoritmos: autocorrelación, Transformada rápida de Fourier, Coeficientes Cepstrales en las Frecuencias de Mel [4], etc. Esto implica que el retardo entre el inicio de una nota y su detección empeora cuanto menor sea la frecuencia de la nota tocada. A continuación, se muestra una tabla donde se comparan los retardos equivalentes para las notas más graves de la guitarra y bajo eléctricos.

Tabla 1. Comparativa de retardos para las notas más graves, para guitarra y bajo.

Instrumento	Nota más grave	F_0 [Hz]	Duración de dos periodos [ms]
Guitarra	E2	82.4	24.3
Bajo	E1	41.2	48.6

Una de las soluciones comerciales más populares entre los guitarristas es el software *MIDI Guitar 2* [5]. Este programa, propiedad de *Jam Origin* se basa en algoritmos de detección de tono, pero al haber sido desarrollado por una empresa, su procesado interno no es de dominio público. El programa permite, tanto en versión VST como *Stand Alone*, la conversión en tiempo real de audio a MIDI, especialmente pensado para guitarra, pero con la posibilidad de usarse con bajo eléctrico. La aplicación permite enrutar el tránsito MIDI de salida dentro del programa de grabación empleado (DAW), o aplicarlo directamente a un instrumento virtual dentro de la aplicación. Esta solución presenta una buena precisión, que se ve afectada a la hora de reconocer acordes complejos. Además, la latencia introducida se hace notable en las notas más graves de la guitarra, caso que empeora al utilizarse con un bajo.

Otro nombre de gran éxito en la industria es el *Axon AX100* [6], un equipo *hardware* desarrollado por la marca Terratec, que emplea redes neuronales para la detección de tono. Este equipo fue lanzado en la década de los 90 y ofrece una calidad de conversión que, hoy en día, sigue contentando a muchos usuarios. Las dos grandes ventajas que ofrece el *Axon AX100* son: una muy baja latencia gracias al uso de redes neuronales y la posibilidad de reconocimiento polifónico de alta precisión. La posibilidad de conversión polifónica viene dada por la utilización de una pastilla hexafónica, que entrega a la entrada del sistema una línea de audio independiente para cada cuerda. Los inconvenientes del equipo son principalmente su elevado precio y la inconveniencia de tener que instalar una pastilla adicional al instrumento.

1.2 Objetivos técnicos y académicos

Los objetivos de este proyecto fin de carrera son, desde el punto de vista técnico:

- Estudiar la posibilidad de reconocer notas musicales de un bajo eléctrico o una guitarra eléctrica mediante el análisis de sus transitorios iniciales con redes neuronales.
- Diseño y configuración de una red neuronal.
- Grabación y creación de un dataset propio.
- Creación de una red neuronal para clasificación de audio.
- Análisis de las características de los transitorios iniciales de nota de bajo y guitarra.

Desde el punto de vista académico, el proyectista adquiere las siguientes competencias y habilidades:

- Conocimientos teóricos y prácticos sobre el campo del aprendizaje automático aplicado a audio.
- Manejo del entorno de trabajo Pytorch, sobre el lenguaje Python.
- Procesado de audio digital por medio de código.

1.3 Estructura del resto de la memoria

En el primer apartado de la memoria se hace un repaso de los avances más relevantes relacionados con la conversión de audio a MIDI aplicada a guitarra y bajo eléctricos, especialmente en su versión en tiempo real, se plantean los objetivos del proyecto y se detalla la estructura del documento.

En el segundo apartado se trata el contexto teórico, describiendo los contenidos relativos al audio y a las redes neuronales empleados en este proyecto. En él se justifican gran parte de las decisiones tomadas durante el diseño de la solución propuesta.

En el tercer apartado se exponen las especificaciones y restricciones del diseño. Para ello se limita el problema a resolver para hacerlo abordable en este proyecto.

En el cuarto apartado se describe el proceso seguido para generar el dataset, diseñar y codificar la red neuronal y finalmente, entrenarla. En este apartado se detallan los tres casos de estudio planteados.

En el quinto apartado se muestran e interpretan las métricas resultantes del entrenamiento de las redes para los distintos casos.

En el sexto apartado se presenta un presupuesto aproximado del coste íntegro del proyecto, contabilizando el material empleado y las horas de trabajo como labor de ingeniero.

En el séptimo apartado se comenta el impacto del proyecto a nivel social y tecnológico, al igual que la relación con los Objetivos del Desarrollo Sostenible.

En el octavo apartado se exponen las conclusiones del proyecto, donde se explica si se han cumplido los objetivos marcados y se proponen futuras líneas de investigación.

En el noveno apartado se expone la bibliografía usada, y en los anexos se expone el código más relevante y el manual de usuario.

2. Marco teórico

A continuación, se detalla el contexto teórico que cimienta este proyecto. Se han dividido los distintos temas en subapartados para una mejor comprensión.

2.1 Transitorios en audio

En audio, un transitorio es un evento de corta duración y alta energía que representa cambios bruscos en la señal. Estos cambios bruscos suelen ser causados por el inicio de una nota o por un golpe percusivo, dando lugar a una alta variabilidad en amplitud y frecuencia. Comúnmente, se suele nombrar al transitorio inicial de una nota musical como ataque u *onset* [7], y es una forma de calificar el timbre de un instrumento (mayor o menor pegada), o de modificarlo si se trata de un sintetizador. Una característica importante de los transitorios es la ausencia de una estructura armónica clara, que los asemeja a una señal de ruido. En la Figura 1 se muestra una nota tocada por una guitarra eléctrica, caracterizada por su amplitud en tiempo y espectrograma.

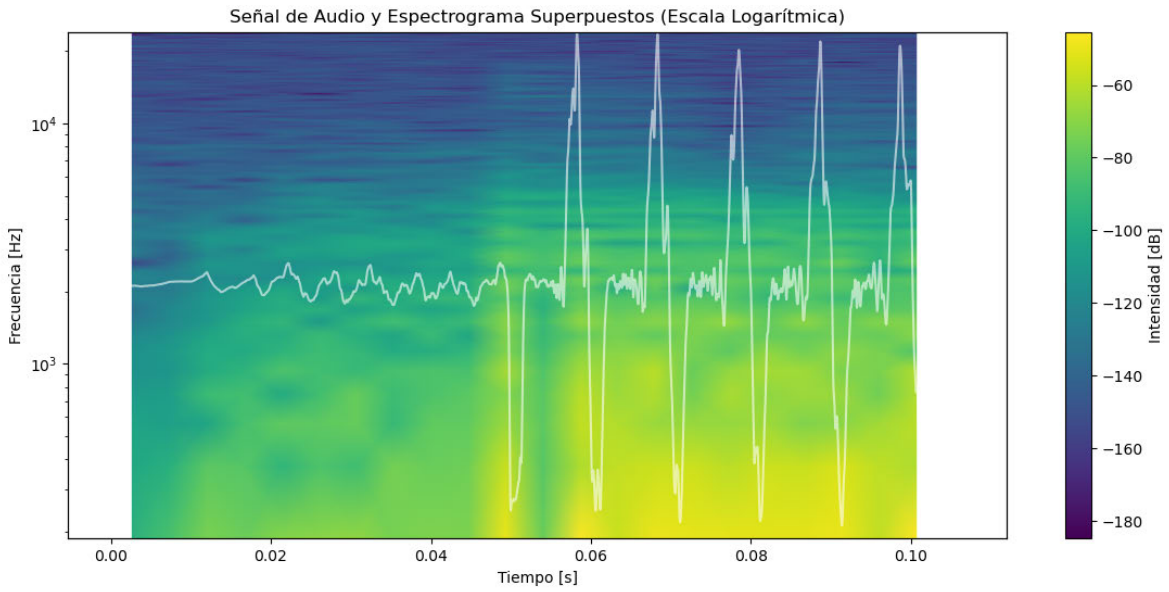

Figura 1. Gráfica que muestra los primeros milisegundos tras la percusión de una nota en una guitarra eléctrica, en amplitud y espectrograma.

La señal grabada ha sido recortada y normalizada en amplitud para esta representación. Se puede apreciar que durante los primeros cincuenta milisegundos la señal tiene poca amplitud y no contiene ninguna periodicidad, la variación de la señal en estos instantes se puede achacar al rozamiento de la púa con la cuerda. Aproximadamente en el milisegundo cincuenta se observa un máximo negativo de amplitud, que corresponde con la percusión de la cuerda. Se ha considerado como transitorio inicial de la nota la ventana temporal comprendida entre el primer mínimo y el primer máximo. Observando el espectrograma se aprecia una franja de tono verdoso tras el pico asociado a la percusión de la nota, de ello se puede extraer que hasta el siguiente máximo de amplitud la cuerda no está oscilando de manera armónica. Además,

durante el primer pico el espectrograma muestra un aumento de energía en alta frecuencia, esto está íntimamente relacionado con el carácter impulsivo del transitorio. Tras el segundo máximo (máximo positivo) se puede considerar que la oscilación de la cuerda se ha estabilizado y empieza la parte armónica de la señal.

Como consecuencia de la inestabilidad armónica de los transitorios con los que se está trabajando, el análisis en frecuencia de la señal para su clasificación no sería efectivo. La ausencia de patrones armónicos en la ventana temporal analizada dificulta la creación de un algoritmo capaz de detectar la nota a partir de la información contenida en dicha ventana.

2.2 Redes neuronales

La inteligencia artificial (Artificial Intelligence, AI) se presenta como un cambio de paradigma respecto a cómo se abordan los problemas a través de la computación. En la programación tradicional, los problemas se diseccionan en otros problemas más pequeños que puedan ser abordados por instrucciones de bajo nivel de un ordenador. El aprendizaje automático (Machine Learning, ML), por otro lado, se basa en la capacidad de los sistemas informáticos para aprender de forma autónoma a partir de unos datos de entrada y sus salidas esperadas, sin la necesidad de haber sido programados para una tarea específica [8]. Esto se logra mediante el uso de algoritmos que logran identificar patrones en grandes cantidades de datos, a partir de los cuales hacer predicciones o toman decisiones [9]. El propio enfoque del ML hace que a ojos de los humanos las soluciones desarrolladas se comporten como "cajas negras", cuya lógica escapa al entendimiento.

En el campo del audio, la IA ha traído consigo numerosos avances relacionados con el procesado de señales, clasificación de eventos sonoros, reconocimiento de voz (*Automatic Speech Recognition, ASR*), etc. Algunos nombres muy relevantes en este campo son LALAL.AI [10]: una aplicación capaz de separar una canción en distintas pistas separadas por instrumentos; o Wav2Vec2 [11]: una red neuronal desarrollada por Facebook, que está logrando marcas históricas en el ámbito del ASR. Así mismo, muchas empresas del mundo de la producción audiovisual, como Adobe, están empezando a incorporar a sus programas herramientas basadas en IA que realizan tareas complejas y ahorra trabajo y tiempo al usuario.

La gran mayoría de las tecnologías de ML desarrolladas actualmente se basan en redes neuronales artificiales (Artificial Neural Networks, ANN), un tipo de modelo computacional que se asemeja al comportamiento de las neuronas biológicas. Dentro de las ANN existen distintas arquitecturas con distintas características: redes secuenciales, convolucionales, recurrentes, etc. Cabe destacar que, en la práctica, es usual combinar capas de distintas arquitecturas, añadiendo complejidad a la red.

El proceso mediante el cual la red aprende a partir de los datos que se le proporcionan se llama entrenamiento. A la hora de entrenar una red neuronal se emplean *datasets*, bases de datos de gran tamaño que contienen un conjunto de datos de entrada que se entregan a la red, y sus respectivas etiquetas, que hacen referencia al valor que se espera obtener a la salida

de esta. Dado el contexto del proyecto, se ha limitado el entrenamiento de la red al aprendizaje supervisado donde todo el conjunto de datos de entrenamiento está etiquetado. Es una práctica habitual dividir el *dataset* en dos conjuntos de datos: el de entrenamiento (*train*) y el de prueba (*test*), teniendo el conjunto de entrenamiento un porcentaje mayor de los datos del *dataset*.

En los siguientes subapartados se comenta la teoría fundamental tras las redes neuronales y se profundiza en las arquitecturas de red que se han empleado para este proyecto.

2.2.1 Funciones de activación

Antes de abordar la definición de las neuronas artificiales y las arquitecturas de red, resulta importante definir las funciones de activación. Estas funciones matemáticas se aplican, por norma general, a la salida de cada capa y cumplen varias funciones siendo las principales la introducción de no-linealidad en el sistema y la normalización de los valores de salida. Comúnmente las funciones de activación se aplican a la salida de cada capa de la red. Por convenio, se ha decidido nombrar a las funciones de activación en este documento con el símbolo ø. Para este proyecto se han empleado dos funciones de activación distintas, que se detallan a continuación.

2.2.1.1 Rectified Linear Unit (ReLU)

Durante el desarrollo e investigación de las redes neuronales, la tendencia a la hora de seleccionar la función de activación para las capas ocultas ha ido evolucionando. En la década de los 90, la función de activación más popular era la sigmoide, que un tiempo más tarde fue sobrepasada por la función tangencial hiperbólica. Estas dos funciones presentan un problema relativo a la distorsión, ya que sus valores de salida están limitados entre 0 y 1 en el caso de la sigmoide, y entre -1 y 1 en el caso de la tangente hiperbólica. Como resultado de esta distorsión, la función provoca pérdidas de información en los cálculos internos de la red. Cuando el valor de entrada es muy superior a 1, o muy inferior a -1, los valores de salida presentarán una variación mínima entorno a los límites de la función [8].

La función comúnmente llamada *Relu* por la comunidad internacional y conocida en español como rectificador es una función no lineal que se define de la siguiente manera:

$$ReLU(x) = max(0, x) = \begin{cases} x & si\ \ x > 0 \\ 0 & si\ \ x \leq 0 \end{cases} \quad (1)$$

Como se observa, la función mantiene a su salida el mismo valor de entrada para todos los valores positivos, mientras que para todo número negativo entrega un valor nulo. El cambio abrupto en la pendiente de la función para $x = 0$ es lo que hace que esta no sea lineal. Esto se puede observar claramente en la Figura 2, donde la derivada de la función da un salto abrupto en el eje de coordenadas:

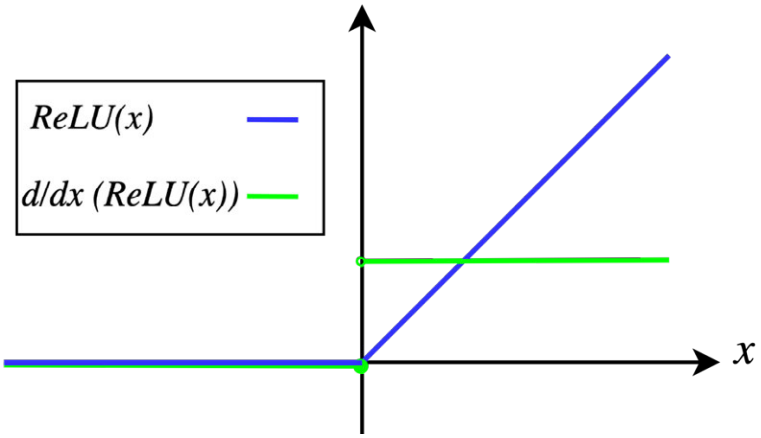

Figura 2. Gráfica de la función de activación ReLU y su primera derivada.

Debido a la simplicidad de computación, ausencia de distorsión, comportamiento lineal siendo una función no lineal y su conveniencia para el cálculo de la retropropagación, se ha seleccionado la función ReLU como función de activación a emplear tras cada capa de la red.

2.2.1.2 Softmax

Softmax es una función matemática lineal, cuyo uso en redes neuronales se limita a la capa de salida de la red. Esta función permite transformar un vector de valores de longitud K, en otro de probabilidades cuyos valores pertenecen al rango $[0, 1]$. La suma de todas las probabilidades siempre es 1. La función Softmax se expresa de forma matemática de la siguiente manera:

$$Softmax(\boldsymbol{x})_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \quad para\ j = 1, \ldots, K \tag{2}$$

Donde $\boldsymbol{x}$ es el vector de entrada, que contiene K elementos.

Esta función resulta vital a la hora de abordar problemas de clasificación con redes neuronales, permitiendo normalizar los valores entregados a la salida de la red y convertirlos en probabilidades asociadas a cada caso contemplado.

2.2.2 Neurona artificial y red neuronal prealimentada

Una red neuronal está formada por (como su propio nombre indica) neuronas artificiales interconectadas entre sí, cuya labor es propagar la información recibida por las neuronas de entrada hasta las de salida. Se considera neurona a la unidad mínima que cimienta las distintas arquitecturas complejas de red. Cada neurona cumple la función de recibir datos de otras neuronas, aplicarles una serie de operaciones matemáticas y transmitirlos a las siguientes neuronas. En la siguiente figura se muestra la arquitectura genérica de una neurona:

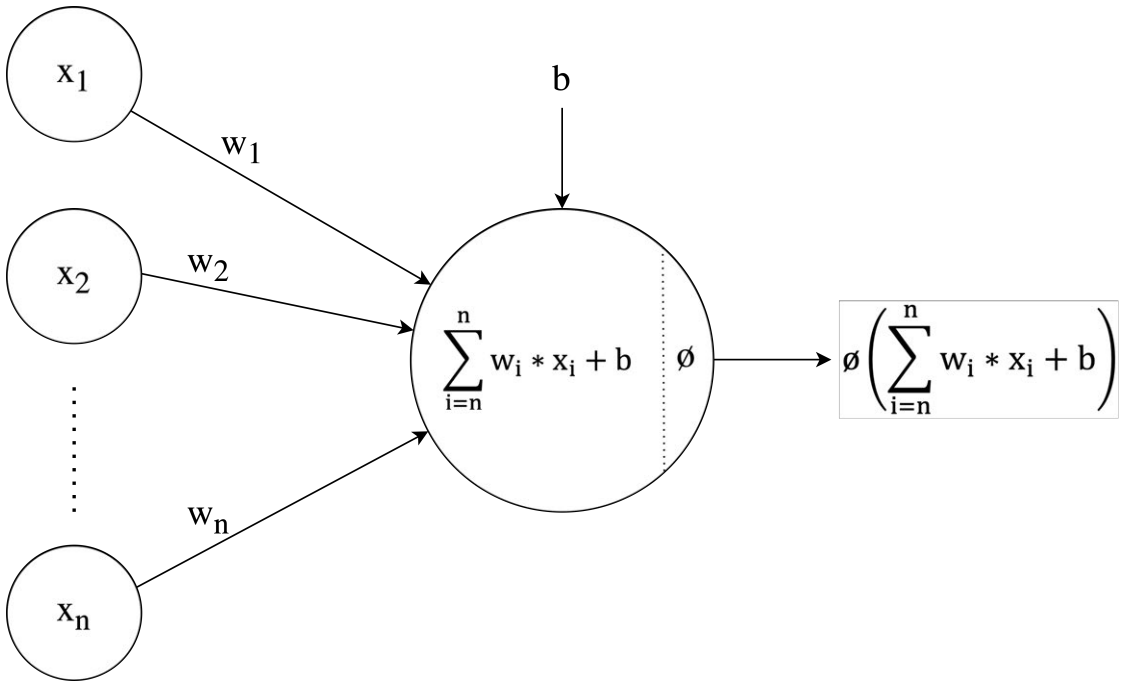

Figura 3. Diagrama de una neurona artificial genérica.

En la Figura 3 se observan las distintas partes de una neurona: los valores de entrada x_n, los pesos de cada conexión neuronal w_n, el sesgo o *bias* b y la función de activación ø. Los pesos pueden entenderse como el valor de importancia que tiene la conexión a la que van asociados. Como se observa en la figura, el valor a la salida de la neurona es el resultado del sumatorio de los productos de cada x de entrada por su peso w asociado, a este valor se le añade el sesgo b y se pasa por la función de activación ø. El sesgo actúa como un parámetro adicional que principalmente ayuda a mejorar el rendimiento de la red para relaciones de entrada/salida complejas. La función de activación y sus características han sido explicadas en el apartado 2.2.1. Los parámetros w y b son variables dinámicas durante el entrenamiento de la red. Esto significa que su valor irá variando conforme la red vaya aprendiendo de forma que, dentro de su topología, habrá neuronas que tengan más influencia que otras.

Por medio de la concatenación de capas formadas por múltiples neuronas artificiales se forma la arquitectura más básica de red neuronal: la red neuronal prealimentada (Feedforward Neural Network, FNN). Esta arquitectura también es conocida en inglés como *Dense, Fully-Connected o MLP (Multi-Layer Perceptron)*. En la Figura 4 se muestra un ejemplo de una arquitectura pequeña de FNN:

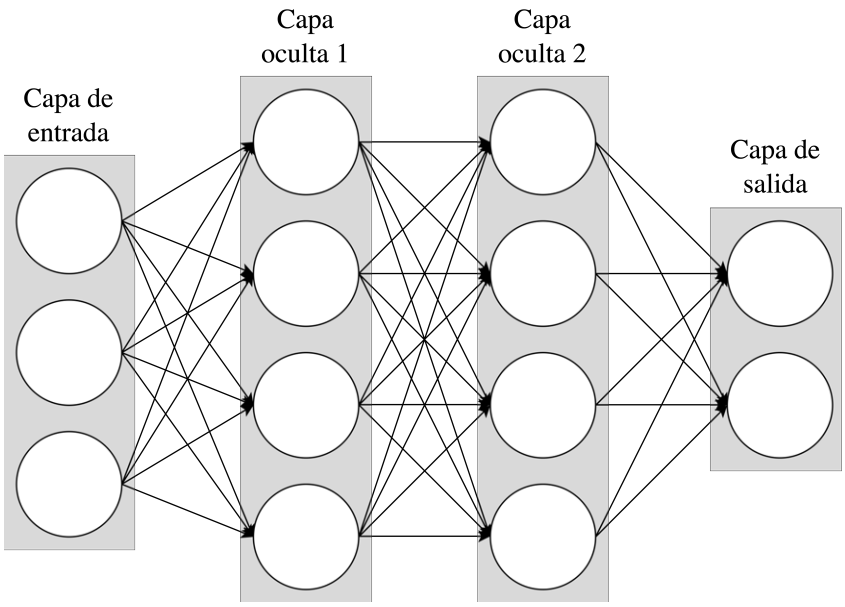

Figura 4. Diagrama de arquitectura de ejemplo FNN.

En el diagrama se puede apreciar la diferenciación entre capas: la primera y la última cumplen la función de entrada y salida a la red, mientras las capas intermedias se conocen como capas ocultas (*Hidden Layers)*. En esta arquitectura las uniones entre capas se realizan de modo que todas las neuronas de una capa están conectadas con todas las neuronas de las capas adyacentes. El motivo principal de diferenciar entre los tipos de capas es que las de entrada y salida deben ser dimensionadas teniendo en mente el tamaño y formato de los datos con los que se va a alimentar a la red, y el tipo de salida que se desea. La capa de salida está íntimamente relacionada con el problema que se desea resolver: si se diseña una red neuronal para un problema de clasificación (a partir de unos datos de entrada se desea determinar un caso entre un grupo de opciones definidas), la red deberá tener a su salida el mismo número de neuronas que de casos contemplados. En el contexto de este ejemplo, suponiendo una red entrenada, es común el uso de la función de activación *Softmax* (ver apartado 2.2.1.2), para obtener una probabilidad por cada neurona de salida, asignada a cada uno de los casos posibles de clasificación. De esta forma, se logra adaptar una red prealimentada a un problema de clasificación.

Las capas ocultas de la red son las encargadas de transportar y transformar los datos de entrada hasta la salida. Gracias a estas capas la red es capaz de reconocer patrones complejos en los datos de entrada. El dimensionamiento de estas capas es difícil de determinar de forma teórica, ya que para cada caso la complejidad varía y suele ser necesario realizar varias pruebas con distintas dimensiones de las capas ocultas. La capacidad de estas capas para reconocer patrones complejos es gracias en gran medida a las funciones de activación que se aplican tras cada neurona artificial. Las funciones de activación (ver apartado 2.2.1.1) hacen que la relación entre la entrada y la salida de la red sea no lineal. Esta no linealidad es la que permite a la red "reconocer" patrones que tampoco son lineales.

2.2.3 Arquitectura Convolucional 1D

2.2.3.1 CNN 1D vs CNN 2D para audio

Una de las arquitecturas de uso más extendido es la red neuronal convolucional (Convolutional Neural Network, CNN), más concretamente en su forma de dos dimensiones. Esto se debe a su aplicación para clasificar y segmentar imágenes con inteligencia artificial, una de las aplicaciones más investigadas en el campo. Esta arquitectura se basa en realizar operaciones de convolución de los datos de entrada de cada capa, por un vector/matriz que contiene los pesos que la red ajustará durante su entrenamiento. Este vector/matriz se suele nombrar como filtro o *kernel* y su forma depende de las dimensiones que tenga la red.

Dependiendo de las dimensiones de los datos de entrada, las CNN se pueden dimensionar desde una, hasta seis dimensiones. Las arquitecturas de mayor orden de dimensión suelen emplearse para procesar conjuntos complejos de datos como la colocación espacio-temporal de un sistema (CNN 4D) [12], interpolación de datos sísmicos (CNN 5D) [13], análisis de vídeo (CNN 3D), etc.

Un gran número de las investigaciones realizadas en el campo de la clasificación de audio mediante redes neuronales se basan en la utilización de CNN 2D. Siendo el audio digital una señal unidimensional: vectores de muestras representativas de la amplitud equiespaciadas en el tiempo, abordar el problema de esta forma resulta anti-intuitivo. Así pues, la forma más común de alimentar la red es por medio de transformaciones de audio como el espectrograma o la MFCC (Mel Frequency Cepstral Coefficient, MFCC) [4], siendo esta última la que tiene un uso más extendido. La MFCC es una transformación ponderada en frecuencia (se aplica un banco de filtros que siguen la Escala de Mel), que trata de emular el comportamiento del oído humano y está especialmente diseñada para reconocimiento de voz. Las dos transformaciones mencionadas entregan a su salida una matriz bidimensional, que representa la energía de la señal en función del tiempo y la frecuencia (o banda de Mel).

El enfoque de la clasificación de señales mediante ANN se fundamenta en la extracción de características (*feature extraction*) representativas de la señal, para la toma de decisiones de la red neuronal en función de dichas características. El preprocesado de audio con transformaciones como las mencionadas es una forma de extraer características de él, pero dichas características son completamente dependientes de la transformación. Por otro lado, la arquitectura CNN 1D, mucho menos explorada que la CNN 2D, realiza el proceso de extraer características por sí misma durante el entrenamiento. Por medio del proceso del descenso de gradiente la red adapta sus pesos para encontrar las similitudes entre muestras con la misma etiqueta y las diferencias con muestras de distinta etiqueta.

2.2.3.2 Arquitectura CNN 1D

Por norma general, dentro de una capa de red convolucional unidimensional se sigue el siguiente flujo de procesamiento de datos:

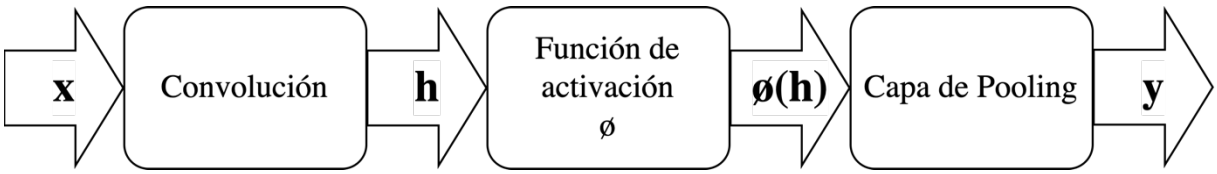

Figura 5. Diagrama de flujo de datos dentro de una capa convolucional 1D.

A continuación se explican los distintos bloques que componen la capa.

Empezando por la convolución, la capa recibe un vector de características de tamaño definido a su entrada. Realiza una operación de convolución de dicho vector con sus filtros/kernel internos que, como se ha mencionado en el subapartado anterior contienen los pesos ajustables de la red. En la Figura 6 se muestra de forma esquemática la operación de convolución de vectores:

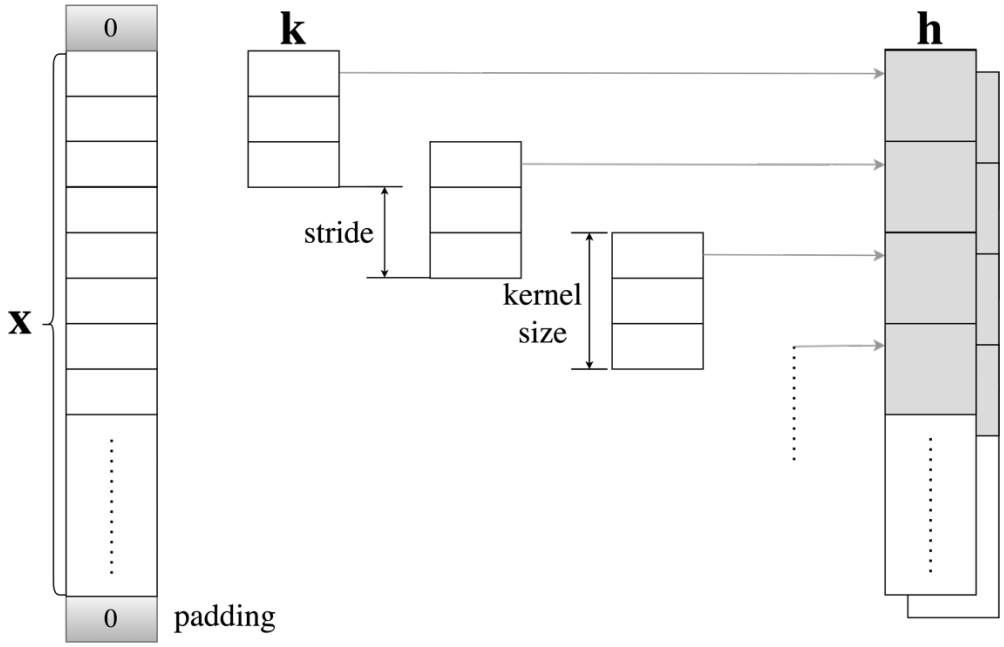

Figura 6. Representación de una capa de red convolucional 1D

En la figura se definen los vectores:

$\mathbf{x}$: Vector de entrada a la capa.

$\mathbf{k}$: Vector filtro/kernel.

$\mathbf{h}$: Vector resultante de la operación de convolución.

Matemáticamente se define la convolución realizada por la capa de la siguiente manera:

$$\mathbf{h}(t) = (\mathbf{x} * \mathbf{k})(t) = \sum_{\tau=-m}^{m} \mathbf{x}(t-\tau) \cdot \mathbf{k}(\tau) \qquad (2)$$

donde:

$$\mathbf{x} \in \mathbb{R}^n \\ \mathbf{k} \in \mathbb{R}^{2m+1} \tag{2}$$

En palabras: cada posición del vector $\mathbf{h}$ se calcula como el sumatorio de los productos de los vectores $\mathbf{x}$ y $\mathbf{k}$ para cada posición desplazada del vector $\mathbf{k}$. En la ecuación (2) se observa que el tamaño de $\mathbf{k}$ debe ser impar, esto se hace por convención y simplicidad a la hora de ajustar el tamaño de las capas. Se definen también cuatro variables para caracterizar la convolución:

stride: Esta variable indica el tamaño del salto (en número de posiciones del vector) que se desplaza el kernel.

kernel_size: Tamaño del filtro interno de la capa, es igual al tamaño del vector $\mathbf{k}$.

out_channels: Cantidad de vectores que se espera a la salida de la capa. Esto implica la capa contendrá el mismo número de vectores $\mathbf{k}$ que de canales de salida. Esto aumenta la cantidad de conexiones neuronales internas y ayuda a dar más profundidad a la red.

padding: Cantidad de valores nulos que se añaden al inicio y final del vector $\mathbf{x}$. El padding ayuda a mantener las dimensiones de los vectores en su paso las diferentes capas y convoluciones de la red.

En la práctica, resulta importante conocer el efecto que tiene la convolución en el tamaño del vector $\mathbf{h}$, ya que es un factor clave a la hora de dimensionar las capas posteriores. El tamaño del vector $\mathbf{h}$ puede ser calculado con la siguiente expresión matemática [14]:

$$\text{Tamaño}(\mathbf{h}) = \left(\frac{\text{X} - \text{kernel\_size} + 2 * padding}{stride} + 1\right) \times \ out\ channels \tag{3}$$

Donde X representa el tamaño del vector $\mathbf{x}$. El término $\times \ out\ channels$ hace referencia al número de vectores que la capa entrega a su salida.

De forma similar al caso de la neurona artificial, es necesario aplicar una función de activación a la salida de la capa para lograr una relación no lineal entre la entrada y la salida de la capa.

Además de la función de activación, es común añadir una capa de *pooling*, en español: agrupación. La misión de la capa de *pooling* es reducir el tamaño del vector que recibe a su entrada, manteniendo las características más importantes. Esto se logra por medio de operaciones tales como el promediado o el valor máximo. En este trabajo se va a limitar el estudio a la capa de *pooling* de valor máximo que recibe el nombre de *Max Pooling*. El funcionamiento de la capa *Max Pooling* se basa en el uso de una ventana deslizante que recorre el vector de entrada y toma el valor máximo dentro de ella. El tamaño de la ventana y del desplazamiento son parámetros ajustables y reciben el nombre de *pool size* y *stride*. En la siguiente figura se muestra su representación gráfica:

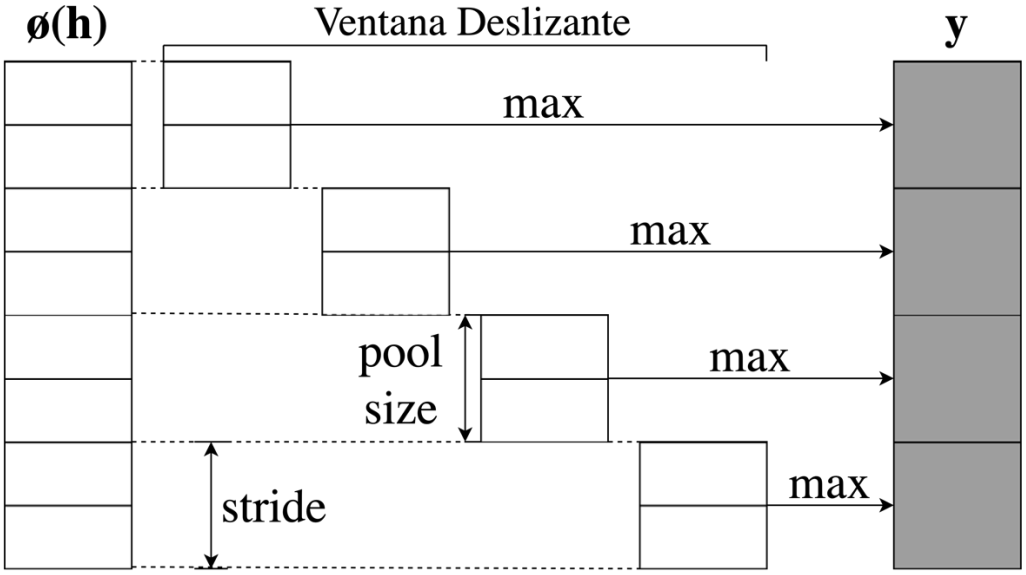

Figura 7. Representación gráfica de la capa MaxPooling.

Al igual que en el cálculo de la convolución, la capa *Max Pooling* altera el tamaño del vector resultante en función de sus parámetros. La fórmula 4 representa la función matemática que permite calcular dicho tamaño:

$$\text{Tamaño salida MaxPool} = \frac{\text{Tamaño}(\mathbf{h}) - \text{pool size}}{stride} + 1 \qquad (4)$$

Tras calcular la salida de la capa de *pooling*, el vector resultante pasa a ser la entrada a la siguiente capa de la red. Cabe mencionar que tanto la función de activación, como la capa de *pooling* pueden ser consideradas como procesados internos de la capa convolucional. Esto se debe a que, en la práctica, una capa convolucional sin estos elementos no permite un buen rendimiento para la red neuronal.

2.2.4 Retropropagación

Como ya se ha comentado en los subapartados anteriores, las distintas capas de la red contienen variables llamadas pesos y biases, cuyo valor debe ser ajustado para mejorar el funcionamiento de la red. Esto se debe a que, al inicializar la red, sus pesos y biases son asignados de forma aleatoria. El ajuste de estas variables se realiza por medio de un algoritmo llamado retropropagación o propagación hacia atrás (*backpropagation*). El proceso mediante el cual se aplica dicho algoritmo recibe el nombre de entrenamiento (*train*). A continuación se explica de forma resumida la teoría detrás del proceso de entrenamiento.

En primer lugar, se le proporciona un conjunto de datos de entrada a la red, llamado *batch*. El tamaño del *batch* (*batch_size*) es un parámetro configurable por el usuario, y afecta únicamente a la velocidad del entrenamiento, en función de la capacidad computacional de la máquina donde se ejecuta. A partir del conjunto de datos de entrada, la red realiza sus cálculos para propagar la información hasta la salida. Una vez obtenidas todas las predicciones, estas se comparan con las etiquetas asociadas a los datos de entrada por medio

una función de error. Las dos funciones de error más empleadas son el Error Cuadrático Medio (Mean Squared Error, MSE) y la Entropía Cruzada (Cross Entropy Loss, CEL). Para abordar este proyecto se ha elegido la función de entropía cruzada. Esta elección se debe a que la función CEL está especialmente pensada para trabajar con probabilidades, lo cual la hace idónea para un problema de clasificación donde se emplea la función de activación *Softmax* a su salida. A continuación se muestra la expresión matemática de la función de entropía cruzada:

$$\mathrm{CEL}(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{C} y_i \cdot \log p_i \tag{5}$$

Donde $\mathbf{y}$ y $\mathbf{p}$ son los vectores de las etiquetas y las predicciones respetivamente y C es el número de clases contempladas.

Una vez definida la función de error, es posible desarrollar la expresión aprovechando que $\mathbf{p}$ (la salida de la red) se puede expresar como una función conocida, dependiente de los valores de entrada a la red ($\mathbf{x}$) y sus pesos ($\mathbf{w}$). La función $\mathbf{p}(\mathbf{x}, \mathbf{w})$ resultante es una combinación de todos los pasos internos de la red, con sus productos, sumatorios y funciones de activación. Al poder expresar la función de error en función de los pesos, se calcula su gradiente respecto a cada peso de la red. Esto se realiza aplicando la regla de la cadena, que permite descomponer la operación en cálculos más manejables. Gracias al gradiente es posible conocer las tendencias de la función de error y, por medio del descenso de gradiente, variar el valor de los pesos en la dirección opuesta al crecimiento máximo. De esta forma se actualizan los valores de los pesos:

$$w_{ij}^{t+1} = w_{ij}^{t} - \eta \frac{\partial\, \mathrm{CEL}(\mathbf{y}, \mathbf{p}(\mathbf{x}, \mathbf{w}))}{\partial w_{ij}} \tag{6}$$

Donde:

- w_{ij}^{t+1} corresponde al peso actualizado.

- w_{ij}^{t} es el peso actual.

- η es la tasa de aprendizaje, parámetro seleccionable por el usuario.

- $\frac{\partial\, \mathrm{CEL}(\mathbf{y},\mathbf{p}(\mathbf{x},\mathbf{w}))}{\partial w_{ij}}$ es el gradiente de la función de error, con respecto al peso w_{ij}.

Esta operación hace que la precisión de la red vaya incrementando progresivamente y se aplica para todo el conjunto de datos de entrenamiento.

Para que la red sea capaz de ajustar sus pesos y biases en todo su potencial, es normal necesitar repetir el proceso de entrenamiento un número determinado de veces sobre el mismo conjunto de datos. A este número de veces se le llama épocas (*epochs)*.

2.2.5 Métricas

Durante el entrenamiento se calculan una serie de métricas que permiten diagnosticar posibles malfuncionamientos de la red, además de cuantificar su éxito. En este proyecto se ha decidido utilizar dos métricas: el error cuadrático medio (MSE) y el porcentaje de aciertos (Accuracy). El error cuadrático medio viene dado por el promediado de los cuadrados de la diferencia de los valores esperados y y los valores predichos por la red p:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - p_i)^2 \tag{7}$$

El porcentaje de aciertos viene definido por la siguiente expresión:

$$Accuracy = \frac{nº\ aciertos}{nº\ entradas} * 100\ [\%] \tag{8}$$

Durante la ejecución del entrenamiento, se almacenan los valores de las métricas para todas las propagaciones de la red. Adicionalmente, cada vez que se completa una época, es decir, cada vez que se entrena la red con el contenido íntegro del dataset de entrenamiento, se pone a prueba la red con el conjunto de prueba. Al testear la red, se desactiva el proceso de propagación de errores y ajuste de pesos y biases, ya que se realiza únicamente para tomar métricas con datos sobre los que la red no ha aprendido nunca.

2.3 Mensajes MIDI

El protocolo MIDI (Musical Instrument Digital Interface) [1] es un estándar que permite la comunicación entre distintos instrumentos musicales electrónicos, ordenadores y otros dispositivos relacionados con la música. Este protocolo se basa en el envío de mensajes que contienen información sobre diversos aspectos de la música de una interpretación musical. Los mensajes MIDI son cruciales para la producción musical moderna, permitiendo una integración fluida entre múltiples dispositivos y software.

En el contexto de este proyecto, los mensajes MIDI de interés son dos: Note ON y Note OFF. Por medio de estos mensajes se ordena el inicio de una nota y el fin de esta. Por lo general el protocolo MIDI se emplea en tiempo real, por lo que un instrumento con esta interfaz comenzará a reproducir una nota al recibir un mensaje Note ON, y la detendrá al recibir un Note OFF asociado a esa misma nota. Los mensajes se codifican como secuencias de bytes, donde el primer bit de cada byte está reservado. A modo de ejemplo en la Figura 8 se muestra la codificación de un mensaje Note ON con sus distintos parámetros:

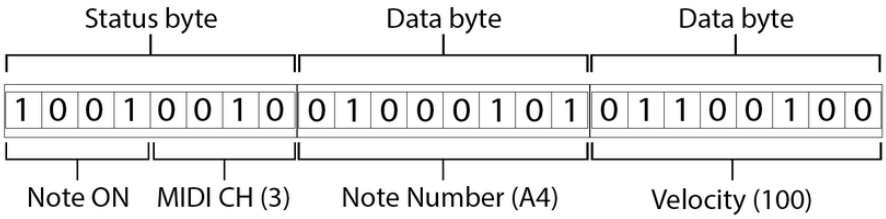

Figura 8. Ejemplo de bytes de un mensaje MIDI Note ON [15].

Los tres parámetros que contiene un mensaje Note ON/OFF son:

- Número de nota: Indica la nota musical que se debe activar o desactivar. Se codifica como un número binario de 7 bits, por lo que se pueden referenciar hasta 128 notas distintas. La nota central del piano C4, se corresponde con el número de nota 60 y cada unidad representa un salto de semitono.

- Velocity: Representa la intensidad con la que se ha tocado una nota, de forma que se pueda mantener la dinámica en la interpretación musical. Este valor se codifica con 7 bits, por lo que se cuantifica con 128 escalones.

- Canal MIDI: Permite organizar y dirigir el flujo de datos entre distintos dispositivos. No resulta de interés para este proyecto.

Con esta información se puede concluir que para la conversión de audio-MIDI, se deberá analizar el audio de modo que se extraiga: la nota tocada y la fuerza con la que se ha tocado.

3. Especificaciones y restricciones de diseño

El proyecto se ha planteado de forma que se satisfagan las siguientes especificaciones:

- Se ha centrado la investigación en la detección de tono, obviando la estimación de la intensidad de cada nota.
- El número de notas distintas que la red debe reconocer es limitado ya que la arquitectura de red y el tamaño del *dataset* dependen de este número.
- El entorno de desarrollo empleado se basa en el lenguaje de programación Python, ya que ofrece una gran cantidad de librerías de código abierto relacionadas con el *machine learning*. Se ha decidido emplear la librería Pytorch para la creación y entrenamiento de la red neuronal.
- Los datasets deben ser originales, capturados con un instrumento musical propio.
- Las muestras de audio empleadas tanto para el entrenamiento como para el testeo deben ser monofónicas.
- Las arquitecturas de ANN propuesta emplean capas CNN 1D y MLP.
- El resultado final consta de una red neuronal capaz de reconocer notas musicales a partir de sus transitorios iniciales, pudiendo asegurarse su eficiencia exclusivamente para el instrumento y las notas con las que se ha entrenado.

4. Descripción de la solución propuesta

En este proyecto se propone el desarrollo de una red neuronal capaz de clasificar eventos transitorios de audio para determinar la nota musical tocada. Para lograr este fin se ha dividido el proceso en tres fases: grabación y generación del *dataset*, diseño y creación de la red neuronal y entrenamiento de la red neuronal. Estas fases de desarrollo se explican en detalle en los siguientes subapartados.

Se han realizado pruebas tanto para guitarra eléctrica como para bajo eléctrico, por lo que ha sido necesario crear *datasets* distintos. Todas las señales de audio tratadas en el proyecto han sido grabadas a una frecuencia de muestreo de 48kHz y en ningún momento han sido re-muestreadas. La resolución a la que han sido grabadas es de 24 bits, pero durante su procesado se han rescaldas a 32 bits, ya que se han tratado como variables de tipo *float32*.

El entrenamiento de las redes se ha realizado para distintos casos propuestos, variando el número de opciones contempladas por la red y el tamaño del *dataset*. Con esto se ha pretendido estudiar el comportamiento en lo que podría ser una solución comercial, donde el usuario entrena la red para reconocer las notas con su propio instrumento.

4.1 Creación de Dataset

Para un correcto entrenamiento de la red resulta primordial recolectar un buen conjunto de datos para generar el *dataset*. Esto implica que se incluyan muestras representativas del problema que se pretende resolver y que sean suficientes para que la red sea capaz de ajustar sus pesos correctamente. Para este proyecto, se ha decidido crear un *dataset* propio, ya que no se puede asegurar que el transitorio inicial producido por dos instrumentos musicales distintos sea igual. Es por ello, que se ha decidido generar *datasets* distintos para cada caso de estudio, diferenciando entre guitarra y bajo.

4.1.1 Enventanado temporal del transitorio

El primer paso para la creación del *dataset* es definir los datos de entrada que alimentarán la red, donde debe figurar el audio correspondiente al evento transitorio inicial de cada nota. Computacionalmente resulta beneficioso emplear tamaños igual a potencias de dos para la entrada a las redes neuronales [16] debido a los propios cálculos del procesador a la hora de abordar convoluciones. Conociendo este hecho se estudia el evento transitorio inicial de una nota producida por una guitarra y un bajo para determinar su duración. En la Figura 9 se muestra una comparación de la señal de audio para el Mi grave de la guitarra y el Mi grave del bajo:

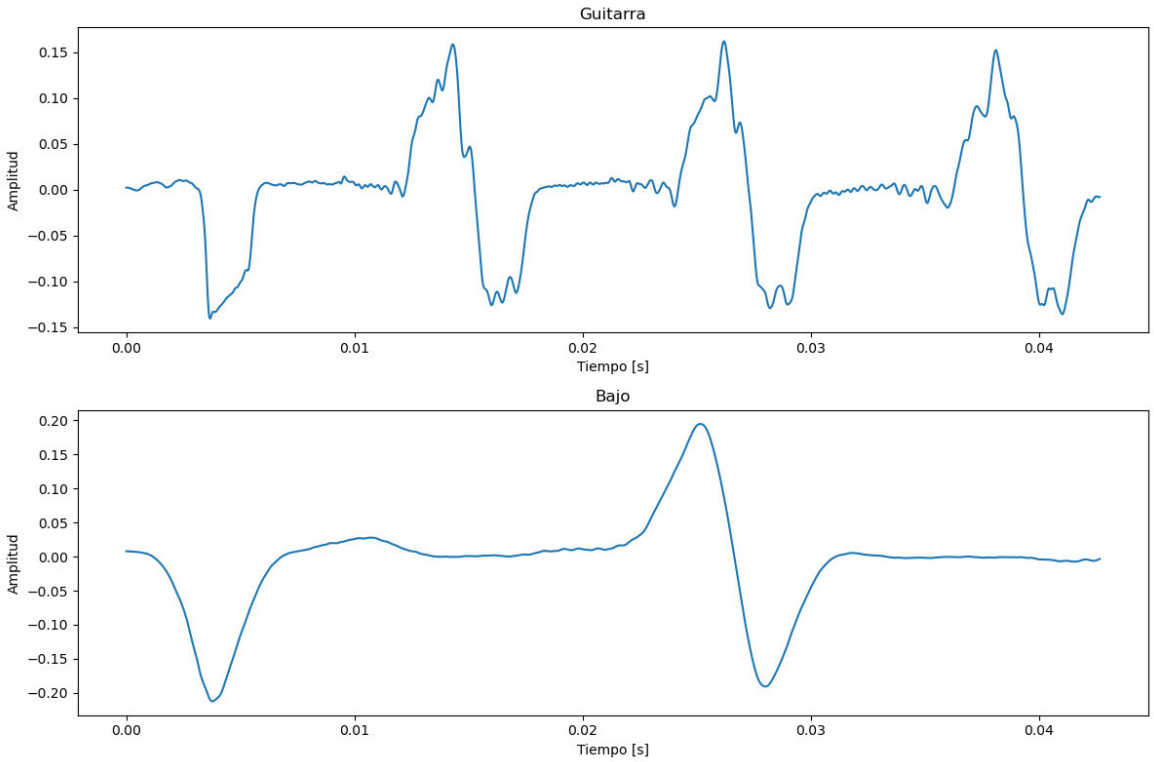

Figura 9. Señales de audio para los instantes iniciales de la nota Mi grave de una guitarra y un bajo.

Se puede apreciar que la frecuencia de la nota tocada por el bajo es la mitad de la tocada por la guitarra, esto se debe a que las cuerdas de un bajo eléctrico están afinadas una octava por debajo de las de la guitarra. Se ha estudiado la nota más grave de ambos instrumentos (E1, cuarta cuerda al aire para bajo y E2, sexta cuerda al aire para guitarra), ya que estas son las que presentan un período más grande. Teniendo en cuenta que se ha respetado que el tamaño de entrada de la red sea potencia de dos, se ha decidido coger una ventana temporal de 512 muestras para la guitarra y de 1024 muestras para el bajo. Conociendo el tamaño en muestras de las ventanas y la frecuencia de muestreo se ha calculado la duración de ambas ventanas de la siguiente manera:

$$Duración\ Ventana = \frac{Muestras}{Fs} = \begin{cases} Dur.Vent._{guitarra} = \dfrac{512}{48000} = 0.010\bar{6}\ seg \\ Dur.Vent._{bajo} = \dfrac{1024}{48000} = 0.021\bar{3}\ seg \end{cases} \quad (9)$$

De esta forma las señales enventanadas quedan como se muestra en la figura 10:

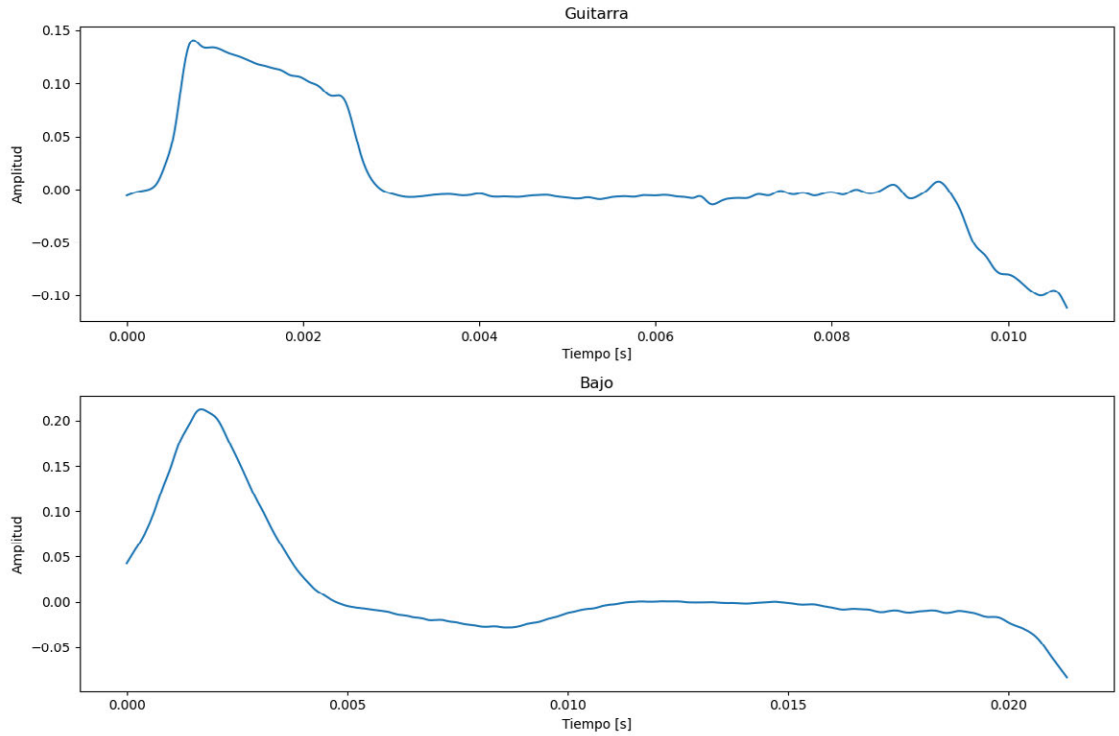

Figura 10. Señales de audio enventanado para los instantes iniciales de la nota Mi grave de una guitarra y un bajo.

Se aprecia que en la ventana temporal de ambas señales se captura el pico inicial, provocado por la pulsación de la cuerda, además de los instantes posteriores. En los últimos milisegundos de la ventana temporal se aprecia una bajada: esto corresponde a los primeros instantes del régimen armónico. Elegir una ventana temporal de duración mitad de las elegidas no presenta suficiente información para que la red sea capaz de distinguir entre notas, ya que la única información incluida es el pico causado por la pulsación, que es idéntico para todas las notas. Una vez definido el tamaño de la ventana temporal que se proporcionará a la red, se ha procedido a captar los datos necesarios para el *dataset*.

4.1.2 Grabación y segmentación del audio

Dado que el entrenamiento de la red requiere un gran volumen de datos, se ha planteado un método semiautomático con el que recopilarlos. Recortar la porción de audio correspondiente al transitorio inicial de forma manual llevaría una cantidad de tiempo muy elevada, ya que el tamaño de los *datasets* suele rondar el orden de miles de entradas etiquetadas. El proceso que se detalla a continuación se resume en grabar un gran número de percusiones de notas para más tarde recortar el trozo de audio de interés por medio de código.

Se han definido tres casos de estudio distintos, a los cuales se hará referencia como Caso 1, Caso 2 y Caso 3. Con estos casos se pretende estudiar el comportamiento de la red y su capacidad para aprender en distintos escenarios. Con el Caso 1 se plantea la clasificación de las cinco notas más graves de la guitarra separadas por un semitono las unas de las otras, todas tocadas en la sexta cuerda. Con el Caso 2 se plantea el mismo escenario que en el Caso 1, pero transportado al bajo. El Caso 3 se ha planteado para comprobar la escalabilidad de la

red con más opciones de clasificación. Para ello se pretende clasificar doce notas tocadas en un bajo, siendo las doce notas las correspondientes a las cuatro cuerdas al aire y sus dos primeros trastes. Para el estudio de los casos se ha generado un dataset distinto por caso y sus características se definen en la siguiente tabla:

Tabla 2. Definición de los casos de estudio.

Caso 1		Caso 2		Caso 3	
Instrumento	Guitarra	Instrumento	Bajo	Instrumento	Bajo
Notas	E2, F2, F#2, G2, G#2	Notas	E1, F1, F#1, G1, G#1	Notas	E1, F1, F#1, A2, A#2, B2, D2, D#2, E2, G2, G#2, A3
Nº Notas	5	Nº Notas	5	Nº Notas	12

Para la captación de audio se ha empleado un montaje básico como se muestra en la figura 11:

Figura 11. Conexionado de equipos para captación de muestras.

Los componentes usados durante el proceso de grabación son los siguientes:

- Guitarra eléctrica *Maybach Stradovari S61*, tipo *Stratocaster*.
- Bajo eléctrico Luxor Japones.
- Interfaz de audio *Focusrite Clarett 2Pre USB.*
- PC MacBook Pro 13 pulgadas, 2018.

Mediante la tarjeta de sonido se realiza la conversión analógico-digital de la señal de audio producida por el instrumento. Como ya se ha mencionado, dicha conversión se ha realizado con una frecuencia de muestreo de 48 kHz. El DAW (Digital Audio Workstation, Estación de Trabajo de Audio Digital) seleccionado es *Logic Pro X* [17].

La grabación se ha realizado separando por pistas las distintas notas: todas las pulsaciones de una misma nota se han grabado en una pista. Se ha hecho así para facilitar el procesado

posterior de los datos. La pulsación de las notas se ha realizado con una púa en el caso de la guitarra, y con el dedo pulgar en el caso del bajo, siempre percutiendo la cuerda de arriba a abajo. Dado que se ha decidido obviar la intensidad (*velocity*) en este proyecto, se han grabado las notas percutidas con distintas intensidades. De esta forma la red aprende a partir de casos representativos y se reducen las posibilidades de que sufra *overfitting* [18].

En la figura 12 se presenta una captura de pantalla de las pistas resultantes tras grabar las muestras para el dataset correspondiente al Caso 1.

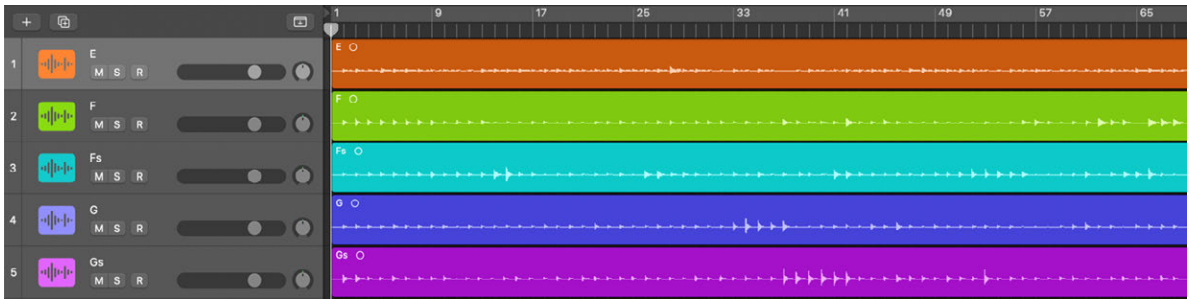

Figura 12. Captura de pantalla de las pistas tras grabar.

Notar que las pistas se han nombrado por la nota que se ha tocado en cada una de ellas, utilizando cifrado americano y utilizando una "s" para hacer referencia al sostenido. Una vez grabadas todas las notas que se desean incluir en el *dataset*, se ha obtenido la información temporal de donde se sitúan los transitorios. Para ello se ha empleado una herramienta interna del propio DAW, llamado "Reemplazo de percusión". Originalmente esta herramienta está pensada para extraer la información de tiempo de donde suceden los golpes de una batería, y así poder mezclarlos con otros sonidos sin perder el "*groove*" de la interpretación. Se ha aprovechado esto, para extraer un archivo MIDI que contiene notas situadas exactamente en los instantes de onset de cada pulsación. La configuración empleada para la herramienta se muestra en la Figura 13.

Figura 13. Configuración del reemplazo de percusión.

Solo ha sido necesario ajustar el umbral relativo, para evitar que el algoritmo detecte golpes falsos. Tras el ajuste, el programa crea una pista nueva donde coloca las notas MIDI asociadas a cada nota (ver Figura 14).

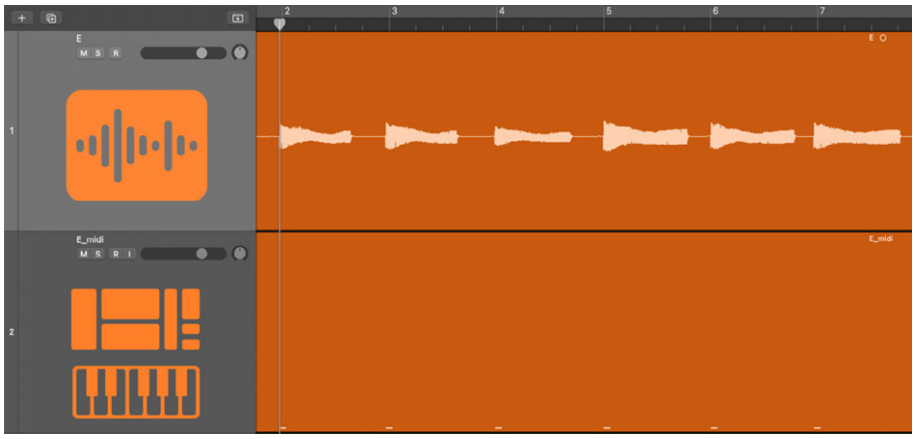

Figura 14. Pistas de audio y MIDI tras aplicar el reemplazo de percusión.

Tras recolectar las pistas de audio, y sus respectivas pistas MIDI con los tiempos de onset asociados a cada nota, se han exportado todas ellas. Las pistas de audio se han exportado como archivos *.wav*, con la frecuencia de muestreo original; y las pistas MIDI se han exportado como archivos *.mid*. Se ha nombrado a cada archivo por el nombre de la nota que contiene, junto al número de octava. Por ejemplo, los archivos asociados a las pistas de la figura 14 se han nombrado: E2.wav y E2.mid, ya que la sexta cuerda de la guitarra está afinada en Mi de la octava número 2.

El pequeño desfase que pueda haber en la colocación de los tiempos de onset se puede considerar despreciable para este proyecto. Esto se debe a una cualidad de las redes neuronales convolucionales, llamada *shift invariant* (Invariante al desplazamiento) [19]. Por el proceso de Max Pooling la red es capaz de reconocer patrones independientemente de su colocación en la capa de entrada.

Una vez se tienen las dos carpetas que recopilan los archivos de audio y MIDI, se ha empleado un programa de Python propio, que recortar la ventana temporal de interés de cada nota y la almacena en una estructura de datos CSV (*Comma-separated values*, Valores separados por comas). Los archivos *.csv* son muy usados en el campo del ML, ya que facilitan mucho almacenar sets de datos en forma de tabla o *dataframe* y su compatibilidad está altamente respaldada por numerosas librerías de código.

La secuencia de operaciones que realiza el código para el preprocesado del *dataset* es la siguiente:

- El programa recorre la carpeta indicada en busca de archivos *.wav*
- Al encontrar un archivo de audio, lo abre y almacena todas las muestras que contiene en un array. También almacena el nombre del archivo en una variable, ya que este indica la nota que ha sido grabada en el archivo. El nombre del archivo se trata con comparación de caracteres y se extrae el número de nota MIDI correspondiente.

- Para el archivo de audio iterado, se lee su respectivo archivo MIDI y se almacenan los tiempos de comienzo de nota en otro array. Para esto se ha empleado la librería pretty_midi [20], que proporciona un método que permite iterar a lo largo del archivo y obtener los instantes en segundos donde hay mensajes NoteOn.
- Se recorre el archivo de audio cargado y se van recortando las ventanas temporales marcadas por los instantes de NoteOn. Se ha realizado una conversión sencilla de segundos a número de muestra, para poder operar sobre la matriz de audio.
- Las ventanas obtenidas se almacenan en una estructura de datos de tipo Lista de Python, asociando cada vector de muestras de audio con su etiqueta: el número de nota.
- Se repite el proceso para cada archivo de la carpeta. Al terminar, se convierte la Lista a una estructura de tipo *dataframe* y se almacena en un archivo *.csv*.

Tras ejecutar el programa de forma satisfactoria, se ha inspeccionado el archivo *.csv* y se observa lo siguiente (figura 15):

```
1   ,label,audio
2   0,43,"[0.010028839111328125, 0.009583711624145508, 0.009215593338012695,
3   1,43,"[0.002613067626953125, 0.002501845359802246, 0.0024241209030151367
4   2,43,"[0.010554909706115723, 0.011181473731994629, 0.011827468872070312,
5   3,43,"[0.007681012153625488, 0.008015275001525879, 0.008304834365844727,
6   4,43,"[0.01014244556427002, 0.010834217071533203, 0.011627197265625, 0.0
7   5,43,"[0.010717391967773438, 0.010680675506591797, 0.010772943496704102,
8   6,43,"[0.009454011917114258, 0.009878873825073242, 0.010297417640686035,
9   7,43,"[-0.0016807317733764648, -0.0017038583755493164, -0.00166773796081
```

Figura 15. Archivo CSV del dataset de guitarra.

Se observan los primeros ocho registros del archivo donde la primera columna muestra el índice del registro, la segunda, nombrada "*label*" almacena la etiqueta de cada registro y la tercera, nombrada "audio" almacena el vector de muestras correspondiente al transitorio de la nota. En el caso del dataset de la guitarra el vector "audio" contiene 512 muestras y en los *datasets* del bajo 1024.

Se ha realizado un recuento de los datasets y se han obtenido el siguiente tamaño para cada uno:

Tabla 3. Recuento de muestras de los datasets (1).

Dataset Caso 1 - Guitarra		Dataset Caso 2 - Bajo	
Etiqueta	Nº Entradas	Etiqueta	Nº Entradas
40	660	28	200
41	664	29	200
42	671	30	200
43	662	31	200
44	662	32	201
Total	3319	Total	1001

Tabla 4. Recuento de muestras de los datasets (2).

Dataset Caso 3 - Bajo	
Etiqueta	Nº Entradas
28	200
29	200
30	200
33	200
34	200
35	200
38	200
39	200
40	200
43	200
44	200
45	200
Total	2400

Como resultado final del subapartado, se han obtenido tres archivos CSV que contienen los *datasets* para los tres casos propuestos. De esta forma, finaliza el preprocesado de datos.

4.2 Diseño de la arquitectura de red

La arquitectura propuesta para la clasificación de transitorios se basa en capas convolucionales de una dimensión y capas densas. Por medio de la concatenación de capas convolucionales multicanal (con sus respectivas funciones de activación) se logra que la red actúe como un extractor de características [21], y por medio de las capas densas la red "toma decisiones" en función de dichas características. Es por esto, que se ha decidido emplear una

arquitectura de red donde, la entrada sea procesada por una serie de capas convolucionales, que entreguen su salida a un grupo de capas densas cuya capa de salida tenga el mismo número de neuronas que etiquetas distintas el *dataset*.

En la figura 16 se muestra una representación de la arquitectura propuesta:

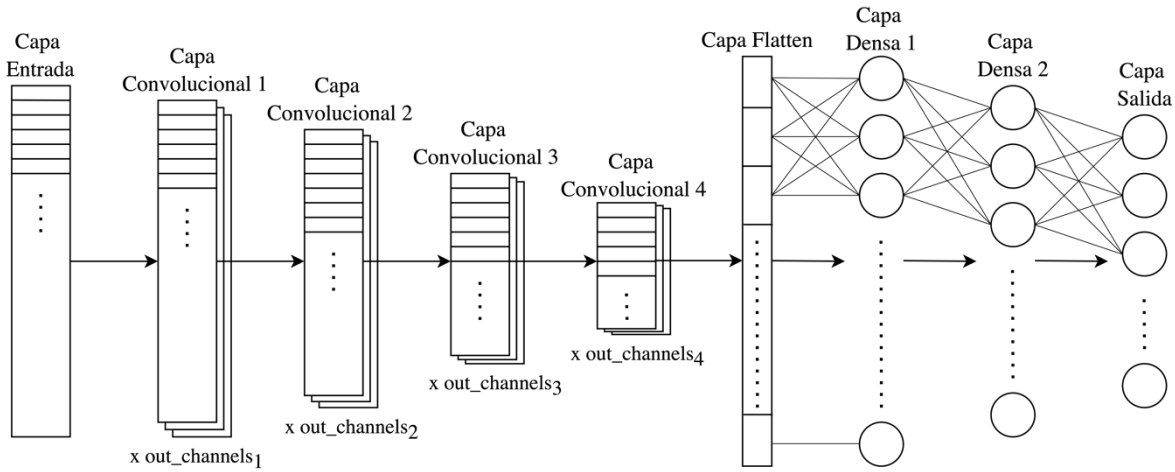

Figura 16. Diagrama de la arquitectura propuesta.

Como se aprecia, la arquitectura esta formada por cuatro capas convolucionales de una dimensión, una capa *flatten* y tres capas densas. La capa *flatten* se encarga de serializar todos los vectores que llegan a su entrada. Dado que las capas convolucionales van decreciendo en tamaño pero creciendo en profundidad con los canales de salida, es necesario recopilar todos los vectores y redistribuirlos en un vector de una única dimensión para entregárselo a las capas densas.

Dentro de las capas convolucionales se incluyen la función de activación elegida (ReLU), y la capa de Max Pooling. Con estos elementos, el flujo interno de datos en la capa convolucional es el siguiente (Figura 17):

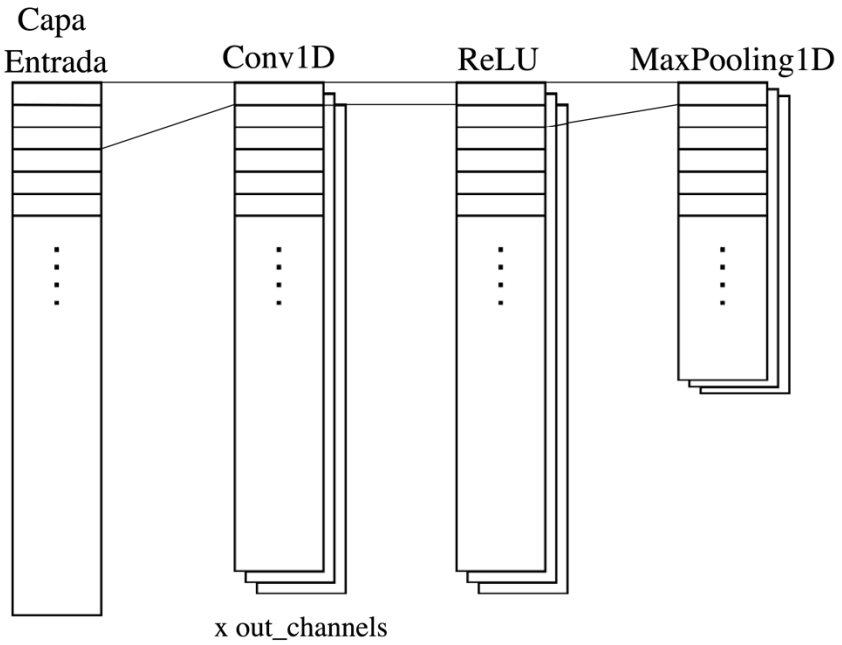

Figura 17. Diagrama de una capa convolucional.

Notar que, dentro de la capa la operación de convolución reduce el tamaño del vector de entrada (3) y le da profundidad (se realiza la convolución un número de veces igual a *out_channels*, con distintos *kernel*). A todos los vectores/canales resultantes de la convolución se les aplica la función de activación ReLU, que no afecta a sus dimensiones, y la capa de pooling, que sí que afecta a sus dimensiones, y las reduce en función de la expresión (4). En la programación de la red solo ha sido necesario calcular el tamaño de salida de la capa *flatten* ya que las capas densas requieren que se especifique su tamaño exacto y que coincida con los valores de entrada, mientras que las capas convolucionales no requieren que se especifique el tamaño de entrada más allá del número de canales de salida y entrada. Esto se debe a que en las capas densas, el número de pesos depende de las conexiones que haya entre capas, mientras que en las capas convolucionales el tamaño de los pesos solo depende del *kernel size* definido.

Internamente las capas densas también incluyen su función de activación ReLU, y en la capa de salida la función Softmax para poder interpretar los datos calculados por la red como probabilidades.

Los parámetros (estudiados en el apartado 2.2.3.2) que gobiernan las operaciones de convolución y pooling elegidos son los que se muestran en la tabla 5.

Tabla 5. Parámetros empleados para la arquitectura propuesta.

	Capa Conv. 1	Capa Conv. 2	Capa Conv. 3	Capa Conv. 4
Convolución	in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=0	in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=0	in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=0	in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=0
Max Pooling	kernel_size=2, stride=2	kernel_size=2, stride=2	kernel_size=2, stride=2	kernel_size=2, stride=2

Se han elegido los números de canales indicados de forma que los canales de salida de una capa coincidan con los canales de entrada de la siguiente. El valor de *kernel_size* y *stride* de la convolución se han elegido teniendo en cuenta que el tamaño de la entrada no es muy grande en su expresión máxima (*datasets* de bajo, 1024 muestras) y que valores más elevados para estas variables reducirían mucho el tamaño de los vectores a su paso por la red. El parámetro *kernel_size* relativo a la capa de Max Pooling hace referencia al parámetro *pool_size* definido en el apartado 2.2.3.2. En este apartado se ha nombrado como *kernel_size* porque así es como se hace referencia a ese parámetro en la librería Pytorch, pero no debe confundirse con el parámetro *kernel_size* relativo a la convolución.

Las dimensiones a la salida de la última capa convolucional se han calculado siguiendo las fórmulas (3) y (4), concatenando la salida de una capa con la entrada de la siguiente, y se han obtenido los siguientes tamaños:

Tabla 6. Tamaño en muestras de entrada y salida a la parte convolucional de la red.

	Tamaño entrada [muestras]	Tamaño salida de la red convolucional [muestras]
Guitarra	512	1920
Bajo	1024	3968

Conociendo las dimensiones de salida de la cuarta capa convolucional, se pueden dimensionar las capas densas que siguen. La primera capa densa contiene obligatoriamente la misma cantidad de neuronas que muestras a su entrada, mientras que las sucesivas se han dimensionado de forma que su tamaño disminuya progresivamente hasta llegar a la última capa. Como ya se ha mencionado, la capa de salida contiene el mismo número de neuronas que casos contemplados en la clasificación. Siguiendo el principio de tratar de mantener los tamaños de las capas como potencias de dos, se han dimensionado las capas densas de la siguiente manera:

Tabla 7. Dimensiones propuestas para las capas densas en función del caso de estudio.

	Capa densa 1	Capa densa 2	Capa densa 3
Caso 1	in_features=1920 out_features=512	in_features=512 out_features=128	in_features=128 out_features=5
Caso 2	in_features=3968 out_features=512	in_features=512 out_features=128	in_features=128 out_features=5
Caso 3	in_features=3968 out_features=512	in_features=512 out_features=128	in_features=128 out_features=12

Tras codificar las tres redes empleando la librería Pytorch (ver código en el Anexo A), se ha extraído un resumen de las distintas arquitecturas, que se muestra en la figura 18:

Layer (type)	Output Shape	Param #
Conv1d-1	[-1, 32, 510]	128
ReLU-2	[-1, 32, 510]	0
MaxPool1d-3	[-1, 32, 255]	0
Conv1d-4	[-1, 32, 253]	3,104
ReLU-5	[-1, 32, 253]	0
MaxPool1d-6	[-1, 32, 126]	0
Conv1d-7	[-1, 64, 124]	6,208
ReLU-8	[-1, 64, 124]	0
MaxPool1d-9	[-1, 64, 62]	0
Conv1d-10	[-1, 64, 60]	12,352
ReLU-11	[-1, 64, 60]	0
MaxPool1d-12	[-1, 64, 30]	0
Flatten-13	[-1, 1920]	0
Linear-14	[-1, 512]	983,552
Linear-15	[-1, 128]	65,664
Linear-16	[-1, 5]	645

Layer (type)	Output Shape	Param #
Conv1d-1	[-1, 32, 1022]	128
ReLU-2	[-1, 32, 1022]	0
MaxPool1d-3	[-1, 32, 511]	0
Conv1d-4	[-1, 32, 509]	3,104
ReLU-5	[-1, 32, 509]	0
MaxPool1d-6	[-1, 32, 254]	0
Conv1d-7	[-1, 64, 252]	6,208
ReLU-8	[-1, 64, 252]	0
MaxPool1d-9	[-1, 64, 126]	0
Conv1d-10	[-1, 64, 124]	12,352
ReLU-11	[-1, 64, 124]	0
MaxPool1d-12	[-1, 64, 62]	0
Flatten-13	[-1, 3968]	0
Linear-14	[-1, 512]	2,032,128
Linear-15	[-1, 128]	65,664
Linear-16	[-1, 5]	645

Layer (type)	Output Shape	Param #
Conv1d-1	[-1, 32, 1022]	128
ReLU-2	[-1, 32, 1022]	0
MaxPool1d-3	[-1, 32, 511]	0
Conv1d-4	[-1, 32, 509]	3,104
ReLU-5	[-1, 32, 509]	0
MaxPool1d-6	[-1, 32, 254]	0
Conv1d-7	[-1, 64, 252]	6,208
ReLU-8	[-1, 64, 252]	0
MaxPool1d-9	[-1, 64, 126]	0
Conv1d-10	[-1, 64, 124]	12,352
ReLU-11	[-1, 64, 124]	0
MaxPool1d-12	[-1, 64, 62]	0
Flatten-13	[-1, 3968]	0
Linear-14	[-1, 512]	2,032,128
Linear-15	[-1, 128]	65,664
Linear-16	[-1, 12]	1,548

Figura 18. Resumen proporcionado por Pytorch de las arquitecturas propuestas.

Al compilar la red para obtener el resumen, Pytorch prueba a hacer una propagación de datos a lo largo de la red, de forma que si hay algún error en la compilación o ejecución significa que las dimensiones de alguna de las capas son erróneas. Al haberse ejecutado sin problemas todos los programas, se puede concluir que el diseño y codificación de las redes se ha realizado con éxito.

4.3 Entrenamiento de la red

Para abordar el entrenamiento, la librería Pytorch ofrece una serie de clases con las que simplificar el proceso, pudiendo obviar los cálculos internos relativos al flujo de vectores, convoluciones, propagaciones, etc. Para ello, se emplean objetos y funciones para implementar conceptos como el dataset, la función de error, funciones de activación, entre muchas otras. En este apartado se detalla el proceso de entrenar y testear la red de forma genérica a los tres casos propuestos, ya que el proceso ha sido el mismo para todos ellos.

El primer paso ha sido codificar una clase de Python que representa el dataset, heredada de la propia clase `Dataset` de Pytorch. En esta clase se definen una serie de funciones que se encargan de: cargar los datos provenientes de los archivos CSV generados en el apartado 4.1, en el caso del constructor `__init__`; entregar datos del dataset en función del índice que se indique `__getitem__` y entregar el tamaño del dataset `__len__`. Estos métodos se declaran con dos guiones bajos delante y detrás porque son métodos "*dunder*". La peculiaridad de estos métodos es que su implementación está ligada a otras operaciones del lenguaje. Por ejemplo, para acceder al valor de una posición `i` de una `matriz`, la nomenclatura empleada es: `matriz[i]`; para poder mantener esta misma nomenclatura al implementar clases que

representen estructuras de datos más complejas como el dataset, se codifica el método dunder `__getitem__`, que permite configurar ese acceso a los datos a conveniencia. En concreto, el método `__getitem__` se ha codificado de forma que entregue una dupla de datos correspondiente a la entrada de la red y la etiqueta de esa entrada.

Tras inicializar el objeto de la clase `Dataset`, se segmenta en otros dos objetos de la misma clase para formar los conjuntos de entrenamiento y de prueba. La proporción usada para esta segmentación es del 80-20%, siendo el conjunto más grande el de entrenamiento. Esto se hace para poder probar el funcionamiento de la red con un conjunto de entradas que nunca ha visto y así poder asegurar su fiabilidad. A partir de estos dos conjuntos de datos se declaran dos objetos tipo `Dataloader`: `train_dataloader` y `test_dataloader`, cuya inicialización se realiza con el objeto `Dataset` y el *batch_size* como parámetros de entrada. Se ha elegido un tamaño de *batch* de 64 entradas, por lo que cada vez que se alimente a la red se le entregará un conjunto de 64 datos de entrada.

Se ha inicializado la red ya creada en el apartado 4.2 (`model`), la función de error (`loss_fn`) y el optimizador (`optimiser`). Como ya se ha mencionado la función de error elegida es la entropía cruzada. El optimizador es una clase cuya función es aplicar la retropropagación de la red y actualizar sus pesos. Se le llama optimizador porque internamente aplica un algoritmo de optimización estocástica para el proceso de retropropagación y descenso del gradiente. El optimizador elegido se llama `Adam` y aplica el algoritmo del mismo nombre, que mejora la velocidad en el proceso de entrenamiento. Al optimizador se le ha pasado como parámetro el *learning_rate* elegido: 0.001, un valor común a la hora de entrenar redes neuronales.

Con todos los elementos necesarios declarados se ha entrenado la red por medio del proceso indicado en el diagrama de flujo de la figura 19:

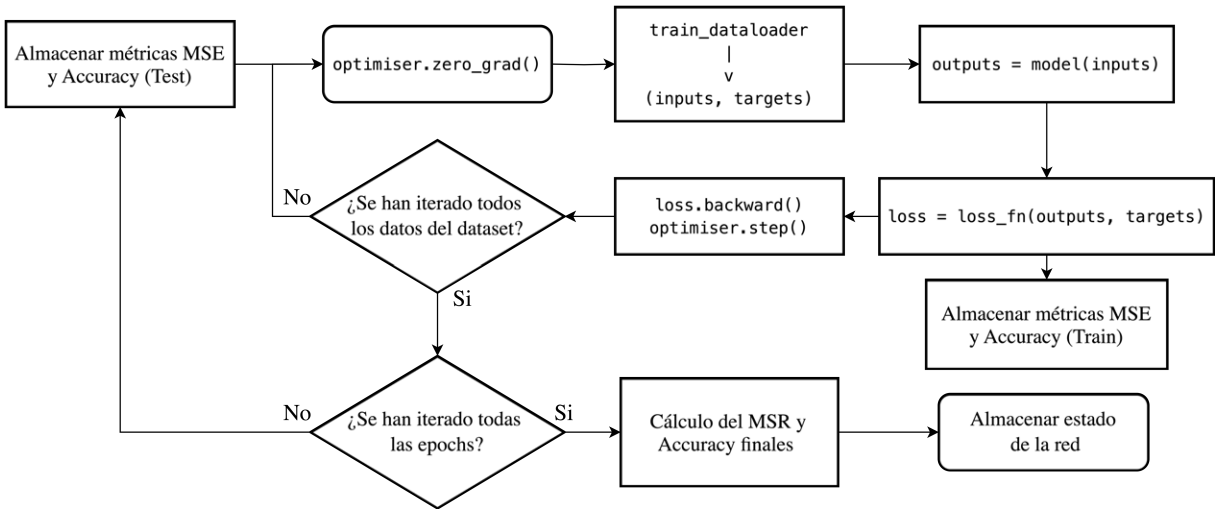

Figura 19. Diagrama de flujo del pseudocódigo del entrenamiento.

Se detalla a continuación el proceso seguido en el diagrama anterior, que corresponde al código empleado en dos bucles que iteran sobre todo el dataset de entrenamiento, un número *epochs* de veces (se ha decidido emplear un número de épocas igual a 25).

- En primer lugar, se ejecuta la función `optimiser.zero_grad()` del optimizador. Esto resetea todos los gradientes de la red a cero, para evitar sobreescrituras entre propagación y propagación.
- Se obtiene, por medio del `train_dataloader`, un *batch* de entradas y salidas (`inputs, targets`) con las que alimentar la red.
- El conjunto de entradas se introduce a la capa de entrada y se obtienen las salidas de la red por medio de la línea de código `outputs = model(inputs)`
- Con el conjunto de salidas obtenidas se calcula el error cometido por medio de la función de error `loss = loss_fn(outputs, targets)`
- Adicionalmente, se calculan las métricas para el estudio de la calidad del entrenamiento, sobre los propios datos de entrenamiento. Se almacenan tanto el error cuadrático medio como la precisión en cada iteración, para más tarde promediar sus valores para cada época.
- Por medio del método `loss.backward()` se realiza la retropropagación de errores, ya explicada en el apartado 2.2.4.
- Como último paso dentro del bucle, se actualizan los pesos y biases por medio de la función `optimiser.step()` del optimizador.
- Cada vez que se completa una época (paso de todo el dataset de entrenamiento por la red), se calculan y almacenan las métricas con el conjunto de datos de prueba.
- Al finalizar los dos bucles, se calculan por última vez las métricas con el conjunto de prueba. Estas últimas métricas son las que caracterizan la red, ya que se toman una vez se ha terminado de ajustar los pesos y biases.

Al finalizar el entrenamiento se almacena el estado de la red neuronal. Esto se realiza guardando todos los pesos y biases resultantes del entrenamiento en un archivo que se ha decidido llamar `cnn1d.pth`.

5. Resultados

El rendimiento y la precisión de una red neuronal son factores que dependen de una gran cantidad de factores: la cantidad de datos con los que se ha entrenado, el dimensionamiento de la red, que los parámetros de la red sean los idóneos o no, etc. Por ello dentro de este apartado, además de cuantificar el éxito de las redes a la hora de enfrentarse a los casos propuestos, se analiza la calidad del entrenamiento. Como se explica en el apartado anterior, se toman métricas del rendimiento de la red para el conjunto de entrenamiento y para el conjunto de prueba. En este apartado se exponen e interpretan las métricas registradas durante los entrenamientos de las redes para los tres casos de estudio, siguiendo métodos de análisis expuestos en el cuarto capítulo del documento "Supervised Machine Learning: A Survey" [22].

Las métricas empleadas para la evaluación de los entrenamientos son el error cuadrático medio (MSE) y la precisión (Accuracy), calculados por medio de las ecuaciones 7 y 8. Para poder comparar las métricas registradas para el conjunto de entrenamiento y para el conjunto de prueba se han generado gráficas que presentan ambas medidas de cada métrica. Más concretamente, se busca estudiar la evolución de las métricas a lo largo del entrenamiento, por ello se ha tomado un valor de cada métrica para cada época.

Por medio de las métricas MSE y Accuracy se ha comprobado el comportamiento de la red durante el entrenamiento, estudiando su tendencia y la relación entre los valores calculados para el conjunto de entrenamiento y el de prueba. Adicionalmente, se ha empleado la métrica Accuracy para cuantificar la precisión final del modelo. Dicha precisión se ha calculado una vez ha terminado el entrenamiento de la red, a partir del número de aciertos resultante de pasar todo el conjunto de prueba. A continuación se hace un repaso de las características de cada caso de estudio, de forma introductoria a los resultados del entrenamiento.

Se recuerda que para el Caso 1, se ha puesto a prueba una red neuronal que debe clasificar porciones de audio de 512 muestras, correspondientes a los primeros 10.6 milisegundos de la señal que produce una guitarra eléctrica tras percutir una nota. Para el experimento se ha empleado un dataset que cuenta con un total de 3319 muestras, divididas entre las cinco notas más graves de una guitarra.

En el Caso 2, se ha proporcionado a la red ventanas temporales de 1024 muestras, que corresponden a los primeros 21.3 milisegundos de la señal de audio que contiene notas tocadas por un bajo eléctrico. El dataset empleado contiene un total de 1001 muestras, contemplado únicamente las cinco notas más graves del instrumento

El último caso (Caso 3) pretende estudiar una ampliación de los casos anteriores, usando un dataset que contiene un total de 12 notas distintas tocadas en el mismo bajo que en el Caso 2. Para este caso se ha empleado el mismo tamaño de ventana de audio y se ha adaptado la red para tener 12 neuronas en la capa de salida. El tamaño del dataset es de 2400 muestras distinta.

Tras entrenar las tres redes se han obtenido las siguientes gráficas a partir de las métricas registradas (figuras 21, 22 y 23):

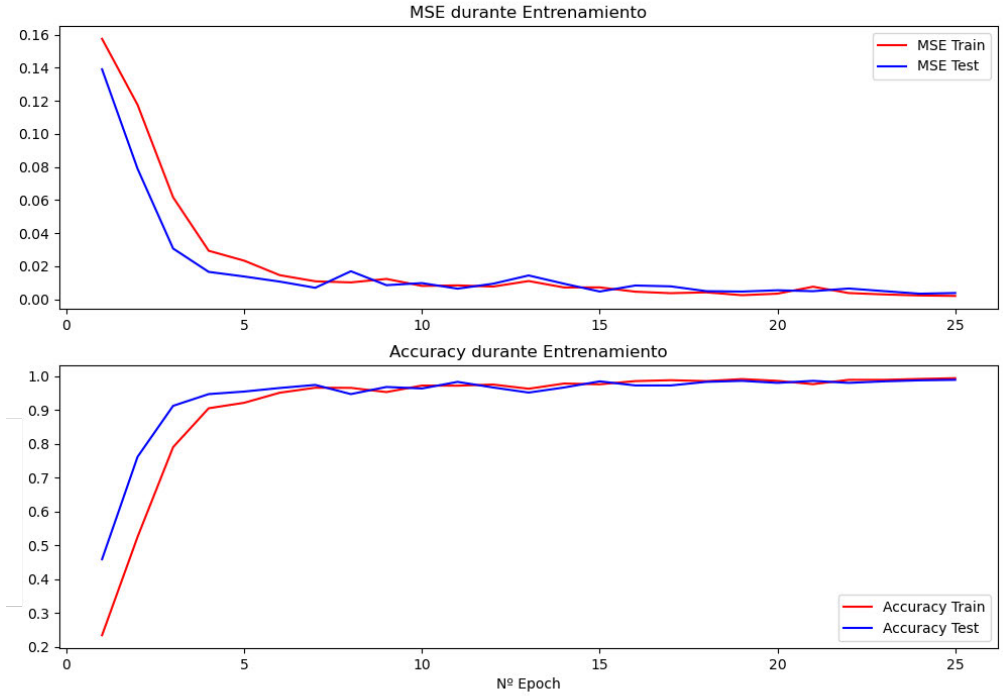

Figura 20. Evolución de las métricas durante el entrenamiento en el Caso 1.

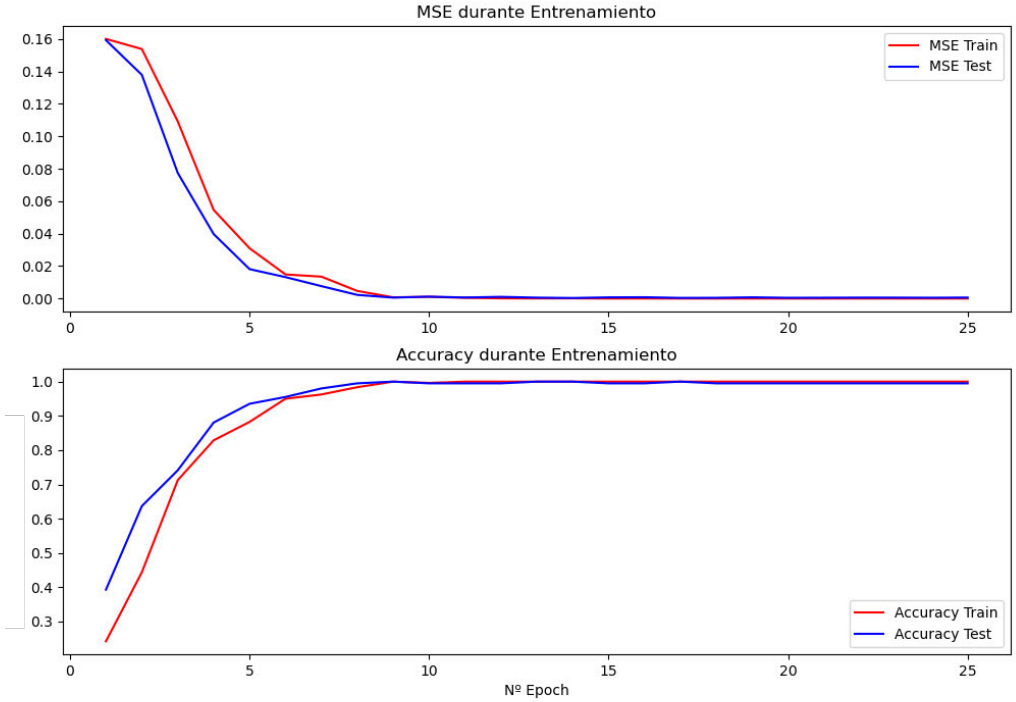

Figura 21. Evolución de las métricas durante el entrenamiento en el Caso 2.

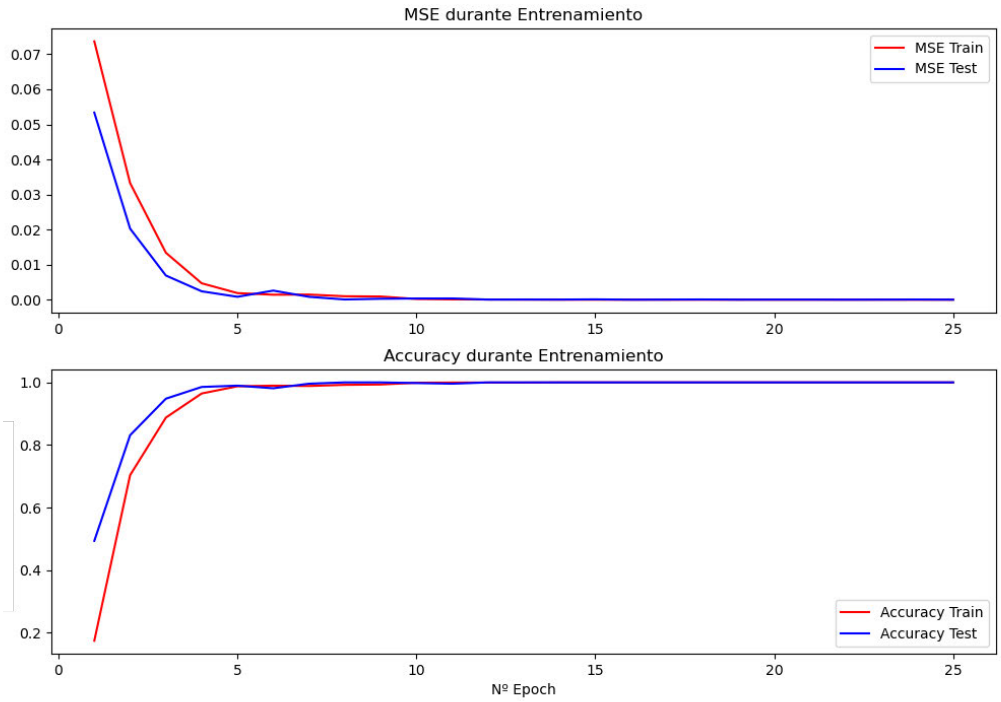

Figura 22. Evolución de las métricas durante el entrenamiento en el Caso 3.

Se observa en la evolución del error cuadrático medio conforme avanzan las épocas de entrenamiento, que su valor cae hasta estabilizarse en valores límite con cero. Dado que al comenzar el entrenamiento, los pesos internos de la red han sido asignados de forma aleatoria, el error cometido es grande, pero conforme avanza pasa las épocas su valor decrece en forma exponencial. La estabilidad y proximidad a cero de la función MSE en puntos avanzados del entrenamiento indica que las predicciones hechas por la red son prácticamente idénticas a los valores esperados. La precisión, al contrario que el MSE, crece de forma logarítmica con las épocas hasta estabilizarse en un valor próximo a uno. La similitud entre las métricas de los conjuntos de entrenamiento y de prueba, una vez se han estabilizado los valores, indica una buena capacidad de la red para generalizar y clasificar datos con los que nunca ha aprendido. Por ello se puede asegurar que no hay un sobreajuste en la red, cosa que pasaría si la precisión de la red fuese mucho mayor para el conjunto de entrenamiento que para el conjunto de prueba.

Al finalizar la ejecución de los entrenamientos se han calculado las precisiones definitivas y se han obtenido los siguientes valores:

Tabla 8. Precisión final calculada para cada red.

Test Accuracy Caso 1	Test Accuracy Caso 2	Test Accuracy Caso 3
98,85 %	99,50 %	100 %

Se ha alcanzado un porcentaje muy elevado de aciertos para los tres casos, siendo el más destacado el del Caso 3, donde se alcanza el 100 % de aciertos de forma consistente durante el entrenamiento. El porcentaje más bajo de aciertos corresponde al caso de la guitarra, con

un porcentaje de aciertos del 98,85 %. Esto se contrasta con la figura 21, donde se aprecia que para todas las métricas existe una pequeña variación. Esta variación puede deberse a la existencia de ruidos, trasteos o diferencias significativas de amplitud en algunas muestras del dataset. Aun así, el porcentaje de aciertos es muy próximo al 100 %, por lo que la red ha sido capaz de aprender y diferenciar entre la gran mayoría de las muestras utilizadas.

6. Presupuesto

Para la realización del proyecto se ha empleado equipamiento de audio de uso semi-profesional, ya que la calidad requerida no es determinante más allá de la tarjeta de sonido. Se ha incluido el conteo final de horas del proyecto de fin de grado, aproximadamente 340 horas, considerando un sueldo medio como ingeniero junior de 13€/hora. En la tabla 9 se presenta el desglose del presupuesto:

Tabla 9. Presupuesto final del proyecto.

Descripción	Precio
MacBook Pro 13 pulgadas, 2018	1.015,77 €
Focusrite Clarett 2Pre USB	404,00 €
Guitarra Maybach Stradovari S61	2099,00 €
Bajo eléctrico 1970's MIJ Luxor Greco J Bass	450,00 €
Labor de ingeniero	4420,00 €
Total	8388,77 €

7. Impacto del proyecto

El desarrollo de este proyecto se enmarca en una época de grandes avances en el campo de la inteligencia artificial. Gran parte del mérito de estos avances se debe a proyectos *Open Source*, que ayudan a esparcir conocimiento y acercarlo a la comunidad no académica. Esto es posible gracias a repositorios de código como GitHub [23] y foros como Stack Overflow [24], donde usuarios de todo el mundo comparten sus proyectos y dudas, permitiendo crear una comunidad activa de alcance mundial. Con este proyecto se pretende aportar el conocimiento y resultados adquiridos a la comunidad investigadora *online*, por ello se ha decidido subir el código fuente producto de este proyecto a un repositorio público de GitHub. De esta forma, se espera que el trabajo realizado sirva de ayuda para futuras investigaciones relacionadas.

Un conversor de audio a MIDI aplicado a instrumentos tradicionales presenta cualidades para ser una gran herramienta educativa para estudiantes de música. Este tipo de tecnología permite a los estudiantes visualizar en tiempo real las notas que están tocando, facilitando el proceso de aprendizaje y corrección. Además, contar con información MIDI extraída con muy baja latencia permite crear aplicaciones didácticas donde, por ejemplo, se pueda aprender y tocar canciones sobre una tablatura que reacciona en tiempo real a lo que el usuario esté tocando. Aunque ya existen aplicaciones así, una mejora en el tiempo de reacción de la estimación de tono permitiría mejorar las prestaciones de la aplicación, pudiendo indicar fallos en el ritmo con gran precisión.

Este proyecto puede contribuir a los Objetivos del Desarrollo Sostenible:

- Educación de calidad (4): con aplicaciones didácticas para las enseñanzas musicales.
- Industria, Innovación e Infraestructura (9): con el desarrollo realizado dentro de la industria del aprendizaje automático.

8. Conclusiones

En este proyecto se ha analizado la posibilidad de mejorar la latencia de los algoritmos de reconocimiento de tono por medio de un enfoque novedoso: el análisis de transitorios iniciales por medio de inteligencia artificial. Para ello se ha diseñado, codificado y entrenado una arquitectura de red neuronal profunda, basada en redes convolucionales unidimensionales y redes densas. Adicionalmente, se ha planteado el estudio de la solución para tres casos distintos, diferenciando entre ellos el instrumento empleado y la cantidad de notas a clasificar. El proceso de desarrollo ha acarreado retos relacionados con el procesado de audio, la grabación y creación del dataset, resolución de errores de código relativos al flujo de datos interno de las redes, entre otros.

Durante el desarrollo del proyecto, la investigación ha tenido un gran peso, debido a que la utilización de redes neuronales convolucionales 1D para la clasificación de audio está muy poco extendida. Los resultados y conocimientos obtenidos con este proyecto pueden emplearse para investigaciones futuras relacionadas.

De acuerdo con los resultados expuestos en el apartado 5, la red neuronal propuesta para la clasificación de transitorios es capaz de cumplir con el objetivo para el que fue diseñada de forma sobresaliente en los tres casos de estudio:

- Para el Caso 1, la red ha sido capaz de diferenciar las cinco notas más graves de una guitarra eléctrica, con una precisión del 98.85% de aciertos. Durante el entrenamiento, la red ha presentado una mayor dificultad para aprender que con los otros dos casos, pero aun así la función de error presenta unos valores correctos.
- Para el Caso 2, la red ha sido capaz de diferenciar las cinco notas más graves de un bajo eléctrico con una precisión del 99.50% de aciertos. El aprendizaje de la red para este caso presenta una tendencia exponencial sin cambios abruptos, tal y como se esperaba.
- Para el Caso 3, la red ha sido capaz de diferenciar entre doce notas repartidas por las cuatro cuerdas de un bajo eléctrico, con una precisión del 100% de aciertos. El aprendizaje de la red para este caso presenta un comportamiento excepcional, reduciendo el error a 0 de forma consistente tras su entrenamiento.

Aun así, existen ciertos matices que comprometen la eficacia del método propuesto y se exponen en el siguiente subapartado.

Dado que en este proyecto se ha limitado el trabajo a plantear los fundamentos de una solución muy compleja, no es posible medir la latencia total del resultado final: no se ha desarrollado una implementación solida en tiempo real. De todos modos, se estima que la latencia final no alcanzaría el objetivo de superar otras soluciones debido al propio lenguaje de programación, Python. La elección de Python como lenguaje de programación sobre el que desarrollar el proyecto viene dada por las funcionalidades de alto nivel que proporciona y han sido de gran ayuda a la hora de desarrollar las redes neuronales. Esto trae el inconveniente de

la eficiencia, ya que a diferencia de lenguajes como C++ (la opción más extendida para procesado de audio en tiempo real), Python no está tan optimizado para consumo de memoria y velocidad de escritura. Por estas razones, el proyecto deja abierta la posibilidad de ampliar la investigación implementando el mismo sistema propuesto, en entornos más veloces y optimizados. Todo el código fuente desarrollado en este proyecto se encuentra en el repositorio de *GitHub* [25].

8.1 Observaciones teóricas

La cuestión principal que cimienta el trabajo es si una red neuronal es capaz de diferenciar notas musicales conociendo tan solo los primeros milisegundos de estas. Tras realizar numerosas pruebas se ha determinado que sí, pero el tamaño de la ventana temporal es muy influyente en la precisión de la red. Se ha observado que para que la red neuronal sea capaz de determinar la nota correctamente, la extensión de la ventana temporal empleada debe abarcar como mínimo, además de la pulsación inicial, parte de la primera oscilación de la nota más grave, como se observa en la Figura 10. Dado que no es posible interpretar la lógica adquirida por la red neuronal tras su entrenamiento (más allá de los resultados obtenidos), no se puede asegurar con certeza qué característica es determinante para la detección de tono. Aun así, es muy probable que el factor "en el que se fija la red neuronal" sea la distancia temporal entre los flancos incluidos en la ventana temporal. De esta forma, la red estaría midiendo el período cero de la oscilación de la nota, entendiendo como período cero el tiempo transcurrido entre la pulsación y la primera oscilación. Este período cero tiene la característica de no ser exactamente igual de largo que el período de la propia nota en su estado estacionario debido a la inestabilidad armónica presente en la oscilación de la cuerda durante los primeros instantes tras su percusión. Si el método elegido para la detección de tono fuera otro esto sería un gran inconveniente, pero con la metodología empleada en este proyecto los resultados son muy favorables.

8.2 Trabajos futuros

8.2.1 Implementación en tiempo real

Una vez entrenada la red, es posible almacenar el estado de sus pesos para exportarla y poder implementarla en otras aplicaciones. Por el propio planteamiento del proyecto, la red está entrenada para detectar el evento transitorio de las notas, no el resto de la señal. Por ello, es necesario implementar un algoritmo de detección de *onset* para determinar el instante en el que comienza la nota, y así poder segmentar la ventana temporal deseada.

8.2.2 Escalabilidad polifónica

Con el sistema propuesto, la detección de tono se limita al audio monofónico y no se puede asegurar su funcionamiento de forma polifónica. Una opción para escalar el proyecto a una funcionalidad polifónica sería emplear una pastilla hexafónica, de forma similar al modelo Axon AX100 [6]. De esta forma se emplearía una red neuronal por cada cuerda, que estarían

funcionando simultáneamente. Otras líneas de investigación sin el uso de pastillas polifónicas requerirían de una propuesta más compleja para poder abordar el problema.

8.2.3 Reducción de la latencia

Debido a la latencia introducida por Python, implementar la red neuronal en tiempo real en este mismo lenguaje no sería óptimo. Por ello se propone que, de cara a futuras investigaciones se implemente el sistema en el lenguaje C++, que proporciona un mejor rendimiento para flujos de datos en tiempo real.

8.2.4 Estimación de la intensidad

Para formar los mensajes MIDI resultantes de la conversión es necesario determinar el parámetro *velocity* que caracteriza la intensidad con la que ha sido la nota. Este parámetro se podría estimar por medio de medición de energía de la señal, o con una red neuronal adicional, expresamente entrenada para ello.

9. Referencias

[1] G. Loy, «Musicians Make a Standard: The MIDI Phenomenon», *Comput. Music J.*, vol. 9, n.º 4, p. 8, 1985, doi: 10.2307/3679619.

[2] A. Klapuri y M. Davy, Eds., *Signal processing methods for music transcription*. New York: Springer, 2006.

[3] L. Michael y B. Jon, «The Effects of Latency on Live Sound Monitoring», *J. Audio Eng. Soc.*, n.º 7198, oct. 2007.

[4] P. Singh, «An Approach to Extract Feature using MFCC», *IOSR J. Eng.*, vol. 4, pp. 21-25, ago. 2014, doi: 10.9790/3021-04812125.

[5] «Jam Origin – Audio to MIDI». Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.jamorigin.com/

[6] «Blue Chip Axon AX100». Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.soundonsound.com/reviews/blue-chip-axon-ax100

[7] C. Duxbury, J. P. Bello, M. Davies, y M. Sandler, «COMPLEX DOMAIN ONSET DETECTION FOR MUSICAL SIGNALS», 2003.

[8] I. J. Goodfellow, Y. Bengio, y A. Courville, *Deep Learning*. MIT Press, 2016.

[9] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[10] O. GMBH, «Extractor de vocal y separador de instrumental IA | LALAL.AI». Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.lalal.ai/es/

[11] A. Baevski, Y. Zhou, A. Mohamed, y M. Auli, «wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations», en *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2020, pp. 12449-12460. Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://proceedings.neurips.cc/paper/2020/hash/92d1e1eb1cd6f9fba3227870bb6d7f07-Abstract.html

[12] C. Choy, J. Gwak, y S. Savarese, «4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks». arXiv, 13 de junio de 2019. doi: 10.48550/arXiv.1904.08755.

[13] W. Fang, L. Fu, W. Xu, A. Bian, y H. Li, «CCNet-5D: 5D convolutional neural network for seismic data interpolation», *GEOPHYSICS*, vol. 88, n.º 4, pp. V333-V344, jul. 2023, doi: 10.1190/geo2022-0420.1.

[14] «Calculate the Output Size of a Convolutional Layer | Baeldung on Computer Science». Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.baeldung.com/cs/convolutional-layer-size

[15] «Figure 1: An example of MIDI message.», ResearchGate. Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.researchgate.net/figure/An-example-of-MIDI-message_fig1_343709022

[16] K. He y J. Sun, «Convolutional Neural Networks at Constrained Time Cost». arXiv, 4 de diciembre de 2014. doi: 10.48550/arXiv.1412.1710.

[17] «Logic Pro for Mac», Apple. Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.apple.com/logic-pro/

[18] X. Ying, «An Overview of Overfitting and its Solutions», *J. Phys. Conf. Ser.*, vol. 1168, p. 022022, feb. 2019, doi: 10.1088/1742-6596/1168/2/022022.

[19] Y. LeCun, «Generalization and network design strategies», *Connect. Perspect.*, jun. 1989, Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://www.academia.edu/2813343/Generalization_and_network_design_strategies

[20] C. Raffel y D. P. W. Ellis, «INTUITIVE ANALYSIS, CREATION AND MANIPULATION OF MIDI DATA WITH pretty_midi».

[21] L. Lu, C. Zhang, K. Cao, T. Deng, y Q. Yang, «A Multichannel CNN-GRU Model for Human Activity Recognition», *IEEE Access*, vol. 10, pp. 66797-66810, 2022, doi: 10.1109/ACCESS.2022.3185112.

[22] M. A. El Mrabet, K. El Makkaoui, y A. Faize, «Supervised Machine Learning: A Survey», en *2021 4th International Conference on Advanced Communication Technologies and Networking (CommNet)*, dic. 2021, pp. 1-10. doi: 10.1109/CommNet52204.2021.9641998.

[23] «Explore GitHub», GitHub. Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://github.com/explore

[24] «Stack Overflow - Where Developers Learn, Share, & Build Careers», Stack Overflow. Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://stackoverflow.com/

[25] nikouriz, «nikouriz/TransientClassificationWithCNN1D». 3 de julio de 2024. Accedido: 3 de julio de 2024. [En línea]. Disponible en: https://github.com/nikouriz/TransientClassificationWithCNN1D

[26] «PyTorch documentation — PyTorch 2.3 documentation». Accedido: 2 de julio de 2024. [En línea]. Disponible en: https://pytorch.org/docs/stable/index.html

Anexo A: Código para la creación de la red neuronal

El siguiente código pertenece al fichero `cnn1d.py`:

```
class CNN(nn.Module):  # Se hereda la clase Module de torch.nn

    def __init__(self):
        super().__init__()  # se mantiene el constructor original
        self.conv1 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=32, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=32, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2),
        )
        self.conv3 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2),
        )
        self.conv4 = nn.Sequential(
            nn.Conv1d(in_channels=64, out_channels=64, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2),
        )
        self.flat = nn.Flatten()
        self.fc1 = nn.Linear(in_features=1920, out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=128)
        self.fc3 = nn.Linear(in_features=128, out_features=5)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, input):
        x = self.conv1(input)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.flat(x)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        return x  # Sin Softmax, para poder calcular MSE durante
entrenamiento
```

Anexo B: Manual de usuario

En este anexo se explica el procedimiento que se debe seguir para recrear la ejecución de la generación del dataset y del entrenamiento de la red neuronal, empleando el material relativo al Caso 1 propuesto a modo de ejemplo.

B.1 Instalación de dependencias

Para facilitar la instalación de librerías para el proyecto, se ha incluido el archivo `requirements.txt`. En este archivo de texto figuran todas las librerías empleadas en el código, junto a la versión de cada una de ellas. Para instalar todas las librerías se debe emplear el siguiente comando en el terminal:

```
> pip install -r /dirección/de/requirements.txt
```

Se debe el texto en cursiva por la dirección al archivo.

B.2 Preprocesado de datos

Para generar el archivo CSV para su posterior uso en el dataset, se ha codificado el programa: `DataPreprocessing.py`. Tras abrirlo con un entorno de edición de código, se observa que en la declaración de las variables constantes figuran las siguientes direcciones de carpetas:

```
MIDI_FOLDER = "dataset/DATASET_STRATO/dataCaso1/midi"
AUDIO_FOLDER = "dataset/DATASET_STRATO/dataCaso1/audio"
OUTPUT_FOLDER = "dataset/DATASET_STRATO/output"
```

Figura 23. Código para la declaración de las direcciones a las carpetas de datos.

En estas carpetas es donde el programa buscará los archivos *.mid* y *.wav* a partir de los cuales generar la estructura de datos que será almacenada en la carpeta indicada en OUTPUT_FOLDER como un archivo *.csv*. Las direcciones a las carpetas pueden ser absolutas o relativas. Tanto los archivos MIDI como los archivos WAV de entrada deben seguir las especificaciones indicadas en el apartado 4.1. Una vez declaradas las variables, se ejecuta el programa mediante el intérprete de código o mediante el siguiente comando (se debe navegar hasta la carpeta que contiene el código desde el intérprete de comandos):

```
> python3 -m DataPreprocessing.py
```

Una vez obtenido el archivo CSV, se puede comprobar su almacenamiento mediante el programa `DataChecking.py`, donde, al igual que en el procedimiento anterior se debe indicar la dirección del archivo de entrada en la constante correspondiente:

```
CSV_PATH = "dataset/DATASET_STRATO/output/StratoDatset.csv"
```

Figura 24. Código para la declaración de la dirección al archivo CSV deseado.

Tras ejecutar este programa, se genera una gráfica que contiene una muestra de cada etiqueta contenida en el archivo CSV, elegida aleatoriamente. A modo de ejemplo se ha ejecutado el programa y se ha obtenido la siguiente gráfica (Figura 25):

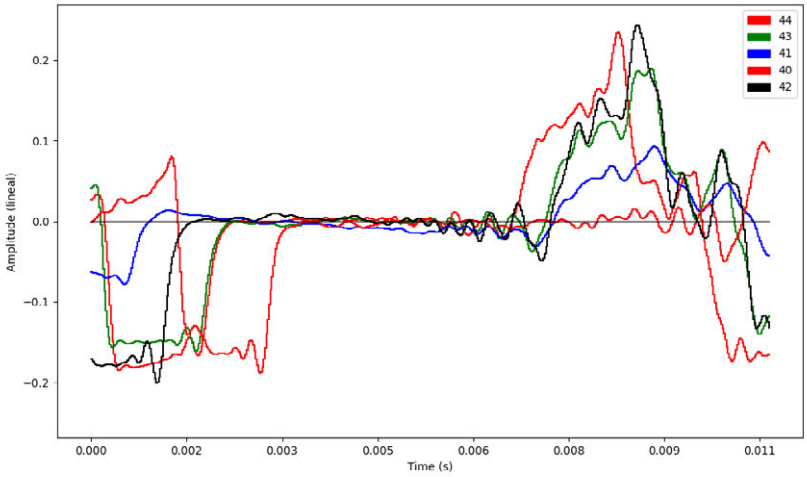

Figura 25. Gráfica resultante tras ejecutar DataChecking.py

B.3 Entrenamiento de la red

Para el entrenamiento de la red resultan necesarios tres archivos de código `cnn1d.py`, `stratodatset.py` y `train.py`. Los dos primeros son los encargados de definir los objetos relativos a la red neuronal y al dataset respectivamente, se importan desde el programa `train.py`, por lo que no deben ser ejecutados. No es necesario editar ninguno de estos dos archivos, a no ser que se desee cambiar algún parámetro de la red o del dataset. El último es el encargado de definir el proceso de entrenamiento y toma de métricas. En él se deben especificar los siguientes parámetros en forma de variables constantes (Figura 26):

```
BATCH_SIZE = 64
EPOCHS = 25
LEARNING_RATE = 0.001
NUM_LABELS = 5
CSV_PATH = "network/stratoConv1d/datasets/StratoDatset.csv"
OUTPUT_PATH = "network/stratoConv1d/output"
```

Figura 26. Código para la declaración de las constantes del programa train.py

Como se observa se deben declarar las variables relativas a las direcciones de entrada (donde se encuentra el archivo CSV) y salida (donde el programa guardará la configuración de la red tras su entrenamiento). El resto de las variables son relativas a los parámetros del entrenamiento de la red. Pueden ser ajustados, pero los presentes en la Figura 26 son los empleados en el Caso 1. Tras ajustar las variables anteriores, se ejecuta el programa de forma

análoga al subapartado anterior. Por el terminal de comandos se puede observar la evolución del entrenamiento en tiempo real, como se muestra en la Figura 27:

```
--------------------------
Epoch 2
loss: 0.9005
accuracy: 68.52 %
--------------------------
Epoch 3
loss: 0.4625
accuracy: 85.54 %
--------------------------
Epoch 4
loss: 0.7540
accuracy: 90.06 %
--------------------------
Epoch 5
loss: 0.2346
accuracy: 93.83 %
--------------------------
Epoch 6
```

Figura 27. Ejemplo de terminal durante la ejecución de train.py

Al concluir la ejecución, el estado de la red neuronal se almacena en un archivo tipo *.pth*, que puede ser importado desde otro programa por medio de Pytorch (ver documentación de Pytorch [26]).