UNTREF UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

INGENIERÍA DE SONIDO

Extracción de cromagramas mediante redes neuronales para el reconocimiento automático de acordes musicales †

Autor: Alejandro Suárez

Tutor: Ing. Leonardo Pepino

(†) Tesis para optar por el título de Ingeniero de Sonido

24 de abril de 2021

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

En este trabajo se explora el estudio de la identificación automática de acordes musicales. A este fin, se presenta el diseño e implementación de un sistema que facilita la extracción automática de cromagramas, combinando técnicas de procesamiento digital de señales junto con modelos de redes neuronales. El mismo presenta dos innovaciones respecto de otros desarrollos similares: la aplicación de entrenamiento *por currículum* para el aprendizaje del modelo, y la utilización de datos generados artificialmente como conjunto de datos de entrenamiento del mismo. Este sistema es evaluado con respecto a técnicas conformes al estado del arte, utilizando métricas reconocidas. Esta evaluación refleja que tanto el entrenamiento por currículum como la utilización de datos artificiales fueron efectivos, dando como resultado un modelo aplicable a información musical no sintética.

Palabras clave: “Detección automática de acordes”; “Redes neuronales”; “Cromagramas”;

ABSTRACT

In this work, the field of automatic chord estimation (ACE) is explored. To this end, this work presents the design and implementation of a system which allows for the automatic extraction of chromagrams, combining digital signal processing techniques as well as neural network models. In relation to previous similar works, two innovative techniques are utilized: the implementation of a *curriculum learning* scheme for training the model, and the use of artificially generated training data. This system is then evaluated against state-of-the-art techniques, using recognized metrics. This evaluation shows that both previously mentioned techniques were effective in the creation of a model that can be applied to non-synthetic musical information.

Keywords: “Automatic Chord Estimation”; “Neural Networks”; “Chromagrams”;

CAPÍTULO 1: INTRODUCCIÓN

1.1 FUNDAMENTACIÓN

Los acordes musicales son estructuras conformadas por varias notas que suenan al mismo tiempo y que describen de manera concisa el contenido armónico de una pieza. En muchos casos, es incluso común que el conocimiento de la secuencia de acordes que componen una pieza musical, sea suficiente como para que músicos puedan tocar en conjunto de manera improvisada [1].

En el ámbito científico, secuencias de acordes son de importancia en relación a varios tópicos, tales como:

- Identificación de diferentes versiones de la misma canción (*identificación de covers*) [2][3].
- Detección de la tonalidad en audio musical [4]-[7].
- Clasificación de canciones según su género musical [8].
- Alineación de audio con contenido lírico [9].

Transcribir manualmente secuencias de acordes es un proceso que consume mucho tiempo y recursos: normalmente requiere de dos o más expertos y un tiempo promedio de transcripción de 8 a 18 minutos por transcriptor por canción [10]. A su vez, se requiere que las personas que transcriben posean un entrenamiento musical suficiente. Por estas razones, la transcripción automática de acordes es un área de investigación importante dentro del campo de la recuperación de información musical (MIR), que puede generar un impacto positivo en la comunidad de músicos.

1.2 OBJETIVOS

1.2.1 Objetivo general

El objetivo de este trabajo es el diseño, la implementación y la evaluación de un sistema que realice el proceso de extracción de cromagramas, para facilitar la transcripción automática de acordes musicales, utilizando técnicas de aprendizaje automático.

1.2.2 Objetivos específicos

- Ahondar en la bibliografía existente respecto de las técnicas y el estado del arte dentro del área de ACE (*Automatic Chord Estimation*).
- Hacer un relevamiento de los conjuntos de datos existentes, que serán utilizados como referencia, tanto para el desarrollo del sistema como para su posterior evaluación.
- Diagramar los esquemas de *pre* y *post* procesamiento de datos posibles, e implementarlos en el lenguaje de programación Python.
- Implementar un modelo de redes neuronales, también utilizando Python, que extraiga cromagramas a partir de representaciones espectrales de audio.
- Ajustar las etapas de procesamiento y el modelo para optimizar los resultados obtenidos.
- Evaluar objetivamente el desempeño de los modelos entrenados utilizando métricas estandarizadas, en una tarea de reconocimiento automático de acordes, comparándolos contra una referencia establecida.

1.3 ESTRUCTURA DE LA INVESTIGACIÓN

En el capítulo 2 de este trabajo se presenta el marco teórico que sirve como punto de partida para el desarrollo posterior. Esta sección pone énfasis en las cuatro aristas principales de este trabajo: la teoría detrás de los acordes musicales, la representación de señales de audio en el dominio espectral (Transformada de Fourier de Tiempo Corto [STFT] y Transformada de Q Constante [CQT]), los modelos de redes neuronales artificiales, y las técnicas tradicionales utilizadas en sistemas de ACE.

En el capítulo 3 se detalla el estado del arte en lo que respecta a técnicas de aprendizaje automático aplicadas a tareas de este tipo. En el capítulo 4 se hace un relevamiento exhaustivo de la metodología utilizada en el presente trabajo. En esta sección se detalla el modelo implementado, su entrenamiento y posterior evaluación. En el capítulo 5 se presentan los resultados obtenidos, mientras que en el 6 se analizan los mismos. Finalmente, el capítulo 7 se centra en las conclusiones extraídas a partir del trabajo realizado, y el capítulo 8 propone futuras líneas de investigación.

CAPÍTULO 2: MARCO TEÓRICO

2.1 NOTAS, ESCALAS E INTERVALOS MUSICALES

Una nota musical es un elemento que representa a un sonido musical. Las notas pueden representar altura y duración de un sonido utilizando la denominada notación musical. Las notas musicales son los bloques fundacionales de gran parte de la música escrita; son discretizaciones de fenómenos musicales que facilitan la ejecución, la comprensión, y el análisis [11].

En la música occidental existen siete notas musicales denominadas *naturales*. Tradicionalmente, estas se definen utilizando nomenclatura de *solfeo*, siendo las mismas Do-Re-Mi-Fa-Sol-La-Si. No obstante, en países anglosajones, las notas se representan utilizando las primeras siete letras del alfabeto, en el siguiente orden: C-D-E-F-G-A-B. En este trabajo, con el fin de mantener la consistencia con respecto al estado del arte, se utilizará esta última nomenclatura.

Dos notas con frecuencias fundamentales de una relación igual a cualquier potencia entera de 2 son percibidas por el oído humano de manera muy similar. Basado en esto, todas las notas con relaciones de este tipo se agrupan bajo el mismo nombre. Por ejemplo, en el caso de dos sonidos con frecuencias fundamentales de 440 Hz y 880 Hz, ambos se corresponden con la nota A (o La), y se dice que el intervalo que conforman es una *octava*.

Cada octava se divide en las 12 notas que conforman el alfabeto musical occidental. Estas serían las 7 notas naturales, junto con sus denominadas *accidentales*: alteraciones que hacen referencia a una nota con una frecuencia fundamental mayor (*sostenido*, denotado con *#*) o menor (*bemol*, denotado con *b*) respecto de la nota natural. De esta forma, las 12 notas resultantes son: C-C#/Db-D-D#/Eb-E-F-F#/Gb-G-G#/Ab-A-A#/Bb-B (nótese que, en el caso de las alteraciones, las mismas pueden escribirse como un sostenido o un bemol en función del contexto, dado que ambas tienen la misma altura).

En el sistema para la notación musical occidental las notas se ubican en un *pentagrama*, el cual está formado por cinco líneas horizontales y cuatro espacios o interlíneas equidistantes que se enumeran de abajo hacia arriba. Un ejemplo de esto se puede ver en la Figura 2.1.

A partir de estas 12 notas se construyen las llamadas *escalas musicales*: conjuntos de notas ordenadas, de forma ascendente (grave a aguda), o descendente (aguda a grave), una a una en posiciones específicas, llamadas *grados*. Las escalas musicales más utilizadas son las escalas *mayores* y *menores*, y se construyen partiendo de una nota dada y agregando

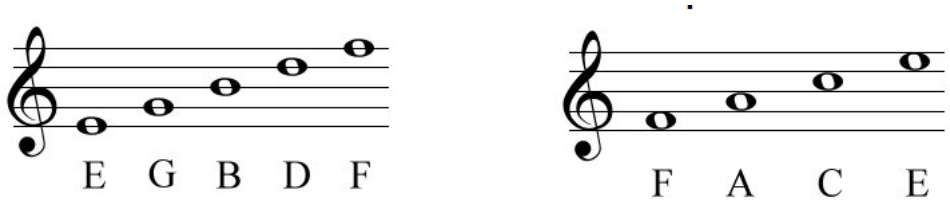

Figura 2.1: Notas musicales dispuestas en un pentagrama, junto con sus nombres.

notas de acuerdo a una secuencia específica.

Por otro lado, se dice que la diferencia entre la altura de dos notas conforma un *intervalo* musical. Esta diferencia o distancia se mide en grados (de acuerdo a la posición de cada nota dentro de la escala) o en tonos y semitonos de separación. Por ejemplo, si se toma la escala mayor de C y la nota C como referencia, se puede decir que la nota G perteneciente a la misma octava forma un intervalo con una distancia de 3 tonos y medio respecto de C o, de manera equivalente, que define un intervalo de *quinta justa*. La tabla 2.1 muestra todas las notas que conforman el alfabeto musical occidental, junto con los intervalos más comunes que las mismas definen en relación a la escala mayor de C, expresados en términos de distancia y denominación.

Tabla 2.1: Intervalos simples más comunes en relación a la escala mayor de C.

Nota	Distancia	Denominación
C	0 semitonos	Unísono
C#	1 semitono	Segunda menor
D	1 tono	Segunda mayor
D#	1 tono y 1 semitono	Tercera menor
E	2 tonos	Tercera mayor
F	2 tonos y un semitono	Cuarta justa
F#	3 tonos	Cuarta aumentada/Quinta disminuida
G	3 tonos y un semitono	Quinta justa
G#	4 tonos	Sexta menor
A	4 tonos y un semitono	Sexta mayor
A#	5 tonos	Séptima menor
B	5 tonos y un semitono	Séptima mayor
C	6 tonos	Octava justa

Estos intervalos se denominan *simples* ya que no son mayores a una octava. Es posible que un intervalo sea mayor a una octava; en ese caso, se dirá que es un intervalo *compuesto*.

2.2 ACORDES MUSICALES

El Diccionario de la Música de Oxford da la siguiente definición [12]:

> *Acorde: Cualquier combinación simultánea de notas, pero usualmente no menos de tres. El uso de acordes es la fundación básica de la armonía.*

En este trabajo se extenderá esta definición para permitir que una o más notas formen un acorde. En consecuencia, notas singulares que son interpretadas a través de arpegios o los denominados acordes *rotos*, en donde las notas son tocadas una tras otra y no de manera simultánea [13], [14], también se considerarán acordes, siendo que se considera que los acordes forman la estructura básica de una pieza, independientemente de si todas sus notas suenan de manera simultánea en un momento dado. Asimismo, se excluyen de esta definición sonidos de carácter no tonal, silencios, y sonidos puramente percusivos.

El uso de acordes y secuencias de acordes (denominadas *progresiones*) prevalecen en la música occidental, siendo una característica representativa de la misma [15]. Cabe aclarar que el foco de este estudio es la música occidental contemporánea, por lo cual el contenido de este marco teórico es válido principalmente dentro de este contexto.

2.2.1 Tríadas

Comúnmente el tipo de acorde más común es la *tríada*. La misma consiste en un conjunto de tres notas que tienen una relación específica entre sí: la *tónica*, la *tercera*, y la *quinta*, siendo que las partes de una tríada obtienen su nombre de su intervalo relativo a la tónica.

Las tríadas más comunes son las llamadas tríadas *mayores*, *menores*, *disminuidas*, y *aumentadas*.

- Una tríada mayor está formada por la tónica, junto con otras dos notas. La primera con un intervalo de tercera mayor, y la segunda con un intervalo de quinta justa (siempre en relación a la tónica).
- Una tríada menor está formada por la tónica, junto con otras dos notas. La primera con un intervalo de tercera menor, y la segunda con un intervalo de quinta justa.
- Una tríada disminuida está formada por la tónica, junto con otras dos notas. La primera con un intervalo de tercera menor, y la segunda con un intervalo de quinta disminuida.
- Una tríada aumentada está formada por la tónica, junto con otras dos notas. La primera con un intervalo de tercera mayor, y la segunda con un intervalo de sexta aumentada.

Estas tríadas se denominan *diatónicas*, ya que están formadas de acuerdo a las notas de una escala, y se construyen partiendo de la tónica y agregando notas cada dos 'pasos' (es decir, en intervalos de terceras).

En la figura 2.2 se pueden observar las tríadas correspondientes a cada grado de la escala mayor de C (que a su vez está compuesta por las notas C, D, E, F, G, A, y B). Las tríadas de este tipo podrán ser *mayores* (grados I, IV, y V), *menores* (grados II, III, y VI), o *disminuidas* (grado VII).

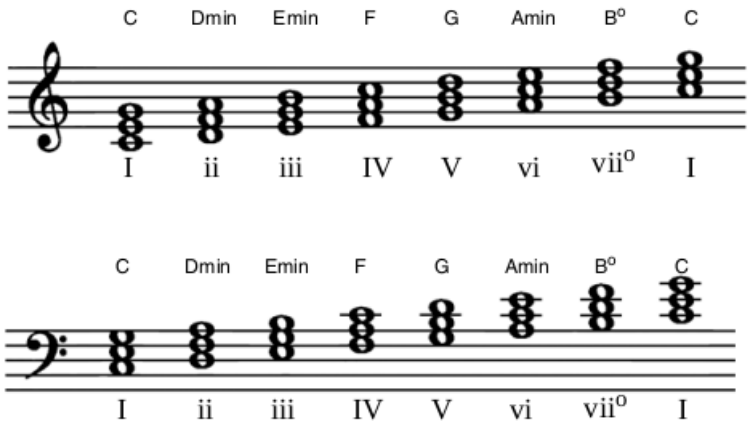

Figura 2.2: Tríadas, en varias posiciones del pentagrama, a partir de los grados de la escala mayor de C. Extraído y adaptado de [16].

Otro tipo de tríadas son los denominados acordes suspendidos, en donde el tercer grado es reemplazado por el segundo o el cuarto grado de la escala.

2.2.2 Inversiones y acordes extendidos

La tónica del acorde no necesariamente tiene que ser la nota más grave. En efecto, un acorde puede tener distintas formas, dependiendo de cuál de las notas del acorde sea la nota más grave (o el *bajo*). La nota más grave del acorde define la posición del mismo, o su *inversión*. La Figura 2.3 muestra las inversiones posibles de una tríada de C mayor.

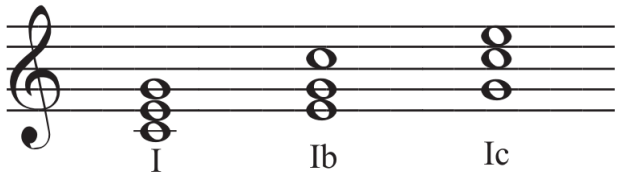

Figura 2.3: Posibles inversiones de una tríada de C mayor. De izquierda a derecha, posición de tónica (I, bajo en C), primera inversión (Ib, bajo en E), y segunda inversión (Ic, bajo en G).

En cuanto a la notación, es común que el acorde se especifique con su nombre acompañado con la nota a utilizar de bajo (por ejemplo, la primera inversión de C puede escribirse como C/E)

Los acordes introducidos hasta ahora han sido tríadas que, por definición, están formados por sólo tres notas. Otras combinaciones más complejas pueden formarse agregando más notas, *extendiendo* de esta forma los acordes. El tipo más común de acorde extendido se produce agregando un intervalo de séptima (respecto de la tónica), aunque también pueden agregarse los intervalos 9, 11, y 13, que se encuentran por fuera de la octava original del acorde.

La tabla 2.2 muestra varios tipos de acordes comunes, junto con su nomenclatura abreviada y los intervalos que los componen.

Tabla 2.2: Acordes comunes con abreviaturas e intervalos constituyentes.

Tipo de acorde		**Abreviatura**	**Listado de intervalos**
Tríadas	Mayor	*maj*	(1, 3, 5)
	Menor	*min*	(1, b3, 5)
	Disminuida	*dim*	(1, b3, b5)
	Aumentada	*aug*	(1, b3, #5)
Acordes de séptima	Séptima Mayor	*maj7*	(1, 3, 5, 7)
	Séptima menor	*min7*	(1, b3, 5, b7)
	Séptima	*7*	(1, 3, 5, b7)
	Séptima disminuida	*dim7*	(1, b3, b5, bb7)
	Séptima semi-disminuida	*hdim7*	(1, b3, b5, b7)
	Menor séptima mayor	*minmaj7*	(1, b3, 5, 7)
Acordes de sexta	Sexta mayor	*maj6*	(1, 3, 5, 6)
	Sexta menor	*min6*	(1, b3, 5, 6)
Acordes extendidos	Novena	*9*	(1, 3, 5, b7, 9)
	Novena mayor	*maj9*	(1, 3, 5, 7, 9)
	Novena menor	*min9*	(1, b3, 5, b7, 9)
Acordes suspendidos	Segunda suspendida	*sus2*	(1, 2, 5)
	Cuarta suspendida	*sus4*	(1, 4, 5)

2.3 REPRESENTACIÓN DE SEÑALES SONORAS Y MUSICALES EN EL DOMINIO ESPECTRAL

2.3.1 Transformada de Fourier Discreta (DFT)

Dada una señal $x[n]$ perteneciente a $\mathbb{C}^N$, su transformada discreta de Fourier (DFT) viene dada por:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi kn}{N}} \qquad k = 0, 1, 2, ..., N-1 \quad (2.1)$$

En donde X es la transformada de Fourier discreta de x, y cada muestra $X[k]$ es el resultado del producto interno entre x y una exponencial compleja de frecuencia $2\pi k/N$. Al formar la familia de exponenciales complejas una base ortonormal, es posible recuperar la señal $x[n]$ a partir de $X[k]$ utilizando la transformada discreta de Fourier inversa (IDFT):

$$x[n] = \frac{1}{N} \sum_{n=0}^{N-1} X[k] e^{j\frac{2\pi kn}{N}} \qquad k = 0, 1, 2, ..., N-1 \quad (2.2)$$

2.3.2 Transformada de Fourier de Tiempo Corto (STFT)

En el caso en que las señales sean no estacionarias (como en el caso de señales de carácter musical), el cálculo de una sola DFT no resulta ser suficiente para representar correctamente la señal. Es por ello que normalmente se hace uso de una ventana temporal, que limita el cálculo de la DFT a una porción determinada de muestras. Moviendo esta ventana a través del tiempo se logra obtener una representación de la señal a lo largo de toda su duración, que contempla las variaciones en frecuencia de la misma.

De esta forma, se define la Tranformada de Fourier de Tiempo Corto (STFT) como:

$$X[n, k] = \sum_{m=0}^{N-1} x[Hn + m] w[m] e^{-j\frac{2\pi km}{N}} \quad (2.3)$$

Donde

- $X[n, k]$ es la transformada de Fourier de tiempo corto de x[n].
- n es el cuadro de la transformada.
- k es el índice frecuencial de la transformada.

- $w[m]$ es la ventana utilizada, que cuenta con una longitud de N muestras.
- H es la cantidad de muestras entre dos ventanas consecutivas. Define cuanto solapamiento existe entre las mismas. También se lo llama tamaño de salto o *hop size*.

Al aplicar una ventana, se producen dos efectos secundarios principales: el manchado espectral (*smearing*), que consiste en una pérdida en la resolución en frecuencia, y la fuga espectral (*leakage*), que es el efecto de los lóbulos secundarios en el espectro de la ventana. La resolución depende del ancho del lóbulo principal del espectro de la ventana ($W(e^{j\omega})$), mientras que la severidad de las fugas depende de la amplitud relativa del lóbulo principal respecto de los lóbulos secundarios. La Figura 2.4 muestra una ventana temporal de tipo Hamming junto con su espectro, en el cual se pueden apreciar estos lóbulos.

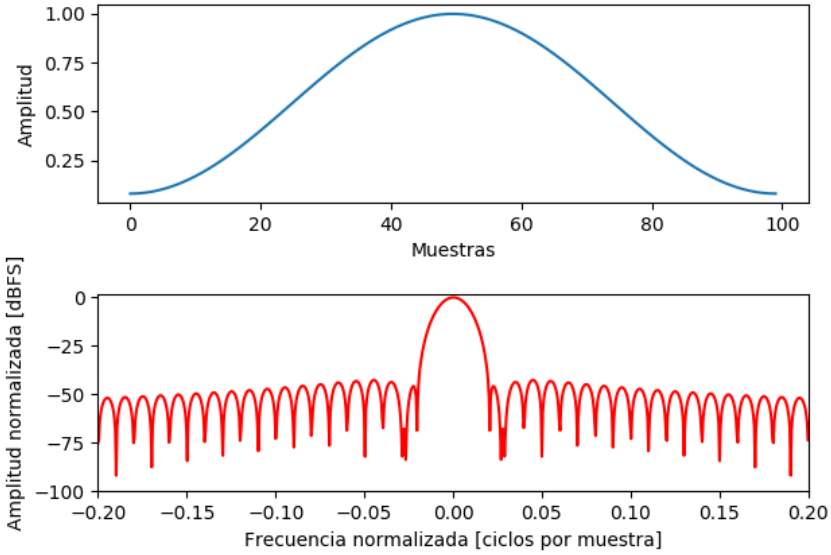

Figura 2.4: Ventana tipo Hamming (arriba), y resultado de aplicar la DFT sobre la misma (abajo).

El propósito principal de la ventana es limitar la extensión de la secuencia a ser transformada, de manera que las características espectrales sean aproximadamente constantes a lo largo de la duración de la ventana. Cuanto más rápidamente cambien las características de la señal, más corta deberá ser la ventana. Asimismo, cuanto más corta es la ventana, menor es la resolución en frecuencia. Por lo tanto, la elección de la longitud de la ventana implica encontrar una relación de compromiso entre resolución en frecuencia y resolución en tiempo.

El resultado de aplicar una STFT sobre una señal es una matriz de números complejos, a partir de la cual pueden extraerse componentes de magnitud y fase. En muchos casos la fase es descartada, y se utiliza la magnitud para construir un gráfico denominado espectrograma, que da una idea del contenido espectral del audio en función del tiempo. La Figura 2.5 muestra un espectrograma extraído a partir del audio de una persona hablando.

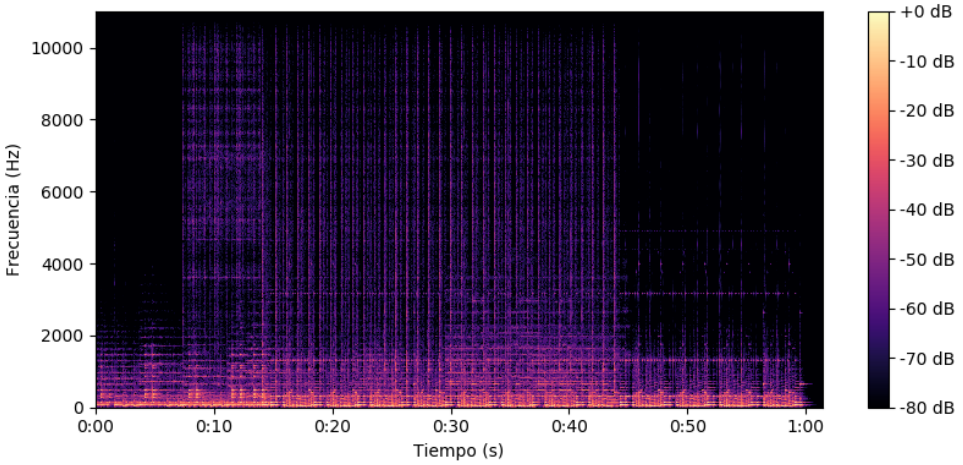

Figura 2.5: Ejemplo de espectrograma, calculado a partir de una señal de habla.

2.3.3 Transformada de Q constante

La representación en un eje de frecuencia lineal dada por la STFT muestra una separación constante entre componentes armónicas de las señales bajo análisis. Esta es la característica principal del patrón producido, y tanto la constante de separación como la posición general de este patrón varían con la frecuencia fundamental. Como resultado de esto, resulta difícil distinguir diferencias en relación a otros aspectos de la señal, tales como timbre, ataque, o decaimiento. Es por esto que usualmente se utiliza un eje de frecuencia logarítmico para representar la STFT de una señal, ya que esta representación permite visualizar de manera más efectiva las características espectrales de la misma y, de esta forma, tareas como identificación de instrumentos o de frecuencias fundamentales se convierten en problemas mucho más simples de reconocimiento de un patrón predeterminado [17].

No obstante, al visualizar de esta manera el resultado de una STFT, es claro que el mapeo de los datos desde el dominio lineal hacia el logarítmico da como resultado muy poca información en frecuencias bajas y excesiva información en frecuencias altas. Lógicamente, esto repercute de manera negativa a la hora de analizar los espectrogramas, especialmente en bajas frecuencias, en donde muchas veces la resolución resulta ser menor que la separación entre notas musicales dentro del rango a analizar.

Propuesta por primera vez en 1991 [18], la Transformada de Q Constante (CQT) busca resolver esta problemática. Esta transformada da como resultado espectrogramas en donde la resolución en un eje de frecuencia logarítmico es constante. A modo de ejemplo, la Figura 2.6 muestra dos espectrogramas, calculados utilizando ambas transformadas, de una señal de audio constituida por 4 tonos puros con frecuencias fundamentales de 250, 500, 1000, y 2000 Hz, respectivamente.

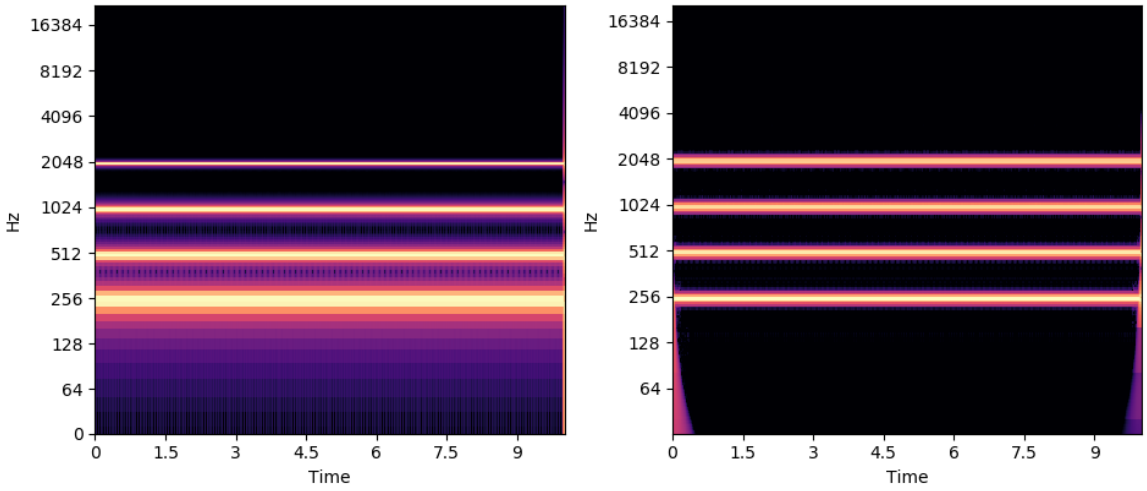

Figura 2.6: Espectrogramas de la misma señal, obtenidos mediante STFT (izquierda) y CQT (derecha).

Para el cálculo de esta transformada, se puede comenzar tomando el ancho de banda δf de la STFT, que está determinado por la frecuencia de muestreo f_s y el tamaño de la ventana N:

$$\delta f = \frac{f_s}{N} \tag{2.4}$$

Como se requiere que los componentes de la CQT estén espaciados de manera exponencial, esto implica una relación constante (Q) entre frecuencia y ancho de banda, que da nombre a la transformada. Es decir:

$$Q = \frac{f}{\delta f} \tag{2.5}$$

A partir de esto, se puede plantear la siguiente ecuación:

$$N[k] = \frac{Qf_s}{f} = \frac{f_s}{\delta f_k} \tag{2.6}$$

En (2.6) k es el índice del componente espectral correspondiente. Se obtiene la ecuación para dicho componente tomando el cálculo del mismo correspondiente a la STFT:

$$X[k] = \sum_{n=0}^{N-1} w[n]x[n]e^{\frac{-j2\pi kn}{N}} \tag{2.7}$$

Y considerando lo siguiente:

- De acuerdo a las ecuaciones (2.4), (2.5) y (2.6), la frecuencia digital $\frac{2\pi k}{N}$ puede reemplazarse por $\frac{2\pi Q}{N[k]}$.
- La ventana $w[n]$ tiene la misma forma para cada componente, pero su longitud está determinada por $N[k]$, por lo cual es una función tanto de k como de n.

- Como el límite superior de la sumatoria pasaría también a ser función de k, entonces el resultado de la misma debería estar normalizado de acuerdo a un factor $N[k]$

De esta manera, la ecuación final que determina el cálculo de la CQT se expresa como:

$$X[k] = \frac{1}{N[k]} \sum_{n=0}^{N[k]-1} w[n,k]x[n]e^{\frac{-j2\pi Qn}{N[k]}} \tag{2.8}$$

2.3.4 Cromagramas

Una de las representaciones del audio que más frecuentemente se utiliza en los sistemas de reconocimiento automático de acordes es el cromagrama. Si bien existen algunas variantes, en líneas generales un cromagrama describe como las notas musicales y su intensidad varían a lo largo de una señal de audio.

Normalmente se representa como una matriz de valores reales entre 0 y 1 donde cada fila representa una nota musical y cada columna representa una ventana temporal de audio, de modo que cada valor se corresponde con la presencia o intensidad de una nota musical en un momento dado. El vector que contiene la intensidad de las notas en un determinado intervalo de tiempo (es decir, una ventana) se conoce como vector croma (*chroma vector* o *chroma feature*). Un ejemplo de cromagrama puede verse en la Figura 2.7, en donde los colores más claros representan mayor intensidad.

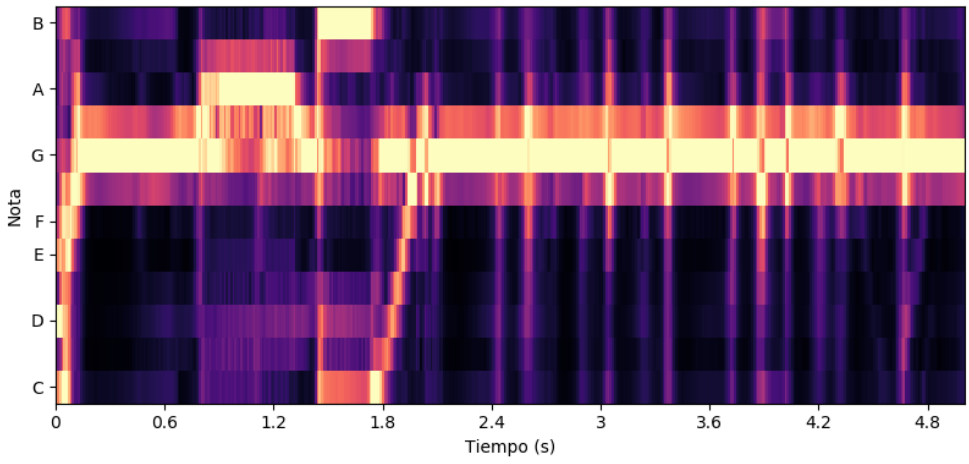

Figura 2.7: Ejemplo de cromagrama, correspondiente a los primeros segundos de la canción Blackbird, de The Beatles.

2.3.5 Separación de componentes percusivas y armónicas

Una herramienta útil para el procesamiento y análisis de audios musicales es la separación de los mismos en sus componentes armónicas y percusivas (HPSS).

Los algoritmos comúnmente utilizados para esta tarea se basan en lo propuesto por Fitzgerald [19], cuyo método se fundamenta en la idea de que un espectrograma muestra eventos musicales percusivos como líneas verticales, y eventos armónicos o estacionarios como líneas horizontales. El procedimiento propuesto es el siguiente:

- Primero, se obtiene la magnitud de la STFT de la señal.
- Luego, se utilizan dos filtros de mediana móvil: el primero, a lo largo del eje de frecuencias, y el segundo, a lo largo del eje de tiempo. Esto da como resultado un espectrograma que resalta los componentes percusivos (S_P), y otro que resalta los componentes armónicos (S_H).
- A partir de estos espectrogramas se calculan máscaras binarias o suaves en cada punto del espectrograma, dependiendo de la predominancia de S_P o S_H.
- Finalmente, se aplican estas máscaras al espectrograma original, y se invierten los espectrogramas resultantes.

A partir de esto se han propuesto diversas mejoras, dentro de las cuales cabe destacar el trabajo realizado por Driedger et al. [20], quienes propusieron dos agregados significativos:

- El agregado de un tercer componente residual, que engloba a los sonidos que no son puramente percusivos ni puramente armónicos (por ejemplo, ruido estacionario).
- La introducción de un *factor de separación* al proceso de decomposición, que determina cuán estricta es la separación, y en qué medida se 'fuerza' a un componente a ser armónico o percusivo.

2.4 REDES NEURONALES ARTIFICIALES

Las redes neuronales artificiales (ANN según sus siglas en inglés) son herramientas de modelado computacional que recientemente han sido utilizadas para la resolución de diversos problemas en una multitud de ámbitos [21].

Pueden ser definidas como estructuras conformadas por elementos adaptativos densamente interconectados (denominados neuronas artificiales o nodos), que son capaces de llevar a cabo cómputos en paralelo para procesamiento y representación de datos [22], [23].

Si bien las redes neuronales artificiales se inspiran en sus contrapartes biológicas, la idea detrás de las ANN no es replicar la operación de sistemas biológicos, sino hacer uso de lo que se conoce sobre el funcionamiento de las redes biológicas para la resolución de problemas complejos.

2.4.1 Modelos de neurona artificial.

En 1958, Rosenblatt introdujo la mecánica de la neurona artificial individual a través del denominado perceptrón, para resolver problemas en el área de reconocimiento de caracteres [24]. Un perceptrón toma un número arbitrario de entradas binarias y produce una única salida, también binaria (Figura 2.8).

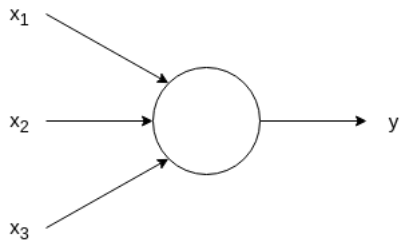

Figura 2.8: Ejemplo de perceptrón con tres entradas.

El perceptrón posee *pesos sinápticos* $w_1, w_2, ...w_n$, los cuales son números reales que expresan la importancia de cada entrada (x_i) en relación a la salida (y). De esta forma, la salida depende de que el valor de la sumatoria $\sum w_j x_j$ sea menor o mayor que un cierto valor de umbral (b). Es decir:

$$y = \begin{cases} 0 & si \ \sum_j w_j x_j + b \leq 0 \\ 1 & si \ \sum_j w_j x_j + b > 0 \end{cases} \tag{2.9}$$

No obstante, el hecho de que la salida del perceptrón sea binaria representa un problema, dado que pequeños cambios en la entrada de una neurona pueden generar grandes cambios en la salida, haciendo que una estructura dependiente de perceptrones sea inestable e impredecible.

Para solucionar esto, se determina que la salida pase de ser binaria a real a través del uso de distintas *funciones de activación*, donde el rango de valores posibles está determinado por la función de activación utilizada. Algunas de las funciones más comunes de activación se pueden ver en la Figura 2.9.

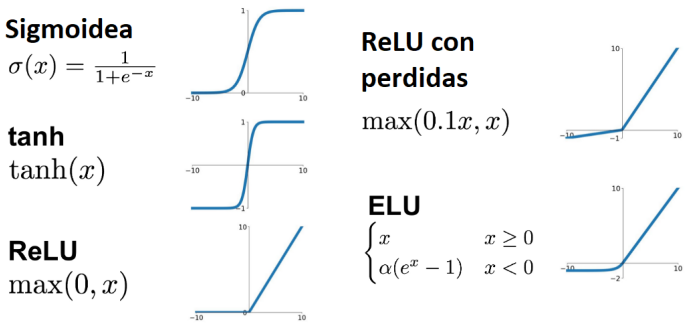

Figura 2.9: Funciones de activación comunes (extraído y adaptado de [25]).

De esta forma, el valor de salida de una neurona dada pasa a ser:

$$A_j = f\left(\sum_j w_j x_j + b\right) \tag{2.10}$$

Donde f es la funcion de activación utilizada, y tanto los pesos sinápticos (w_j) como el umbral (b) son variables cuyos valores cambian durante el proceso de aprendizaje, el cual se describirá en la sección 2.4.3.

2.4.2 Capas de neuronas

Frecuentemente, se distribuyen las neuronas en capas en las cuales cada neurona se conecta con todas las neuronas de la capa anterior (con excepción de la primera capa, la capa de entrada) y todas las neuronas de la capa posterior (con excepción de la capa de salida). Las capas intermedias se denominan capas ocultas. En esta disposición, se dice que las capas están *densamente conectadas*. Un ejemplo de esto puede verse en la Figura 2.10. De esta forma, se introduce una entrada a la red, y la misma produce una salida en base al umbral y el peso de cada neurona que forma parte de la red.

2.4.3 Aprendizaje de una red neuronal artificial

Para el entrenamiento de una red neuronal, en primer lugar se requiere un conjunto de datos destinados a este propósito. En caso de que el aprendizaje sea supervisado (en el cual se hace uso de datos previamente etiquetados), los mismos consisten en pares (x, y) donde cada valor de y es el valor de salida correspondiente a cada entrada x. Por ejemplo, si se

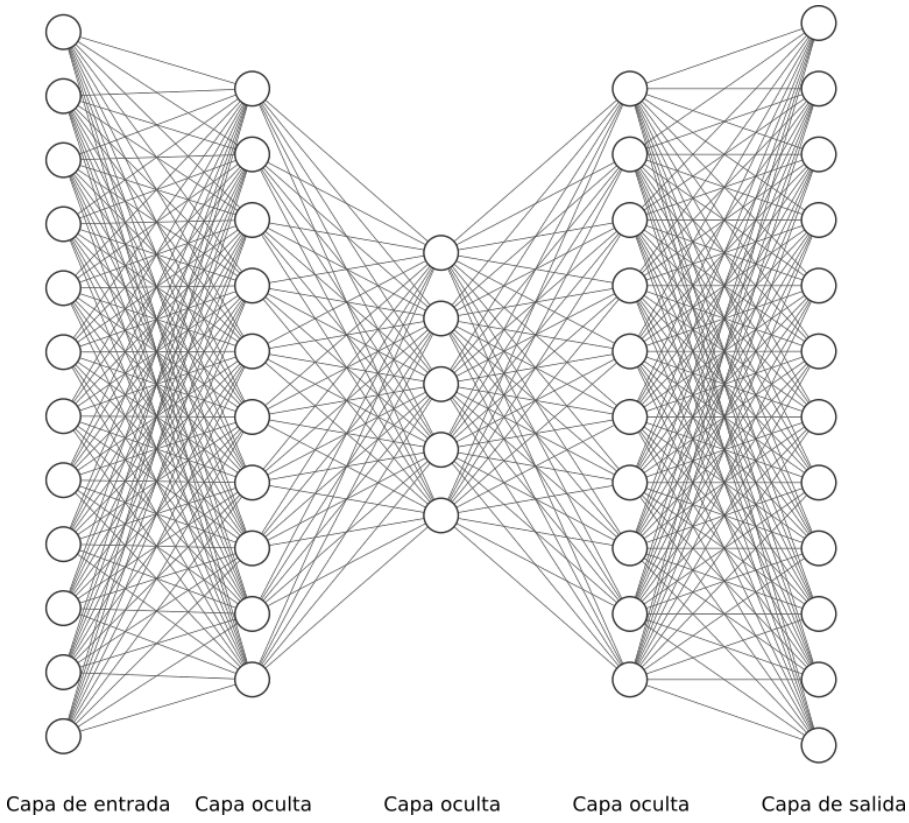

Figura 2.10: Arquitectura de red neuronal con capas densamente conectadas.

quisiera entrenar una red para reconocer dígitos escritos a mano, sería necesario tener un conjunto de pares en donde x sean las imágenes de los dígitos, e y los rótulos o etiquetas correspondientes, considerando que la red neuronal es una función matemática $f(x)$. De esta forma, se pueden comparar los resultados entregados por el sistema contra los valores esperados, y ajustar de manera acorde los parámetros de la red neuronal.

Para realizar esta comparación y determinar el grado de error presente en la salida de la red se define una función de error o *función de costo*. La expresión matemática de la misma dependerá de la arquitectura de la red y del propósito de la misma. Un ejemplo común de función de costo es el error cuadrático medio:

$$C(w,b) = \frac{1}{n}\sum_{x=1}^{x=n} ||f(x) - y||^2 \tag{2.11}$$

donde w es el conjunto de pesos sinápticos de la red, b el conjunto de umbrales (o *biases*), n es la cantidad total de vectores de entrada de la red, $f(x)$ el resultado de ingresar a la red una entrada x, e y el valor de salida correspondiente a esa entrada. Asimismo, la suma contempla la totalidad de los vectores de entrada en el conjunto de datos de entrenamiento. Puede observarse que, si la diferencia entre la salida obtenida y la salida verdadera decrece, también decrece el resultado de la función de costo.

Ahora bien, habiendo determinado una medida del error, el siguiente paso es, para todas

las neuronas que forman parte de la red, determinar valores óptimos de w y b que posibiliten la minimización de la función de costo.

Para determinar los valores óptimos de los parámetros b y w del modelo, se utiliza el algoritmo de descenso por gradiente [26], el cual consiste en ajustar los parámetros P del modelo en forma iterativa de acuerdo a la expresión:

$$\Delta P = -\eta \nabla_P (C(P)) \tag{2.12}$$

donde η es la tasa de aprendizaje, la cual determina el grado de actualización de los parámetros en cada iteración del algoritmo.

El algoritmo de propagación hacia atrás de los errores, o retropropagación [27], permite calcular de forma eficiente $\nabla_P (C(P))$ y realizar el descenso por gradiente.

2.4.4 Épocas, lotes y mini-lotes.

El término *época* hace referencia a un ciclo de entrenamiento en el cual fueron utilizadas todas las muestras del conjunto de datos de entrenamiento. Se denomina *lote* al conjunto total de datos de entrenamiento. El mismo puede dividirse en porciones de un cierto tamaño de muestras, las cuales se denominan *mini-lotes*.

El proceso de ajuste descrito en la sección anterior puede implementarse de varias maneras, donde la elección del método dependerá del caso particular:

- Realizando el ajuste de parámetros luego de cada muestra en el conjunto de datos de entrenamiento (descenso por gradiente estocástico, o *stochastic gradient descent*).
- Realizando el ajuste de parámetros al finalizar cada época de entrenamiento (lotes).
- Realizando el ajuste de parámetros cada cierta cantidad de muestras (mini-lotes).

2.4.5 Entrenamiento, validación, y prueba.

Normalmente, el conjunto de datos del cual se dispone se divide en tres partes:

- **Datos de entrenamiento.** Ya mencionados con anterioridad, se utilizan para entrenar al modelo. Suelen ser la porción más grande, representando entre un 80 % y 90 % del total de datos.
- **Datos de validación.** Conjunto de datos (usualmente entre 5 % y 10 % del total) que se utiliza para evaluar el modelo entre épocas. Es utilizado para poder tomar ciertas decisiones, como determinar cuándo detener el entrenamiento o qué arquitectura de red neuronal utilizar.

- **Datos de prueba.** Conjunto de datos (usualmente entre 5 % y 10 % del total) que se utiliza una sola vez, cuando el modelo ha sido seleccionado y el entrenamiento concluido. Sirve para evaluar el rendimiento del modelo. Tanto el conjunto de datos de prueba como el de validación están compuestos por datos que no fueron utilizados para el entrenamiento del modelo (es decir, que el mismo nunca 'vio').

No obstante, dependiendo de los datos disponibles, puede que este esquema no se respete en todos los casos; por ejemplo, si se dispone de un conjunto de datos reducido, puede resultar más útil obviar el uso de un conjunto de pruebas y utilizar solo el de validación.

2.4.6 Sobreajuste y subajuste.

Al entrenar un modelo, pueden darse dos fenómenos no deseados: el sobreajuste (*overfitting*), y el subajuste (*underfitting*).

Se dice que un modelo sobreajusta cuando el mismo presenta un rendimiento muy superior con el conjunto de datos de entrenamiento respecto de los conjuntos de validación y pruebas. Esto implicaría que el mismo no ha desarrollado la capacidad de generalizar a partir del entrenamiento, y simplemente ha 'memorizado' la información provista. Este fenómeno puede ocurrir cuando el modelo posee más parámetros y poder expresivo de lo que era necesario para caracterizar al modelo subyacente a los datos [28].

Asimismo, un modelo que subajusta muestra un alto error en el aprendizaje en la etapa de entrenamiento. Esto se debe a que el mismo es muy simple o cuenta con poca *capacidad* para aprender a partir de la información dada, o bien a que dicha información resulta demasiado compleja para el modelo utilizado.

La Figura 2.11 muestra los tres casos posibles: ajuste correcto, subajuste, y sobreajuste.

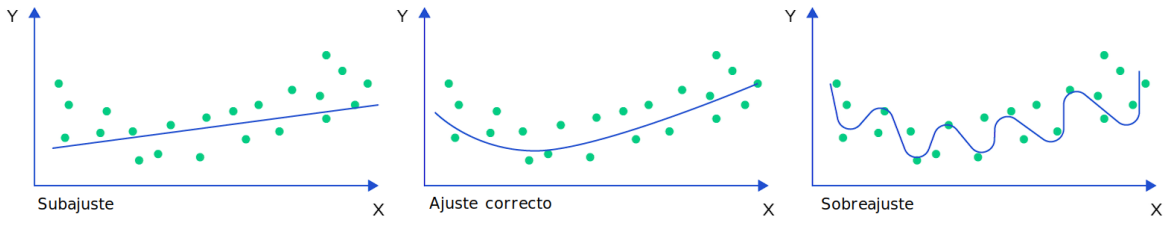

Figura 2.11: Ejemplo de modelo que presenta subajuste (izquierda), sobreajuste (derecha), y un ajuste adecuado (centro) (Extraído y adaptado de [29]).

Algunas técnicas comunes para mitigar el sobreajuste son:

- **Uso de más datos de entrenamiento**. Utilizar conjuntos de entrenamiento más grandes es una de las mejores formas de mejorar el rendimiento del modelo a la hora de generalizar. No obstante, dependiendo del tipo de datos requerido, puede que sea

muy difícil o incluso imposible la obtención de conjuntos grandes de datos. Una estrategia para aumentar la cantidad de datos de entrenamiento es generarlos de manera sintética manipulando datos ya existentes.

- **Dropout**. Se conoce con este nombre a la práctica de anular de manera aleatoria las conexiones de un cierto porcentaje de neuronas de una o más capas de la red durante el entrenamiento [30]. Esto impide que el modelo sea excesivamente dependiente de la influencia de una o unas pocas neuronas.

- **Regularización de pesos y umbrales**. Esta práctica implica agregar un nuevo término a la función de costo de la red, cuyo fin es el de penalizar a las neuronas con pesos y/o umbrales que toman valores altos. De igual manera que con la utilización de Dropout, se busca que la red aprenda valores pequeños de pesos y umbrales.

- **Interrupción anticipada del entrenamiento**. Esta técnica consiste en detener el entrenamiento antes de que el modelo comience a sobreajustar los datos [31]. Para ello, normalmente se toma como referencia el rendimiento del modelo sobre el conjunto de validación a través de las épocas de entrenamiento.

2.4.7 Interpretabilidad de los modelos de redes neuronales.

Dada la complejidad de las redes neuronales artificiales, resulta dificultoso determinar de qué manera las mismas han asimilado la información dada y si han realmente aprendido de la forma esperada, independientemente de su desempeño final.

Por ejemplo, en el caso mencionado anteriormente de una red neuronal que clasifique dígitos escritos a mano, es importante saber a qué píxeles de una imagen de entrada dada el modelo 'presta más atención', siendo que se espera que el mismo reconozca una imagen de (por ejemplo) un número siete a partir de las regiones de la imagen en donde aparecen los trazos correspondientes al mismo, en lugar de realizar sus predicciones a partir de otras características de la imagen (independientemente de la exactitud de dichas predicciones).

Para ello, en algunos casos se buscó generar entradas que maximizaran la actividad neural de interés [32]. En otros, se propuso reconstruir la entrada de cada capa a partir de su salida [33]. En el contexto de este trabajo en particular, se utilizaron los llamados mapas de saliencia [34]. Esencialmente, la idea es que para cada elemento de una entrada determinada se calcule la derivada de la salida respecto de la entrada:

$$\frac{\delta_{salida}}{\delta_{entrada}} \tag{2.13}$$

donde estos gradientes se calculan utilizando algoritmos de propagación inversa (de igual manera que el error se propaga hacia atrás en el entrenamiento de la red). El resultado de

esto es una matriz que resalta áreas de la entrada que tienen mayor influencia en una salida determinada. De manera intuitiva, esto pone el foco en las regiones 'salientes' del vector de entrada que contribuyen al resultado.

Continuando con el ejemplo anterior, la Figura 2.12 muestra una imagen de entrada del dígito uno (1), y el mapa de saliencia correspondiente a dicha entrada.

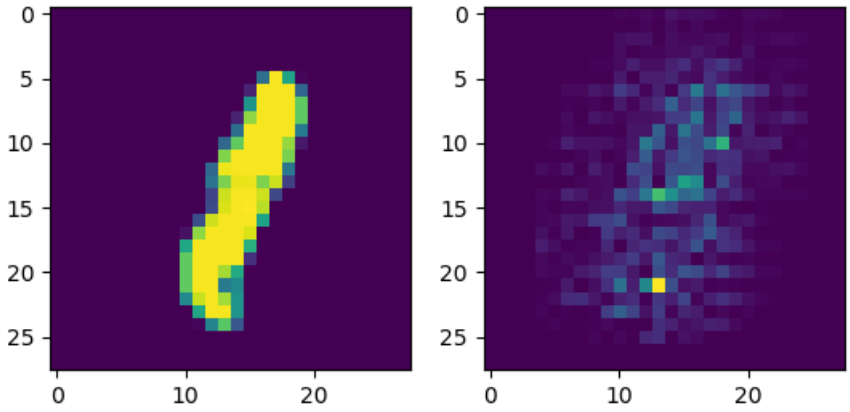

Figura 2.12: Imagen de entrada (izquierda), y mapa de saliencia correspondiente (derecha).

2.5 ESTIMACIÓN AUTOMÁTICA DE ACORDES

El campo de ACE en su forma moderna tiene su inicio en 1999 [35]. Desde ese entonces, se ha establecido un flujo de trabajo asociado a este campo. El mismo se puede observar en la Figura 2.13.

En las siguientes subsecciones se dará una descripción general de cada una de las etapas de este proceso.

2.5.1 Extracción de cromagramas

Es en este paso en donde se enfoca el presente trabajo, y al mismo tiempo es el de mayor importancia y el que mayores consideraciones requiere. Consiste en la obtención de cromagramas a partir de los audios a analizar, comúnmente utilizando diversas técnicas de procesamiento digital de señales. Dichas técnicas, junto con el flujo de trabajo utilizado serán analizadas en detalle en la siguiente sección.

2.5.2 Estrategias de modelado

El siguiente paso consiste en asignar etiquetas a los vectores croma, de acuerdo al acorde que se estima. Algunas técnicas para lograr esto son:

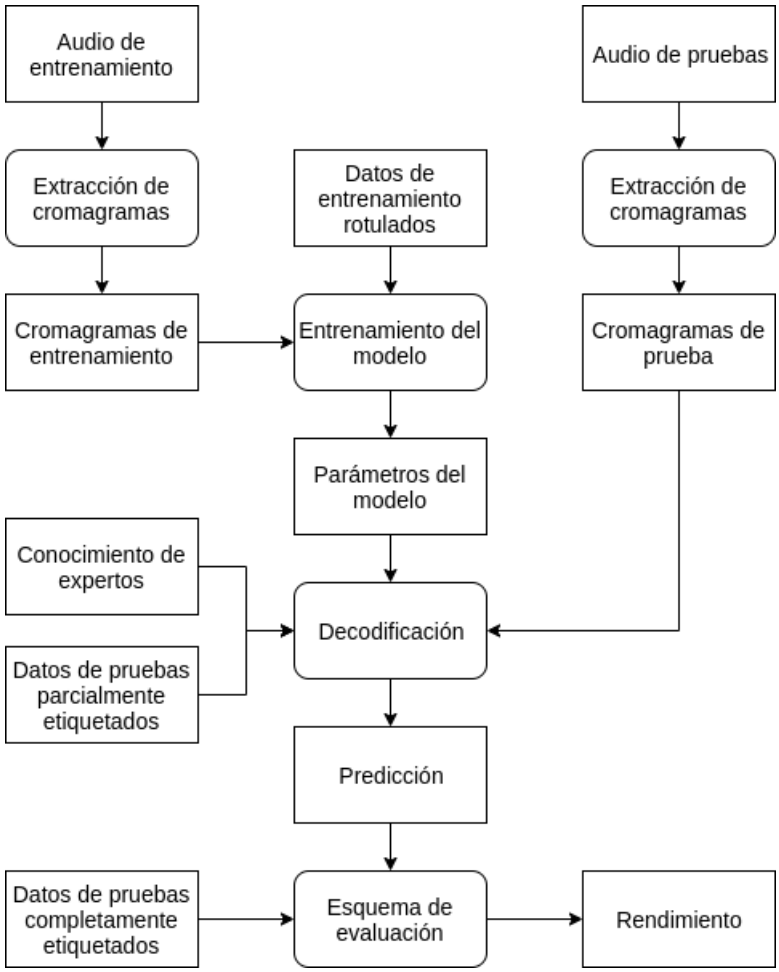

Figura 2.13: Diagrama de flujo de trabajo de un sistema de ACE. Los datos se muestran en rectángulos, y los procesos en rectángulos con esquinas redondeadas.

- **Comparación contra plantillas.** Es la alternativa más simple, y consiste en comparar los vectores croma contra la distribución conocida de notas en un acorde, bajo la presunción de que el cromagrama tendrá una cierta similitud a los acordes presentes en la pieza.

- **Modelos ocultos de Márkov (HMMs).** El uso de comparaciones, tal como fueron descritas en el ítem anterior, presentan limitaciones al modelar la naturaleza continua de secuencias de acordes. Esto puede mitigarse utilizando métodos para *suavizar* los cromagramas, o bien puede hacerse uso de alguna noción temporal en el modelo subyacente. Una de las formas más comunes de lograr esto último es utilizando *modelos ocultos de Márkov*. Si bien los detalles matemáticos y las complejidades de estos modelos exceden el enfoque de este trabajo (aunque un buen punto de partida puede encontrarse en [36]), cabe mencionar que, en los últimos tiempos, el uso de este tipo de modelos se ha convertido en el método más común para asignar acordes a ventanas de audio en el dominio de ACE [37].

Asimismo, es posible extender una arquitectura basada en HMMs para estimar de manera simultánea acordes y tonalidades.

- **Otras variantes.** Se suelen usar estrategias de modelado tales como el uso de *redes Bayesianas dinámicas* [38], el desarrollo de modelos específicos para distintos géneros musicales [39], o el uso de modelos discriminativos [40].

2.5.3 Entrenamiento del sistema y conjuntos de datos

Poseer información conocida y validada respecto de los acordes presentes en una pieza es esencial a la hora de determinar la eficacia del sistema. Normalmente este tipo de información consiste en dos códigos de tiempo asociados a un intervalo, junto con una etiqueta del acorde correspondiente.

Por otro lado, existen dos enfoques posibles para llevar a cabo el entrenamiento y ajuste del sistema: haciendo uso del conocimiento de expertos, o que el entrenamiento esté impulsado por los mismos datos (*data-driven*).

- **Conocimiento de expertos.** Implica ajustar manualmente ciertos parámetros del sistema, de acuerdo al criterio de expertos en el tema. Por ejemplo, Shenoy y Wang [41] le dieron mayor preponderancia a acordes comunes dentro de la tonalidad, especificando también que, si los primeros tres tiempos dentro de un compás se correspondiesen con un acorde, entonces el cuarto tiempo también estaría asociado a ese acorde. Este enfoque prevaleció principalmente en las primeras investigaciones dentro del campo de ACE, dada la escasez de datos de entrenamiento validados.

- **Entrenamiento impulsado por datos.** Implica que los parámetros del sistema estén determinados, en cierta medida, por los datos disponibles. Por ejemplo, en una red neuronal artificial, los pesos sinápticos y umbrales, que son los parámetros del modelo, se ajustan en función de los datos de entrenamiento. Este tipo de entrenamiento requiere de una cierta cantidad de datos de entrenamiento para ser viable.

No obstante, ambos enfoques pueden tener sus limitaciones, especialmente a la hora de generalizar los resultados obtenidos a otros conjuntos de datos.

2.5.4 Técnicas de evaluación del sistema

A pesar de que el reconocimiento automático de acordes es un campo de investigación relativamente activo, las métricas de evaluación continúan siendo tópico de discusión [42]. Como resultado, no hay un consenso respecto del mejor enfoque para comparar dos secuencias de etiquetas de acordes y, en su lugar, la comparación se realiza en base a ciertas

reglas. La métrica usada con mayor frecuencia para esta tarea es el denominado *Weighted Chord Symbol Recall* (WCSR) [42].

El denominado *Chord Symbol Recall* (CSR) se define como:

$$\frac{|S \cap S^*|}{|S^*|} \tag{2.14}$$

donde S y $S*$ representan las predicciones y los valores verdaderos de los acordes, respectivamente, y la intersección entre S y S^* es la parte en donde las mismas se solapan [43]. El criterio de acuerdo al cual se considera que hay un solapamiento está definido por alguna de las siguientes reglas:

- **Tónica.** Requiere solo que las tónicas sean equivalentes.
- **Terceras.** Requiere que tanto las tónicas como las terceras (sean mayores o menores) coincidan.
- **Tríadas.** Requiere que las tríadas (tónica-tercera-quinta) coincidan.
- **Séptimas.** Igual que con las Tríadas, pero agregando la séptima.
- **Tétradas.** Todos los intervalos coinciden.
- **Mayor-Menor.** Coincide la calidad (Mayor o menor).
- **Mayor-Menor-Inv.** Es igual a Mayor-Menor pero requiere ademas que la nota más grave también coincida.
- **Séptimas-Inv.** Es igual a Mayor-Menor-Inv, pero ademas requiere Séptimas.
- **MIREX.** Coinciden al menos tres notas.

Dado esto, el WCSR es simplemente el promedio ponderado de los CSRs de todos los audios utilizados de acuerdo a la longitud en segundos de los mismos:

$$WCSR = \frac{\sum Longitud_i * CSR_i}{\sum Longitud_i} \tag{2.15}$$

CAPÍTULO 3: ESTADO DEL ARTE

3.1 EXTRACCIÓN DE CROMAGRAMAS: TÉCNICAS Y PROCESOS

El proceso de extracción de cromagramas consta de varios sub-procesos que se han vuelto estándar a lo largo de los años. Los mismos se describen a continuación:

- **Pasaje al dominio de la frecuencia.** Implica obtener, a través de alguna transformada (STFT, CQT), una representación espectral adecuada del audio a analizar.

- **Pre-procesado.** Al considerar una señal de audio musical, es claro que no todo el contenido resulta relevante con respecto al análisis armónico del mismo. Técnicas para remover el llamado contenido espectral *de fondo* [44] y el contenido armónico correspondiente a las notas individuales (es decir, extrayendo solamente f_0)[45] son herramientas comunes que investigadores han utilizado para la extracción de cromagramas. Asimismo, en muchos casos se han utilizado algoritmos de HPSS para aislar las componentes armónicas de los audios.

- **Afinación.** En algunos casos, los audios musicales a analizar no fueron grabados utilizando una afinación estándar (A4 = 440 Hz), o bien existieron pequeñas imperfecciones en la afinación. Para compensar esto, es posible computar espectrogramas de mayor resolución (que permiten mayor flexibilidad con la afinación de la pieza) [46], o implementar algoritmos más complejos que ajusten los espectrogramas obtenidos a la afinación original [47].

- **Obtención de saliencia tonal.** Aunque el espectrograma pre-procesado y afinado es intuitivamente una buena representación de la evolución tonal de una pieza, algunos autores han explorado maneras de modificar esto para representar los sonidos de manera más fiel a la percepción humana. Esto ha implicado la implementación de diversos procesamientos en el dominio frecuencial, como por ejemplo el uso de la ponderación A [48], lo cual ha dado resultados satisfactorios [49].

- **Suma de octavas y normalización.** El paso final en el cálculo de cromagramas involucra la suma del contenido espectral correspondiente a cada nota musical, seguido de una normalización en amplitud. El primer paso permite que se trabaje con una representación concisa de 12 dimensiones, mientras que la normalización hace que el resultado sea independiente respecto de cambios en la amplitud original de la pieza.

- **Suavizado/Sincronización con pulsos.** Un descubrimiento temprano en el estudio de ACE, fue que el uso de cromagramas instantáneos (es decir, por cada ventana) llevaba a que, debido al ruido y las señales transitorias, los sistemas predijeran cambios de acordes con gran frecuencia [35]. El proceso de suavizado de los vectores croma reduce este fenómeno. Por otro lado, los acordes suelen ser estables entre cambios de ritmo musicales [50]. Este hecho fue explotado para la creación de cromagramas síncronos respecto del pulso de la pieza, donde la ventana temporal se reduce a la mitad del pulso principal [51]. De esta manera, también se evita inestabilidad en las secuencias de acordes predichas por el sistema.

La Figura 3.1 muestra un diagrama del flujo de trabajo realizado para la extracción de cromagramas.

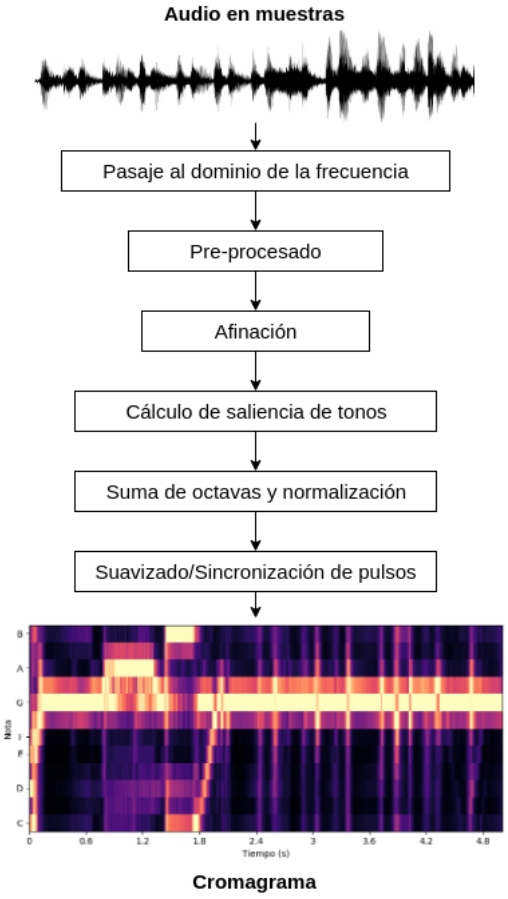

Figura 3.1: Pasos comunes a tomar para convertir una señal de audio digital en su correspondiente representación en forma de cromagrama.

3.2 APRENDIZAJE AUTOMÁTICO APLICADO A ACE

Si bien existe una amplia variedad de casos en los cuales técnicas de aprendizaje automático han sido aplicadas a tareas de ACE (principalmente en lo que respecta a identificar

acordes o secuencias a partir de cromagramas [49], [52]), el proceso de extracción de cromagramas se ha mantenido largamente exento de esta tendencia.

Algunos investigadores han obviado el esquema de trabajo tradicional, optando por desarrollos basados completamente en técnicas de aprendizaje automático. Tal es el caso de Bello y McFee [53], que utilizaron una arquitectura de estilo *codificador/decodificador* [54]. De esta forma, el codificador mapea una entrada variante en el tiempo (audio) a un espacio latente, mientras que el decodificador toma este espacio latente y lo mapea al espacio de salida (conformado por etiquetas de acordes). Obtuvieron resultados satisfactorios de WCSR, de hasta 0.861 en predicción de tónicas y hasta 0.671 en tétradas, aunque el modelo desarrollado no tiene en cuenta inversiones de acordes.

Otro trabajo de relevancia es el realizado por Korzeniowski y Widmer [55], quienes desarrollaron un sistema de extracción de cromagramas que bautizaron como extractor de croma (*chroma extractor*). Utilizaron una red neuronal para extraer cromagramas a partir de espectrogramas de STFTs modificadas (espectrogramas de medio semitono, o *quarter-tone*). La red implementada consistía en una estructura simple de una capa de entrada, seguida por tres capas ocultas densamente conectadas de 512 neuronas, que concluían en una capa de salida de 12 neuronas, correspondiente a las 12 notas del alfabeto musical occidental. El conjunto de datos utilizado consistió en 383 canciones de música de artistas como The Beatles [56], Queen y Zweieck [57], entre otros. En cuanto al rendimiento del sistema, se estableció una tarea simple de reconocimiento (solamente comparando las notas del cromagrama contra plantillas establecidas); se compararon los resultados de esta tarea utilizando como entrada los cromagramas obtenidos mediante el sistema desarrollado, contra los resultados obtenidos utilizando cromagramas estimados a través de métodos tradicionales.

Zhou y Lerch [58] implementaron una Red Neuronal de Aprendizaje Profundo (DNN) para la tarea de predicción de acordes. Esta red toma varias ventanas de CQT (conjunto denominado como super ventana, *superframe*), obteniéndose una salida que puede interpretarse directamente como la probabilidad de cada tipo de acorde. En lo que respecta al diseño de la red, los investigadores llevaron a cabo experimentos con dos arquitecturas distintas: un modelo de 6 capas de 1024 neuronas cada una, y un modelo de 6 capas en donde las capas medias contaban con un número menor de neuronas (512, y 256), estructura que denominaron como *bottleneck*, o cuello de botella. Asimismo, para mejorar la predicción, experimentaron ingresando esta salida a dos clasificadores: siendo el primero un HMM, y el segundo un modelo conocido como Support Vector Machine (SVM). Utilizaron WCSR como métrica de evaluación del sistema, obteniendo valores de hasta 0.919 en un subconjunto de pruebas extraído del mismo conjunto de datos utilizado por [55].

CAPÍTULO 4: METODOLOGÍA

4.1 CONJUNTOS DE DATOS

4.1.1 Generación de audios de entrenamiento, validación y prueba

Para el entrenamiento y la validación del sistema se creó un conjunto de datos específico para esta tarea, utilizando el conjunto de datos NSynth [59].

El mismo consiste en 305979 audios de notas musicales, cada uno con una envolvente, altura, y timbre únicos. Cada muestra es un audio monofónico muestreado a 16000 Hz (con una profundidad de 16 bits PCM), correspondiente a uno de 1006 instrumentos de librerías comerciales de muestras musicales. El conjunto cubre todas las notas de un piano MIDI estándar (21-108), con 5 *velocidades* (intensidad de la nota) diferentes (25, 50, 75, 100, 127). El mismo se distribuye en conjuntos de entrenamiento, validación y prueba de la siguiente manera:

- **Entrenamiento**. Conjunto con 289205 audios. Los instrumentos utilizados no se solapan con los utilizados por los conjuntos de validación y prueba.
- **Validación**. Conjunto con 12678 audios. Los instrumentos utilizados no se solapan con los utilizados por los conjuntos de entrenamiento y prueba.
- **Prueba**. Conjunto con 4096 audios. Los instrumentos utilizados no se solapan con los utilizados por los conjuntos de entrenamiento y validación.

De manera análoga, para la generación de cada uno de los conjuntos de datos a utilizar en el presente trabajo, se utilizaron audios correspondientes al conjunto de datos que se debía generar. Es decir, para la generación del conjunto de entrenamiento se utilizaron audios del conjunto de entrenamiento de NSynth y, de igual manera, audios de los conjuntos de NSynth de validación y prueba para la generación de los conjuntos análogos.

Las Figuras 4.1, 4.2 y 4.3 muestran histogramas de los audios de los tres conjuntos de NSynth, en función de sus características más relevantes: *nota (MIDI)*, *velocidad*, *tipo de instrumento*, y *fuente*.

La estrategia para la creación del conjunto de datos de entrenamiento se basó en el aprendizaje por currículum (*curriculum learning*). Esto consiste en entrenar al modelo con una serie de conjuntos de datos progresivamente más complejos, lo cual posibilita que el aprendizaje suceda de manera más controlada y eficaz [60].

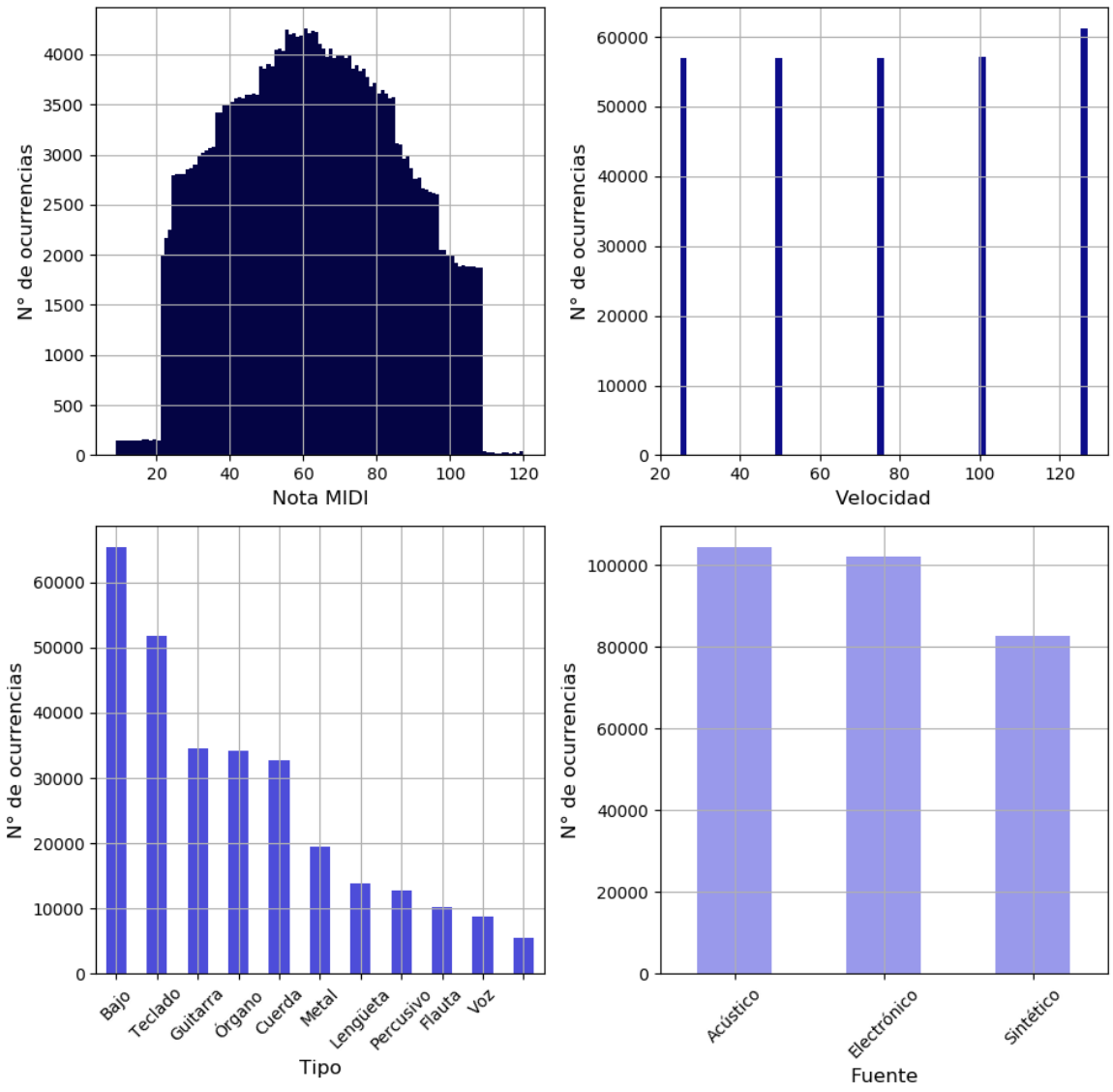

Figura 4.1: Histogramas del conjunto de entrenamiento de NSynth en función de sus principales características.

En función de esto, se crearon 6 conjuntos de datos para el entrenamiento, donde cada conjunto se diferenció por la cantidad de notas que conforman los audios del mismo. Para lograr esto, primero se seleccionó un audio del conjunto de NSynth correspondiente que contenga una nota específica. Luego, en base a la complejidad deseada, este audio se mezcló con una cierta cantidad de audios correspondientes a otras notas del mismo conjunto, seleccionadas al azar. El poder especificar una de las notas de cada audio permitió garantizar, a la hora de la creación del conjunto de datos, que las 88 notas posibles estén presentes al menos la cantidad deseada de veces dentro del mismo (en este caso, se determinó que cada nota apareciera al menos en 400 audios del conjunto de entrenamiento).

De esta forma, el primer conjunto de datos consistió de audios que cuentan con la presencia de una sola nota con un instrumento elegido al azar, mientras que el último consistió

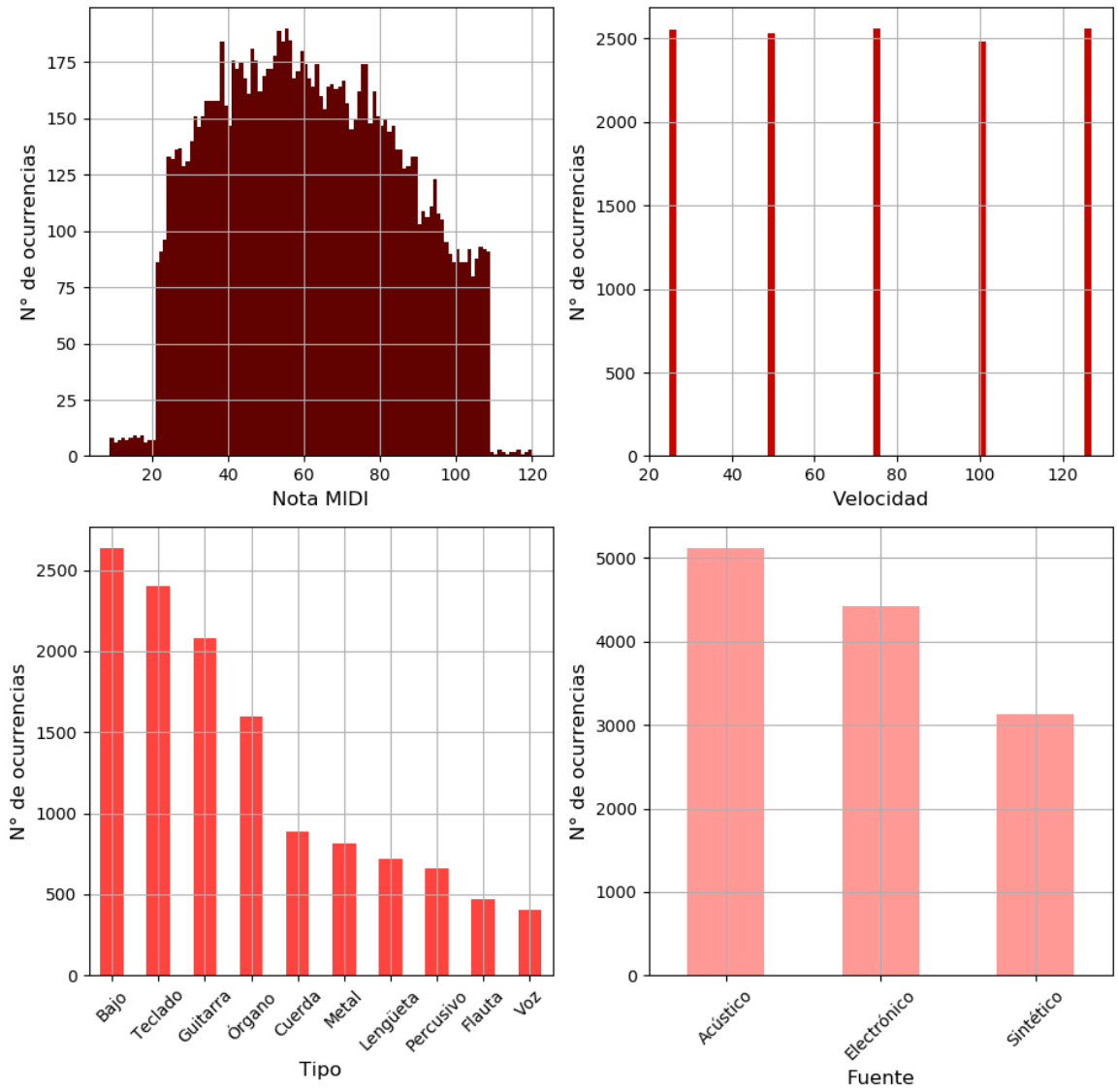

Figura 4.2: Histogramas del conjunto de validación de NSynth en función de sus principales características.

de audios en donde suenan 6 notas que pueden repetirse de manera simultánea en 6 instrumentos al azar. Todos los audios resultantes se normalizaron en una escala de amplitud de -1 a 1 previo a ser exportados. Los mismos fueron guardados utilizando una nomenclatura con estructura { id }_{ tipo }_{ nota({ valor RMS }) }.wav, donde:

- **Id**: Número entero único que representa el audio obtenido.
- **Tipo**: Tipo del instrumento correspondiente a la nota especificada.
- **Nota**: Índices de un vector que representa cada una de las 12 notas musicales: [C, C#, D, D#, E, F, F#, G, G#, A, A#, B].
- **Valor RMS**: Valor RMS (*Root Mean Square*) del audio correspondiente a la nota, con una precisión de 5 dígitos decimales. Representa una medida de su valor energético.

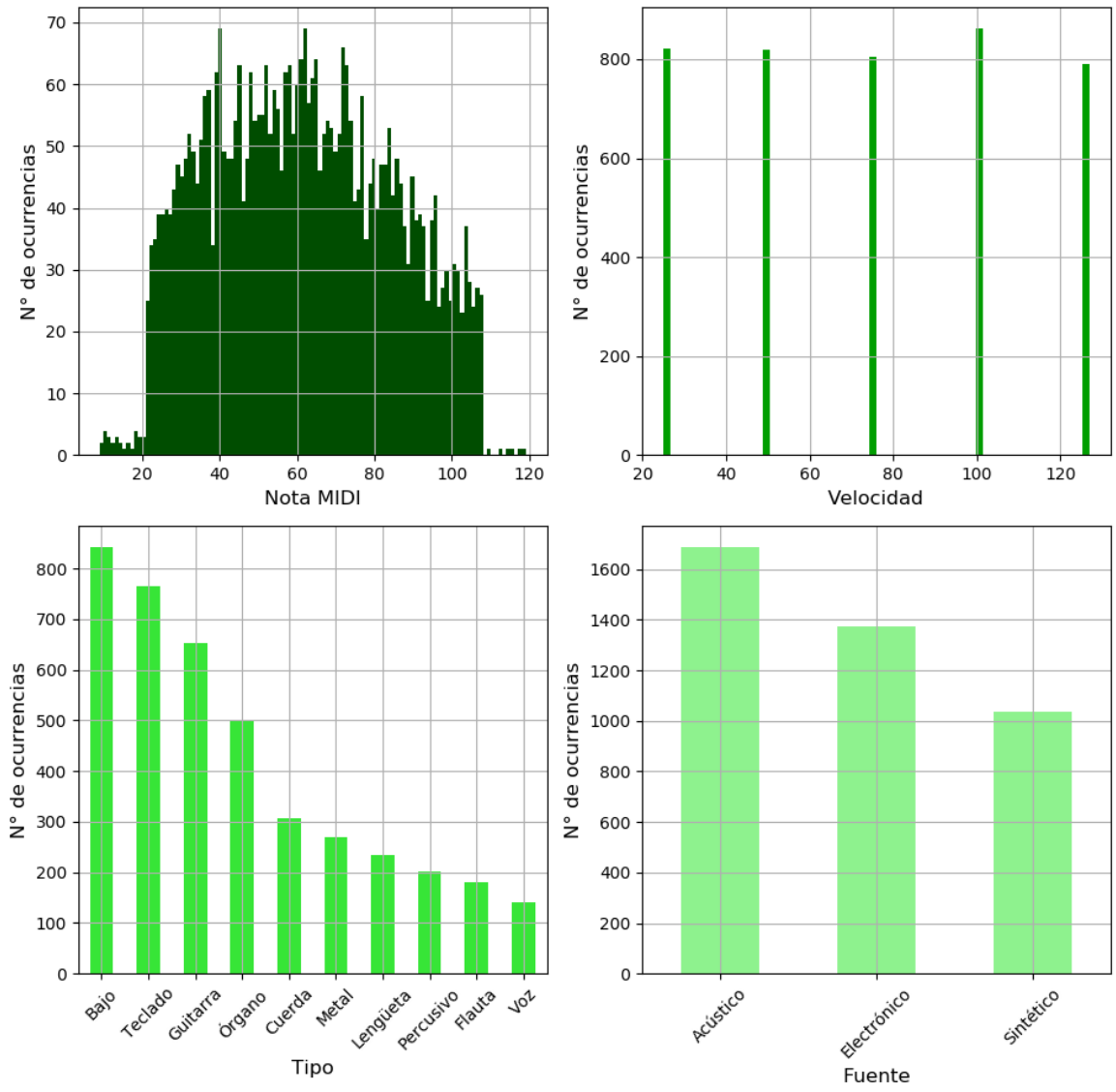

Figura 4.3: Histogramas del conjunto de pruebas de NSynth en función de sus principales características.

Asimismo, múltiples notas se separan con un guión (-). Por ejemplo, un archivo con el nombre *0_brass_8(0.54545)-10(0.43212)-4(0.52345)-0(0.23333).wav* tendría un id de 0, un timbre asociado a un instrumento de metal, y las notas *G#* con valor RMS de 0.54545, *A#* con valor RMS de 0.43212, *E* con valor RMS de 0.52345, y *C* con valor RMS de 0.23333.

Cada uno de los audios de NSynth consiste originalmente en 3 segundos de audio y 1 segundo de decaimiento y silencio. Esto podría ser problemático, ya que a cada audio se le asignará una sola etiqueta con las notas constituyentes del mismo. Por lo tanto, si las mismas no estuviesen presentes durante la totalidad del audio, entonces esta etiqueta no sería siempre válida. Asimismo, para la generación de un conjunto de datos dado, es deseable que exista la mayor variación posible en cuanto a las muestras del mismo, con el fin de cubrir la mayor cantidad de casos posibles. Es por esto que se determinó el uso de una mayor cantidad de muestras de audio de corta duración, siendo la duración de cada audio de 0.25

segundos.

En total se generaron 35200 audios de entrenamiento, 11733 de validación, y 7040 de prueba para cada uno de los 6 conjuntos de datos, totalizando de esta manera 211200 audios de entrenamiento, 70398 de validación y 42240 de prueba.

4.1.2 Obtención de espectrogramas

Para cada audio se calculó una CQT de resolución espectral de 25 valores por octava, a partir de una frecuencia mínima de 27.5 Hz (correspondiente a A0, la nota más grave en un piano tradicional), y un salto temporal entre columnas de 1024 muestras, resultando en una matriz de 175 x 4 valores reales por audio. La implementación particular de esta CQT está basada en el método de sub-muestreo recursivo [61], y fue computada mediante la biblioteca de Python LibROSA [1] [62]. Finalmente, para eliminar componentes espectrales no deseados, se eliminó la primera columna de esta CQT, descartando así la influencia espectral del transitorio inicial del audio.

El último paso en el pre-procesamiento de la información de audio, fue la normalización de los espectrogramas. Para ello, primero se transformaron los valores de amplitud a una escala de decibeles a escala completa (*dBFS*). Luego, se anularon los valores menores a un umbral de -40 dB respecto de la amplitud máxima (0 dB), descartando así la influencia de los mismos. Finalmente, los valores resultantes se trasladaron a una escala de valores reales entre 0 y 1.

La Figura 4.4 muestra los espectrogramas normalizados obtenidos a partir de audios de cada uno de los conjuntos de datos (previo al truncado de la primera columna de los mismos).

Finalmente, dado que por cada audio se extrajeron 3 muestras, el total de muestras de cada conjunto resultó ser:

- **Entrenamiento**: 633600 muestras.
- **Validación**: 211194 muestras.
- **Prueba**: 126720 muestras.

Aunque, como será detallado más adelante, la evaluación final del modelo no se realizó utilizando este conjunto de datos de prueba, sino que se utilizaron audios musicales no sintéticos en el marco de una tarea de reconocimiento de acordes, de manera que el mismo serviría solo como referencia.

[1] Más información en https://librosa.github.io/librosa/generated/librosa.core.cqt.html

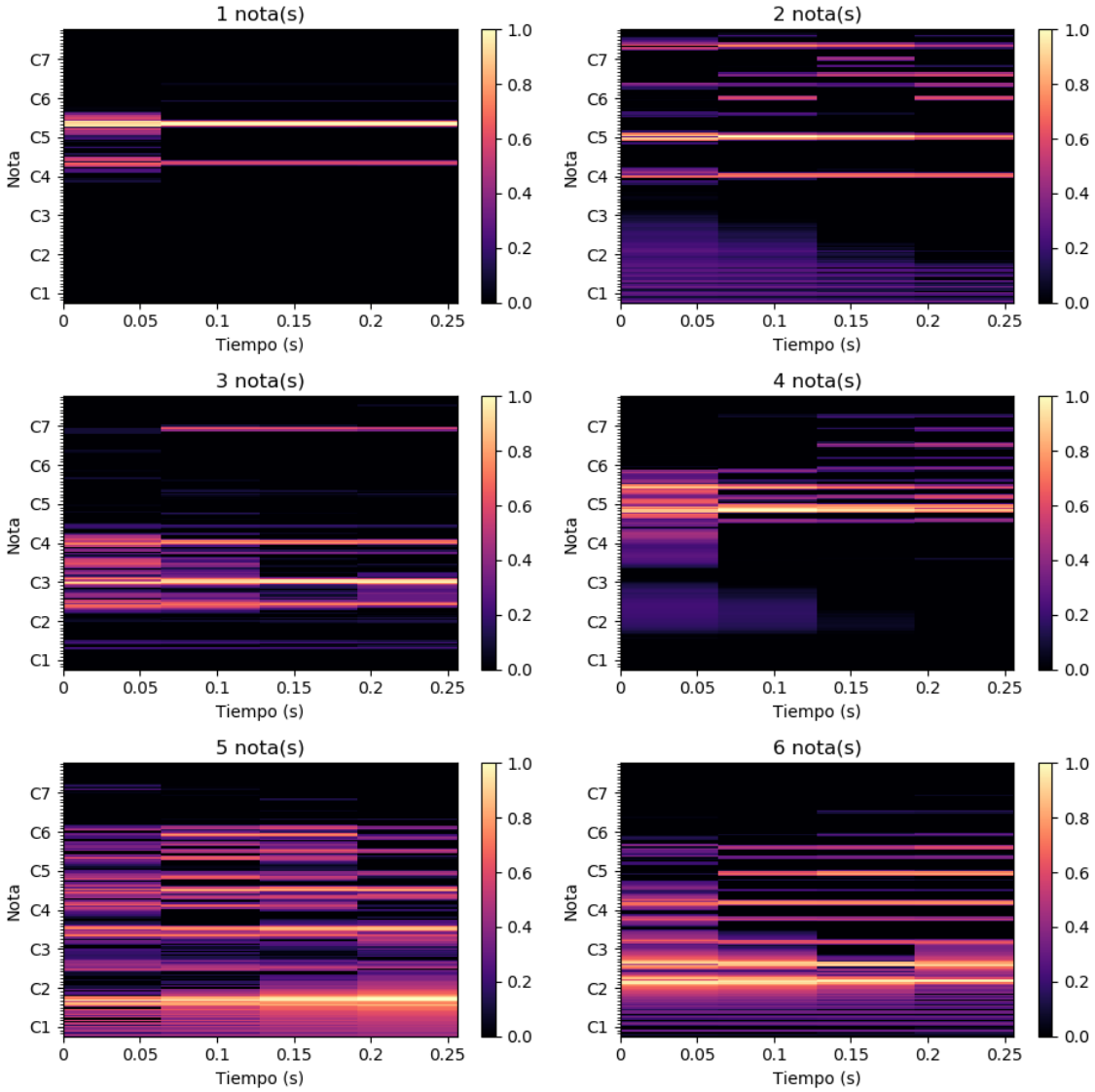

Figura 4.4: Espectrogramas (CQT) de 6 audios correspondientes a los distintos conjuntos de entrenamiento generados.

4.1.3 Definición de los valores de salida

El siguiente paso fue definir los valores de salida que se corresponderían a una entrada dada. Esto se llevó a cabo asociando, a partir del nombre del archivo, un vector de 12 elementos en donde cada uno de ellos estuviese asociado a una nota del alfabeto musical occidental, donde los elementos correspondientes a las notas presentes tendrían un valor unitario, mientras que el resto de los elementos sería nulo (0). Es decir, un vector croma binario.

Se definió que el primer elemento de este vector representara a la nota *C*. De esta manera, el audio del ejemplo anterior, constituido por las notas *G#*, *A#*, *E*, y *C*, estaría representado por el vector $[1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0]$, siendo que las notas correspondientes a

cada índice son $[C, C\#, D, D\#, E, F, F\#, G, G\#, A, A\#, B]$. Estos vectores se consideraron representativos de la totalidad del fragmento de audio, de manera que cada columna de la CQT se asoció al mismo vector croma.

4.1.4 Extracción de espectrogramas y valores de salida para el modelo entrenado con información energética

Una de las variantes propuestas fue la de entrenar un modelo que tuviera en cuenta los valores RMS de los audios a procesar. Para ello, debió modificarse el proceso de normalización, tanto de los espectrogramas como de los valores de salida.

Para los espectrogramas, mientras que en el caso anterior cada muestra se normalizaba de acuerdo al valor máximo del espectrograma (constituido por 3 muestras), ahora el modelo necesitaría ser capaz de aprender valores absolutos de amplitud o energía, por lo cual la normalización debería hacerse en base a una referencia 'global' o absoluta. Asimismo, en este tipo de modelos de redes neuronales es deseable que los valores de entrada estén en un rango de números reales entre 0 y 1 o -1 y 1.

En base a estos dos requerimientos, se buscó un valor de referencia para escalar los espectrogramas. Primero, se realizó un relevamiento de 1000 audios de cada uno de los subconjuntos del conjunto de datos de entrenamiento, totalizando 6000 audios. Estos audios constituyen 18000 muestras (columnas) de 175 valores cada una. Es decir, se tomaron 3150000 valores para analizar, los cuales se expresaron en una escala de dB (calculados utilizando una referencia arbitraria de 0.01).

A partir de estos valores se extrajo un histograma y se calcularon los percentiles más relevantes. Estos se pueden ver en la Tabla 4.1, mientras que el histograma se muestra en la Figura 4.5.

Tabla 4.1: Principales percentiles correspondientes a los valores observados en los espectrogramas de 6000 audios del conjunto de datos de entrenamiento.

P10	P25	P50	P75	P90	P95
-28.11	-10.84	3.60	14.99	23.84	28.88

En base a estos percentiles y a la forma de la distribución, se eligió el valor correspondiente a P90 como factor de escala (es decir, todos los valores se dividirían por este factor). Se tomó un percentil en lugar del valor máximo para disminuir la influencia de potenciales outliers en el escalado. De esta manera, se buscó que los valores escalaran a un rango cercano a $[-1, 1]$.

El mismo procedimiento se repitió para los valores RMS de salida, aunque en este caso se tomaron 30000 muestras del conjunto de audios con 6 notas, totalizando 180000 valores

RMS (1 por nota). Los percentiles e histogramas se pueden ver en la Tabla 4.2 y la Figura 4.5, respectivamente.

Tabla 4.2: Principales percentiles correspondientes a los valores RMS observados para 180000 notas distintas.

P10	P25	P50	P75	P90	P95
0.11	0.17	0.26	0.35	0.45	0.52

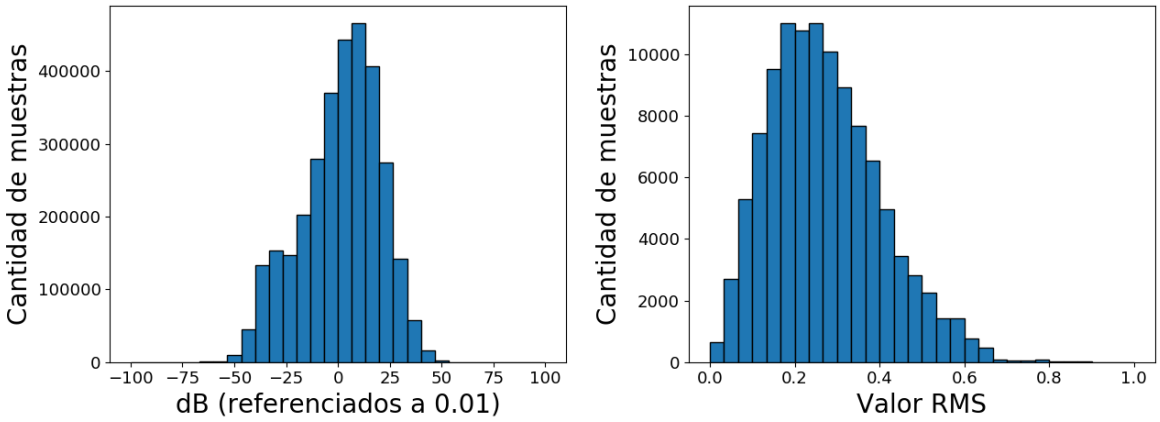

Figura 4.5: Histograma correspondiente a los valores observados en los espectrogramas de 6000 audios del conjunto de datos de entrenamiento (izquierda), e histograma correspondiente a los valores RMS observados para 180000 notas distintas (derecha).

De igual manera, en base a estos valores se eligió como factor de escala el valor correspondiente a P75.

4.2 ARQUITECTURA DEL MODELO

El modelo utilizado se basó en una arquitectura de Perceptrón Multicapa (*Multilayer Perceptron*, o MLP), que no es más que una serie de capas densamente conectadas. La estructura de la red se inspiró en el desarrollo realizado por Korzeniowski y Widmer [55], quienes utilizaron una red de arquitectura MLP con 3 capas de 512 neuronas cada una.

En este caso, el modelo se conformó por una capa de entrada que permite el ingreso de 175 valores (es decir, del tamaño de cada una de las muestras calculadas), seguida de 3 capas ocultas de 256 neuronas cada una. Intercaladas a estas capas se incorporaron 3 capas de *Dropout*, que reducen a 0 el 25 % de las salidas de la capa anterior correspondiente. Finalmente, se incorporó una capa de salida de 12 neuronas, correspondiente a las 12 notas posibles. Un esquema de la arquitectura utilizada puede verse en la Figura 4.6.

Se utilizó ReLU (Unidad Lineal Rectificada) como función de activación para las capas ocultas, mientras que para la capa de salida se definió una activación a través de una función

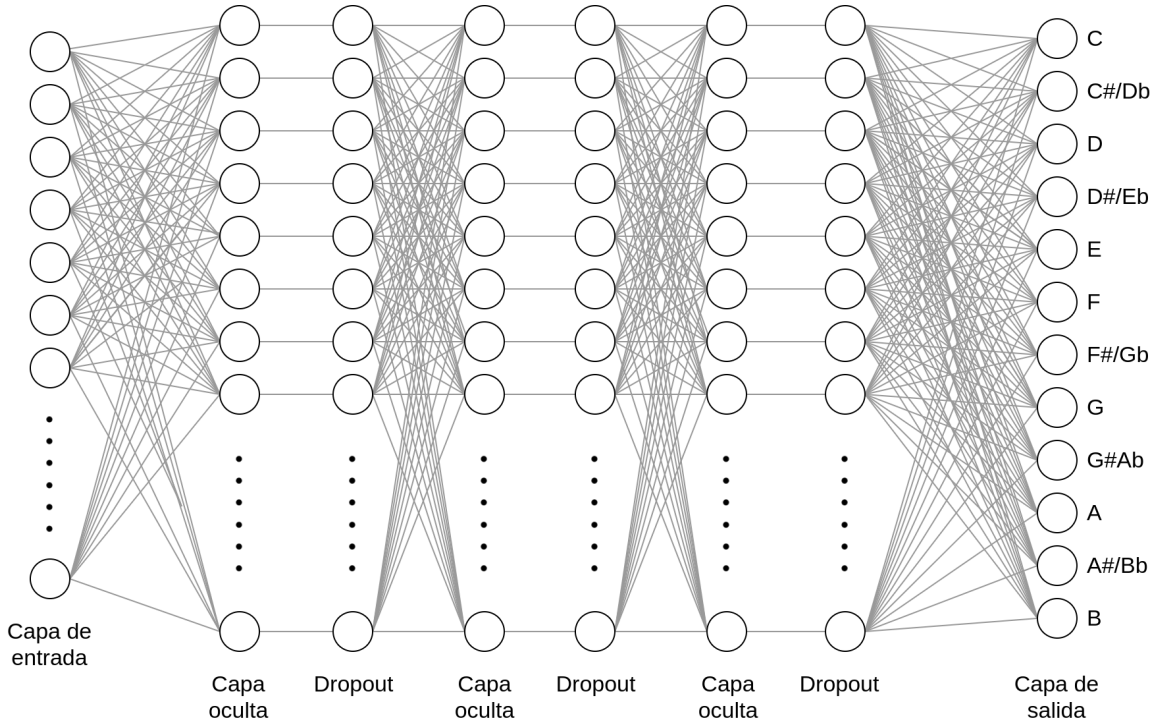

Figura 4.6: Arquitectura de la red neuronal utilizada.

sigmoide.

El código pertinente a la implementación del modelo fue desarrollado utilizando el lenguaje de programación Python (en su versión 3.7), junto con las bibliotecas de aprendizaje automático *Tensorflow* (versión 1.14.0) [63] y *Keras* [64].

4.3 ENTRENAMIENTO DEL MODELO

El entrenamiento se realizó por mini-lotes de distintos tamaños: se entrenaron variantes del modelo utilizando 256, 512, y 1024 muestras por mini-lote. Asimismo, se estableció un número de épocas de 150 para cada conjunto de datos de entrenamiento, con la condición de que el entrenamiento se detendría si no se observaba una mejora mayor a 0.001 en el valor de la función de costo aplicada sobre el conjunto de validación a lo largo de 50 épocas.

Para la optimización y ajuste del sistema se utilizó una variante del algoritmo de descenso por gradiente denominado Adam [65], con una tasa de aprendizaje de 0.001. Como función de costo, se utilizó la denominada entropía cruzada binaria, definida como:

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=0}^{N}(y * log(\hat{y}_i) + (1 - y) * log(1 - \hat{y}_i)) \tag{4.1}$$

donde y es el valor real (es decir, la etiqueta correspondiente al acorde), e $\hat{y}$ la predicción realizada por el modelo. Esta función evalúa cuán lejos del valor real (que es obligatoriamente

0 o 1) está la predicción, para cada una de las clases posibles (12 en este caso). Finalmente, excepto en los casos en donde se indique lo contrario, el entrenamiento se realizó de manera incremental mediante aprendizaje por currículum, pasando de manera secuencial del conjunto de datos con menor cantidad de notas (1) al de mayor cantidad de notas (6), siendo que, dentro de cada conjunto, las muestras individuales se ordenaban de manera aleatoria.

En total, se entrenaron 5 modelos con distintas variaciones:

- Modelo con tamaño de mini-lotes de 1024 muestras.
- Modelo con tamaño de mini-lotes de 512 muestras.
- Modelo con tamaño de mini-lotes de 256 muestras.
- Modelo con tamaño de mini-lotes de 1024 muestras sin aprendizaje por currículum (con los 6 conjuntos de datos de entrenamiento englobados en uno solo, con sus muestras ordenadas de manera aleatoria).
- Modelo con tamaño de mini-lotes de 1024 muestras, utilizando durante el entrenamiento vectores croma no binarios (es decir, que tienen en cuenta el valor RMS de la(s) nota(s) correspondientes).

4.4 EVALUACIÓN DEL MODELO

Una vez concluido el entrenamiento del modelo en todas sus variantes, el mismo debió ser evaluado. Si bien en otras aplicaciones podría evaluarse un modelo de características similares utilizando el conjunto de datos de prueba, se consideró que en este caso no bastaría con ello, puesto que:

- Si bien los instrumentos utilizados fueron distintos, los datos de prueba fueron generados de igual manera que los datos de entrenamiento y validación: utilizando audios constituidos por notas aisladas con ciertas características tímbricas, pero fuera de un contexto musical. Dado que el modelo sería utilizado con audios musicales no sintéticos, se consideró que la evaluación también debería de ser llevada a cabo utilizando datos de este tipo.
- Dado que el foco de este trabajo se pone sobre los cromagramas como elemento constituyente de sistemas de estimación automática de acordes, resulta sensato evaluar la utilidad del modelo desarrollado en el marco de una tarea de esta índole.

Es por esto que se decidió implementar una tarea simple de reconocimiento de acordes, utilizando como conjunto de datos, una porción de la discografía de la banda inglesa The Beatles.

4.4.1 Conjunto de datos para la evaluación

MIREX (*Music Information Retrieval Evaluation eXchange*) es una competencia realizada anualmente en donde se exponen y evalúan diversos sistemas y algoritmos relacionados al campo denominado Recuperación de Información Musical (*Music Information Retrieval*, o MIR). En el marco de la categoria de ACE, los conjuntos de datos para la evaluación se extraen principalmente de dos fuentes:

- **Isophonics**. Colección de datos del Centro para la Música Digital de la Universidad de Londres. Cuenta con transcripciones de acordes correspondientes a:
  - **The Beatles**. Como parte de su tesis doctoral, Christopher Harte llevó a cabo la transcripción de acordes y su correspondiente verificación para los 12 discos oficiales de estudio de The Beatles, así como también la definición de la sintaxis utilizada para definir los acordes [56]. Estas transcripciones fueron revisadas en múltiples ocasiones, tanto por su autor como por la comunidad dedicada a MIR, por lo cual son de alta fiabilidad [38].
  - **Zweieck**. Transcripciones del album *Zwielicht*. Confianza moderada en la precisión de las transcripciones.
  - **Queen**. Transcripciones del los albumes *Greatest Hits I* y *Greatest Hits II*. Confianza moderada en la precisión de las transcripciones.
  - **Carole King**. Transcripciones del album *Tapestry*. No han sido revisadas rigurosamente, por lo cual deben utilizarse con cuidado.
- **Billboard**. Este conjunto de datos se divide en dos partes: una de acceso público, y otra que se reserva para la evaluación interna en MIREX. Actualmente la parte pública consiste en transcripciones de 740 canciones distintas que han aparecido en el ranking de Billboard a lo largo de los años [10].

En este trabajo se optó por la utilización del conjunto de datos de Isophonics, en particular la discografía de The Beatles. Se decidió no utilizar los conjuntos de Queen, Zweieck, y Carole King debido a su media-baja fiabilidad, mientras que se desistió de utilizar el conjunto de Billboard debido a que, dado que las canciones que conforman el mismo están en su mayoría fuera del dominio público, la recopilación de los audios correspondientes resulta impráctica y dificultosa. Asimismo, dado que estos audios solo se utilizarían para la evaluación del sistema, el volumen de datos requerido no es tan grande.

El audio utilizado para las transcripciones se corresponde con la tirada original de cada disco en formato CD, emitida por primera vez en 1987. Esta colección consiste en 180 canciones a lo largo de 13 discos (el disco titulado 'The Beatles' es un disco doble), totalizando

8 horas 8 minutos y 53 segundos de audio (29333 segundos en total). La duración de cada disco puede verse en la tabla 4.3, así como también el código que identifica a la edición en particular.

Tabla 4.3: Discos que conforman el conjunto de datos, con su duración total y código de catálogo.

N°	Album	Código	Duración(s)	Duración(mm:ss)
01	*Please Please Me*	CDP 7 46435 2	1965.84s	32:45
02	*With the Beatles*	CDP 7 46436 2	2004.32s	33:24
03	*A Hard Day's Night*	CDP 7 46437 2	1830.01s	30:30
04	*Beatles for Sale*	CDP 7 46438 2	2053.41s	34:13
05	*Help!*	CDP 7 46439 2	2061.06s	34:21
06	*Rubber Soul*	CDP 7 46440 2	2148.1s	35:48
07	*Revolver*	CDP 7 46441 2	2099.3s	34:59
08	*Sgt. Pepper's Lonely Hearts Club Band*	CDP 7 46442 2	2390.57s	39:50
09	*Magical Mistery Tour*	CDP 7 48062 2	2209.88s	36:49
10	*The Beatles (CD1)*	CDS 7 46443 8	2781.91s	46:21
11	*The Beatles (CD2)*	CDS 7 46443 8	2834.08s	47:14
12	*Abbey Road*	CDP 7 46446 2	2844.08s	47:24
13	*Let It Be*	CDP 7 46447 2	2110.88s	35:10

Las transcripciones utilizadas consisten en archivos de texto (uno por canción) en donde cada fila consiste en un tiempo inicial y final que definen un intervalo, junto con una etiqueta correspondiente al acorde presente durante dicho intervalo. Por ejemplo:

$$
\begin{aligned}
&0{,}000000 \quad 2{,}612267 \quad N\\
&2{,}612267 \quad 11{,}459070 \quad E\\
&11{,}459070 \quad 12{,}921927 \quad A\\
&12{,}921927 \quad 17{,}443474 \quad E\\
&17{,}443474 \quad 20{,}410362 \quad B\\
&20{,}410362 \quad 21{,}908049 \quad E\\
&21{,}908049 \quad 23{,}370907 \quad E:7/3\\
&23{,}370907 \quad 24{,}856984 \quad A\\
&24{,}856984 \quad 26{,}343061 \quad A:min/b3
\end{aligned}
$$

Son las anotaciones correspondientes a los primeros 26 segundos de la canción *I Saw Her Standing There* del álbum *Please Please Me*. Nótese la presencia de una etiqueta 'N'; la

misma hace referencia a intervalos en donde no está presente ningún acorde (en este caso debido a que hay un silencio al principio de la canción).

Se encontró cierta dificultad al obtener la edición específica del álbum *The Beatles* a los cuales las anotaciones hacen referencia, por lo cual ciertas canciones del conjunto de datos tuvieron que ser descartadas para su utilización, ya que su duración no coincidía con la especificada en las anotaciones. Las mismas son *Wild Honey Pie*, *The Continuing Story of Bungalow Bill*, *Cry Baby Cry*, *Revolution 9*, *Rocky Raccoon* y *Don't Pass Me By*.

4.4.2 Tarea de reconocimiento de acordes propuesta

Una vez establecido el conjunto de datos para la evaluación, se procedió a crear un sistema simple que realice el reconocimiento de acordes a partir de las predicciones del modelo.

El mismo consiste en tomar cada uno de los vectores del cromagrama de salida y calcular la distancia euclídea del mismo con respecto a plantillas preestablecidas. Por ejemplo, dado un vector croma de la forma

$$[0{,}55, 0{,}71, 0{,}22, 0{,}11, 0{,}35, 0{,}03, 0{,}43, 0{,}22, 0{,}12, 0{,}78, 0{,}90, 0{,}33]$$

y un vector de la plantilla, correspondiente al acorde *C mayor*,

$$[1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]$$

la distancia euclídea entre ellos estará dada por:

$$\sqrt{\sum_{n=1}^{N}(x_i - y_i)^2} \tag{4.2}$$

En (4.2) x_i e y_i son los elementos correspondientes al vector de salida y al vector de la plantilla, respectivamente. En este caso, el resultado sería 1.88.

En base a esto, se calcula la distancia euclídea del vector de salida contra todos los vectores de la plantilla, y se determina que el acorde predicho es aquel respecto del cual la distancia es menor. Esto se realiza para cada uno de los vectores obtenidos a través del modelo para el total de cada una de las canciones del conjunto de datos.

Las plantillas utilizadas están constituidas por todas las tríadas posibles. Es decir, se consideran:

- Acordes de tríada mayor
- Acordes de tríada menor
- Acordes suspendidos (sus2 y sus4)

- Acordes de sexta aumentada (aug)
- Acordes de tríada disminuidos (dim)

Es muy probable que el resultado de asignar un acorde a cada ventana de manera independiente resultase en una predicción con grandes variaciones, debido a que los vectores croma se asocian a un intervalo relativamente corto de tiempo, producto del tamaño de 1/16 de segundo de la ventana utilizada en el cálculo de los espectrogramas. Es por ello que se aplicó un filtro de moda móvil a la predicción de acordes, con el fin de 'suavizar' el resultado final. Para este filtro se realizaron pruebas con un tamaño de ventana de 1, 10, 20, y 30 muestras, siendo 1 el caso donde no se aplica suavizado, tomando el tamaño que diera mejores resultados.

Por ejemplo, la Figura 4.7 muestra un diagrama de acordes a través del tiempo (comúnmente denominado *chordgram*), donde se observan dos secuencias: la primera sin suavizar, y la segunda una vez aplicado un suavizado con una ventana de 10 muestras; al principio y al final de la secuencia se toma un número de muestras progresivamente menor, siendo 5 la cantidad de muestras más baja.

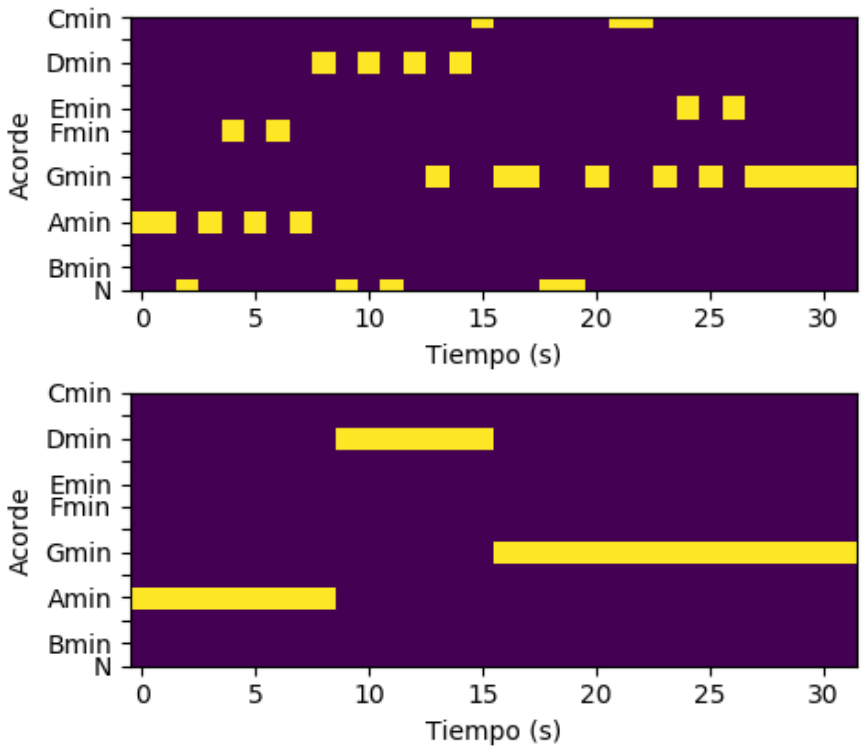

Figura 4.7: Secuencia de acordes sin suavizado (arriba), y con suavizado (abajo), utilizando una ventana de 10 muestras.

Finalmente, a partir de estas predicciones se calcula el WCSR, que sirve como métrica final para la evaluación del modelo. El mismo se calcula utilizando tres criterios de comparación: tónica (ROOT), acordes mayores y menores (MAJMIN), y tríadas (TRIADS).

Dado que este sistema de reconocimiento de acordes a partir de cromagramas es muy simple en relación a sistemas modernos que cumplen la misma función, para establecer una referencia se calcularon también cromagramas a través de métodos tradicionales de procesamiento digital de señales (utilizando nuevamente funcionalidades de la biblioteca LibROSA [2]), con el fin de comparar los resultados finales obtenidos. De aquí en adelante estos cromagramas serán denominados cromagramas de referencia.

Asimismo, con el fin de mejorar el rendimiento del sistema de reconocimiento, se realizaron pruebas eliminando los componentes percusivos de los espectrogramas utilizados. Esto se logró mediante un algoritmo de HPSS, cuya implementación se basa en [19] y [20].

[2] https://librosa.github.io/librosa/generated/librosa.feature.chroma_cqt.html

CAPÍTULO 5: RESULTADOS

5.1 RESULTADOS DEL ENTRENAMIENTO

La figura 5.1 muestra las curvas de aprendizaje para el modelo entrenado utilizando mini-lotes de 1024 muestras y aprendizaje por currículum, para cada uno de los conjuntos de datos de entrenamiento. En las mismas se observa la convergencia del error en los conjuntos de validación con respecto a las épocas de entrenamiento. Tal como se hubiese supuesto, el error mínimo al cual convergen las curvas incrementa junto con la complejidad de los conjuntos de datos de entrenamiento (recordando que el primer conjunto consistio en audios de una sola nota, y el sexto de seis).

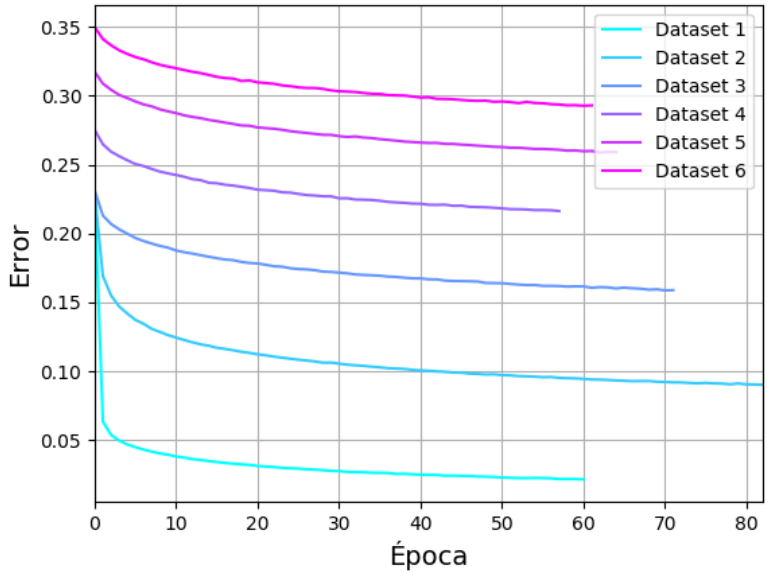

Figura 5.1: Curvas de aprendizaje para el primer modelo, para cada uno de los conjuntos de datos de entrenamiento.

5.2 CROMAGRAMAS OBTENIDOS

La Figura 5.2 muestra una comparación entre cromagramas obtenidos a través del modelo, utilizando un tamaño de mini-lotes de 1024 muestras y aprendizaje por currículum, y cromagramas de referencia.

En los mismos ya se puede apreciar una mayor fidelidad y claridad en los cromagramas

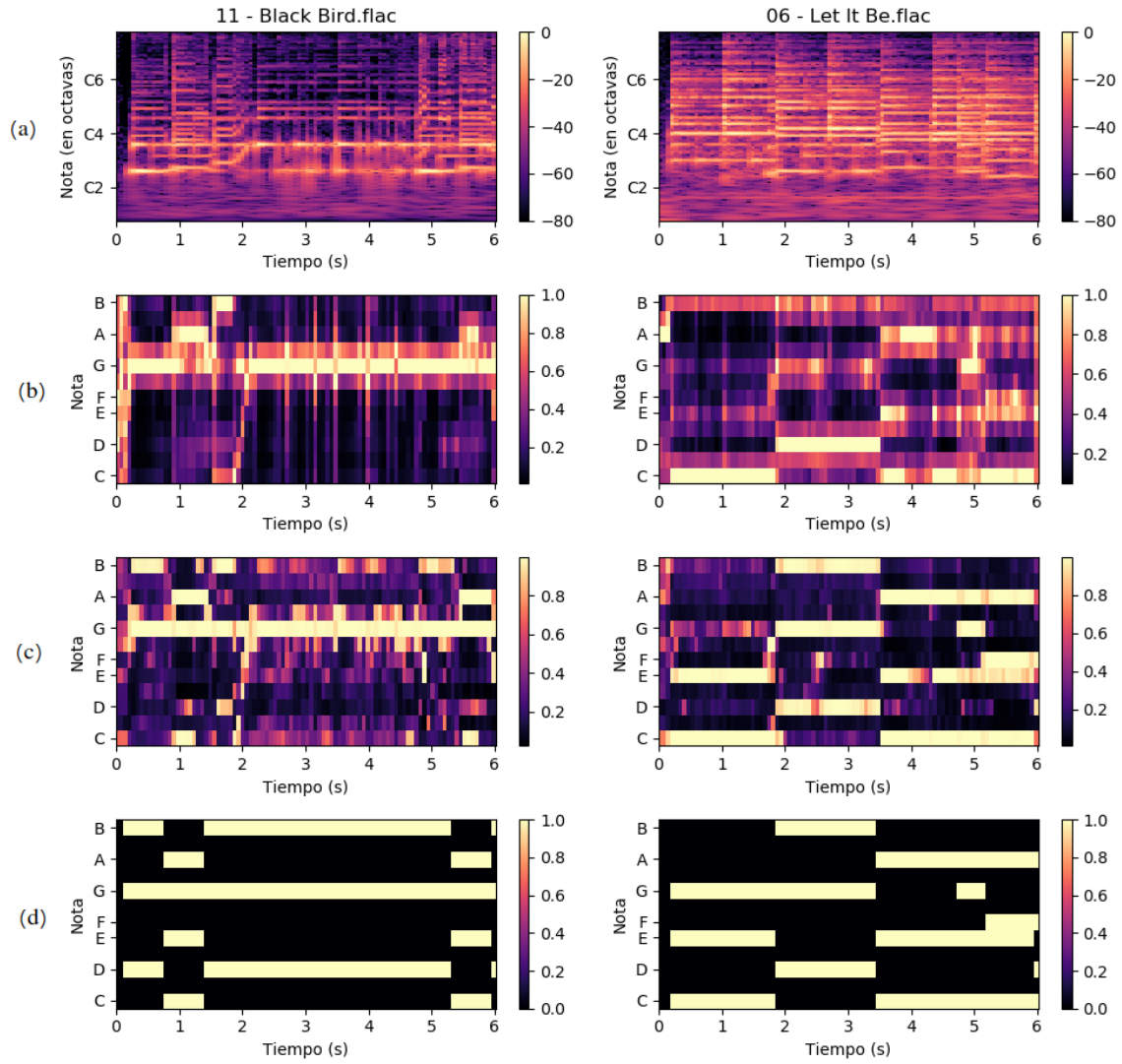

Figura 5.2: Espectrogramas con amplitud en dBFS (a), cromagramas de referencia (b), cromagramas extraídos por el modelo (c) y cromagramas 'ideales' construidos a partir de las etiquetas del conjunto de datos de evaluación (d), para los primeros 6 segundos de las canciones *Let It Be* y *Black Bird*.

extraídos por el modelo. No obstante, debe tenerse en cuenta que los cromagramas "ideales" no necesariamente representan lo que sucede a nivel melódico en el audio, sino que describen la estructura armónica subyacente. Por lo tanto, no necesariamente estarán presentes, en un intervalo de audio, todas las notas de un acorde dado por una etiqueta, y ambos cromagramas reflejarán esto.

La Figura 5.3 muestra los cromagramas obtenidos con el modelo entrenado con valores RMS de las notas, comparados contra los obtenidos por el modelo entrenado sin la utilización de estos valores; ambos con tamaño de mini-lotes de 1024 muestras y aprendizaje por currículum.

Cabe destacar que, en el caso de este modelo, los espectrogramas que se utilizan como entrada se normalizaron utilizando un factor de escala fijo tal como se hizo en el entrena-

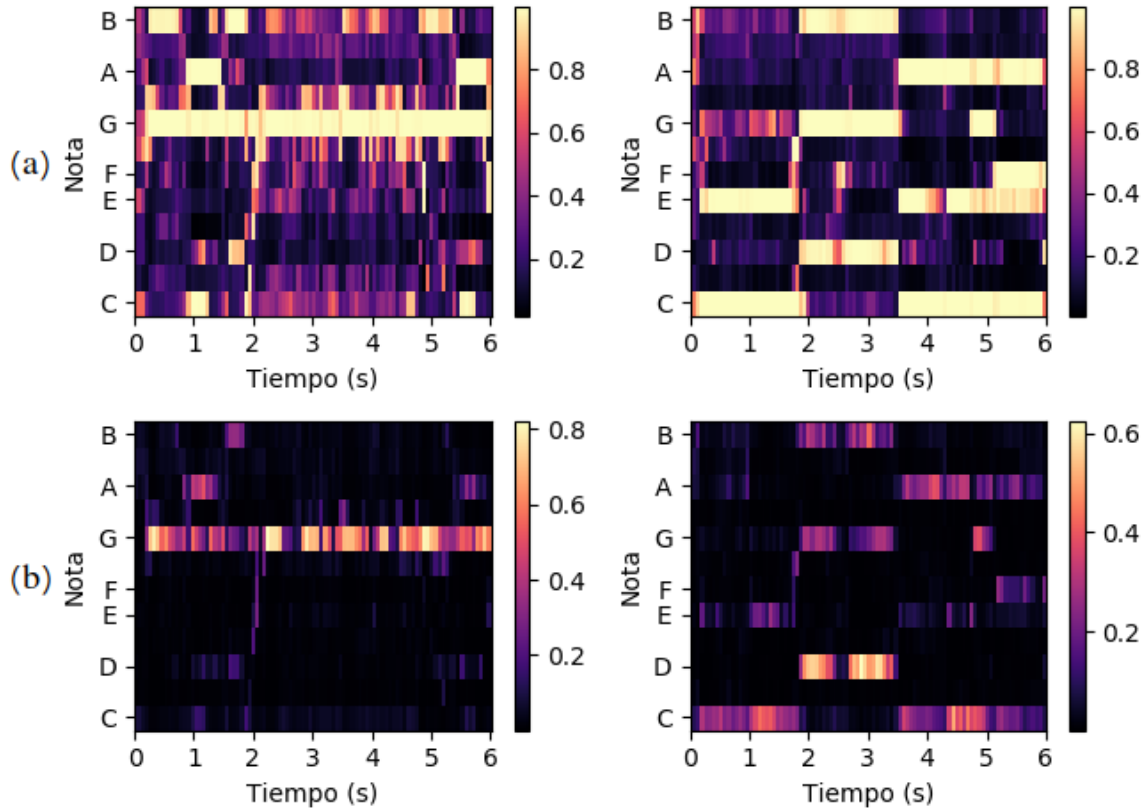

Figura 5.3: Cromagramas del modelo entrenado sin valores RMS (a) y con valores RMS (b), para los primeros 6 segundos de las canciones *Let It Be* y *Black Bird*.

miento, utilizando el valor correspondiente al P90 obtenido sobre una porción del conjunto de entrenamiento.

No obstante, dado que este valor se obtuvo a partir de un conjunto de datos radicalmente distinto a audios musicales reales, el uso del mismo factor de escala puede no ser adecuado. Es por esto que se experimentó utilizando diversos valores, siendo estos los correspondientes a los percentiles P50 y P75, extraídos bajo las mismas condiciones que el P90 (ver Tabla 4.1). Los cromagramas obtenidos pueden verse en la Figura 5.4.

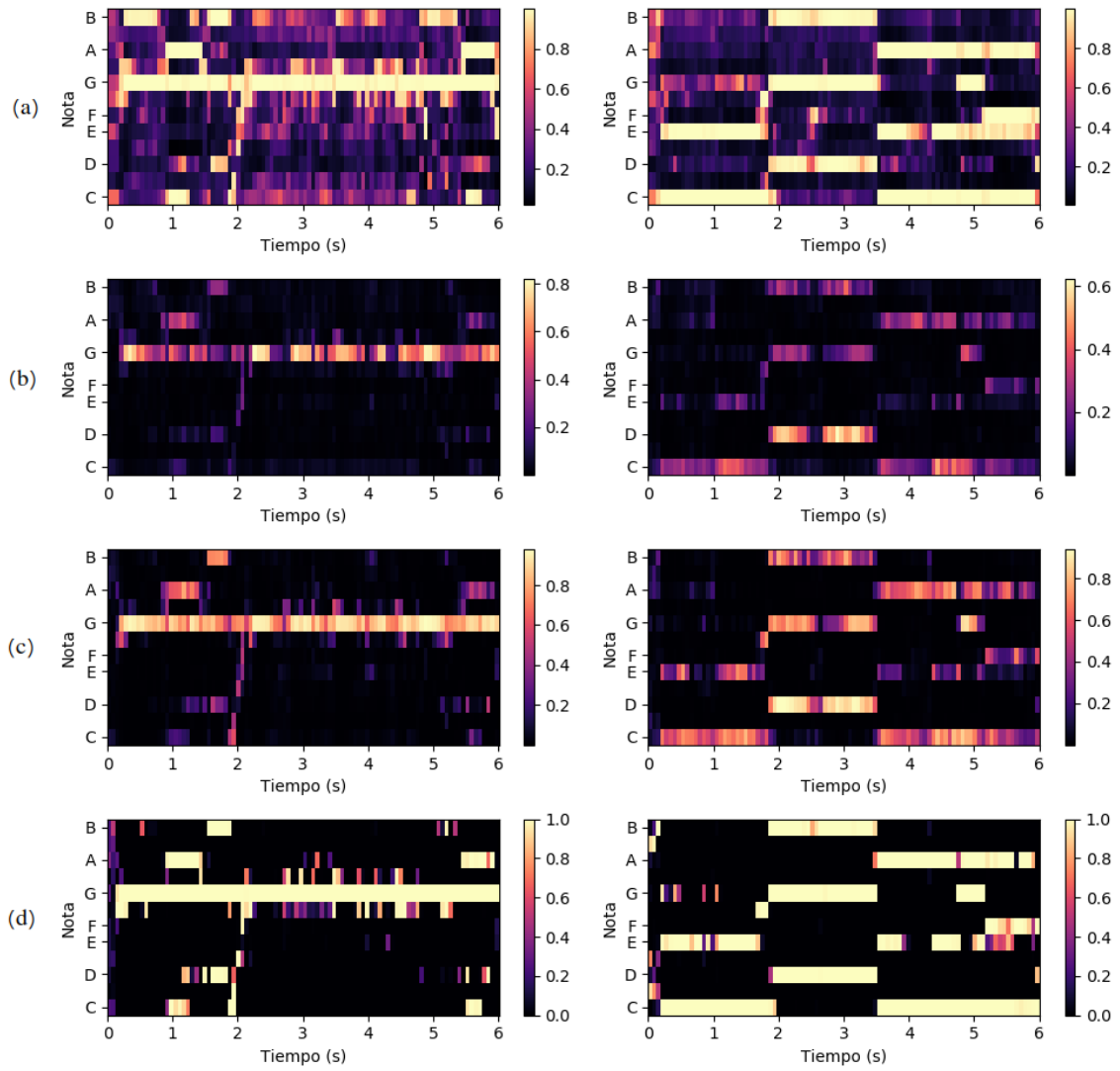

Figura 5.4: Cromagramas del modelo entrenado sin valores RMS (a) y con valores RMS, utilizando un factor de escala de 23.84 (b), 14.99 (c) y 3.60 (d), para los primeros 6 segundos de las canciones *Let It Be* y *Black Bird*.

5.3 PREDICCIONES REALIZADAS

La Figura 5.5 muestra los resultados de WCSR obtenidos a partir de los cromagramas extraídos, utilizando modelos con los tres tamaños de mini-lotes para cada conjunto de reglas utilizado, haciendo uso de distintas cantidades de muestras al aplicar el filtro de moda móvil a las predicciones originales. Los tres modelos se entrenaron por currículum, tal como se describió en la sección anterior.

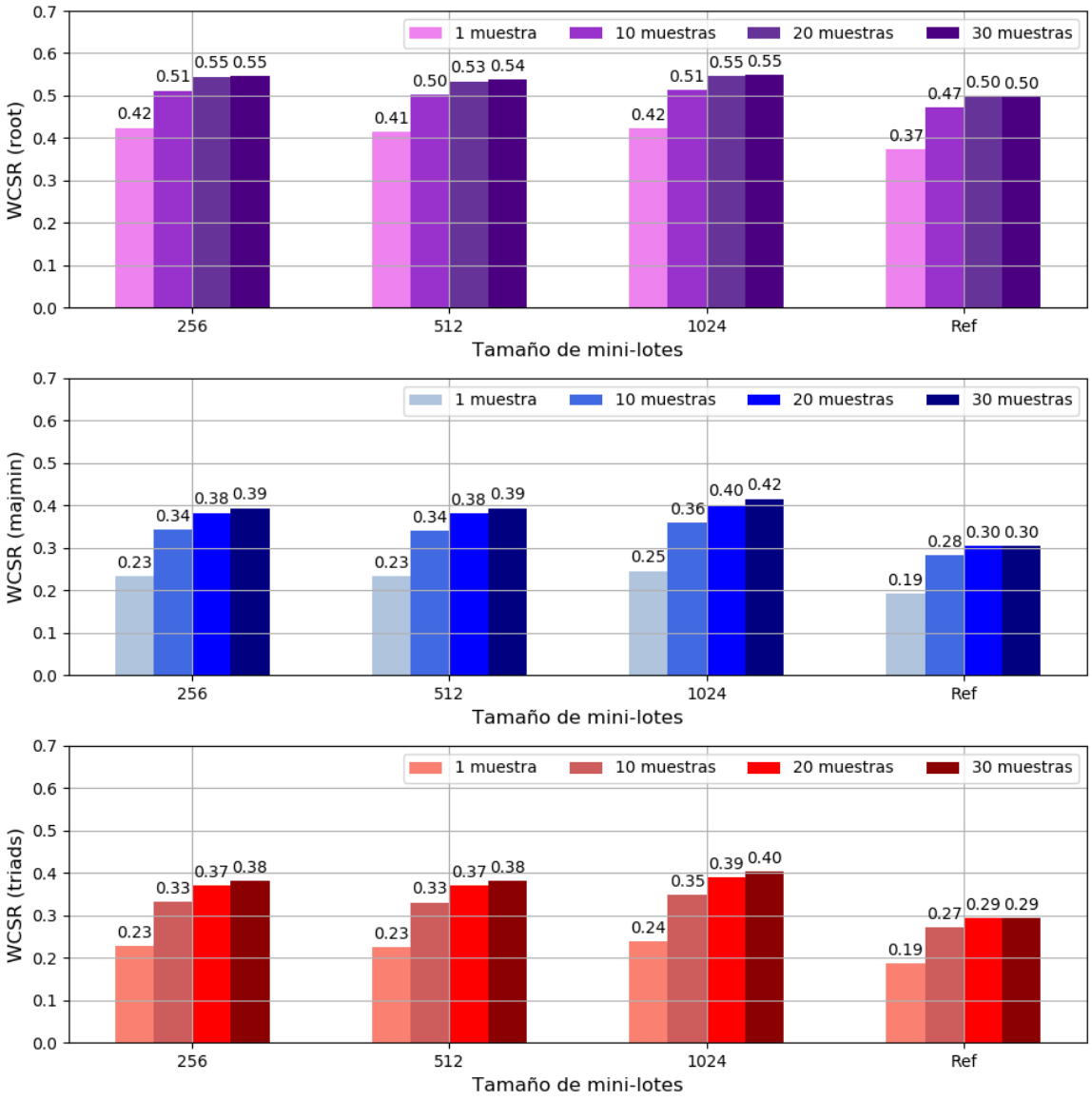

Figura 5.5: Resultados obtenidos de WCSR con modelos entrenados utilizando aprendizaje por currículum para tres tamaños de mini-lotes distintos, variando el número de muestras en el filtro de moda móvil (MM) de postprocesado de etiquetas.

En base a estos resultados, de aquí en adelante, se tomó el modelo entrenado por currículum con un tamaño de mini-lotes de 1024 muestras como modelo *base* para posteriores

comparaciones. Asimismo, para todos los modelos mencionados, el entrenamiento se realizó utilizando un tamaño de mini-lotes de 1024 muestras.

La Figura 5.6 muestra la comparación entre los puntajes obtenidos utilizando modelos sin y con el uso de aprendizaje por currículum.

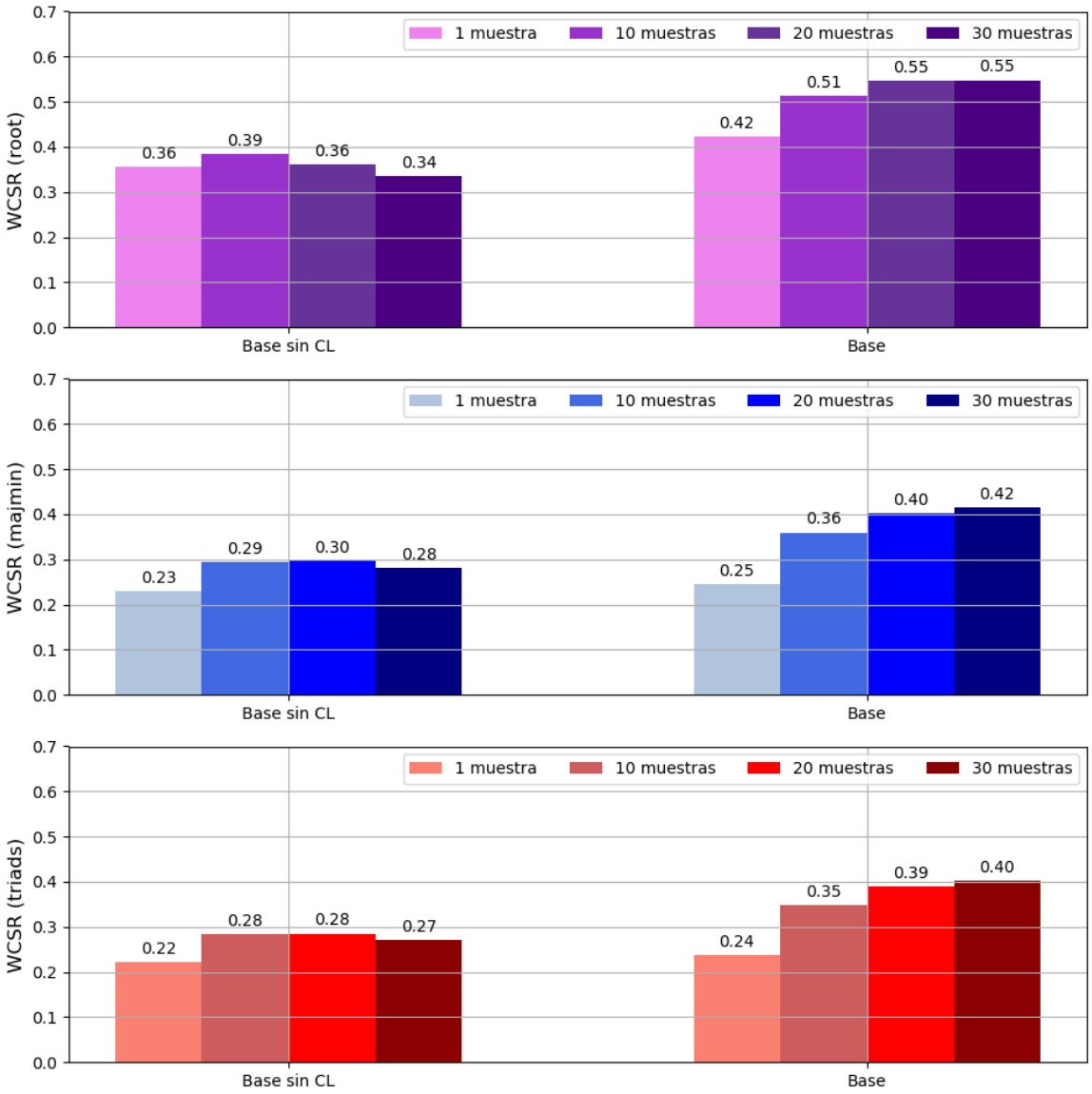

Figura 5.6: Resultados obtenidos de WCSR sin y con el uso de aprendizaje por currículum (CL), variando el número de muestras en el filtro de moda móvil (MM).

Respecto a los resultados obtenidos con y sin la eliminación de los componentes percusivos, la Figura 5.7 muestra los resultados obtenidos utilizando este pre-procesamiento.

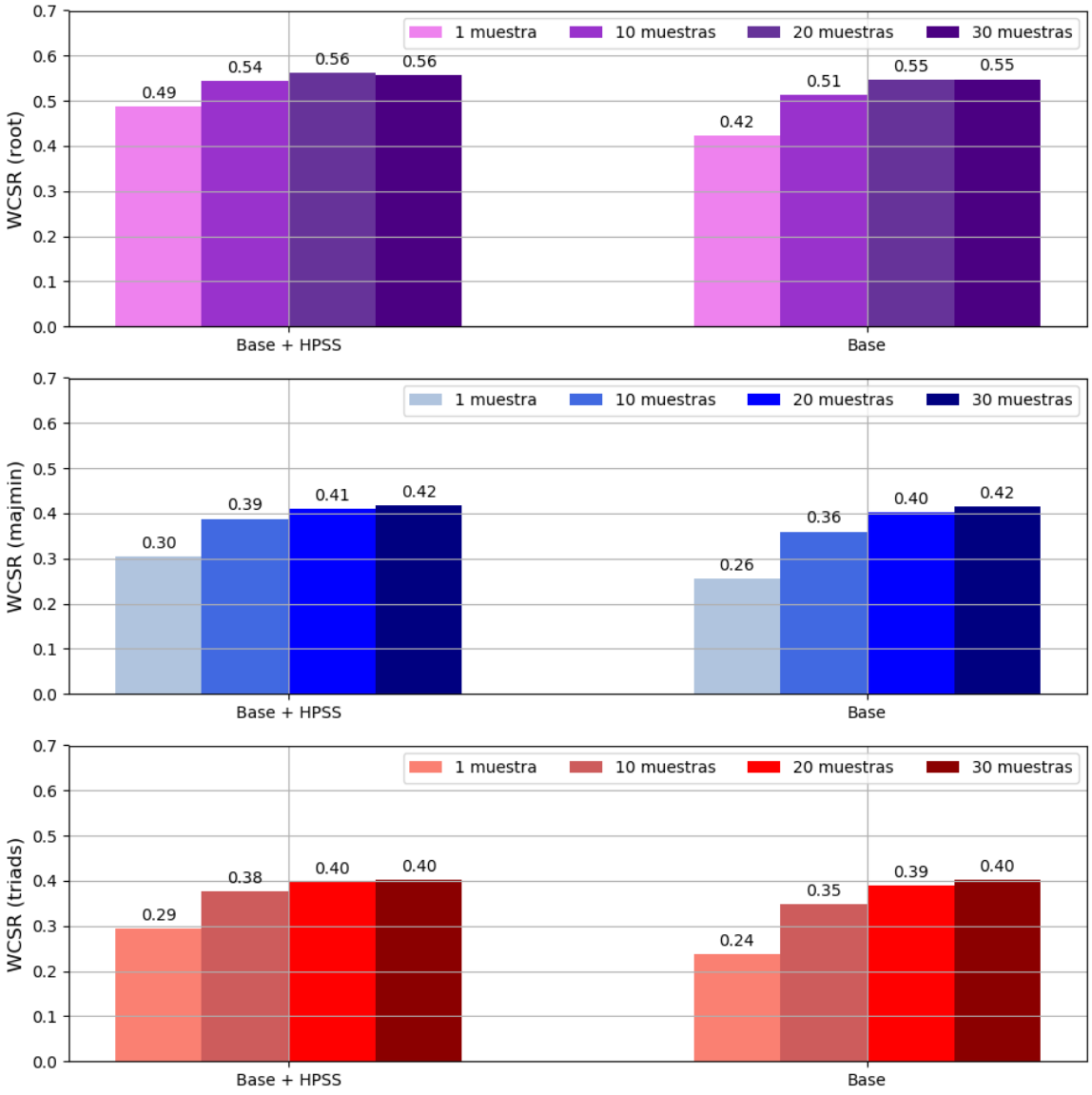

Figura 5.7: Resultados obtenidos de WCSR con y sin la eliminación de componentes percusivas, variando el número de muestras en el filtro de moda móvil (MM).

Finalmente, se presentan resultados obtenidos a partir del modelo entrenado con valores RMS. La Figura 5.8 muestra los valores de WCSR para varios factores de escala aplicados a los espectrogramas de entrada y eliminando las componentes percusivas.

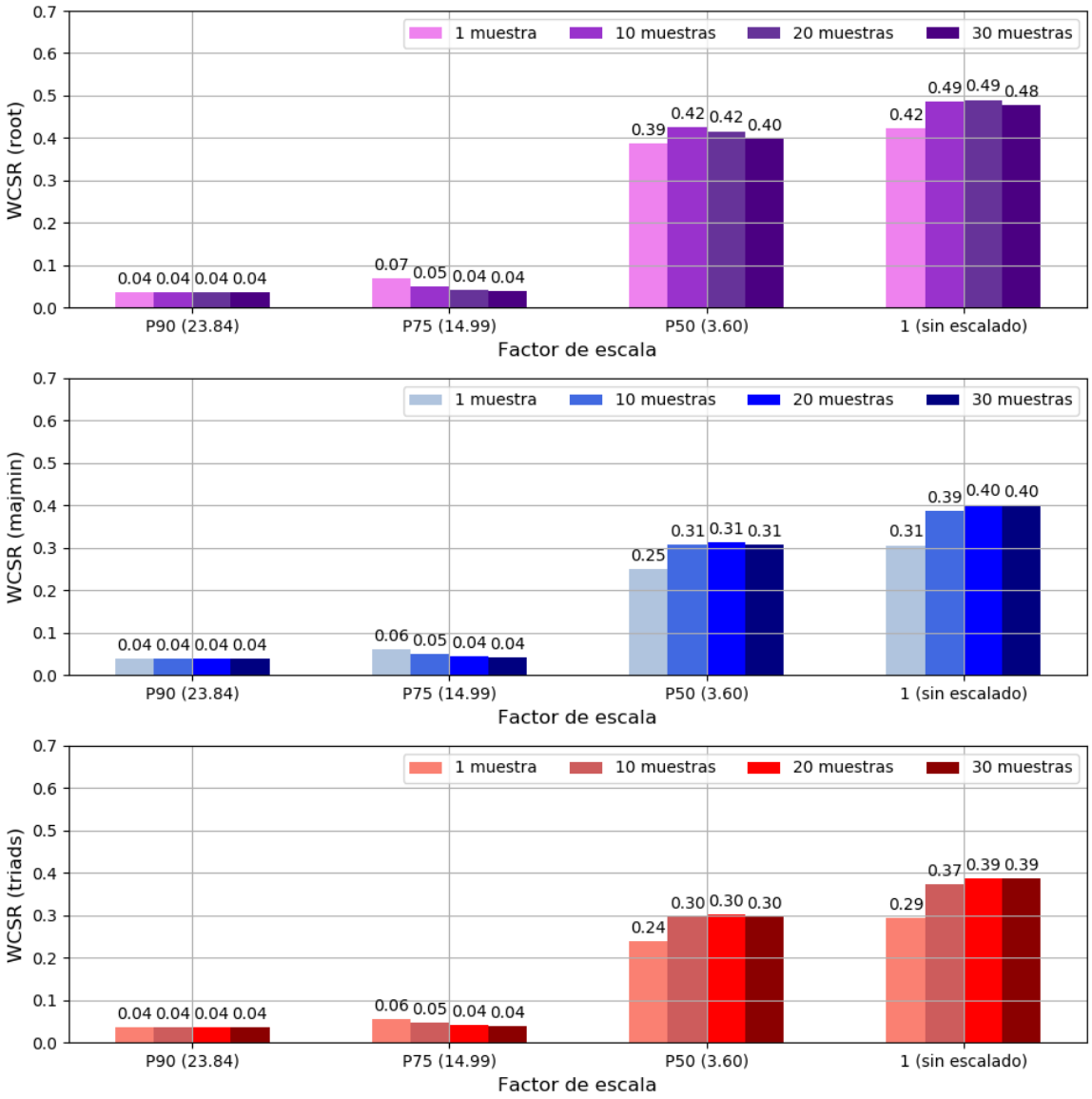

Figura 5.8: Resultados obtenidos de WCSR a partir del modelo entrenado con valores RMS utilizando distintos factores de escala y eliminando componentes percusivas, con varios tamaños de muestras en el filtro de moda móvil (MM) utilizado para el postprocesado de las etiquetas.

En base a la mejora obtenida con factores de escala más pequeños, también se evaluó este último modelo con factores menores a 1, hasta que la mejora dejase de ser significativa (<0.005). Los resultados se pueden ver en la Figura 5.9

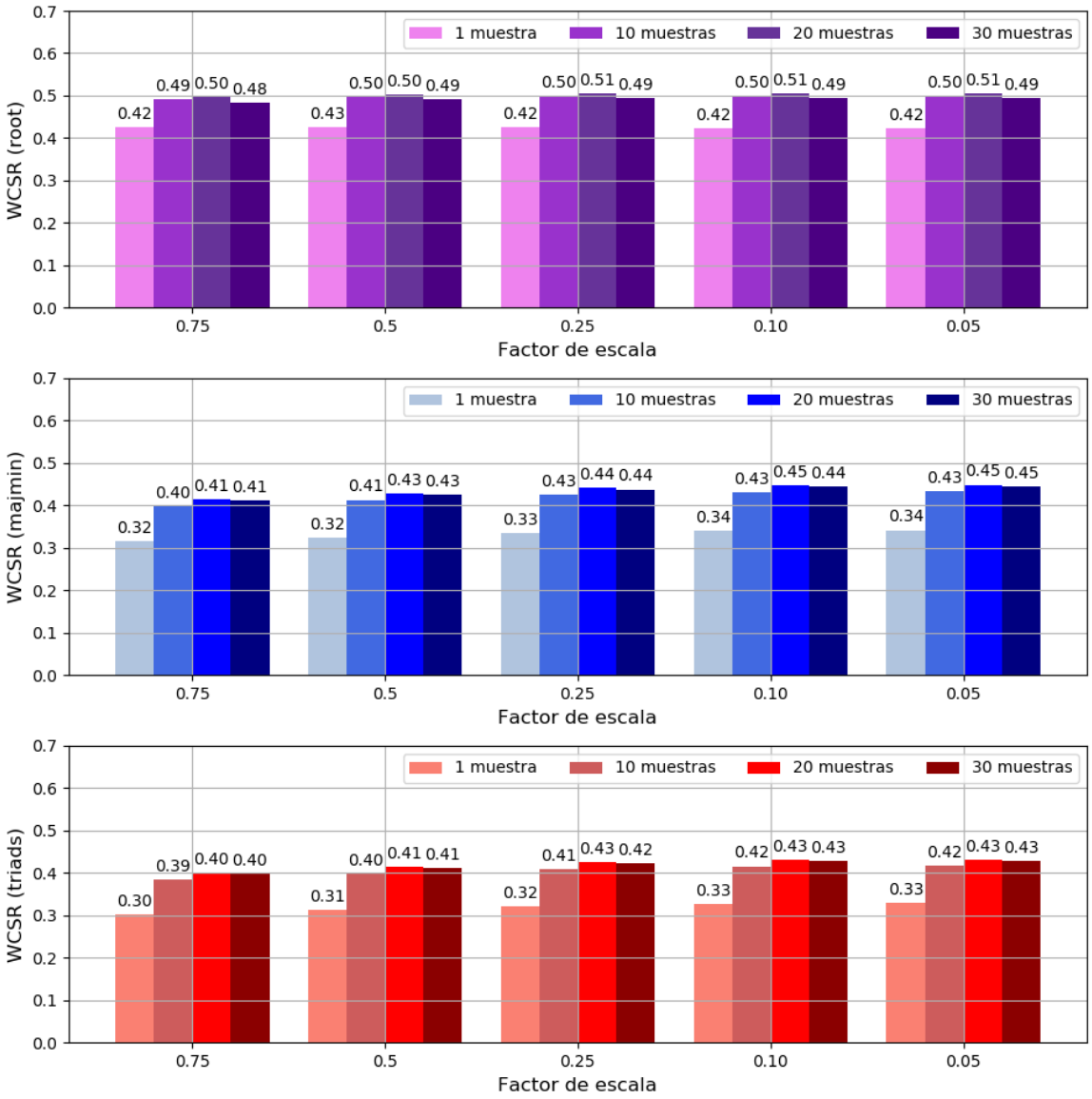

Figura 5.9: Resultados obtenidos de WCSR a partir del modelo entrenado con valores RMS utilizando factores de escala menores a 1 y eliminando componentes percusivas, con varios tamaños de muestras en el filtro de moda móvil (MM) utilizado para el postprocesado de las etiquetas.

La Figura 5.10 resume estos resultados en un gráfico de factor de escala contra WCSR promedio (entre los tres criterios utilizados), para todos los tamaños de filtro de moda móvil.

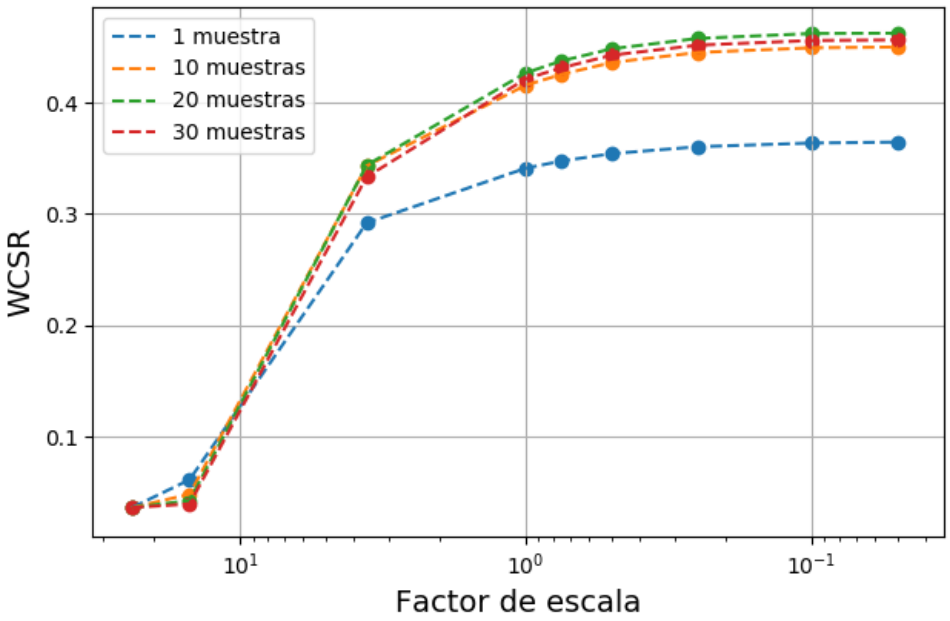

Figura 5.10: WCSR promedio en función del factor de escala para el modelo entrenado con valores RMS.

CAPÍTULO 6: DISCUSIÓN DE LOS RESULTADOS

6.1 ANÁLISIS DE RESULTADOS

Previo al análisis, cabe destacar que todos los cromagramas obtenidos a partir de los modelos entrenados superaron a los cromagramas de referencia en su desempeño en la tarea propuesta. En el caso del modelo entrenado con valores RMS, se requirió una etapa de pre-procesamiento adicional para escalar los espectrogramas.

6.1.1 Análisis de resultados del modelo entrenado sin valores RMS

La utilización de tamaños de mini-lotes mayores, en principio, posibilita una estimación más precisa del gradiente y acelera el entrenamiento. Para los distintos tamaños de filtro de moda móvil, se observaron diferencias poco notorias en los resultados al variar el tamaño de mini-lotes, aunque se observa una tendencia de que usar mini-lotes más grandes da resultados ligeramente mejores.

La eliminación de componentes percusivas (mediante HPSS) también dio como resultado leves mejoras en el rendimiento, aunque no se vio una diferencia sustancial. Esto fue sorpresivo, principalmente debido a que el modelo fue entrenado con audios de contenido casi exclusivamente armónico, y muy pocos elementos percusivos, razón por la cual se esperaba una mejora mayor al eliminar estos componentes del conjunto de prueba.

Donde sí se vieron grandes diferencias, fue en relación a la utilización de entrenamiento por currículum, técnica cuya utilización mostró una gran mejora en los resultados. Esto se condice con lo esperado, ya que el propósito del entrenamiento por currículum es guiar el aprendizaje del modelo de manera progresiva y controlada. Por ejemplo, al aprender a identificar notas individuales primero, el modelo podría tener mayor facilidad para luego aprender a identificar secuencias de dos o más notas, ya que haría uso de la previa identificación individual de notas. Tal es la diferencia observada, que el modelo entrenado sin aprendizaje por currículum no produjo cromagramas con puntajes superiores a los cromagramas de referencia, lo cual indica la eficacia del uso de esta técnica para este caso en particular.

6.1.2 Análisis de resultados del modelo entrenado con valores RMS

El uso de información relacionada con el contenido energético de los audios en el entrenamiento, produjo un modelo que, a priori, pareció producir cromagramas con mayor correlación con la intensidad de las notas en los audios de prueba. En función de esta representación más precisa de la intensidad de las notas, puede que este modelo también sea de utilidad en tareas de otro tipo, como en reconocimiento de tonalidades.

No obstante, a la hora de predecir acordes de acuerdo a la tarea implementada, se obtuvieron valores de WCSR sumamente bajos. Esto se mitigó utilizando factores de escala menores en la normalización de los espectrogramas de entrada; se pudieron observar resultados comparables a los del modelo original, al aplicar un factor igual a 1 (sin escalado), así como también utilizando valores más pequeños (0.75, y 0.5).

Sin embargo, al hacer esto, los cromagramas obtenidos pierden información de intensidad de las notas, y se observa una especie de “saturación” de la salida, donde las notas presentes toman la mayor intensidad posible, mientras el resto tiende a valores muy bajos.

6.2 ANÁLISIS DEL MODELO BASE

Con base al análisis de resultados, en esta sección se explora en mayor detalle el comportamiento de uno de los modelos con mejores resultados: el entrenado con un tamaño de mini-lotes de 1024 muestras, con aprendizaje por currículum (modelo base). Dicho análisis resulta de interés para entender el funcionamiento del mismo, tanto para proponer mejoras, como modelos más eficaces.

Para visualizar el aprendizaje del modelo, se calcularon mapas de saliencia utilizando la biblioteca *keras-vis* [66]. La Figura 6.1 muestra la suma de la saliencia de las notas C, E, y G (notas prominentes) durante los 10 primeros segundos de la canción *Let It Be*. En la misma se puede observar como la activación de cada nota de salida está ligada a las “filas” del espectrograma de entrada correspondientes a dicha nota, en cada una de las octavas a considerar.

Otro fenómeno que puede verse es un incremento en la amplitud de la saliencia en altas frecuencias. Esto puede observarse mejor en el mapa de saliencia correspondiente a la nota G que, junto con el espectrograma normalizado del segmento de audio, puede verse en la figura 6.2.

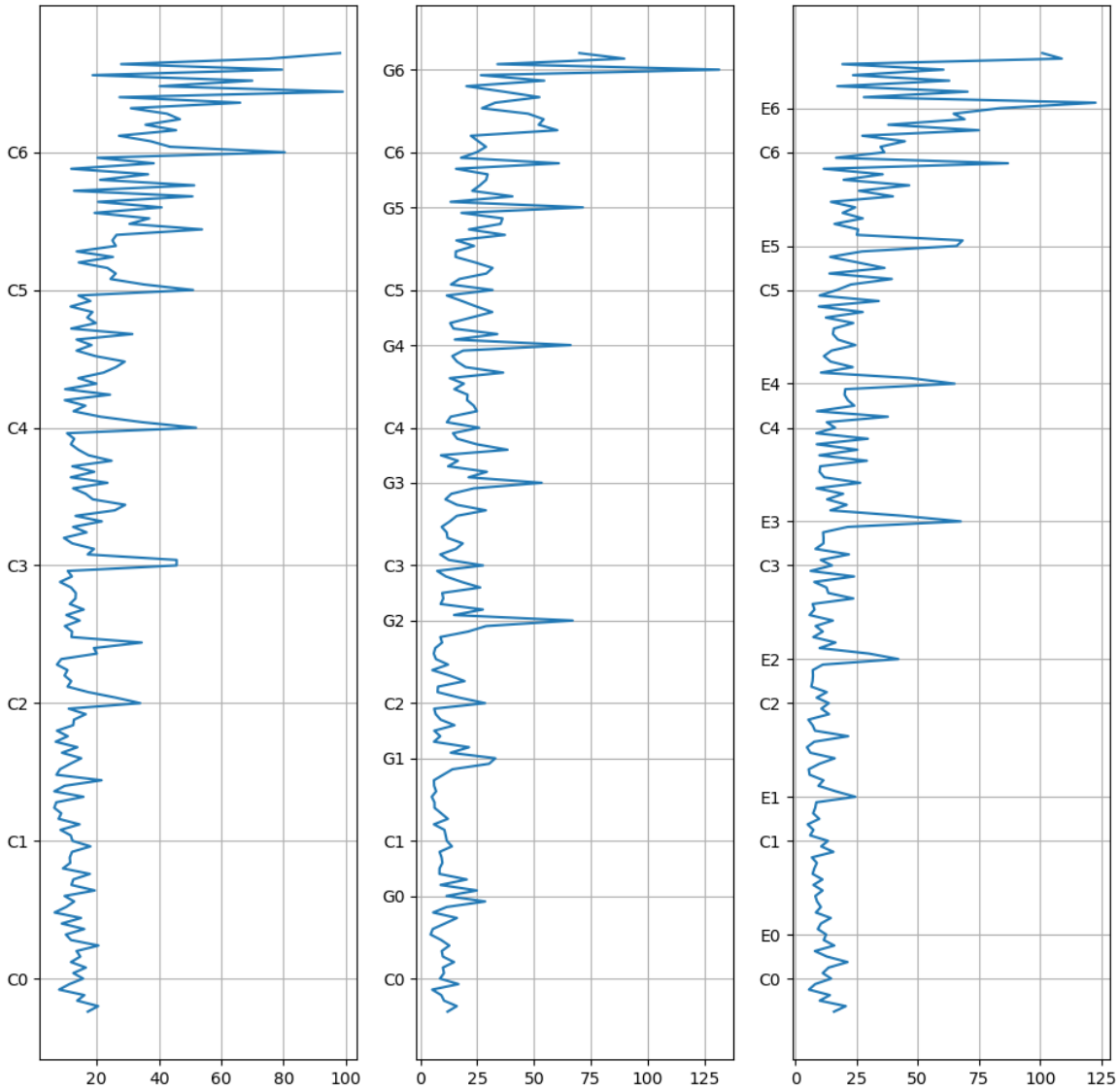

Figura 6.1: Suma de las saliencias para las notas C (izquierda), G (centro) y E (derecha), correspondientes a los primeros 10 segundos de la canción *Let It Be*.

Esto indica que la red es muy sensible al contenido en alta frecuencia. Esto puede corroborarse al observar que los resultados muestran una mejora significativa al utilizar un filtro paso alto para pre-procesar los audios (ver Anexo 2), y puede deberse a que la densidad de armónicos de una frecuencia fundamental aumenta conforme aumentan las octavas.

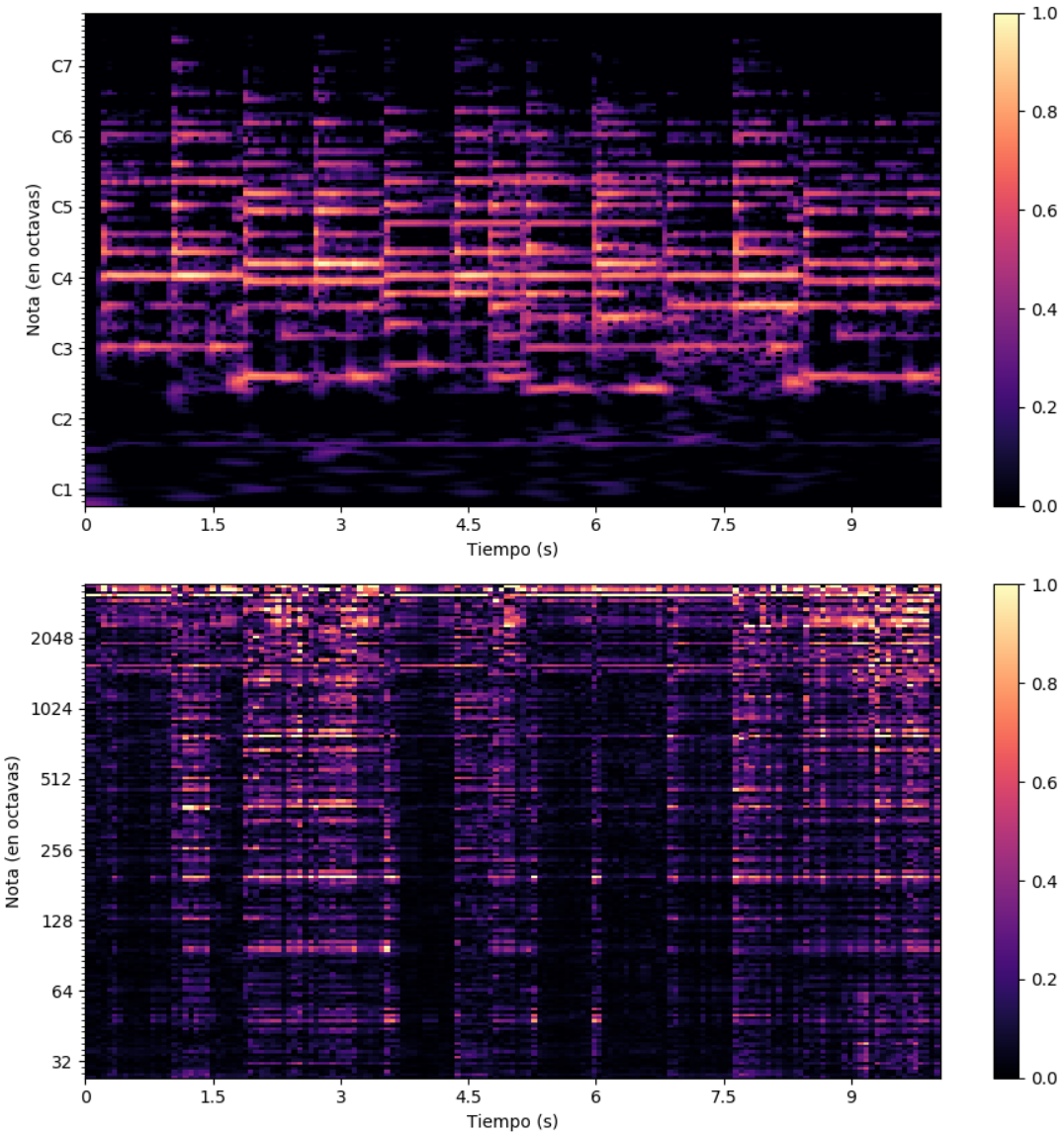

Figura 6.2: Espectrograma (arriba) y mapa de saliencia para la nota G (abajo), correspondientes a los primeros 10 segundos de la canción *Let It Be*.

CAPÍTULO 7: CONCLUSIONES

En este trabajo se ha diseñado e implementado un sistema extractor de cromagramas a partir de audios, a través de una red neuronal con una arquitectura de Perceptrón Multicapa. Este sistema presenta dos particularidades respecto de otros desarrollos similares:

- En lugar de audios musicales reales, el conjunto con el cual se entrenó el modelo consistió en audios artificiales generados a partir del dataset NSynth. Esto permitió contar con un gran volumen de datos de entrenamiento.
- Se utilizó una técnica conocida como entrenamiento por currículum, en donde el modelo fue entrenado utilizando conjuntos de datos cada vez más complejos. Esto se tradujo en la utilización de conjuntos de audios en los cuales la cantidad de notas presentes aumentaba de manera progresiva.

A nivel cualitativo, los cromagramas obtenidos mostraron en general una mejor definición de las notas presentes en relación a cromagramas obtenidos a través de técnicas tradicionales de procesamiento digital de señales.

Para la evaluación objetiva de los modelos de redes neuronales propuestos, se implementó una tarea simple de reconocimiento automático de acordes mediante similitud contra patrones de acordes establecidos. En esta tarea, el desempeño de los cromagramas de casi todas las variantes del modelo fue ampliamente superior al de los cromagramas de referencia. La excepción fueron los cromagramas del modelo RMS utilizando factores de escala grandes, que dieron resultados sumamente bajos de la métrica analizada (WCSR). Esto demuestra que ambas variantes principales del modelo (con y sin valores RMS) fueron capaces de generalizar, es decir, de aprender a partir de audios artificiales y “no musicales” información suficiente como para actuar de manera efectiva sobre audios musicales reales.

Asimismo, se vio una notable diferencia a favor de la utilización de entrenamiento por currículum en relación a entrenamiento tradicional, demostrando así la efectividad del mismo para este tipo de tareas. Los modelos también mostraron mejoras al eliminar las componentes percusivas de los espectrogramas de entrada, aunque dicha mejora no fue de gran magnitud.

CAPÍTULO 8: LÍNEAS FUTURAS DE INVESTIGACIÓN

Si bien los espectrogramas obtenidos a través de CQT son de uso común para este tipo de tareas, se podrían realizar experimentos entrenando e implementando el mismo modelo haciendo uso de STFT para la obtención de espectrogramas, y verificar las posibles diferencias.

Normalmente, los sistemas de reconocimiento automático de acordes a partir de cromagramas poseen un nivel de complejidad y sofisticación mayor al establecido en este trabajo, y asimismo también funcionan de distinta manera. Es por ello que la evaluación del modelo debería también realizarse utilizando un sistema de estas características, con el fin de comprobar la adaptabilidad de los cromagramas extraídos a sistemas más modernos, comúnmente utilizados hoy en día. Junto con esto, se propone también la ampliación de las categorías de acordes utilizadas (comprendiendo, por ejemplo, acordes de séptima), y la utilización de un conjunto de datos mayor para la evaluación.

Además, si bien este trabajo se centra en el reconocimiento automático de acordes como aplicación central de los cromagramas extraídos por el modelo, los mismos pueden utilizarse en otras tareas de similar índole, en particular el modelo entrenado con información energética (valores RMS).

Otra de las limitaciones del modelo actual sobre la cual se podrían realizar mejoras, es que los cromagramas se extraen para ventanas temporales fijas. Dadas las rápidas variaciones espectrales que se pueden observar en un audio musical, es posible que algunas de estas ventanas capturen contenido irrelevante en relación a las notas musicales presentes; por ejemplo, puede que una ventana capture solamente el golpe generado por un instrumento percusivo, o un instante de silencio transitorio. Para mitigar esto, puede implementarse una variación sobre el modelo que tenga en cuenta una cierta ventana contextual de muestras, para que el mismo pueda utilizar información temporal para sus predicciones.

Finalmente, el uso de datos generados de forma sintética parece ser útil y es a su vez una fuente infinita de información. En futuros trabajos se podría expandir sobre lo realizado en la etapa de generación de datos para incluir más tipos de audios, como eventos percusivos, ruido, señales moduladas en amplitud y/o frecuencia, entre otros.

BIBLIOGRAFÍA

[1] The Real Book, 6ta Edición, Hal Leonard Corp., Milwaukee, WI, USA, (2004).

[2] Ellis, D. y Poliner, G., Identifying “Cover Songs” With Chroma Features and Dynamic Programming Beat Tracking, 2007 IEEE International Conference on Acoustics, Speech and Signal Processing, 4, 1429-1433, (2007).

[3] Gómez, E. y Herrera, P., The song remains the same: identifying versions of the same piece using tonal descriptors, Proceedings of the 7th International Society for Music Information Retrieval Conference (ISMIR), 180-185, (2006).

[4] Catteau, B., Martens, J.-P. y Leman, M., A Probabilistic Framework for Audio-Based Tonal Key and Chord Recognition, Proceedings of the 30th Annual Conference of the Gesellschaft für Klassifikation e.V, 637-644, (2006).

[5] Shenoy, A. y Wang, Y., Key, Chord, and Rhythm Tracking of Popular Music Recordings, Computer Music Journal, 29, 75-86, (2005).

[6] Lee, K. y Slaney, M., Acoustic Chord Transcription and Key Extraction From Audio Using Key-Dependent HMMs Trained on Synthesized Audio, IEEE Transactions on Audio, Speech, and Language Processing, 16, 291-301, (2008).

[7] Zenz, V. y Rauber, A., Automatic Chord Detection Incorporating Beat and Key Detection, 2007 IEEE International Conference on Signal Processing and Communications, 1175-1178, (2007).

[8] Pérez-Sancho, C., Rizo, D. e Iñesta, J. M., Genre Classification Using Chords and Stochastic Language Models, Connection Science, 21, 145-159, (2009).

[9] Mauch, M., Fujihara, H. y Goto, M., Integrating Additional Chord Information Into HMM-Based Lyrics-to-Audio Alignment, IEEE Transactions on Audio, Speech, and Language Processing, 20, 200-210, (2012).

[10] Burgoyne, J., Wild, J. y Fujinaga, I., An Expert Ground Truth Set for Audio Chord Recognition and Music Analysis, Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR), 633-638, (2011).

[11] Nattiez, J. y Abbate, C., Music and Discourse: Toward a Semiology of Music, Princeton University Press, Princeton, NJ, USA, (1990).

[12] Kennedy, M., Bourne, J. y Rutherford-Johnson, T., The Oxford Dictionary of Music, 6ta Edición, Oxford University Press, Oxford, England, UK, (2012).

[13] Benward, B. y Saker, M., Music in Theory and Practice, 7ma Edición, McGraw-Hill, New York, NY, USA, (2003).

[14] Zamacois Soler, J., Tratado de Armonía, 1ra Edición, Idea Books S.A., Ámsterdam, Países Bajos, (2002).

[15] P. Mailm, W., Music Cultures of the Pacific, the Near East, and Asia, 3ra Edición, Pearson, New York, NY, USA, (1995).

[16] Harnum, J., Basic Music Theory: How to Read, Write, and Understand Written Music, 4ta Edición, CreateSpace Independent Publishing Platform, Scotts Valley, CA, USA, (2013).

[17] Purwins, H., Blankertz, B. y Obermayer, K., A new method for tracking modulations in tonal music in audio data format, Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, 6, 270-275 vol.6, (2000).

[18] Brown, J., Calculation of a Constant Q Spectral Transform, The Journal of the Acoustical Society of America, 89, 425, (1991).

[19] Fitzgerald, D., Harmonic/Percussive Separation using Median Filtering, (2010).

[20] Driedger, J., Müller, M. y Disch, S., Extending Harmonic-Percussive Separation of Audio Signals, Proceedings of 15th International Society for Music Information Retrieval Conference (ISMIR), (2014).

[21] Basheer, I. A. y Hajmeer, M. N., Artificial neural networks: fundamentals, computing, design, and application. Journal of microbiological methods, 43, 3-31, (2000).

[22] Hecht-Nielsen, R., Neurocomputing, 1era Edición, Addison-Wesley, Reading, MA, USA, (1990).

[23] Schalkoff, R. J., Artificial Neural Networks, 1era Edición, McGraw-Hill Companies, New York, NY, USA, (1997).

[24] Rosenblatt, F., The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review, 65, 386-408, (1958).

[25] Pepino, L., Separación de fuentes musicales mediante redes neuronales convolucionales, Tesis de Grado, Universidad Nacional de Tres de Febrero, (2019).

[26] Ruder, S., An overview of gradient descent optimization algorithms, Computing Research Repository, abs/1609.04747, (2016).

[27] Rumelhart, D. E., Hinton, G. E. y Williams, R. J., Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, 1era Edición, MIT Press, Cambridge, MA, USA, (1986).

[28] Nielsen, M. A., Neural networks and deep learning, Determination Press, San Francisco, CA, USA, (2015).

[29] Lever, J., Krzywinski, M. y Altman, N., Points of Significance: Model selection and overfitting, Nature Methods, 13, 703-704, (2016).

[30] Srivastava, N. et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research, 15, 1929-1958, (2014).

[31] Bengio, Y., Practical recommendations for gradient-based training of deep architectures, Computing Research Repository, abs/1206.5533, (2012).

[32] Erhan, D. et al., Visualizing Higher-Layer Features of a Deep Network, Technical Report, Univeristé de Montréal, (2009).

[33] Zeiler, M. D. y Fergus, R., Visualizing and Understanding Convolutional Networks, Lecture Notes in Computer Science, 818-833, (2014).

[34] Simonyan, K., Vedaldi, A. y Zisserman, A., Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Computing Research Repository, abs/1312.6034, (2013).

[35] Fujishima, T., Realtime chord recognition of musical sound: A system using Common Lisp Music, Proceedings of International Computer Music Conference, 464-467, (1999).

[36] Rabiner, L. R., A tutorial on hidden Markov models and selected applications in speech recognition, Proceedings of the IEEE, 77, 257-286, (1989).

[37] McVicar, M. et al., Automatic Chord Estimation from Audio: A Review of the State of the Art, IEEE Transactions on Audio, Speech, and Language Processing, 22, 556-575, (2014).

[38] Mauch, M., Automatic Chord Transcription from Audio Using Computational Models of Musical Context, Tesis de Doctorado, University of London, (2010).

[39] Lee, K., A System for Automatic Chord Transcription from Audio Using Genre-Specific Hidden Markov Models, Adaptive Multimedia Retrieval: Retrieval, User, and Semantics: 5th International Workshop, 134-146, (2007).

[40] Ashley, J. et al., A Cross-Validated Study of Modelling Strategies for Automatic Chord Recognition in Audio, Proceedings of 8th International Society for Music Information Retrieval Conference (ISMIR), 251-254, (2007).

[41] Shenoy, A. y Wang, Y., Key, Chord, and Rhythm Tracking of Popular Music Recordings, Computer Music Journal, 29, 75-86, (2005).

[42] Raffel, C. et al., mir_eval: A Transparent Implementation of Common MIR Metrics, Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), (2014).

[43] Deng, J.-q. y Kwok, Y.-K., Large Vocabulary Automatic Chord Estimation with an Even Chance Training Scheme, Proceedings of 18th International Society for Music Information Retrieval Conference (ISMIR), (2017).

[44] Pauws, S., Musical key extraction from audio, Proceedings of 5th International Society for Music Information Retrieval Conference (ISMIR), (2004).

[45] Lee, K. y Slaney, M., Automatic Chord Recognition from Audio Using a HMM with Supervised Learning, 7th International Conference on Music Information Retrieval, 133-137, (2006).

[46] Sheh, A. y P.W. Ellis, D., Chord Segmentation and Recognition using EM-Trained Hidden Markov Models, International Symposium on Music Information Retrieval, (2003).

[47] Harte, C. y Sandler, M., Automatic Chord Identification Using a Quantised Chromagram, Audio Engineering Society Convention 118, (2005).

[48] Smith, M. T., Audio Engineer's Reference Book, 2da Edición, Focal Press, Abingdon, UK, (2001).

[49] Ni, Y. et al., An End-to-End Machine Learning System for Harmonic Analysis of Music, IEEE Transactions on Audio, Speech, and Language Processing, 20, 1771-1783, (2011).

[50] Goto, M. y Muraoka, Y., Real-time beat tracking for drumless audio signals: Chord change detection for musical decisions, Speech Communication, 27, 311-335, (1999).

[51] Bello, J. y Pickens, J., A Robust Mid-level Representation for Harmonic Content in Music Signals, Proceedings of 6th International Society for Music Information Retrieval Conference (ISMIR), 304-311, (2005).

[52] Lee, K. y Slaney, M., A Unified System for Chord Transcription and Key Extraction Using Hidden Markov Models, Proceedings of 8th International Society for Music Information Retrieval Conference (ISMIR), 245-250, (2007).

[53] McFee, B. y Bello, J. P., Structured Training for Large-Vocabulary Chord Recognition, Proceedings of 18th International Society for Music Information Retrieval Conference (ISMIR), 188-194, (2017).

[54] Cho, K., Courville, A. C. y Bengio, Y., Describing Multimedia Content using Attention-based Encoder-Decoder Networks, Computing Research Repository, abs/1507.01053, (2015).

[55] Korzeniowski, F. y Widmer, G., Feature Learning for Chord Recognition: The Deep Chroma Extractor, Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 37-43, (2016).

[56] Harte, C., Towards automatic extraction of harmony information from music signals, Tesis de Doctorado, University of London, (2010).

[57] Mauch, M. et al., OMRAS2 metadata project 2009, Proceedings of the 10th International Society for Music Information Retrieval Conference (ISMIR), (2009).

[58] Zhou, X. y Lerch, A., Chord Detection Using Deep Learning, Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), (2015).

[59] Engel, J. et al., Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders, Proceedings of the 34th International Conference on Machine Learning, 70, 1068-1077, (2017).

[60] Bengio, Y. et al., Curriculum learning, Proceedings of the 26th Annual International Conference on Machine Learning, 41-48, (2009).

[61] Schörkhuber, C. y Klapuri, A., Constant-Q transform toolbox for music processing, Proceedings of the 7th Sound and Music Computing Conference, (2010).

[62] McFee, B. et al., librosa: Audio and Music Signal Analysis in Python, Proceedings of the 14th Python in Science Conference, 18-24, (2015).

[63] Abadi, M. et al., TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, https://www.tensorflow.org, (2015).

[64] Chollet, F. et al., Keras, GitHub, https://keras.io, (2015).

[65] Kingma, D. y Ba, J., Adam: A Method for Stochastic Optimization, International Conference on Learning Representations, (2014).

[66] Kotikalapudi, R. et al., keras-vis, GitHub, https://github.com/raghakot/keras-vis, (2019).

ANEXO A: CÓDIGO IMPLEMENTADO

En este anexo se encuentra parte del código implementado, y se detalla la función de los scripts más relevantes. La totalidad del mismo puede encontrarse en `https://github.com/AleSua93/tesis_scripts_v2`.

A.1 GENERACIÓN DE DATOS DE ENTRENAMIENTO Y VALIDACIÓN

Estos scripts se corresponden con la generación del conjunto de datos de entrenamiento a partir de los audios de NSynth. A continuación se describe la función de los scripts más importantes:

- *generate.py*. Script principal que genera los audios de entrenamiento, validación y pruebas, a partir de los audios de NSynth. Hace uso de instancias de la clase NsynthGenerator.
- *nsynth_generator.py*. Contiene a la clase NsynthGenerator, cuyos métodos definen la generación de audios compuestos por una o más notas.
- *compile_dataset.py*. A partir de los audios generados, realiza el pre-procesamiento de los mismos (CQT, normalización) y almacena cada conjunto de datos en formato HDF5.
- *utils.py*. Funciones y variables globales de uso general, tanto para esta etapa del proceso como para las siguientes.

generate.py

```
1  from nsynth_generator import NsynthGenerator
2  from dicts import PITCHES, MIDI_2_CHROMA
3  import random, os
4
5  dir = os.path.dirname(__file__)
6  train_input_dir = os.path.join(dir, "../data/nsynth/nsynth-train/")
7  val_input_dir = os.path.join(dir, "../data/nsynth/nsynth-valid/")
8  test_input_dir = os.path.join(dir, "../data/nsynth/nsynth-test/")
9  train_output_dir = os.path.join(dir, "../data/nsynth/output/training/")
10 val_output_dir = os.path.join(dir, "../data/nsynth/output/validation/")
11 test_output_dir = os.path.join(dir, "../data/nsynth/output/test/")
12
13 training_generator = NsynthGenerator(input_dir=train_input_dir, output_dir=train_output_dir
       )
14 validation_generator = NsynthGenerator(input_dir=val_input_dir, output_dir=val_output_dir)
```

```
15 test_generator = NsynthGenerator(input_dir=test_input_dir, output_dir=test_output_dir)
16
17 pitches = range(21, 109) # All 88 notes on the piano. The added intervals can have the
       0-127 pitches however
18 pitches = [i for i in pitches for _ in range(400)] # The pitches are repeated: [21, 21, 22,
        22, 23, 23, ...]
19 samples_num = len(pitches)
20 samples_duration = 0.25
21
22 train_number_outputs = {
23     1: pitches,
24     2: pitches,
25     3: pitches,
26     4: pitches,
27     5: pitches,
28     6: pitches,
29 }
30
31 val_number_outputs = {
32     1: random.sample(pitches, int(samples_num/3)),
33     2: random.sample(pitches, int(samples_num/3)),
34     4: random.sample(pitches, int(samples_num/3)),
35     5: random.sample(pitches, int(samples_num/3)),
36     3: random.sample(pitches, int(samples_num/3)),
37     6: random.sample(pitches, int(samples_num/3)),
38 }
39
40 test_number_outputs = {
41     1: random.sample(pitches, int(samples_num/5)),
42     2: random.sample(pitches, int(samples_num/5)),
43     4: random.sample(pitches, int(samples_num/5)),
44     5: random.sample(pitches, int(samples_num/5)),
45     3: random.sample(pitches, int(samples_num/5)),
46     6: random.sample(pitches, int(samples_num/5)),
47 }
48
49 # Delete existing files
50 training_generator.delete_files()
51 validation_generator.delete_files()
52 test_generator.delete_files()
53
54 global_idx = 0
55
56 print("Generating training data")
57 for key, value in train_number_outputs.items():
58     for idx,i in enumerate(value):
59         if (global_idx % 1000 == 0):
60             print('Id: {}'.format(global_idx))
61         training_generator.generate(number=key, root=i, duration=samples_duration,
               output_id=global_idx)
62         global_idx += 1
63
64 print("Generating validation data")
65 for key, value in val_number_outputs.items():
66     for idx,i in enumerate(value):
67         validation_generator.generate(number=key, root=i, duration=samples_duration,
```

```
            output_id=global_idx)
68          global_idx += 1
69
70  print("Generating testing data")
71  for key, value in test_number_outputs.items():
72      for idx,i in enumerate(value):
73          test_generator.generate(number=key, root=i, duration=samples_duration, output_id=
                global_idx)
74          global_idx += 1
```

nsynth_generator.py

```
1   from dicts import MIDI_2_CHROMA
2   import os, json
3   import librosa, librosa.feature
4   import mir_eval.chord as chord
5   import random
6   import numpy as np
7   from utils import SR
8   from decimal import *
9
10  class NsynthGenerator():
11
12      def __init__(self, input_dir, output_dir):
13
14          self.output_dir = output_dir
15          self.input_dir = input_dir
16
17          with open(self.input_dir + "examples.json") as examples:
18              data = json.loads(examples.read())
19
20          self.ids = []
21          self.names = []
22          self.pitches = []
23          self.families = []
24          self.velocities = []
25          self.sources = []
26          for id, key in enumerate(data.keys()):
27              self.ids.append(id)
28              self.names.append(data[key]['note_str'])
29              self.pitches.append(data[key]['pitch'])
30              self.families.append(data[key]['instrument_family_str'])
31              self.velocities.append(data[key]['velocity'])
32              self.sources.append(data[key]['instrument_source_str'])
33          self.families = [s.replace('_', '-') for s in self.families] # This is so names
                like synth_lead turn into synth-lead
34
35      def generate(self, number, root, duration, output_id):
36          audio_files_dir = self.input_dir + "audio/"
37          output_dir = self.output_dir + str(number) + "/"
38
39          root_id = self._choose_random(root)
40          main_audio, sr = librosa.load(audio_files_dir + self.names[root_id] + ".wav", sr=SR
                , duration=duration)
41          main_rms = np.sqrt(np.mean(np.square(main_audio)))
42
```

```
43          notes = ["{}({:.5f})".format(MIDI_2_CHROMA[self.pitches[root_id]], main_rms)]
44
45          for i in range(number - 1):
46              random_id = random.choice(self.ids)
47              audio, sr = librosa.load(audio_files_dir + self.names[random_id] + ".wav", sr=
                    SR, duration=duration)
48              rms = np.sqrt(np.mean(np.square(audio)))
49              main_audio += audio
50
51              notes.append("{}({:.5f})".format(MIDI_2_CHROMA[self.pitches[random_id]], rms))
52
53          notes = [note.replace('.', ',') for note in notes]
54          notes = "-".join(notes)
55
56          filename = output_dir + str(output_id) + "_" + self.families[root_id] + "_" + notes
                + ".wav"
57          librosa.output.write_wav(filename, main_audio, sr, norm=True)
58
59      def _choose_random(self, pitch):
60          audio_ids = []
61          for idx, p in enumerate(self.pitches):
62              if p == pitch:
63                  audio_ids.append(idx)
64          return random.choice(audio_ids)
65
66      def delete_files(self):
67          dirs = os.listdir(self.output_dir)
68          for directory in dirs:
69              files = os.listdir(self.output_dir + "/" + directory)
70              for f in files:
71                  os.remove(self.output_dir + "/" + directory + "/" + f)
72          print("All files deleted!")
```

compile_dataset.py

```
1   import os
2   import h5py
3   import numpy as np
4   import librosa
5   import matplotlib.pyplot as plt
6   from utils import extract_cqt, intervals_2_chroma, sort_by_id, normalize_cqt,
        normalize_cqt_rms
7   from utils import HOP_LENGTH, SR
8
9   def delete_datasets(datasets_dir):
10      subdirs = os.listdir(datasets_dir)
11      subdirs = [os.path.join(datasets_dir, d) for d in subdirs]
12      for d in subdirs:
13          files = os.listdir(d)
14          for f in files:
15              os.remove(os.path.join(d, f))
16      print("All files deleted!")
17
18  def generate_dataset(input_dir, output_dir, subdir, output_file):
19      sample_length = '' # This will be the size of time samples for each audio clip
20
```

```
21      full_output_file = os.path.join(output_dir, subdir, output_file)
22      full_input_dir = os.path.join(input_dir, subdir)
23      full_info_file = os.path.join(output_dir, subdir, output_file.split('.')[0] + "-info.
            txt")
24
25      with h5py.File(full_output_file, 'w') as f:
26          dset = f.create_dataset("data", (175,0), dtype='float64', maxshape=(None, None))
27          dset = f.create_dataset("labels", (12,0), dtype='float64', maxshape=(None, None))
28
29      info_file = open(full_info_file, "w")
30
31      print("Ids:", file=info_file)
32
33      files = sorted(os.listdir(full_input_dir), key=sort_by_id)
34
35      print("\nDirectory: {}".format(full_input_dir), file=info_file)
36
37      for f in files:
38          filepath = os.path.join(full_input_dir, f)
39          intervals = f.split('_')[2].split('.')[0].split('-')
40
41          cqt = extract_cqt(filepath)
42          cqt = normalize_cqt_rms(cqt)
43          cqt = cqt[:, 1:] # THIS IS TEMPORARY # (it wasn't)
44
45          sample_length = cqt.shape[1] if sample_length == '' else sample_length
46
47          chroma = intervals_2_chroma(intervals)
48          chroma = np.transpose(np.tile(chroma, (cqt.shape[1], 1)))
49
50          print(f.split('_')[0], end=' - ', flush=True, file=info_file)
51
52          with h5py.File(full_output_file, 'r+') as f:
53              data = f['data']
54              labels = f['labels']
55              data.resize((data.shape[0],data.shape[1]+cqt.shape[1]))
56              data[:,-cqt.shape[1]:] = cqt
57              labels.resize((labels.shape[0],labels.shape[1]+chroma.shape[1]))
58              labels[:,-chroma.shape[1]:] = chroma
59
60      print("\nThe spectrogram length in time samples of each audio clip is: {}".format(
            sample_length), file=info_file)
61
62      info_file.close()
63
64
65  dir = os.path.dirname(__file__)
66  data_path = train_data = os.path.join(dir, "../data/final_dataset/rms_dataset")
67  train_file = "training.hdf5"
68  val_file = "validation.hdf5"
69  test_file = "test.hdf5"
70
71  train_dir = os.path.join(dir, "../data/nsynth/output/training")
72  val_dir = os.path.join(dir, "../data/nsynth/output/validation")
73  test_dir = os.path.join(dir, "../data/nsynth/output/test")
74
```

```
75 delete_datasets(data_path)
76
77 datasets = range(1, 7)
78 for d in datasets:
79     print("Generating {}({})".format(train_file, d))
80     generate_dataset(input_dir=train_dir, output_dir=data_path,  subdir=str(d), output_file
       =train_file)
81     print("Generating {}({})".format(val_file, d))
82     generate_dataset(input_dir=val_dir, output_dir=data_path, subdir=str(d), output_file=
       val_file)
83     print("Generating {}({})".format(test_file, d))
84     generate_dataset(input_dir=test_dir, output_dir=data_path, subdir=str(d), output_file=
       test_file)
```

utils.py

```
1  import librosa
2  import librosa.display
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  # Some constants
7
8  HOP_LENGTH = 1024
9  SR = 16000
10 N_BINS = 175
11 BINS_PER_OCTAVE = 25
12 FMIN = 27.5
13 AUDIO_LEN = 3
14 CQT_THRESHOLD = -40
15
16
17 def extract_cqt(filepath):
18     audio, sr = librosa.load(filepath, sr=SR, mono=True)
19     C = np.abs(librosa.cqt(audio, fmin=27.5, sr=sr, hop_length=HOP_LENGTH, n_bins=175,
       bins_per_octave=25))
20     return C
21
22 def normalize_cqt(C):
23     C_db = librosa.power_to_db(C**2, ref=np.max)
24     C_db[C_db <= CQT_THRESHOLD] = CQT_THRESHOLD
25     C_db = C_db - CQT_THRESHOLD
26     C_norm = C_db / np.max(C_db)
27     return C_norm
28
29 def normalize_cqt_rms(C):
30     C_db = librosa.power_to_db(C**2, ref=0.01)
31     # print("Min {} Max {}".format(np.min(C_db), np.max(C_db)))
32     # print("Min {} Max {}".format(np.min(C), np.max(C)))
33     C_db_norm = C_db / 23.84
34     return C_db_norm
35
36 def intervals_2_chroma(intervals):
37     notes = [int(i.split('(')[0]) for i in intervals]
38     rms_values = [float(i.split('(')[1].split(')')[0].replace(',', '.')) for i in intervals
       ]
```

```
39
40      notes_rms = dict((notes[i], rms_values[i]) for i in range(len(notes)))
41      # notes_rms = dict((notes[i], 1) for i in range(len(notes))) # This line ignores energy
            , comment it out for including it
42
43      chroma = np.zeros(12) # Number of possible pitches
44      for idx, pitch in enumerate(chroma):
45          if idx in notes:
46              chroma[idx] = notes_rms[idx]
47      return chroma
48
49
50  def sort_by_id(filename):
51      return int(filename.split('_')[0])
```

A.2 ENTRENAMIENTO DEL MODELO

Scripts pertinentes a la implementación y el entrenamiento del modelo de redes neuronales, utilizando los conjuntos de datos generados con anterioridad.

- *train.py*. Entrena un modelo determinado con un conjunto de datos de entrenamiento determinado. En una primera instancia crea al modelo, en posteriores instancias carga el modelo existente y continúa el entrenamiento con un nuevo conjunto de datos.
- *main.sh*. Ejecuta el script de entrenamiento de manera secuencial, para posibilitar el entrenamiento por currículum.
- *data_generator.py*. Contiene la clase DataGenerator, cuya función es entregar al modelo los mini-lotes de datos para el entrenamiento, a partir del conjunto de datos correspondiente.

train.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from data_generator import DataGenerator
4  import os, sys, shutil, random, pickle
5  import h5py
6  from utils import AUDIO_LEN
7  os.environ['KMP_WARNINGS'] = 'off'
8
9  current_dir = os.path.dirname(__file__)
10 dataset_dir = "../data/final_dataset/rms_dataset/"
11 models_dir = "../models/"
12 dataset_subdir = sys.argv[1]
13
14 full_dataset_path = os.path.join(current_dir, dataset_dir, dataset_subdir)
15 train_file = os.path.join(full_dataset_path, "training.hdf5")
16 validation_file = os.path.join(full_dataset_path, "validation.hdf5")
```

```
17 test_file = os.path.join(full_dataset_path, "test.hdf5")
18 training_plots_dir = os.path.join(current_dir, "../graphs/training_plots", dataset_subdir)
19
20 logdir = os.path.join(current_dir, "../logs/", dataset_subdir)
21 # Remove and recreate log subdirectory
22 shutil.rmtree(logdir)
23 os.mkdir(logdir)
24
25 BATCH_SIZE = 1024
26 NUM_EPOCHS = 150
27 NUM_WORKERS = 8
28
29 # DATA
30
31 partition = {
32     'training': [],
33     'validation': [],
34     'test': [],
35 }
36 dimension = ''
37
38 with h5py.File(train_file, 'r') as f:
39     partition['training'] = range(f['data'].shape[1])
40     dimension = f['data'].shape[0]
41
42 with h5py.File(validation_file, 'r') as f:
43     partition['validation'] = range(f['data'].shape[1])
44
45 with h5py.File(test_file, 'r') as f:
46     partition['test'] = range(f['data'].shape[1])
47
48 print("Samples in training dataset: {}".format(len(partition['training'])))
49 print("Samples in validation dataset: {}".format(len(partition['validation'])))
50 print("Samples in test dataset: {}".format(len(partition['test'])))
51
52 train_params = {
53     'dim': (dimension,),
54     'batch_size': BATCH_SIZE,
55     'n_channels': 1,
56     'shuffle': True,
57     'data_filename': train_file,
58 }
59 val_params = {
60     'dim': (dimension,),
61     'batch_size': BATCH_SIZE,
62     'n_channels': 1,
63     'shuffle': True,
64     'data_filename': validation_file,
65 }
66 test_params = {
67     'dim': (dimension,),
68     'batch_size': BATCH_SIZE,
69     'n_channels': 1,
70     'shuffle': True,
71     'data_filename': test_file,
72 }
```

```
73
74  training_generator = DataGenerator(partition['training'], **train_params)
75  validation_generator = DataGenerator(partition['validation'], **val_params)
76  testing_generator = DataGenerator(partition['test'], **test_params)
77  training_generator.prepare()
78  validation_generator.prepare()
79  testing_generator.prepare()
80
81  # MODEL
82
83  import tensorflow as tf
84  from tensorflow.keras.optimizers import Adam
85  from tensorflow.keras.models import Sequential, load_model
86  from tensorflow.keras.layers import Dense, Dropout, Input
87  from tensorflow.keras.regularizers import l1, l2
88  from tensorflow.keras.callbacks import TensorBoard, EarlyStopping
89  from callbacks import PredictCallback
90
91  with h5py.File(test_file, 'r+') as f:
92      dset = f['data']
93      labels = f['labels']
94      predict_data = dset[:, 0:AUDIO_LEN*10]
95      predict_labels = labels[:, 0:AUDIO_LEN*10]
96
97  callbacks = [
98      PredictCallback(predict_data, predict_labels, plots_dir=training_plots_dir),
99      TensorBoard(log_dir=logdir),
100     EarlyStopping(monitor='val_loss', min_delta=0.001, patience=50, restore_best_weights=
            True)
101 ]
102 optimizer = Adam(lr=0.001)
103
104 # Load previous model if it exists, otherwise create new model
105 if (int(dataset_subdir) > 1):
106     previous_subdir = str(int(dataset_subdir)-1)
107     print('Loading model from: {}'.format(previous_subdir))
108     model = load_model(os.path.join(current_dir, models_dir) + "{}.h5".format(
            previous_subdir))
109 else:
110     print("Starting training...")
111     model = Sequential()
112     model.add(Dense(256, activation='relu', input_shape=(175,), kernel_initializer='
            glorot_uniform'))
113     model.add(Dropout(0.25))
114     model.add(Dense(256, activation='relu', kernel_initializer='glorot_uniform'))
115     model.add(Dropout(0.25))
116     model.add(Dense(256, activation='relu', kernel_initializer='glorot_uniform'))
117     model.add(Dropout(0.25))
118     model.add(Dense(12, activation='sigmoid', kernel_initializer='glorot_uniform'))
119
120 model.compile(loss='kullback_leibler_divergence', optimizer=optimizer, metrics=['accuracy'
        ])
121 model.summary()
122 history = model.fit_generator(generator=training_generator,
123                               validation_data=validation_generator,
124                               epochs=NUM_EPOCHS,
```

```
                                max_queue_size=20,
                                callbacks=callbacks,
                                workers=NUM_WORKERS
                               )
scores = model.evaluate_generator(testing_generator)
print("Metrics are: {}".format(model.metrics_names))
print("Final scores are: {}".format(scores))
model.save(os.path.join(current_dir, models_dir) + "{}.h5".format(dataset_subdir))

# Save the history
with open(os.path.join(current_dir, models_dir) + "{}_history".format(dataset_subdir), 'wb'
    ) as f:
    pickle.dump(history.history, f)

# Clean up
training_generator.finish()
validation_generator.finish()
testing_generator.finish()
```

main.sh

```
#!/usr/bin/env bash

# conda env must be activated beforehand

echo "-- Running 1st training --"
python ./train/train.py 1
echo "-- Running 2nd training --"
python ./train/train.py 2
echo "-- Running 3rd training --"
python ./train/train.py 3
echo "-- Running 4th training --"
python ./train/train.py 4
echo "-- Running 5th training --"
python ./train/train.py 5
echo "-- Running 6th training --"
python ./train/train.py 6
echo "-- Training completed --"
```

data_generator.py

```
from tensorflow.keras.utils import Sequence
import numpy as np
import mir_eval.chord as chord
import re, os
from scipy import stats
import h5py

dir = os.path.dirname(__file__)
data_filepath = os.path.join(dir, "../data/final_dataset/")


class DataGenerator(Sequence):

    def __init__(self, list_IDs, data_filename, batch_size=32, dim=(32,32,32), n_channels
        =1, shuffle=False):
```

```
15          'Initialization'
16          self.dim = dim
17          self.batch_size = batch_size
18          self.list_IDs = list_IDs
19          self.n_channels = n_channels
20          self.shuffle = shuffle
21          self.data_filename = data_filename
22          self.on_epoch_end()
23
24      def prepare(self):
25          self.data_file = h5py.File(self.data_filename, 'r+')
26          self.dset = self.data_file['data']
27          self.labels = self.data_file['labels']
28
29      def finish(self):
30          self.data_file.close()
31
32      def __len__(self):
33          'Denotes the number of batches per epoch'
34          return int(np.floor(len(self.list_IDs) / self.batch_size))
35
36      def __getitem__(self, index):
37          'Generate one batch of data'
38          #  Generate indexes of the batch
39          indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]
40          # Find list of IDs
41          list_IDs_temp = [self.list_IDs[k] for k in indexes]
42
43          # Generate data
44          X, y = self.__data_generation(list_IDs_temp)
45
46          return X, y
47
48      def on_epoch_end(self):
49          'Updates indexes after each epoch'
50          self.indexes = np.arange(len(self.list_IDs))
51          if self.shuffle == True:
52              np.random.shuffle(self.indexes)
53
54
55      def __data_generation(self, list_IDs_temp):
56          'Generates data containing batch_size samples' # X : (n_samples, *dim, n_channels)
57          # Num channels was used here before
58          X = np.empty((self.batch_size, *self.dim))
59          y = np.empty((self.batch_size, 12), dtype=int)
60
61          # Generate data
62          for i, ID in enumerate(list_IDs_temp):
63              X[i,] = self.dset[:, ID]
64              y[i, :] = self.labels[:, ID] / 0.35
65
66          return X, y
```

A.3 EVALUACIÓN: RECONOCIMIENTO AUTOMÁTICO DE ACORDES

Código de implementación de la tarea de ACE propuesta, y posterior obtención de puntajes de WCSR.

- *template_generator.py*. Contiene a la clase TemplateGenerator, cuyos métodos devuelven las plantillas utilizadas en la tarea de reconocimiento.
- *chord_recognizer.py*. Contiene a la clase ChordRecognizer, que reconoce acordes a partir de un vector croma y un juego de plantillas.
- *evaluate.py*. Calcula y almacena el CSR de cada una de las canciones en el conjunto de evaluación.
- *compute_final_score.py*. Toma los puntajes de CSR y calcula el puntaje de WCSR correspondiente al modelo en cuestión.

template_generator.py

```
1
2  import numpy as np
3  import mir_eval.chord as chord
4
5  class TemplateGenerator():
6
7      @staticmethod
8      def get_maj_min():
9          templates = {}
10         majors = ["C","Db","D","Eb","E","F","F#","G","Ab","A","Bb","B"]
11         minors = ["C:min","Db:min","D:min","Eb:min","E:min","F:min","F#:min","G:min","Ab:
               min","A:min","Bb:min","B:min"]
12
13         for c in majors:
14             t = chord.encode(c)
15             a = t[1][12 - t[0]:].tolist()
16             b = t[1][:12 - t[0]].tolist()
17             templates[c] = np.array(a+b)
18
19         for c in minors:
20             t = chord.encode(c)
21             a = t[1][12 - t[0]:].tolist()
22             b = t[1][:12 - t[0]].tolist()
23             templates[c] = np.array(a+b)
24
25         templates['N'] = np.array([0,0,0,0,0,0,0,0,0,0,0,0])
26         # templates['NC'] = np.array([1,1,1,1,1,1,1,1,1,1,1,1])
27
28         return templates
29
30     @staticmethod
31     def get_triads():
```

```
32          templates = TemplateGenerator.get_maj_min()
33
34          sus2 = ["C:sus2","Db:sus2","D:sus2","Eb:sus2","E:sus2","F:sus2","F#:sus2","G:sus2",
                "Ab:sus2","A:sus2","Bb:sus2","B:sus2"]
35          sus4 = ["C:sus4","Db:sus4","D:sus4","Eb:sus4","E:sus4","F:sus4","F#:sus4","G:sus4",
                "Ab:sus4","A:sus4","Bb:sus4","B:sus4"]
36          diminished = ["C:dim","Db:dim","D:dim","Eb:dim","E:dim","F:dim","F#:dim","G:dim","
                Ab:dim","A:dim","Bb:dim","B:dim"]
37          augmented = ["C:aug","Db:aug","D:aug","Eb:aug","E:aug","F:aug","F#:aug","G:aug","Ab
                :aug","A:aug","Bb:aug","B:aug"]
38
39          for c in diminished:
40              t = chord.encode(c)
41              a = t[1][12 - t[0]:].tolist()
42              b = t[1][:12 - t[0]].tolist()
43              templates[c] = np.array(a+b)
44
45          for c in augmented:
46              t = chord.encode(c)
47              a = t[1][12 - t[0]:].tolist()
48              b = t[1][:12 - t[0]].tolist()
49              templates[c] = np.array(a+b)
50
51          for c in sus2:
52              t = chord.encode(c)
53              a = t[1][12 - t[0]:].tolist()
54              b = t[1][:12 - t[0]].tolist()
55              templates[c] = np.array(a+b)
56
57          for c in sus4:
58              t = chord.encode(c)
59              a = t[1][12 - t[0]:].tolist()
60              b = t[1][:12 - t[0]].tolist()
61              templates[c] = np.array(a+b)
62
63          return templates
```

chord_recognizer.py

```
1   import mir_eval.chord as chord
2   import numpy as np
3   import librosa.display
4   import matplotlib.pyplot as plt
5   from scipy import spatial
6   from scipy.stats import mode
7   from utils import SR
8   from scipy.stats import mode
9
10
11  class ChordRecognizer():
12
13      @staticmethod
14      def predict_chord(chroma_vector, templates):
15          distances = []
16          for key, template in templates.items():
17              distances.append(spatial.distance.euclidean(chroma_vector, template))
```

```
18          predicted_chord = list(templates.keys())[np.argmin(distances)]
19          return predicted_chord
20
21      @staticmethod
22      def moving_mode(chords, num_samples):
23          mm = []
24          offset = int(num_samples / 2)
25          for idx, chord in enumerate(chords):
26              if (idx < offset):
27                  interval = chords[0:idx+offset]
28                  mm.append(mode(interval)[0][0])
29              elif (idx + offset > len(chords)):
30                  interval = chords[idx-offset:len(chords)-1]
31                  mm.append(mode(interval)[0][0])
32              else:
33                  interval = chords[idx-offset:idx+offset]
34                  mm.append(mode(interval)[0][0])
35          return mm
```

evaluate.py

```
1
2  from template_generator import TemplateGenerator
3  from chord_recognizer import ChordRecognizer
4  import os, json, sys
5  from pprint import pprint
6  import mir_eval.chord
7  import matplotlib.pyplot as plt
8  import librosa, librosa.feature, librosa.util
9  from utils import *
10 from tensorflow.keras.models import Sequential, load_model
11 import tensorflow as tf
12 tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
13
14 mm_samples = int(sys.argv[1])
15
16 current_path = os.path.dirname(__file__)
17 beatles_discography_basepath = os.path.join(current_path, '../data/discographies/beatles')
18 beatles_labels_basepath = os.path.join(current_path, '../data/final_labels/The Beatles')
19 results_path = os.path.join(current_path, "./results")
20 results_file = os.path.join(results_path, 'results-{}-samples.json'.format(mm_samples))
21
22 models_dir = os.path.join(current_path, "../models/sixth_iteration/") # change this to
       select the model
23 model_num = 6
24 model = load_model(models_dir + "{}.h5".format(model_num))
25
26 songs_data = []
27 songs_labels =  []
28 excluded_songs = [
29     '05 - Wild Honey Pie.flac',
30     '06 - The Continuing Story of Bungalow Bill.flac',
31     '11 - Cry Baby Cry.flac',
32     '12 - Revolution 9.flac',
33     '13 - Rocky Raccoon.flac',
34     "14 - Don't Pass Me By.flac",
```

```
]

albums = os.listdir(beatles_labels_basepath)
for album in albums:
    songs = sorted(os.listdir(os.path.join(beatles_labels_basepath, album)))
    for song in songs:
        song_fullpath = os.path.join(beatles_discography_basepath, album, song.split('.lab')[0] + '.flac')
        label_fullpath = os.path.join(beatles_labels_basepath, album, song)
        songs_data.append(song_fullpath)
        songs_labels.append(label_fullpath)

final_scores = {}
duration = None #None equals to full duration
templates = TemplateGenerator.get_triads()

for idx, (song, label) in enumerate(zip(songs_data, songs_labels)):

    if (song.split('/')[-1] in excluded_songs):
        print("Excluding: {}".format(song.split('/')[-1]))
        continue

    print("{} - Processing: {}".format(idx, song.split('/')[-1]), end =" ")

    audio, _ = librosa.load(song, sr=SR, duration=duration)
    # audio = highpass_filter(audio, SR) # Apply the highpass filter

    audio = librosa.util.normalize(audio)
    # First, calculate the librosa chroma
    song_length = librosa.get_duration(audio, SR)
    librosa_cqt = np.abs(librosa.cqt(audio, sr=SR,
                                     hop_length=HOP_LENGTH,
                                     fmin=FMIN,
                                     n_bins=7 * 12,
                                     bins_per_octave=12,
                                     tuning=None))
    librosa_cqt_harmonic, _ = librosa.decompose.hpss(librosa_cqt)
    librosa_chroma = librosa.feature.chroma_cqt(y=None, sr=SR, C=librosa_cqt_harmonic, hop_length=HOP_LENGTH, fmin=FMIN)

    librosa_chords = np.empty(shape=(len(templates), librosa_chroma.shape[1]))
    librosa_predicted_chords = []

    for idx, col in enumerate(librosa_chroma.T):
        predicted_chord = ChordRecognizer.predict_chord(chroma_vector=col, templates=templates)
        chord_index = list(templates.keys()).index(predicted_chord)
        chord_vector = np.zeros( shape=(len(templates)) )
        chord_vector[chord_index] = 1
        librosa_chords[:, idx] = chord_vector
        librosa_predicted_chords.append(predicted_chord)

    # Now, calculate the chroma through the model
    model_cqt = np.abs(librosa.cqt(audio, fmin=FMIN, sr=SR, hop_length=HOP_LENGTH, n_bins=N_BINS, bins_per_octave=BINS_PER_OCTAVE))
    model_cqt_harmonic, _ = librosa.decompose.hpss(model_cqt)
```

```
87      model_cqt_norm = normalize_cqt_rms(model_cqt_harmonic, scale_factor=0.05) # Modify the
            scaling factor here
88      model_chroma = model.predict(model_cqt_norm.T).T
89
90      model_chords = np.empty(shape=(len(templates), librosa_chroma.shape[1]))
91      model_predicted_chords = []
92
93      for idx, col in enumerate(model_chroma.T):
94          predicted_chord = ChordRecognizer.predict_chord(chroma_vector=col, templates=
                templates)
95          chord_index = list(templates.keys()).index(predicted_chord)
96          chord_vector = np.zeros( shape=(len(templates)) )
97          chord_vector[chord_index] = 1
98          model_chords[:, idx] = chord_vector
99          model_predicted_chords.append(predicted_chord)
100
101     # Finally, load the ground truth, and the interval duration
102     start_times = []
103     end_times = []
104     ground_truth_raw = []
105     with open(label) as f:
106         for line in f:
107             start_times.append(float(line.split(' ')[0]))
108             end_times.append(float(line.split(' ')[1]))
109             ground_truth_raw.append(line.split(' ')[2].rstrip())
110
111     ground_truth_final = []
112
113     for chroma_idx, col in enumerate(librosa_chroma.T):
114         current_time = chroma_idx*HOP_LENGTH/SR
115         for labels_idx, label_end_time in enumerate(end_times):
116             if current_time < label_end_time:
117                 ground_truth_final.append(ground_truth_raw[labels_idx])
118                 break
119
120     # We continue if the lists have different lengths; this would mean that the labels do
            not correspond to the song
121     if (len(ground_truth_final) > len(model_predicted_chords)):
122         print("Ground truth labels are longer! Trimming...")
123         ground_truth_final = ground_truth_final[:len(model_predicted_chords)]
124     elif (len(ground_truth_final) < len(model_predicted_chords)):
125         print("Chromagrams are longer! Trimming...")
126         model_predicted_chords =  model_predicted_chords[:len(ground_truth_final)]
127         librosa_predicted_chords =  librosa_predicted_chords[:len(ground_truth_final)]
128
129     if mm_samples > 1:
130         model_predicted_chords = ChordRecognizer.moving_mode(model_predicted_chords,
                mm_samples)
131         librosa_predicted_chords = ChordRecognizer.moving_mode(librosa_predicted_chords,
                mm_samples)
132
133     # Finally, calculate the comparisons and the final score for each set of predictions
134     durations = np.ones(shape=(len(model_predicted_chords)))
135
136     # ROOT
137     root_model_comparisons = mir_eval.chord.root(reference_labels=ground_truth_final,
```

```
            estimated_labels=model_predicted_chords)
138     root_model_score = mir_eval.chord.weighted_accuracy(root_model_comparisons, durations)
139     root_librosa_comparisons = mir_eval.chord.root(reference_labels=ground_truth_final,
            estimated_labels=librosa_predicted_chords)
140     root_librosa_score = mir_eval.chord.weighted_accuracy(root_librosa_comparisons,
            durations)
141     # MAJMIN
142     majmin_model_comparisons = mir_eval.chord.majmin(reference_labels=ground_truth_final,
            estimated_labels=model_predicted_chords)
143     majmin_model_score = mir_eval.chord.weighted_accuracy(majmin_model_comparisons,
            durations)
144     majmin_librosa_comparisons = mir_eval.chord.majmin(reference_labels=ground_truth_final,
             estimated_labels=librosa_predicted_chords)
145     majmin_librosa_score = mir_eval.chord.weighted_accuracy(majmin_librosa_comparisons,
            durations)
146     # TRIADS
147     triads_model_comparisons = mir_eval.chord.triads(reference_labels=ground_truth_final,
            estimated_labels=model_predicted_chords)
148     triads_model_score = mir_eval.chord.weighted_accuracy(triads_model_comparisons,
            durations)
149     triads_librosa_comparisons = mir_eval.chord.triads(reference_labels=ground_truth_final,
             estimated_labels=librosa_predicted_chords)
150     triads_librosa_score = mir_eval.chord.weighted_accuracy(triads_librosa_comparisons,
            durations)
151
152     final_scores.update({
153         song.split('/')[-1]: {
154             'length'    : song_length,
155             'root'      : {'librosa': root_librosa_score,
156                            'model'  : root_model_score},
157             'majmin'    : {'librosa': majmin_librosa_score,
158                            'model'  : majmin_model_score},
159             'triads'    : {'librosa': triads_librosa_score,
160                            'model'  : triads_model_score},
161         }
162     })
163     print("TRIADS  - librosa: {} model: {}".format(triads_librosa_score, triads_model_score
            ))
164     # pprint(final_scores, indent=4)
165
166 with open(results_file, 'w') as f:
167     json.dump(final_scores, f, indent=4)
```

compute_final_score.py

```
1  import json, os
2
3  current_path = os.path.dirname(__file__)
4  # results_path = os.path.join(current_path, "./results/first_iteration")
5  # results_path = os.path.join(current_path, "./results/first_iteration/highpass")
6  # results_path = os.path.join(current_path, "./results/second_iteration")
7  # results_path = os.path.join(current_path, "./results/third_iteration")
8  # results_path = os.path.join(current_path, "./results/fourth_iteration")
9  # results_path = os.path.join(current_path, "./results/fifth_iteration")
10
11 # results_path = os.path.join(current_path, "./results/sixth_iteration/factor_p50")
```

```
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_p75")
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_p90")
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_1")
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_0.75")
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_0.50")
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_0.25")
# results_path = os.path.join(current_path, "./results/sixth_iteration/factor_0.10")
results_path = os.path.join(current_path, "./results/sixth_iteration/factor_0.05")

results_files = [
    os.path.join(results_path, 'results-1-samples.json'),
    os.path.join(results_path, 'results-10-samples.json'),
    os.path.join(results_path, 'results-20-samples.json'),
    os.path.join(results_path, 'results-30-samples.json'),
]

for results_file in results_files:

    with open(results_file, 'r') as f:
        results = json.load(f)

    librosa_values_root = []
    model_values_root = []
    librosa_values_majmin = []
    model_values_majmin = []
    librosa_values_triads = []
    model_values_triads = []
    lengths = []

    for key, value in results.items():
        lengths.append(value['length'])
        model_values_root.append(value['root']['model'])
        librosa_values_root.append(value['root']['librosa'])
        model_values_majmin.append(value['majmin']['model'])
        librosa_values_majmin.append(value['majmin']['librosa'])
        model_values_triads.append(value['triads']['model'])
        librosa_values_triads.append(value['triads']['librosa'])

    total_length = sum(lengths)
    final_librosa_score_root = 0
    final_model_score_root = 0
    final_librosa_score_majmin = 0
    final_model_score_majmin = 0
    final_librosa_score_triads = 0
    final_model_score_triads = 0

    for idx, length in enumerate(lengths):
        final_librosa_score_root += librosa_values_root[idx] * length / total_length
        final_model_score_root += model_values_root[idx] * length / total_length
        final_librosa_score_majmin += librosa_values_majmin[idx] * length / total_length
        final_model_score_majmin += model_values_majmin[idx] * length / total_length
        final_librosa_score_triads += librosa_values_triads[idx] * length / total_length
        final_model_score_triads += model_values_triads[idx] * length / total_length

    print("File: {}".format(results_file.split('/')[-1]))
    # print("ROOT - Librosa score: {} - Model score: {}".format(final_librosa_score_root,
```

```
        final_model_score_root))
68      # print("MAJMIN - Librosa score: {} - Model score: {}".format(
            final_librosa_score_majmin, final_model_score_majmin))
69      # print("TRIADS - Librosa score: {} - Model score: {}".format(
            final_librosa_score_triads, final_model_score_triads))
70
71      print("Model score: {}".format(final_model_score_root))
72      print("Model score: {}".format(final_model_score_majmin))
73      print("Model score: {}".format(final_model_score_triads))
```

ANEXO B: RESULTADOS OBTENIDOS UTILIZANDO FILTRO PASO ALTO

A continuación, se muestran los resultados obtenidos utilizando el modelo entrenado con un tamaño de mini-lotes de 1024 muestras y aprendizaje por currículum, aplicando HPSS y un filtro paso alto (HPF) con una frecuencia de corte de 70 Hz.

Muestras de MM	Criterio	Con HPF	Sin HPF
1 (sin filtro)	ROOT	0.523	0.423
	MAJMIN	0.378	0.246
	TRIADS	0.364	0.238
10	ROOT	0.586	0.513
	MAJMIN	0.464	0.360
	TRIADS	0.447	0.349
20	ROOT	0.598	0.546
	MAJMIN	0.483	0.402
	TRIADS	0.465	0.390
30	ROOT	0.593	0.548
	MAJMIN	0.484	0.416
	TRIADS	0.466	0.403