Universidad Zaragoza
1542

Trabajo Fin de Máster

Análisis Musical mediante Inteligencia Artificial

Musical Analysis through Artificial Intelligence

Autor

Carlos Hernández Oliván

Director

José Ramón Beltrán Blázquez

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2019

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

A mi familia, que siempre me ha apoyado a lo largo de estos años.
A José Ramón, por confiar en mí desde el primer momento dándome la oportunidad de hacer este apasionante trabajo y por seguir haciéndolo.
A David, por estar siempre a disposición para ayudarme con este trabajo y a Cinzia por apoyarme siempre.

RESUMEN

Las Redes Neuronales son una herramienta muy potente para clasificar, procesar y generar nuevos datos. Con respecto a la música, estas redes se han utilizado para componer nuevas melodías, armonizar temas, etc., pero solo unas pocas investigaciones han tenido en cuenta la importancia del análisis musical. En este proyecto se han desarrollado dos modelos de Redes Neuronales que identifican las transiciones de las diferentes partes de la estructura de las piezas musicales y las diferencias entre las transiciones para etiquetarlas. Para ello, se ha realizado un etiquetado de las partes de la estructura formal de piezas musicales a través de una red neuronal y se han detectado las transiciones en la estructura musical a través de técnicas de aprendizaje profundo y aprendizaje automático con Pytorch. Los resultados obtenidos son similares al estado del arte de este trabajo que se ha tomado como ejemplo para desarrollar este software.

Este proyecto consta de un primer capítulo de introducción, el segundo capítulo explica las características de la teoría de las Redes Neuronales que se han utilizado en este proyecto, el tercer capítulo expone el caso del etiquetado de estructuras, el cuarto capítulo estudia el caso de detección de transiciones y el quinto capítulo compara los resultados obtenidos con el estado del arte. El sexto capítulo expone las conclusiones y las líneas futuras.

ABSTRACT

Neural Networks are a strong tool to classify, process and generate new data. Regarding to music these networks have been used in order to compose new melodies, harmonize, etc, but only a few investigations have taken into account the importance of music analysis. In this project it has been developed two models of Neural Networks which identifies the transitions of the different parts of the structure of the musical pieces and the differences between the transitions in order to label them. To do that, it has been done a transition labeling through a neural network and the detection of music structure boundaries through Deep Learning and Machine Learning techniques with Pytorch. The obtained results are similar to the state of art decrived in this work which has been taken as an example to develop this software.

This project consists on one chapter of introduction, the second chapter explains the characteristics of the Neural Networks theory which have been used in this project, the third chapter exposes the case of structure labeling, the fourtth chapter studies the case of boundary detection and the fifth chapter compares the results with the state of the art. The sixth chapter exposes the conclusion and future work.

Índice de Contenido

Índice de Figuras

Índice de Expresiones

Índice de Tablas

Capítulo I

Introducción

En este capítulo se realiza una introducción al trabajo exponiendo los objetivos fijados al comienzo del mismo, así como la presentación de la Inteligencia Artificial y su relación e importancia en su aplicación al campo de la música como el análisis musical, principal objeto de estudio en este trabajo.

1.1. Motivación

La Inteligencia Artificial está creciendo considerablemente dados sus múltiples campos de aplicación en economía, medicina, artes, etc, y sus extraordinarias posibilidades como la predicción de enfermedades o mejora y optimización de procesos. Uno de los padres de la Inteligencia Artificial fue Marvin Minsky, que junto a a John McCarthy fundó uno de los primeros laboratorios en el MIT dedicados a este campo, además de SNARC en 1951, el primer simulador de redes neuronales artificiales.

Dentro de la Inteligencia Artificial se encuentra el denominado *Machine Learning*, que es un subcampo de la Inteligencia Artificial donde los algoritmos son capaces de aprender sin haber sido previamente programados para ello explícitamente. Por otro lado, el *Deep Learning* es un subcampo del *Machine Learning* donde las redes neuronales son capaces de aprender en base a datos existentes.

Esto conceptos han sido introducidos a lo largo de la historia reciente donde la utilización de técnicas de *Machine Learning* y *Deep Learning* han dado paso a nuevas líneas de investigación.

En la última década el crecimiento del interés en *Machine Learning* y *Deep Learning* ha sido exponencial. Esto es debido al potencial de estas técnicas y sus amplios campos de aplicación. En el caso de este trabajo se van a utilizar técnicas de *Machine Learning* y *Deep Learning* en el ámbito musical, más concretamente en el análisis musical ya que es un problema todavía por resolver y que puede ser de utilidad para posteriores estudios donde se busque crear obras a partir de estas técnicas, ya que una vez analizada una pieza es mucho más fácil componer basándose en su estructura. Además, puede servir de utilidad para profesionales de la música como herramienta de análisis musical, siempre verificando sus resultados con un análisis realizado por un profesional de la música.

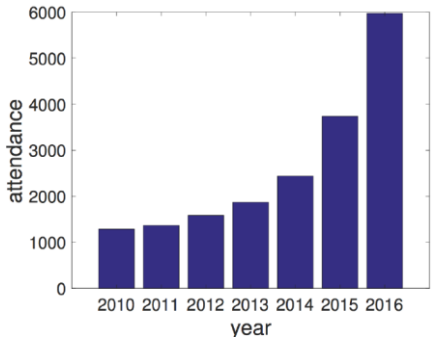


Figura 1. Asistentes a congresos sobre Machine Learning en los últimos años [1]

1.2. Objeto

El análisis musical es necesario para la composición musical, ya que los humanos componen en base a las obras que han analizado anteriormente, de ahí el interés en diseñar un sistema de análisis que sirva después en líneas futuras para la ccomposición musical automática. Cabe destacar a su vez que el análisis musical es algo subjetivo ya que, en obras no clásicas, no existe un acuerdo sobre las estructuras formales de las piezas

musicales y, depende del músico que analice la obra, los resultados pueden variar por lo que los resultados que se obtengan en este trabajo serán tendrán también un lado de subjetividad.

El objeto de este trabajo es desarrollar una red neuronal en Pytorch que permita realizar un anális estructural de piezas musicales a partir de la detección de transiciones en la estructura de la pieza, con el objetivo de desarrollar una herramienta que pueda ser utilizada tanto como por profesionales de la música como por investigadores en Inteligencia Artificial y que además pueda servir como base para futuras investigaciones. El software desarrollado está formado por diversos módulos o *scripts* que sirven para modelizar y etrenar la red, que son:

- Módulos previos donde se descarga la base de datos y se transforma en la forma de datos de entrada de la red neuronal utilizada.
- Un módulo donde se genera el *dataloader* con el que se entrenará y validará la red, los modelos de las redes neuronales a utilizar.
- Un módulo con los bucles de entrenamiento y validación de la misma.
- Un módulo donde se comprueban los resultados del entrenamiento.
- Un módulo de test donde se obtiene la precisión final del modelo.

En los capítulos III y IV se nombran los módulos específicos programados para los dos casos de estudio de este trabajo.

1.3. Alcance

El trabajo se centra en dos casos dentro del proceso completo de composición musical mediante Inteligencia Artificial, que son la identificación de las transiciones en la estructural formal de las piezas musicales y la obtención de un análisis estructural de dichas piezas teniendo en cuenta la información temporal del audio. Para ello, en primer lugar, se han revisado los estudios previos en análisis musical con redes neuronales pudiendo observarse que hasta ahora en ninguno se resuelve el problema planteado de reconocer estructuras musicales mediante el uso de redes neuronales. Después, en base a estos estudios previos se ha decidido así trabajar con el formato audio .mp3 y no MIDI como proponen diversos estudios previos, se ha escogido una base de datos de audio .mp3 etiquetada conforme a la estructura de las canciones en .txt y se ha procesado la señal de audio transformando el audio de las canciones a matrices como se describirá más adelante. Posteriormente, se han transformado las etiquetas mediante la programación en Python a un formato que pueda entender la red y se ha desarrollado el esquema de la red neuronal, programando a posteriori su correspondiente modelo en Pytorch. Para finalizar se ha entrenado y validado el modelo y se han ajustado los parámetros necesarios para hacer el modelo lo más eficiente posible, y se han evaluado los resultados calculando la efectividad del modelo.

En la figura 2 se muestra un posible diagrama de bloques con las etapas principales del proceso global de composición musical automática, que sería el objetivo a muy largo plazo. El orden de las etapas no tiene por qué ser necesariamente el mostrado en la figura 2 ya que existen estudios de, por ejemplo, identificación armónica o separación de

instrumentos que no realizan un análisis formal de la obra musical previo. El diagrama muestra cuál sería el orden lógico según la teoría de análisis musical.

En este trabajo, se ha realizado el caso del etiquetado de transiciones (capítulo III) antes que el de detección de transiciones (capítulo IV), ya que se ha intentado hacer el etiquetado de las transiciones sin previamente haber realizado la detección pero al obtener resultados no idóneos se ha decidido realizar la detección para en líneas futuras realizar el etiquetado a partir de la detección y mejorar los resultados obtenidos en este trabajo.

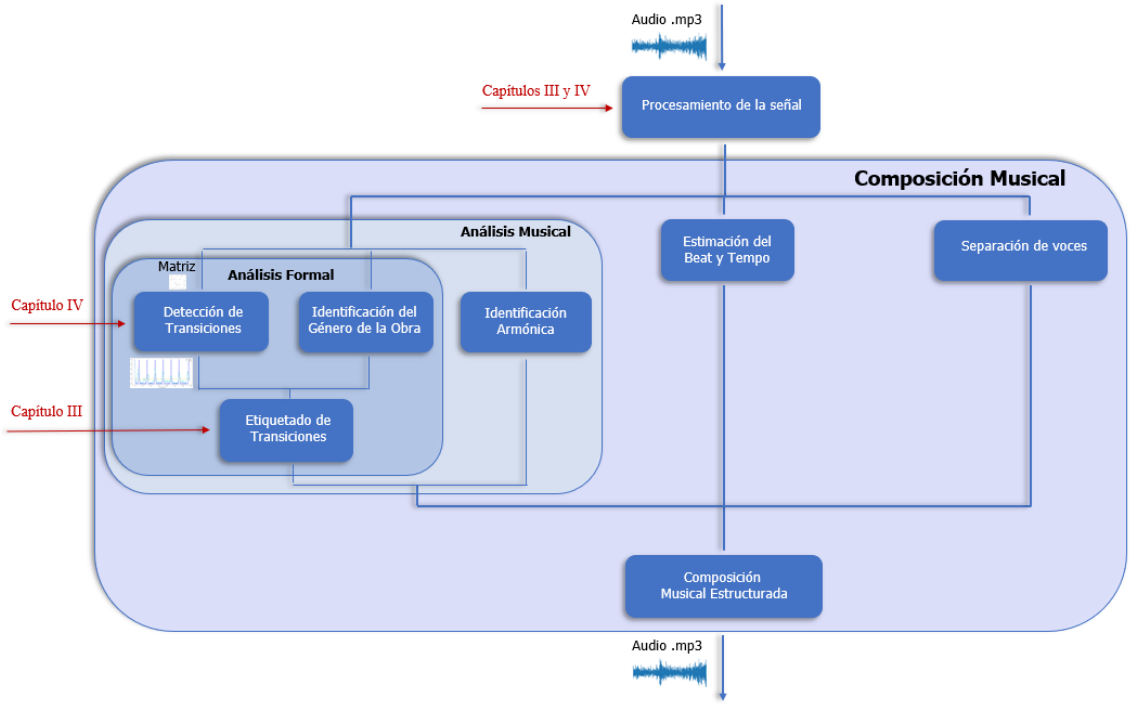


Figura 2. Esquema general del proceso global de composición con Inteligencia Artificial. La zona resaltada en naranja corresponde a los pasos realizados en este trabajo

1.4. Descripción de la Memoria

La memoria se compone de cinco capítulos, este primer capítulo de introducción donde se expone la motivación que ha llevado a realizar este trabajo de investigación, sus objetivos, alcance y fases de desarrollo.

A continuación, en el capítulo II se realiza una descripción del estado del arte en el cual se sitúa este trabajo y a continuación se realiza una introducción general al Análisis Musical y su importancia en esta línea de investigación, se introducen los conceptos de Inteligencia Atificial, *Machine Learning* y *Deep Learning* y se realiza una introducción detallada de la tipología de redes neuronales que se utilizan en los dos casos particulares de este trabajo.

En el capítulo III se expone el caso de **etiquetado de la estructura** formal de piezas musicales, es decir, dar nombre a cada transición en una pieza musical. Las etapas que se han seguido para llevar a cabo el trabajo van desde la obtención de la base de datos hasta la validación del modelo, pasando por la elección de los parámetros y tipología concreta

de la red neuronal, y la descripción de las principales funciones utilizadas en Pytoch para su ejecución.

En el capítulo IV se realiza el segundo caso planteado en este trabajo que es la **detección de transiciones** en la estructura formal de piezas musicales. Las etapas seguidas son las mismas que en el capítulo anterior.

En el capítulo V se realiza una comparación de los resultados de los casos estudiados en este trabajo y se comparan con los resultados obtenidos en los modelos de referencia.

Finalmente, se exponen las conclusiones y líneas futuras del trabajo.

Capítulo II

Inteligencia Artificial, Deep Learning y Machine Learning

En este capítulo se describen el estado del arte y las técnicas de Deep Learning y Machine Learning a nivel general y específico en el campo de la composición musical. Además, se describen las características y las tipologías de redes neuronales utilizadas para la resolución de los casos planteados.

2.1. Introducción al Análisis Musical

El análisis musical [2] es una disciplina musical que estudia la estructura musical tratando de obtener un conocimiento total de la obra. Su objetivo es la detección de la estructura interna de la pieza buscando comprender la forma en que el compositor encuadra la misma en un estilo concreto y comprender así las características de cada estilo. El concepto de análisis musical ha ido evolucionando a lo largo de la historia y ha ido incluyendo distintos tipos de análisis que se pueden realizar sobre las obras musicales. Sobre una obra musical se puede realizar un **análisis estilístico**, un **análisis formal**, que es el que se encarga de extraer las partes en las que el compositor ha estructurado la pieza y es el que se busca realizar en este trabajo y un **análisis armónico**, entre otros.

Un ejemplo análisis de la estructura formal de una obra se muestra en la figura 3:

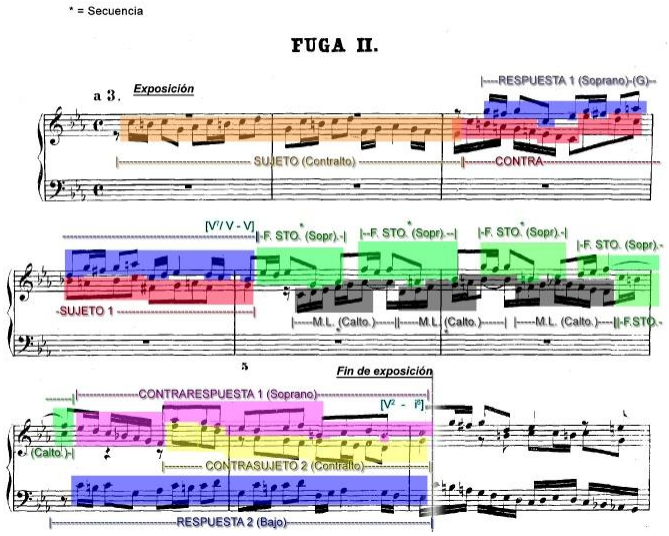


Figura 3. Ejemplo de análisis de la exposición de la Fuga II en Do m BWV874 de Bach [3]

2.2. Orígenes de la Inteligencia Artificial

La Inteligencia Artificial es un campo de la ciencia e ingeniería que trata de construir entidades inteligentes [4]. Existen cuatro categorías que definen la Inteligencia Artificial: pensar humanamente, actuar humanamente, pensar racionalmente y actuar racionalmente. La primera prueba creada para comprobar que una máquina pudiese actuar humanamente fue el Test de Turing propuesto en 1950 por Alan Turing, considerado como el creador de la Inteligencia Artificial. El test consistía en responder a una serie de preguntas formuladas por un interrogador humano que debía especificar al final del test si habían sido respondidas por un humano o una máquina. Para ello, la máquina ha de poseer un lenguaje de procesamiento natural, una forma de representar el conocimiento, una forma automática de razonar y el conocido como *Machine Learning*, que se basa en la idea de que la máquina se autoprograme, es decir, abandonar la idea de programar la máquina de tal forma que realice aquello que se desea y proveer a la máquina de conocimiento para que ella misma aprenda a programarse prediciendo posibles estados futuros. Unos años antes, en 1943, Warren McCulloch y Walter Pitts [5] ya habían publicado el primer trabajo que existe sobre Inteligencia Artificial que proponía un modelo de redes de neuronas artificiales cuyo estado podía ser ”on” u ”off”. También sostuvieron la idea de

que estas redes de neuronas podían aprender, y en 1949, Donald Hebb desarrolló su conocida como regla de Hebb. Más tarde, en 1950, Marving Minsky y Dean Edmonds construyeron la primera red neuronal en un ordenador y desde entonces el interés y desarrollo de aplicaciones basadas en la Inteligencia Artificial no ha hecho más que crecer estructurando sus algoritmos en diversas técnicas. En la figura 4 se puede ver la estructura de los conceptos de Inteligencia Artificial, *Machine Learning* y *Deep Learning*.

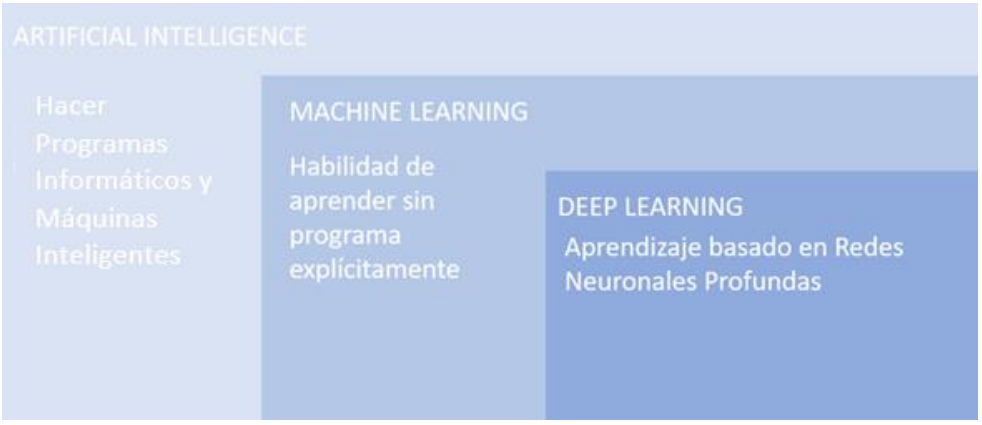


Figura 4. Inteligencia Artificial, *Deep Learning* y *Machine Learning*

2.2.1. Machine Learning

El *Machine Learning* [4] es una disciplina científica del ámbito de la Inteligencia Artificial que crea sistemas que aprenden automáticamente en base a la experiencia e intuición. El *Machine Learning* nace de la mano de la Inteligencia Artificial y se desarrolla principalmente en los años 50 cuando Arthur Samuel programó el primer software de aprendizaje informático. En los años 60 se crea el algoritmo *Nearest Neighbor* que permitía el reconocimiento automático de patrones básicos. A partir de ahí en los años 90 el *Machine Learning* gana popularidad gracias a la inserción de modelos estadísticos e informáticos y sigue en auge en la actualidad con la incorporación del *Big Data* y minería de datos.

2.2.2. Fundamentos del Deep Learning

Una parte de la Inteligencia Artificial es el *Deep Learning* o Aprendizaje Profundo. El Deep Learning se basa en la utilización de capas de algoritmos estructuradas en redes neuronales que procesan y transmiten la información de una capa a otra, desde la capa de entrada hasta la capa de salida pasando por capas ocultas. En la figura 5 se muestra una línea temporal del desarrollo de las distintas técnicas de *Deep Learning*.

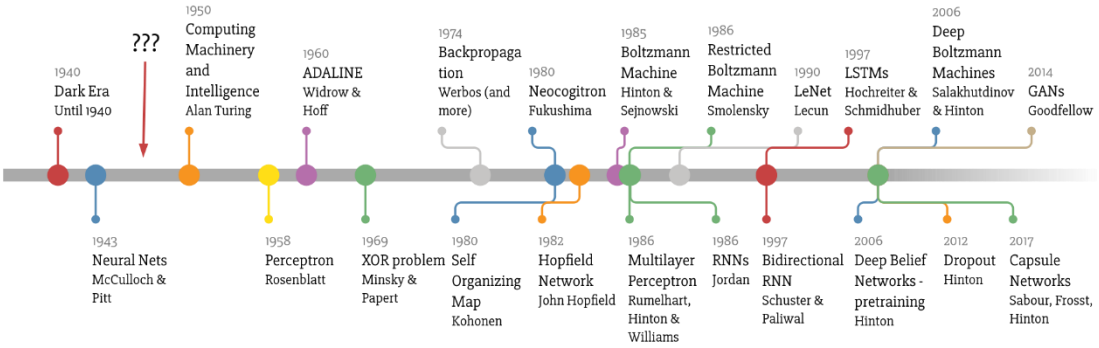


Figura 5. Línea temporal del *Deep Learning* [6]

A pesar de que las redes neuronales llevan décadas desarrolladas, el concepto de *Deep Learning* es bastante reciente. En 2006 Geoffrey Hinton, Simon Osindero y Yee-Whye Teh publican el artículo "*A fast learning algorithm for deep belief nets*" [7] con el que asientan las bases del *Deep Learning*, y es a partir de ese momento cuando comienza el desarrollo de las denominadas como Redes Neuronales Profundas o *Deep Neural Networks* y optimización de los modelos de redes neuronales que resuelven problemas actuales complejos como el reconocimiento de imágenes, procesado de texto o reconocimiento de objetos, gracias al desarrollo de nuevos algoritmos de optimización, nuevo Hardware con unidades de procesamiento gráficas o GPUs y al denominado como *Big Data*.

Actualmente y desde 2012 cuando Hinton con su equipo consiguen ganar a Imagenet utilizando una Red Neuronal Profunda, se combinan técnicas de *Machine Learning* y *Deep Learning* para crear modelos que se aproximen mejor a los problemas que se requieren en la actualidad.

Dentro del *Deep Learning* se tienen tres tipos de aprendizaje, el aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje reforzado dependiendo si se le provee de información en forma de objetivos a la máquina para que aprenda intentando llegar a esos objetivos o no.

2.3. Introducción a las Redes Neuronales Artificiales

Las Redes Neuronales tratan de imitar el complejo procesamiento que realiza el cerebro humano para resolver las múltiples tareas que intentan imitar estas redes artificiales. A raíz de la información existente sobre las neuronas y sus conexiones en el cerebro humano se puede realizar un modelo computacional del cerebro que trate de imitar estas conexiones con el fin de crear una herramienta que trate la información de forma más eficaz y eficiente que la programación estándar.

Antes de comenzar con la descripción formal de las redes neuronales es necesario saber cómo se distribuyen los datos de entrada de la red y cómo se realizan predicciones o estimaciones a partir de dichos datos. Una red neuronal toma datos de entrada de una base de datos distribuida en conjuntos de datos de entrenamiento, validación y test. Los datos de **entrenamiento** sirven como entrada de la red para que ésta optimice sus pesos durante

el entrenamiento mediante los procedimientos que se describirán posteriormente en este capítulo. Una vez la red ha calculado sus parámetros en base a los datos de entrenamiento, se introducen como entrada a la red los datos de **validación**, que que sirven para comprobar si la red es capaz de generalizar o no, es decir, si la red ha aprendido características específicas de los datos de entrenamiento (esto es conocido como sobreentrenamiento u *overfitting*) o por el contrario, si ha aprendido características globales que le permiten comprender datos que no ha visto durante el entrenamiento, lo que quiere decir que el modelo es correcto. También puede suceder que el modelo no se haya entrenado lo suficiente, lo que se conoce como *underfitting*. Por último, los datos de **test** sobre los cuales se evalúa la eficacia del modelo. Normalmente un dataset se distribuye en 60% de datos para entrenamiento, 15% de validación y 25% de test, aunque estos porcentajes dependen del tamaño del dataset. En la figura 6 se muestran ejemplos de entrenamiento de una red neuronal.

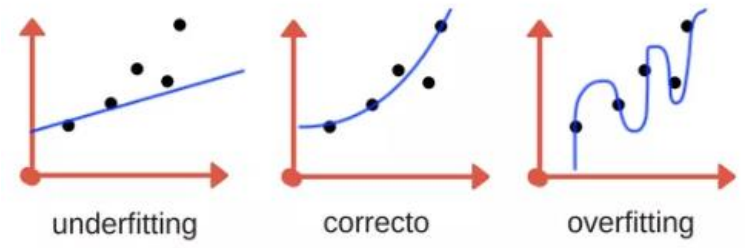


Figura 6. Tipos de resultados de entrenamiento (a) *underfitting* (b) correcto (c) *overfitting* [8]

Una vez introducido el marco de cómo se estructuran los datos de entrada de una red neuronal, es necesario conocer cómo es la estructura básica de la misma. Se puede representar la neurona artificial como se muestra en la figura 8, donde w_i son los pesos, que son unos parámetros que se actualizan conforme la red aprende de los datos de entrada, b_o es el bias y $g(z)$ la función de activación, que siendo no-lineal ayuda a modelar funciones complejas, es decir, la función de activación introduce la no-linealidad al modelo. La entrada se denota como x_i y la salida como y.

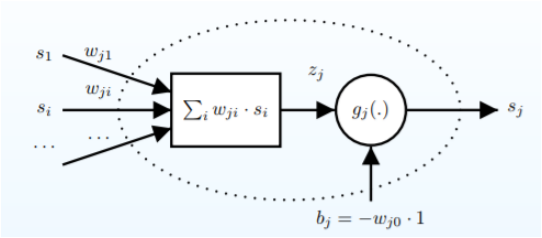


Figura 7. Estructura de la neurona artificial [9]

Así pues, la salida de la neurona viene dada por la siguiente expresión:

$$y = g\left(b_o + \sum_{i=1}^{N} x_i . w_i\right) \tag{1}$$

Si se escribe la expresión anterior en forma matricial se tiene la expresión (2):

$$Y = g(X.W + b_o) \tag{2}$$

donde W es la matrix correspondiente a los pesos y X el vector de entrada:

$$W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \qquad \text{y} \qquad X = [x_1 \quad \dots \quad x_n]$$

Para una red con varias neuronas, existirán varias salidas por lo que las expresiones (1) y (2) se escribirán como las expresiones (3) y (4) respectivamente, siendo esta última la expresión (3) en forma matricial:

$$y_j = g_j\left(b_{oj} + \sum_{i=1}^{N} x_i.w_{ij}\right) \tag{3}$$

$$y_j = g_j\left(X.W_j + b_{oj}\right) \tag{4}$$

Durante el entrenamiento de las redes neuronales, el error en cada ejemplo de entrenamiento se calcula mediante la **función de pérdida**, de error, de coste o ***Loss Function*** $L(\hat{y}, y)$ que se calcula comparando la salida de la red y con el objetivo $\hat{y}$. Esta función es la media de las funciones de pérdida para cada dato de entrenamiento. Existen distintas funciones de pérdida dependiendo del problema a resolver y de cómo sean los datos de entrada y salida que se quieren obtener, como la MSE (Error Cuadrático Medio o *Mean Squared Error*), BCE (*Binary Cross Entropy*), etc. La expresión de la función de coste MSE se muestra a continuación:

$$L(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{5}$$

donde m es el número de entradas de entrenamiento, $y^{(i)}$ es el vector de la i-ésima entrada del dataset entrenamiento[1], $\hat{y}^{(i)}$ es la etiqueta asociada a la entrada i-ésima del dataset entrenamiento.

Estas funciones de coste se minimizan con los **algoritmos de optimización**. Estos algoritmos son procesos iterativos que buscan los valores óptimos de los parámetros del modelo. Existen varios algoritmos de optimización dependiendo de cómo se busca minimizar la función de coste siendo los más utilizados el gradiente descendente y el gradiente descendente estocástico.

Las redes neuronales artificiales aprenden gracias a la propagación hacia atrás o algoritmo de *Backpropagation*. Este algoritmo introducido en 1986 por Geoffrey Hinton, David Rumelhart y Ronald Williams [10] se basa en el cálculo y optimizacion de los pesos de la red desde la entrada hasta la salida y su propagación hacia atrás que consiste en invertir la dirección de cálculo desde la salida hasta la entrada para optimizar los pesos de la red.

[1] Conjunto de datos de entremaniento

2.4. Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales (CNN) [11] son conocidas por sus aplicaciones en imagen. En este tipo de redes [12], cada capa de convolución está organizada en tres etapas principales: etapa de convolución, activacion y *pooling*, y cada estructura de CNN tiene más o menos etapas dependiendo del problema que se quiera resolver.

La **etapa de convolución** consiste en aplicar un filtro o *kernel* a las imágenes de entrada de tal forma que sea capaz de extraer un determinado número de características que elige el usuario. Estas características se denominan mapas de características o *feature maps*, donde cada una de ellos comparte los mismos pesos. A continuación, se transfiere la información a una **etapa de activación** no-lineal, que en las CNN suele ser una función ReLU (ver apartado 2.4.2). Finalmente se utiliza una **etapa de *pooling*** que lo que hace es reducir las dimensiones de los datos para que sean más fáciles de computar. Cada una de estas tres etapas conforman una capa de convolución y la sucesiva aplicación de capas convolucionales conforman la CNN. Al final de las capas de convolución se introducen capas lineales o *fully connected* que clasifican las características extraídas por las etapas convolucionales (ver figura 8). Al final de estas capas se introduce una función de activación que dependerá del tipo de datos y etiquetas del que se provea a la red, siendo las más utlizadas la función sigmoide, si se realiza una clasificación binaria, o una softmax si la clasificación es muticlase (ver figura 9).

En las CNN lo habitual es que las imágenes de entrada tengan las mismas dimensiones. En el caso de este trabajo y como se detallará en el capítulo III, al introducir imágenes obtenidas a partir de canciones en la red los tamaños de entrada son distintos para lo que existen técnicas que se describirán más adelante en este capítulo.

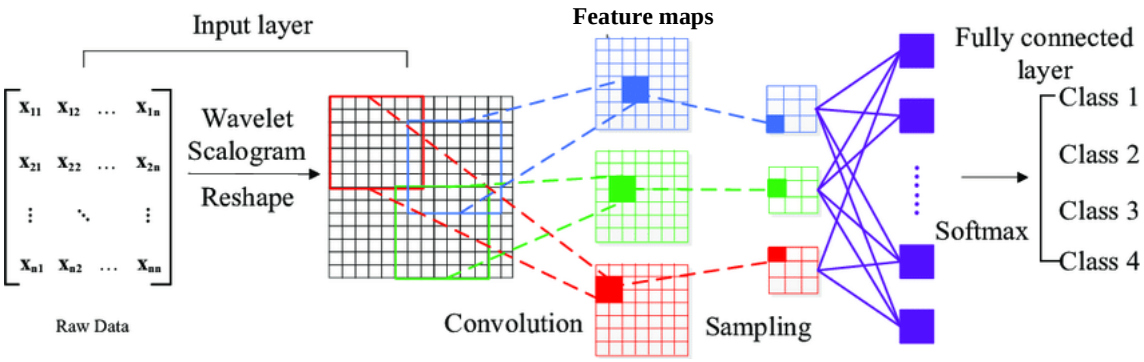


Figura 8. Estructura de una red convolucional [13]

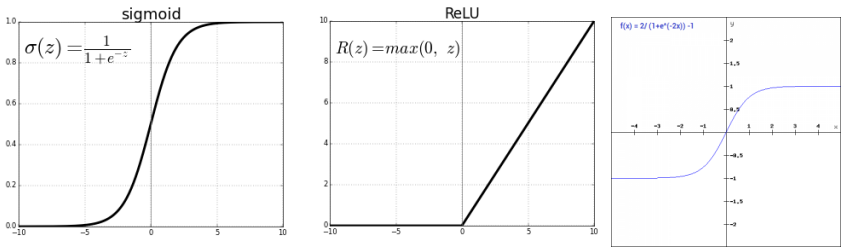


Figura 9. Funciones de activación (a) Sigmoide (b) ReLU (c) Softmax [14]

2.4.1. Etapa de Convolución. Kernel, Stride y Padding

Una convolución es una función lineal que transforma dos señales de entrada en una señal de salida (ver figura 10). Para definir una convolución lo primero que se tiene que definir es el tamaño del núcleo o ***kernel***. En Pytorch se realiza la convolución mediante la función `nn.Conv2d`. Para calcular las dimensiones de salida se tienen en cuenta los siguientes parámetros:

- ***Input*** = tamaño de la imagen de entrada de dimensiones (N, C_{in}, H, W), donde N es el número de imagen del *dataset* a procesar y C_{in} el número de mapas de características o canales de la imagen (N es el *batch* que en este trabajo es igual a 1 ya que todas las imágenes tienen tamaños distintos y hay que introducirlas una a una en la red, por tanto, N será la imagen a procesar).
- ***Output*** = dimensiones de salida (N, C_{out}, H, W) que se calculan como se muestra en la expresión 3. El número de *kernels* que se fijen en como parámetro será el números de mapas de características (C_{out}) que tenga la salida.
- **Número de *kernels*** = Número de mapas de características o *feature maps* que se quieren en la salida.
- ***Kernel*** = tamaño de núcleo utilizado. Un kernel 2-d se denomina también filtro.
- ***Stride*** = Indica el desplazamiento en píxeles del *kernel* sobre la matriz de entrada.
- ***Padding*** = Número de ceros que se añaden a los márgenes de la matriz de entrada para modificar el tamaño de salida de la capa.

$$\text{out}(N_i, C_{out}) = \text{bias}\left(C_{out_j}\right) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) * \text{input}(N_i, k) \quad (6)$$

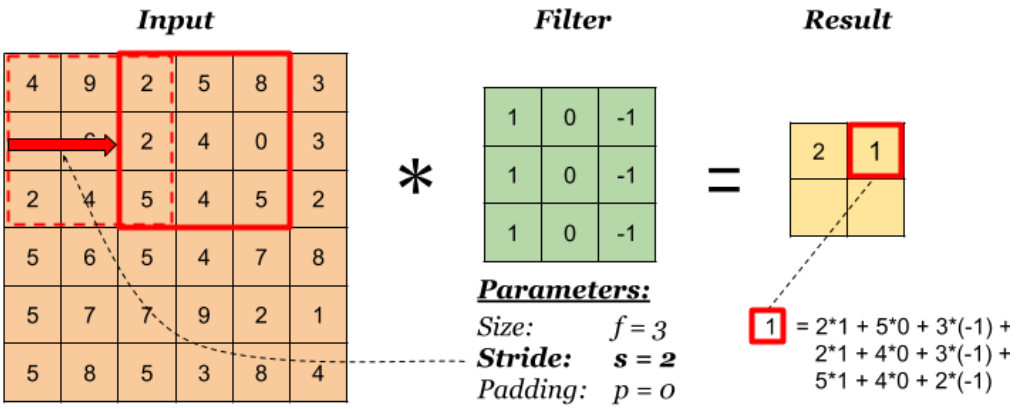


Figura 10. Operación de convolución [15]

2.4.2. Etapa no lineal. Función ReLU y LeakyReLU.

La etapa siguiente a la etapa de convolución es la etapa no lineal. Esta capa añade no linealidad a los datos provenientes de la etapa convolucional. Esta función normalmente es la función ReLU, que iguala a cero los elementos negativos de la matriz tras la

aplicación de la etapa de convolución y deja iguales los elementos positivos. Así pues, la salida de esta etapa se calcula como:

$$f(x) = \max(0, x) \tag{7}$$

En la figura 11 se muestran las funciones de transferencia de la función ReLU y la función leakyReLU que es una variante de la función anterior y que se explica más adelante:

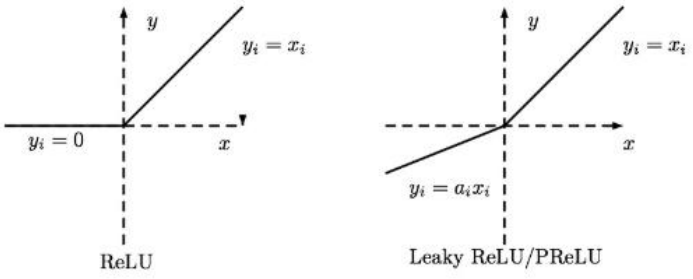


Figura 11. Funciones (a) ReLU (b) LeakyReLU [16]

Esta función acelera la convergencia del gradiente comparado con las funciones sigmoide y softmax. Se utiliza la función ReLU en las CNN en lugar de otras funciones de activación porque no satura, es decir, el gradiente es siempre alto (igual a uno si la neurona se activa) frente a gradientes bajos que dan las funciones sigmoide y softmax que saturan si la entrada es muy alta o baja y por consecuencia dan gradientes muy bajos.
En vez de utilizar la función ReLU se pueden utilizar funciones similares como la PreLU, RReLU o la denominada **LeakyReLU**, cuya ventaja respecto a la función ReLU es que resuelve el problema que tiene esta última cuando los gradientes son altos que pueden llevar a la activación de todos los pesos de tal forma que se llegue al punto de no activar las neuronas y, por tanto, dar todos los gradientes nulos. La salida de la función LeakyReLU se calcula como:

$$\text{LeakyReLU}(x) = \begin{cases} 0.01x & \text{para } x \leq 0 \\ x & \text{para } x > 0 \end{cases} \tag{8}$$

2.4.3. Etapa de Pooling

La tercera y última etapa de una convolución es un *subsampling* o *pooling*. Cuando la convolución ha extraído las características de la imagen el siguiente paso es la aplicación de un *pooling* mediante el cual se reduce el número de parámetros de salida de la capa convolucional. Existen diversas formas de realizar esta etapa, las más habituales son: *max pooling*, *average pooling* y *sum pooling*. En este trabajo se utilizará el *max pooling*.

Un ejemplo de estas técnicas se puede ver en la figura 12:

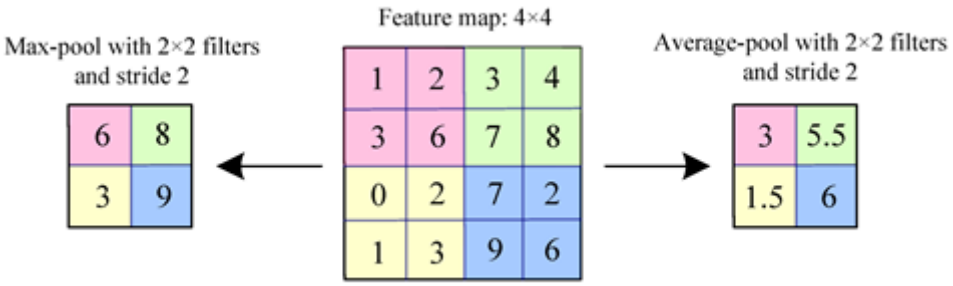


Figura 12. Ejemplos de *pooling* [17]

El *max pooling* en Pytorch se puede definir como `nn.MaxPool2d`. Los parámetros de esta etapa son:

- **Input** = tamaño de la imagen de entrada de dimensiones (N, C, H_{in}, W_{in}), donde N es el número de imagen del *dataset* a procesar y C el número de mapas de características o canales de la imagen.
- **Output** = dimensiones de salida (N, C, H_{out}, W_{out}) que se calculan como [18]:

$$H_{out} = \left[\frac{H_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1}{stride[0]} + 1\right] \quad (9)$$

$$W_{out} = \left[\frac{W_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1}{stride[1]} + 1\right]$$

El parámetro *dilation* hace referencia a los espacios que se dejan entre los píxeles que coge el *kernel*.

2.4.3.1. Adaptive Pooling

Para el caso particular de este trabajo, dado que los tamaños de las imágenes de entrada de la red son variables ya que cada canción tiene una duración distinta, es necesario introducir una capa de *pooling* especial en la red que no fije el tamaño del vector de entrada sino el de salida, de tal forma que la propia red ajuste la entrada para obtener las dimensiones de salida deseadas. Una técnica que permite hacer esto es el *adaptive pooling*. En la red de este trabajo se utilizará esta etapa de *pooling* en una de las capas de convolución para unificar las dimensiones de los vectores de las imágenes para obtener al final de la red una salida de iguales dimensiones para todas las imágenes. En Pytorch se implementa este tipo de *pooling* mediante la función `nn.AdaptiveMaxPool1d`.

2.4.4. Fully Connected Layers

Una vez las imágenes son procesadas y se han extraído las características principales mediante las capas de convolución se utilizan capas *fully connected* para clasificarlas. La función principal de estas capas es extraer las características de la salida de la CNN y conectarlas a una función de clasificación como la *softmax* o la *sigmoide*. La salida final de la capa *fully connected* se transformará en un vector unidimensional de *n* valores correspondientes a las *n* clasificaciones si se trata de un problema multiclase.

La expresión (4) descrita anteriormente en este capítulo es la ecuación matricial referente a una red neuronal con más de una capa capa de neuronas.

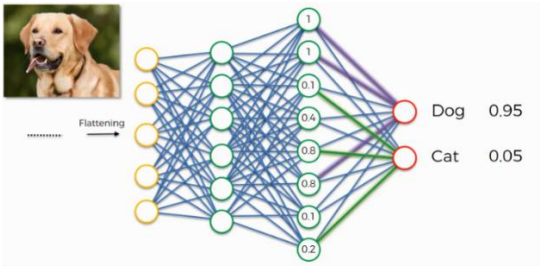


Figura 13. Ejemplo de fully connected layers en clasificación de imágenes [19]

2.4.5. Dropout

El *Dropout* [20] consiste en anular un porcentaje de los pesos de la red para evitar que el modelo aprenda características muy específicas del dadtaset de entrenamiento y que sobreentrene. Esta capa se puede añadir después de cada convolución como se verá en los capítulos III y IV de este trabajo.

2.4.6. Batch normalization

La capa *Batch Normalization* [21] se utiliza para reducir la *Covariate Shift*, que es el cambio en la distribución de las variables de entrada presentes en cada lote o *batch* de entrenamiento y validación. Esta capa permite tener una distribución fija en cada neurona de las capas ocultas. Como el *batch size* en este trabajo es igual a 1 (se introduce una imagen por lote a la red ya que cada imagen tiene una resolución distinta), esta capa no es estrictamente necesaria.

2.5. Redes Neuronales Long Short Term Memory (LSTM)

Las redes *Long Short Term Memory* (LSTM) son un tipo de redes recurrentes propuestas en 1997 por Sepp Hochreiter y Jürgen Schmidhuber [22]. Las redes recurrentes son un tipo de red neuronal que se diferencian por formar ciclos o bucles denominados conexiones recurrentes. Las conexiones recurrentes pueden ser de una neurona con ella misma (retroalimentación), entre neuronas de una misma capa o entre neuronas de una capa con neuronas de la capa anterior. Así como el campo de aplicación de las CNN es el procesamiento de imágenes, las redes LSTM se basan en el análisis de señales con memoria y, por tanto, sus campos de aplicación son más variados que otro tipo de redes y se aplican, por ejemplo, en la composición musical, procesamiento de texto, etc.

Existen dos formas de representar las redes neuronales recurrentes. En la figura que se muestra a continuación se muestran las dos formas, una representando la recurrencia en forma de realimentación y la otra mediante conexiones en la dirección de computación, donde el parámetro *x* es la entrada de la red y *h* el estado en un tiempo *t*.

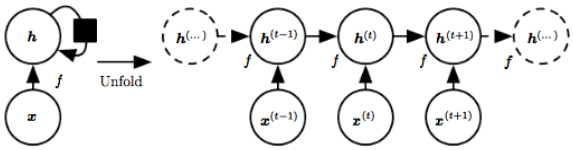


Figura 14. Representación de una red neuronal recurrente [23]

La figura anterior se puede desglosar en la figura 15. El parámetro *y* es el objetivo, *U*, *W* y *V* los pesos entre la capa de entrada y la primera capa oculta, entre la primera capa oculta y la segunda, y entre la segunda capa oculta y la salida respectivamente. *L* es la pérdida o error que mide la distancia entre la salida (*o* en la figura 15) y el objetivo *y*.

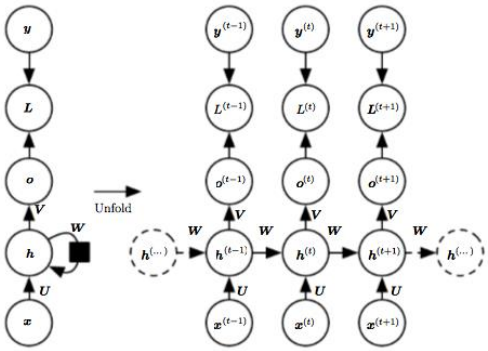


Figura 15. Representación de una red neuronal recurrente [23]

Entrando más a fondo en la arquitectura de una red LSTM, el elemento básico de esta tipología de redes es la célula de memoria o *memory-cell* (figura 16), que se compone de 4 elementos principales: una puerta de entrada, una neurona con retroalimentación, una puerta de salida y una puerta denominada *forget gate*. El valor del peso de la retroalimentación es 1, y esto asegura que el estado de la célula de memoria permanezca constante en cada paso de tiempo o *timestep*. En la figura 17 se muestran varias células de memoria LSTM concatenadas.

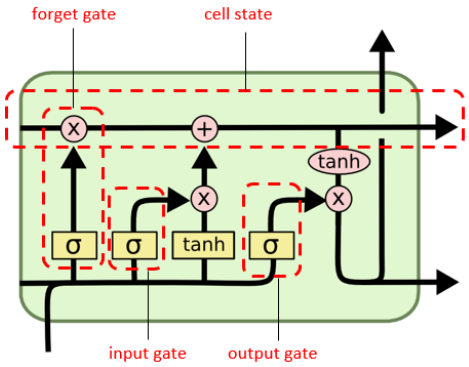


Figura 16. Célula individual de memoria LSTM con sus respectivas puertas [24]

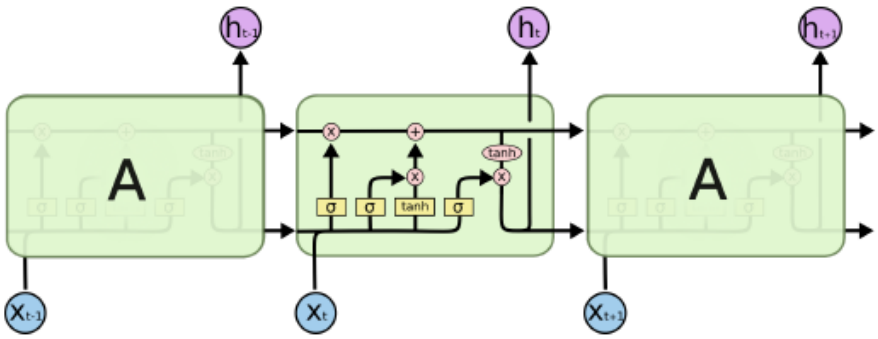


Figura 17. Células de memoria LSTM concatenadas [24]

Las células de memoria LSTM tienen distintas puertas que son unidades multiplicativas con activación continua y son compartidas por todas las celdas que pertenecen a un mismo bloque de memoria. Las puertas controlan la célula de estado o *cell state*. Existen 3 tipos de puertas, la **puerta de entrada** se encarga de permitir o impedir el acceso de los valores provenientes de la entrada y salidas de la red en el instante anterior al interior de la celda. La **puerta de salida** tiene una acción similar a la de entrada y la puerta ***forget gate*** puede modular la retroalimentación de la célula permitiéndole recordar u olvidar su estado previo, cuando sea necesario.

La célula de memoria LSTM puede añadir o quitar información de la célula de estado C_t mediante la regulación de las puertas. Las puertas contienen una capa sigmoide seguida de un operador multiplicación. La sigmoide, que comprime los valores en el rango 0, 1 dará una salida más cercana a 1 o a 0 dependiendo de cuánto considere que la información de la célula de estado es o no valiosa para la red, respectivamente.
Así pues, el primer paso en una célula de memoria LSTM es decidir qué información se debe descartar de la célula de estado C_{t-1}, siendo tomada esta decisión por la capa sigmoide.

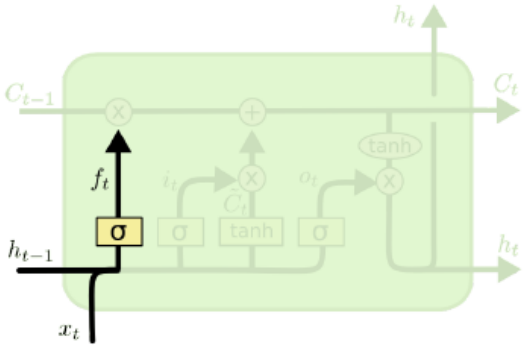


Figura 18. *Forget gate* [24]

La salida f_t de la puerta se puede obtener mediante la expresión (10), donde, como se ha comentado en expresiones anteriores W_t es la matriz de pesos, h_{t-1} es el vector de estado oculto en el instante de tiempo anterior, x_t es la entrada en el instante de tiempo *t* y σ la función de activación sigmoide.

$$f_t = \sigma(W_t.[h_{t-1}, x_t] + b_j) \tag{10}$$

El segundo paso es decidir qué nueva información se va a guardar en la célula de estado. Una sigmoide (puerta de entrada) decide cuántos valores se deben actualizar y una tangente hiperbólica (*tanh*) crea un vector de nuevos candidatos posibles $\tilde{C}_t$ pueden ser añadidos al estado. La expresión (11) corresponde a la activación i_t de la puerta de entrada expresión (12) corresponde a la obtención de los nuevos candidatos posibles $\tilde{C}_t$.

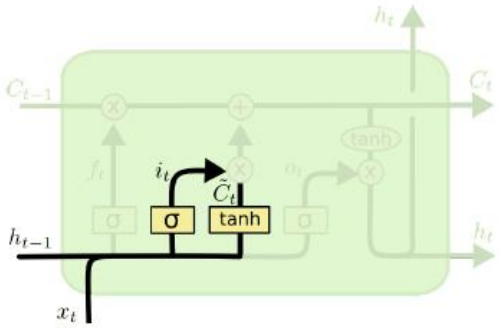


Figura 19. Puerta de entrada o *input layer* [24]

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i) \tag{11}$$

$$\tilde{C}_t = tanh(W_C.[h_{t-1}, x_t] + b_C) \tag{12}$$

El tercer paso consiste en la actualización de la célula de estado anterior C_{t-1} a la nueva C_t. Multiplicando el estado anterior que contiene la información seleccionada por las sigmoides por la expresión 10 (f_t) y sumando posteriormente el término $i_t * \tilde{C}_t$ se obtienen los nuevos candidatos.

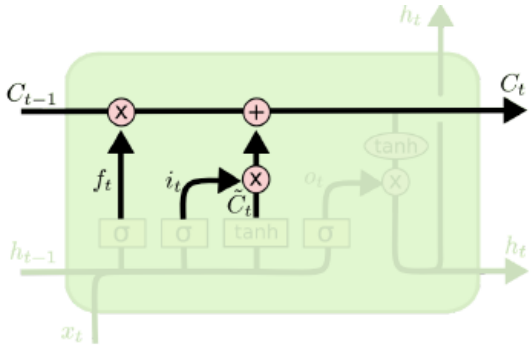


Figura 20. Elección de nuevos cancidatos [24]

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{13}$$

El cuarto paso consiste en decidir qué información se extrae a la salida de la célula de memoria. La salida o_t se obtendrá, primero, aplicando una sigmoide que decide qué partes de la célula de estado se extraerán a la salida y después, con una tangente hiperbólica la célula de estado se comprime entre -1 y 1 y se multiplica por la salida de la sigmoide anterior.

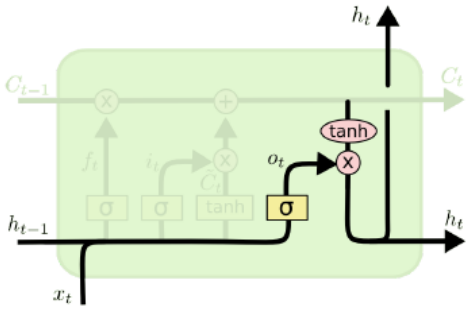


Figura 21. Salida de la célula de memoria LSTM [24]

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o) \tag{14}$$

$$h_t = f_t * \tanh(C_t) \tag{15}$$

2.6. Estado del Arte

A día de hoy no existe mucha documentación sobre el análisis musical mediante Inteligencia Artificial. Los estudios más relacionados en el ámbito de la Inteligencia Artificial y la música son, en gran parte, estudios que tienen como objetivo la composición musical pero muy pocos hablan de la importancia que supondría enseñar a una red neuronal a analizar música cuando es gracias al análisis y al referenciarse en otras obras como se aprende a componer. Los estudios que se centran en el análisis musical lo hacen mediante *clustering* y algoritmos no supervisados, pero son muy pocos los que utilizan redes neuronales. En este trabajo se realiza el estudio de análisis musial con redes neuronales con la finalidad de crear un modelo mucho más general y aplicable a más géneros que los modelos realizados mediante algoritmos no supervisados [25] [26].

Los estudios en el ámbito de la composición musical comienzan desde la composición automática y se extienden al uso de redes neuronales complejas como utiliza el sistema *DeepBach* [27], que combina técnicas como Cadenas de Markov con redes neuronales *Long-Short-Term-Memory* (LSTM) y que es, por el momento, el que mejor resultados ha dado en composición musical mediante redes neuronales. Este sistema es capaz de crear corales al estilo de J.S. Bach prediciendo la nota sucesiva en función de las notas cercanas.

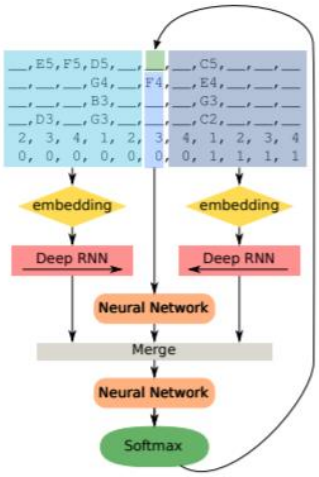


Figura 22. Estructura del modelo DeepBach [27]

Estos algoritmos mencionados anteriormente trabajan con archivos de audio normalmente en forma de archivo MIDI, pero existen estudios que trabajan directamente con la señal de audio en .wav o .mp3, la procesan y extraen información que se utiliza para alimentar la red neuronal. Este es, por ejemplo, el caso del estudio que más se acerca a este trabajo, solo que en este caso se detectan cambios en la estructura de la obra musical pero no se clasifican las partes de la misma. Thomas Grill y Jan Schlüter [28] plantean el uso de matrices de similaridad o *Self Similarity Matrixes* (SMM) y *Lag Matrixe*s extraídas a partir de la señal de audio como entrada a una estructura de redes neuronales convolucionales, que se alimentan también con el espectrograma logarítmico de Mel (MLS). Estas matrices bidimensionales que se pueden observar en la figura 23 serán los datos de entrada que utilizarán los modelos de redes neuronales empleados en este trabajo. Este es el punto de partida de este trabajo que se detallará más adelante en los capítulos III y IV.

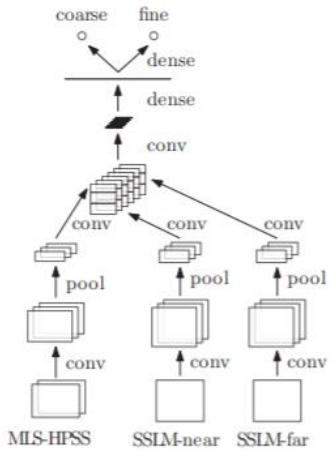


Figura 23. Estructura del modelo de detección de fronteras de T. Grill y J. Schlüter [28]

Lo que se propone en este trabajo es un modelo que tome como entrada solo una *Lag Matrix* por canción, y no dos matrices y un espectograma como en el modelo propuesto por Thomas Grill y Jan Schlüter [28], es decir, se propone una simplificación del modelo del estado del arte para obtener, o bien, una clasificación de las partes de la estructura musical de una pieza, o bien, una identificación de las transiciones entre partes de una obra.

Capítulo III

Análisis de la Estructura Formal de Piezas Musicales

En este capítulo se describe el primer caso de estudio realizado en este trabajo correspondiente a la identificación y etiquetado de las distintas partes de la estructura formal de piezas musicales.

3.1. Proceso de Desarrollo

El objetivo de este capítulo es la identificación y etiquetado de las distintas partes de la estructura formal de piezas musicales. Actualmente, no existen modelos de redes neuronales que resuelvan este problema de etiquetado de la estructura de piezas musicales, todos los métodos utilizan algoritmos no supervisados y teoría estadística para resolverlo [25] [26].

Antes de comenzar a entrar en detalle en cada paso seguido en la elaboración y validación del dataset y del modelo desarrollados en este trabajo es necesario tener una visión general del proceso seguido para su elaboración.

Primero, se obtiene un dataset que se divide en entrenamiento y validación, y que contiene canciones en formato .mp3 y etiquetas e formato .txt organizadas en columnas de tiempo en segundos y nombre de la etiqueta como se mostrará más adelante en este capítulo. Después, se calculan las matrices que se introducen en forma de imágenes como entrada a la red y se convierten las etiquetas a vectores en *frames*. A continuación, se desarrolla el modelo de la red neuronal, se entrena y se optimizan los parámetros del modelo. Por último se valida el modelo.

En este capítulo se explican todos y cada uno de los pasos, desde la obtención del dataset hasta el entrenamiento para este caso concreto de estudio, la optimización de parámetros del modelo y se exponen los resultados obtenidos, y en el capítulo V se compararán los resultados con el estado del arte.

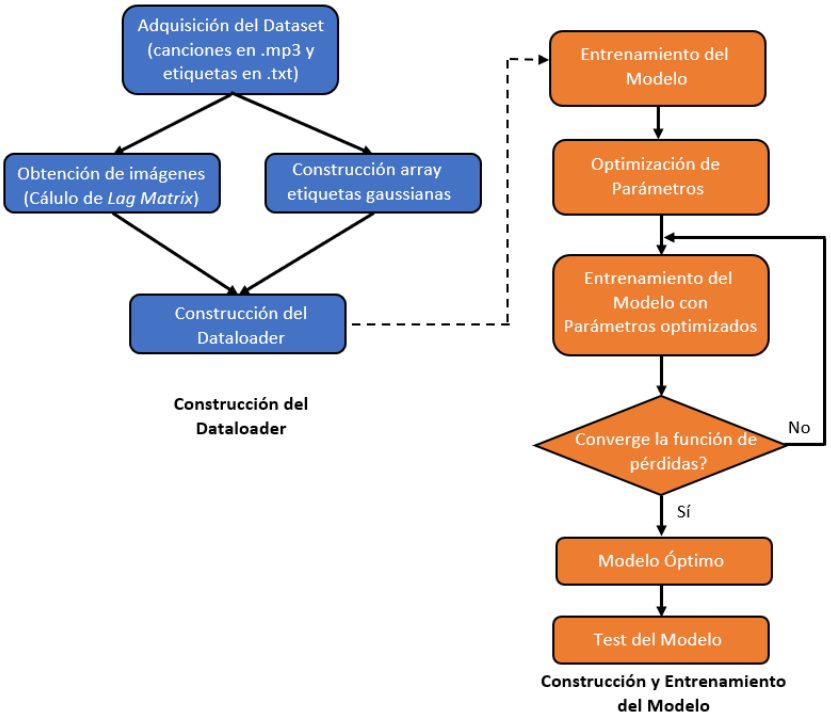


Figura 24. Esquema del proceso seguido en este trabajo

Para llevar a cabo el proceso descrito anteriormente se han programado los siguientes *scripts* de creación propia que son:

- `extract_labels_from_txt.py`: extracción de las etiquetas y tiempos del .txt.
- `create_histogram.py`: creación de los histogramas de etiquetas a partir del .txt.
- `songs_to_SSM.m`: conversión de las canciones .mp3 a *Self Similarity Matrix*.
- `data.py`: creación del *Dataloader* en Pytorch que carga las etiquetas y las imágenes.
- `model_CNN_LSTM.py`: modelo de la red neuronal CNN con LSTM.
- `train.py`: entrenamiento y validación del modelo.
- `song_results.py`: evaluación de los resultados del modelo en una canción y gráfico de la misma.

3.2. Conceptos Teóricos. *Self Similarity Matrix* (SSM)

El primer paso a realizar tanto en este caso de etiquetado de estructura como en el caso estudiado en el capítulo IV consiste en el procesamiento de audio, es decir, convertir la señal de audio en formato .mp3 a un formato de entrada que la red pueda comprender. Hay diversas formas de procesar el audio para introducirlo como entrada a la red. Para este caso de estudio, se va a realizar mediante la construcción de la Matriz de Similaridad o *Self Similarity Matrix* (SSM).

La matriz de similaridad o Self Similarity Matrix (SSM) es una forma de representación gráfica de secuencias similares en series de datos. Jonathan Foote en 1999 [29] calculó estas matrices a partir del espectrograma de Mel para ilustrar la estructura del audio.

Para calcular la medida de similaridad entre dos instantes de audio se parametrizan en Coeficientes Cepstrales en las Frecuencias de Mel (MFCC) a modo de vectores de correlación. En la figura 25 se puede observar el proceso de obtención de los MFCCs.

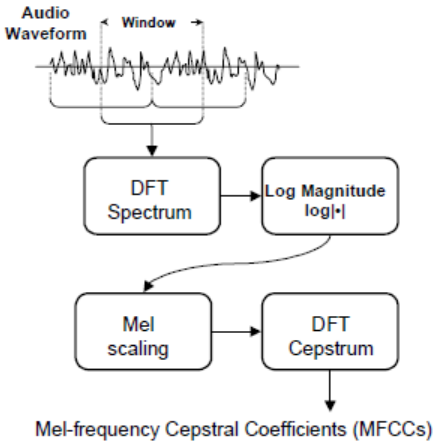


Figura 25. Proceso de extracción de los MFCCs [29]

Una vez obtenidos los MFCCs, la medida de similaridad [29] se calcula a partir de un vector de correlación que recoge los datos relevantes de un conjunto de datos de un pequeño intervalo. Dados dos vectores de características o *feature vectors* se define la función de similaridad a partir de la expresión:

$$S(i,j) = s(v_i, v_j) \qquad i,j \in (1,\dots,\mathrm{n}) \tag{16}$$

donde $s(v_i, v_j)$ es una función que mide la similaridad entre dos vectores. Esta función puede ser la distancia Euclideana, el producto escalar, etc. Foote [29] toma como función el producto escalar de dos vectores de correlación de ventanas de audio *i* y *j*. Para calcular la dependencia temporal entre los vectores Foote propone la siguiente expresión donde S_w es el vector de correlación para la ventana w:

$$S_w(i,j) = \frac{1}{w}\sum_{k=0}^{w-1}(v_{i+k} \bullet v_{j+k}) = \frac{1}{w}\sum_{k=0}^{w-1}\left(\|v_{i+k}\|\,\|v_{j+k}\|\cos\theta\right) \tag{17}$$

Para calcular las matrices de las canciones se toman muestras de una determinada duración de audio con una superposición específica. La resolución de la imagen serán las ventanas de la canción que se calculan dividiendo los milisegundos que dura la canción por los milisegundos que dura una ventana. Un ejemplo de SSM se muestra en la figura 26.

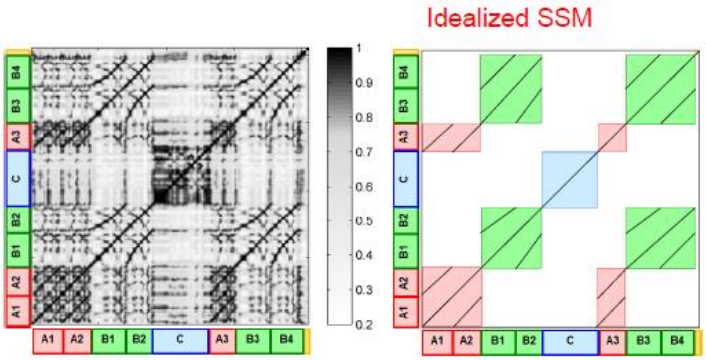


Figura 26. Matriz de Similaridad o Self Similarity Matrix de la Danza Húngara No. 5 de Brahms [30]

3.3. Base de Datos

SALAMI [31] es una base de datos con más de 1.000 canciones de diversos géneros la mayoría anotadas por 2 autores siguiendo 3 formas diversas de anotación: *lower case*, *upper case* y *functions*. La columna de la derecha corresponde al tiempo de inicio de cada parte de la canción en segundos y la de la izquierda el nombre dicha parte. Un ejemplo se puede ver en la figura 27.

(a)

Time	Label
0.0	Silence
0.1992517	A
13.501904761	B
38.61430839	C
63.621836734	B
88.55430839	C
113.373015873	B
137.900226757	C
168.23941043	Silence
170.596916099	End

(b)

Time	Label
0.0	Silence
0.1992517	b
13.501904761	c
26.061882086	c
38.61430839	d
51.192607709	d
63.621836734	e
76.086145124	e
88.55430839	f
100.948299319	f
113.373015873	g
125.552494331	g
137.900226757	h
150.152267573	h
168.23941043	Silence
170.596916099	End

(c)

Time	Label
0.0	Silence
0.1992517	Intro
13.501904761	Verse
38.61430839	Chorus
63.621836734	Verse
88.55430839	Chorus
113.373015873	Solo
137.900226757	Chorus
168.23941043	Silence
170.596916099	End

Figura 27. Ejemplo de ls fichero de etiquetas "anotaciones" de SALAMI de la canción 6 de SALAMI (a) *lower case* (no se muestra entero debido a su tamaño) (b) *upper case* (c) *functions*

En este caso se han escogido 437 canciones de dicha base de datos que corresponden a obras entre las canciones 955 y la 1498 SALAMI.

3.3.1. Etiquetas

Para las etiquetas se han escogido las anotaciones correspondientes al anotador 2 y para aquellas canciones sin anotar por el anotador 2, se han cogido las etiquetas correspondientes a las del anotador 1.

En la figura 28 se muestran los histogramas de etiquetas correspondientes a los tres tipos de etiquetado de la base de datos SALAMI.

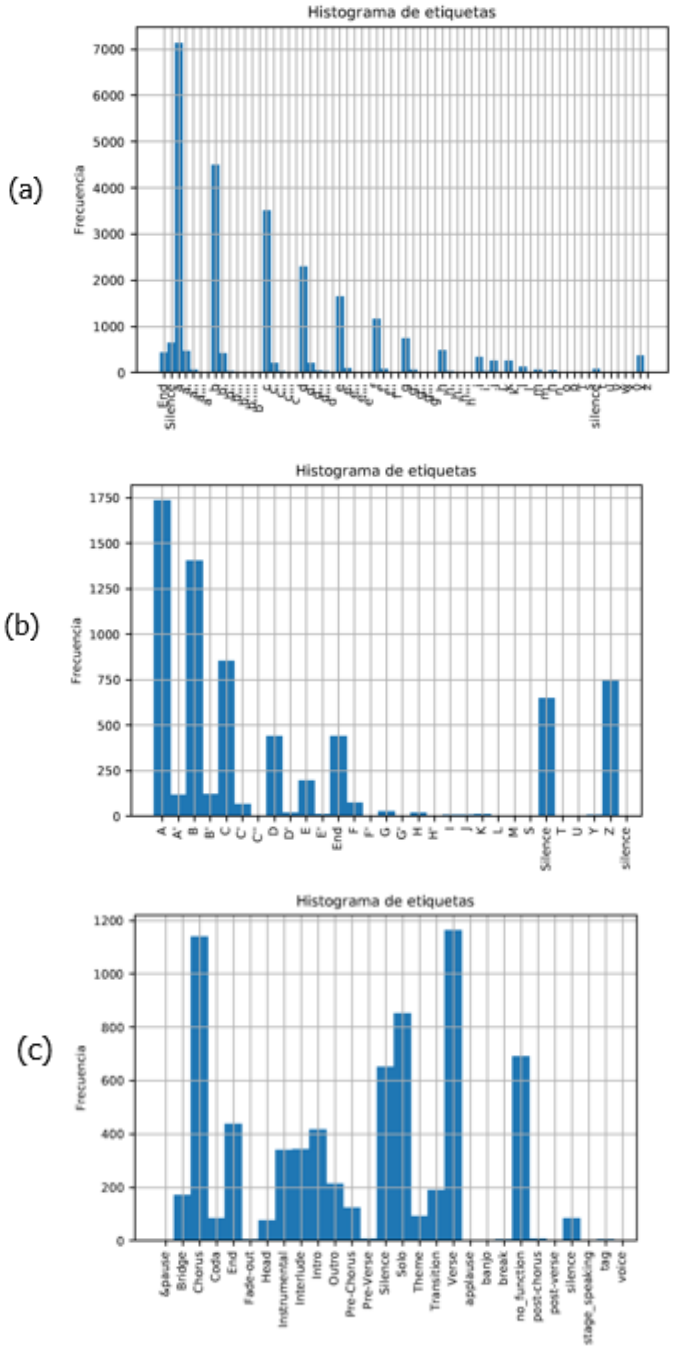


Figura 28. Histogramas de las etiquetas para la canciones 955 a 1498 de SALAMI de los ficheros de etiquetas (a) *lower case* (b) *upper case* (c) *functions*

Para la elección del tipo fichero de etiquetas a utilizar se ha escogido el último histograma mostrado anteriormente (figura 28c correspondiente al fichero *functions*) ya que es el más homogéneo y representa claramente las partes de la estructura de las piezas de la base de datos. Como hay demasiadas clases y algunas por su temática similar se pueden agrupar con otras, se ha realizado un *script* de creación propia en Python que agrupa las etiquetas de la siguiente forma y las iguala a los siguientes valores:

1 = no_function: pause, applause, banjo, break, silence, stage_speaking, voice, Silence, no_function
2 = Intro: Head, Intro
3 = Verse: Pre-Verse, post-verse, Theme, Verse
4 = Chorus: Pre-Chorus, post-chorus, Chorus
5 = Fade-out: Coda, End, Fade-out
6 = Transition: Interlude, Bridge
7 = Solo
8 = Instrumental
9 = Outro

Una vez están agrupadas las etiquetas, mediante otro script de creación propia en Python se han leído los tiempos en segundos de los ficheros functions.txt y se han transformado a *frames*, dando a cada *frame* el valor de su etiqueta correspondiente siguiendo la numeración mostrada anteriormente. El nuevo etiquetado es un vector que se ha guardado en un archivo .cvs que será leído por el *Dataloader* de Pytorch posteriormente. El vector de etiquetas es de la forma:

$$labels = [label_{frame\ 1}, \dots, label_{frame\ n}]^T$$

3.3.2. Imágenes

En este caso de estudio, las imágenes de entrada a la red son *Self Similarity Matrixes* SSM como se ha descrito anteriormente en este capítulo y se han obtenido con la MATLAB_SM-Toolbox_1.0 [32]. El tamaño de ventana o *window length* escogido es de 200ms y el salto o *hop size* de 100ms (50% de *overlap*). A las imágenes se les ha aplicado un *smoothing* para evitar dar a la red demasiada información con la que no pueda generalizar.

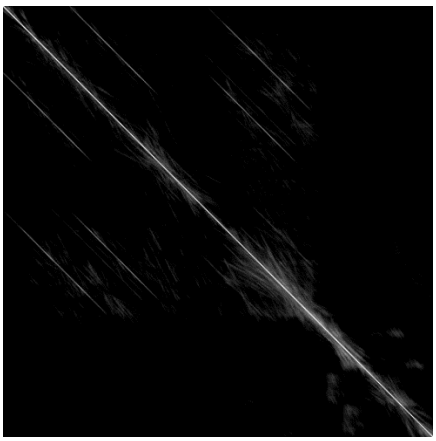


Figura 29. Self Similarity Matrix para la canción 955 de SALAMI

3.4. Implementación del Modelo

3.4.1. Estructura de la Red

Una vez presentados la entrada de la red y de la base de datos utilizada se procede a la implementación del modelo de red. La principal dificultad en este caso de estudio es que se trata de un problema **multiclase**, pero además hay una dificultad añadida ya que cada canción no tiene por qué tener el mismo número ni tipo de clases, por tanto, se trata de un problema de clasificación ***multiclass label***.

En este tipo de problema multiclase a diferencia del caso que se estudiará en el capítulo IV, se utilizará como función de activación a la salida de la red una función *softmax* (capítulo II, figura 9c) y no sigmoide como para cuando se tiene una sola clase, y una función de pérdida o coste (*loss function*) *Negative Log Likelihood Loss* (NLLL) que se utiliza en este tipo de problemas multiclase.

En este trabajo se ha realizado el modelo que se muestra en figura 30 que consiste en una red CNN a la que se le ha añadido una LSTM después, siendo la CNN de dos capas a la que le sigue una capa LSTM. Las funciones aplicadas para estructurar el modelo en Pytorch que se encuentran en el módulo nn de Torch son las siguientes [33]:

- **Convolución**: Conv2d (input_channels, output_channels[2], kernel_size, stride, padding).
- ***Pooling***: MaxPool2d (num_kernel_size, stride, padding).
- **LeakyReLU**: leaky_relu (ouput_previous_conv[3]).
- **Dropout**: Dropout2d (percentage).
- **Batch Normalization**: BatchNorm1d
- **LSTM**: LSTM (sequence_len, batch, input_size)

[2] número de *kernels* a aplicar en la entrada = número de mapas de características deseado a la salida.

[3] vector de dimensiones [N, C, H, W]

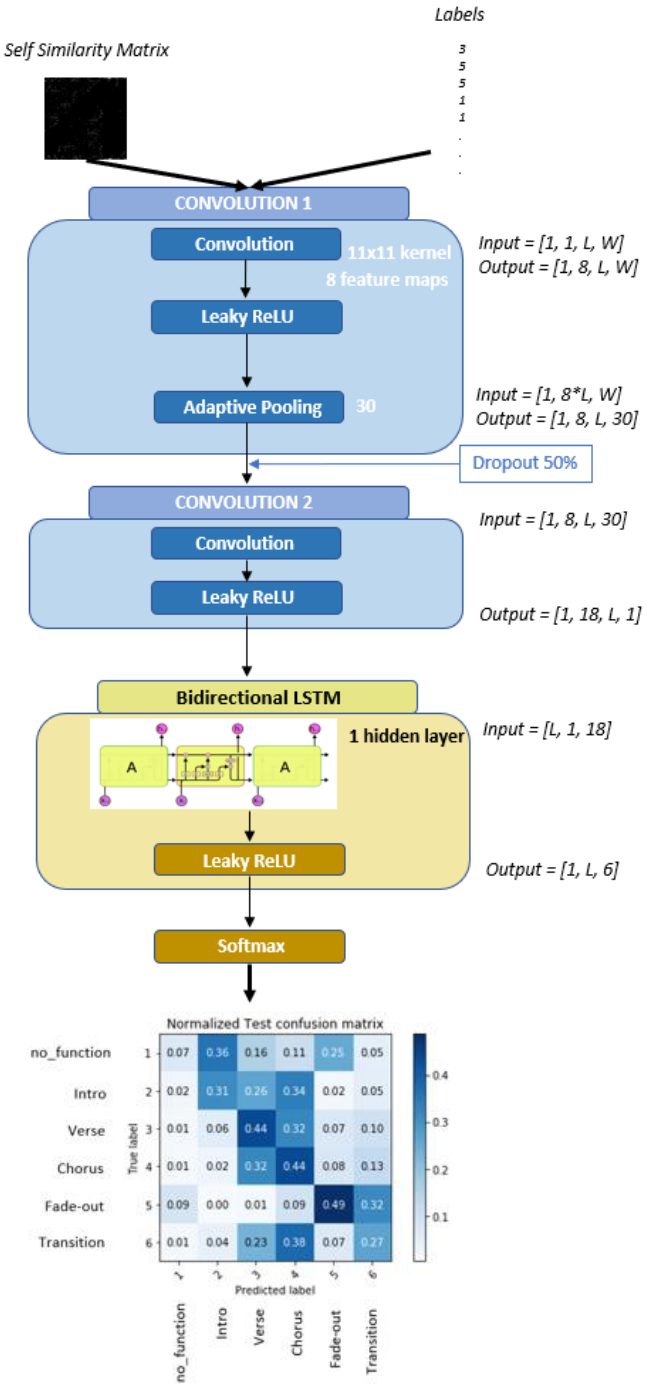


Figura 30. Modelo CNN con LSTM implementado (6 clases)

3.4.2. Entrenamiento del Modelo. Optimización de Parámetros

En este caso, al tener pocas obras (437 canciones en total) se ha dividido en los datasets 350 canciones para entrenamiento (80%) y (20%) 87 canciones para validación, y el test se ha realizado sobre las obras de validación.

El entrenamiento con el dataset de entrenamiento ajustará los pesos de la red y en validación se podrán optimizar los parámetros del modelo como el *learning rate*, el optimizador, la función de pérdida o el número de épocas a entrenar. Dado que la

resolución de las imágenes es distinta, el *batch size* será igual a 1, es decir, se suministrará cada imagen a la red por separado y no por lotes. El orden de optimización y elección de los parámetros se expone a continuación.

El ***learning rate*** se ha estimado realizando entrenamientos de un pequeño número de épocas y viendo cómo evoluciona el error de entrenamiento y validación. El valor final se ha fijado en 0.001.

Como **optimizador de la función de coste** se ha escogido Adam, que es un optimizador que combina dos extensiones del algoritmo de gradiente estocástico descendente: el AdaGrad y el RMSProp.

El **número de épocas** se ha determinado lanzando un entrenamiento más largo y viendo cuándo convergen las curvas de pérdida de test y entrenamiento. En este caso particular entrenando el modelo con 9 clases como se puede observar en la figura 31 hay un claro sobreentrenamiento desde el principio del entrenamiento, por lo que habrá que o, reajustar el modelo, o reagrupar las clases de una forma distinta. En la figura 31 la curva naranja corresponde al dataset de entrenamiento y la azul al de validación.

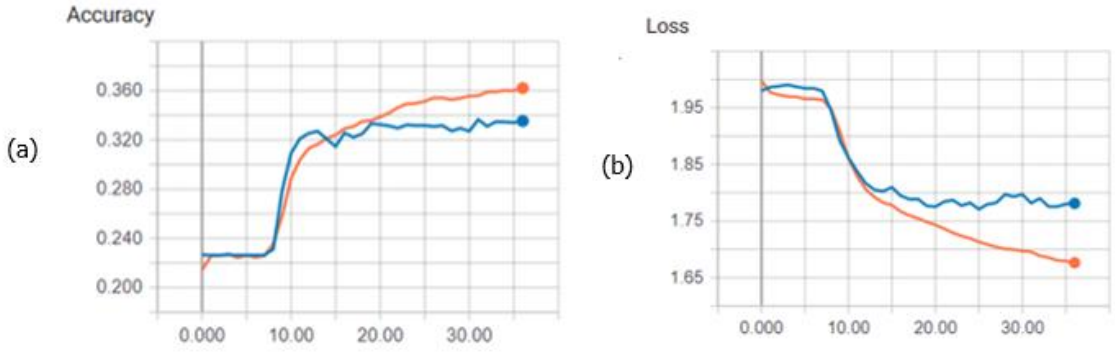


Figura 31. (a) *Accuracy* (b) *Loss* para el modelo entrenado con 9 clases

Como **función de pérdida** o *loss function*, como se ha comentado anteriormente se ha escogido la *Negative Log Likelihood Loss* (NLLL), que es la función utilizada en este tipo de problemas multiclase.

3.5. Análisis de los Resultados

Los resultados para la aplicación del modelo con las 9 clases se muestran en la figura 32.

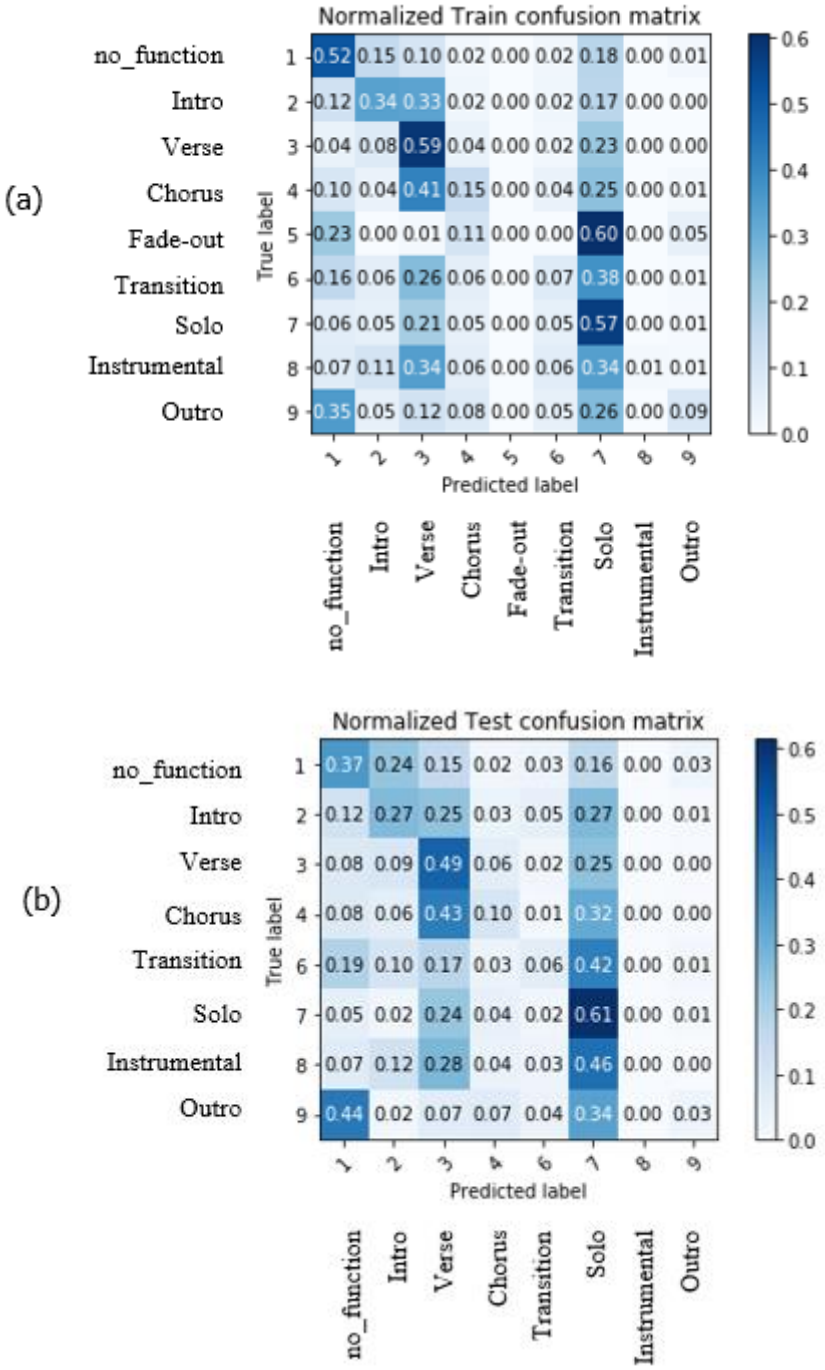


Figura 32. Matrices de confusión para (a) entrenamiento (b) test para el modelo entrenado. La curva naranja corresponde al test y la azul al entrenamiento. En la matriz de confusión de test la etiqueta 5 (Fade-out) no se muestra porque todas las probabilidades son igual a cero

A la vista de los resultados que se obtienen en las matrices de confusión se pueden sacar como conclusiones que los errores más grandes que el modelo comete son la confusión de las etiquetas:

- no_function con Intro
- Intro con Verse y Solo
- Verse con Solo
- Chorus con Verse y Solo
- Transition con Solo
- Instrumental con Solo
- Outro con no_function

➢ Fade-out, Chorus, Transition, Instrumental y Outro son etiquetas que no identifica con ellas mismas ya que todas sus probabilidades son prácticamente igual a cero.

Como se puede observar en la figura 31 el modelo sobreentrena, pero además, como se ha comentado anteriormente analizando los resultados de la figura 32 se puede concluir que el modelo con 9 clases no es capaz de dar una buena precisión, y, dado que hay varias etiquetas que no predice, se ha decidido hacer una reagrupación de las clases que son más similares entre sí de la siguiente forma y asignando los valores que se muestran a continuación a cada etiqueta:

1 = no_function: pause, applause, banjo, break, silence, stage_speaking, voice, Silence, no_function
2 = Intro: Head, Intro
3 = Verse: Pre-Verse, post-verse, Theme, Verse
4 = Chorus: Pre-Chorus, post-chorus, Chorus
5 = Fade-out: Coda, Outro, End, Fade-out
6 = Transition: Interlude, Bridge, Instrumental, Solo, Transition

Una vez reagrupadas las etiquetas, se realiza el histograma de todos los ficheros que se puede ver en la figura 33.

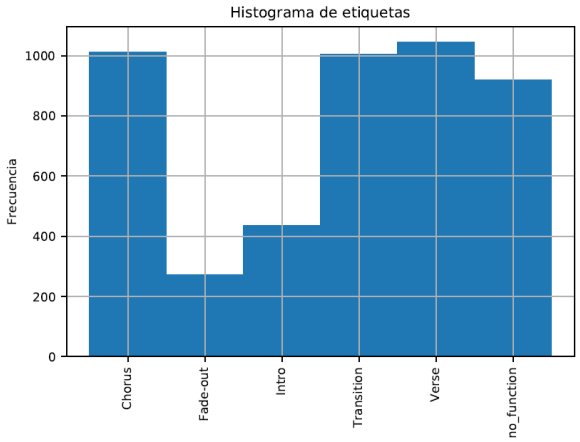


Figura 33. Histogramas de las etiquetas agrupadas para la canciones 955 a 1498 de SALAMI de los ficheros de etiquetas *functions*

El mismo modelo (figura 30) se ha entrenado con las etiquetas reagrupadas y los resultados de entrenamiento del modelo con 6 etiquetas se muestran en las figuras 34 y 35.

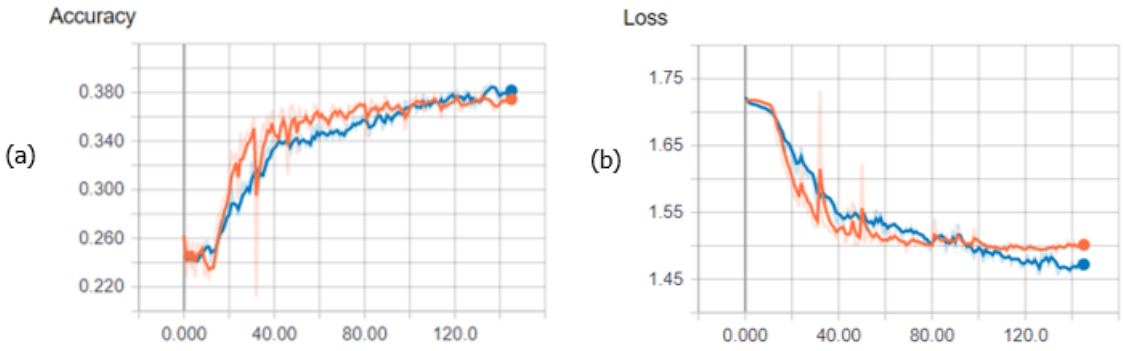


Figura 34. Curvas (a) *accuracy* (b) *loss* para el modelo entrenado. La curva naranja corresponde al test y la azul al entrenamiento

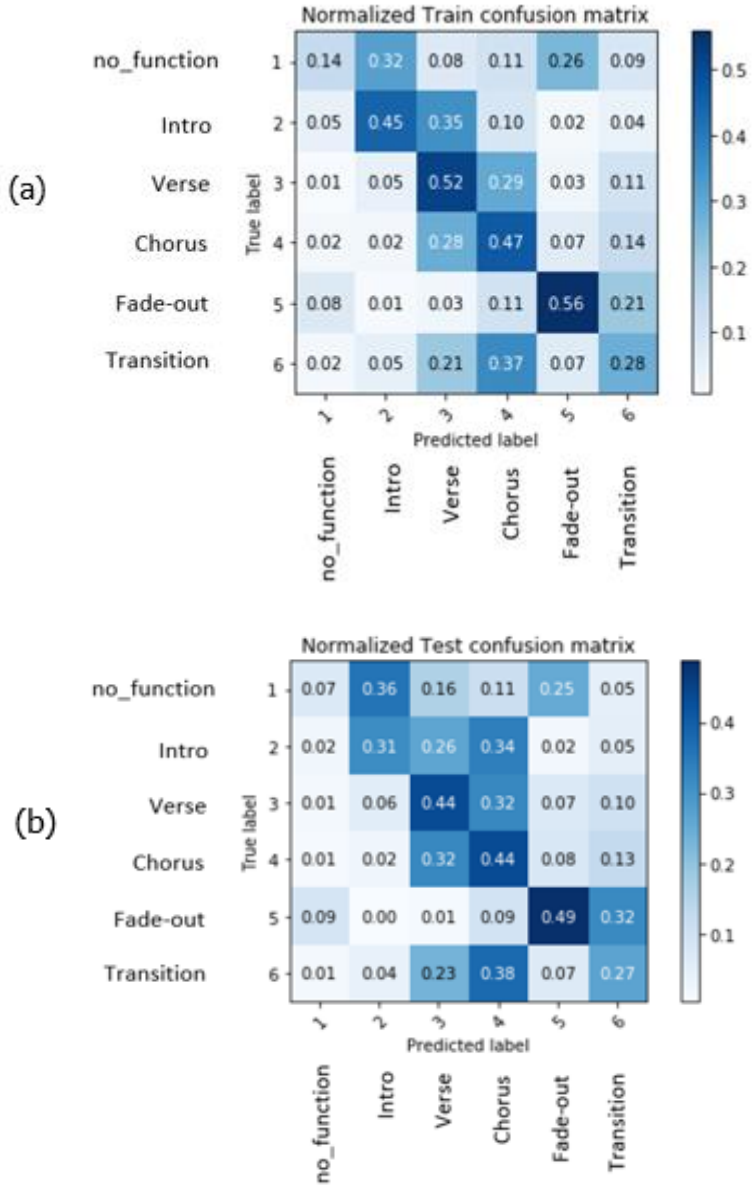


Figura 35. Matrices de confusión para (a) entrenamiento (b) test para el modelo entrenado. La curva naranja corresponde al test y la azul al entrenamiento

Los resultados que se obtienen en las matrices de confusión entrenando el modelo con 5 clases son mejores que los obtenidos con 9 clases, per aún así se siguen cometiendo errores significativos a la hora de estimar las etiquetas. Los errores más grandes que el modelo comete son la confusión de las etiquetas:

- no_function con Intro y Fade-out
- Intro con Verse y Chorus
- Verse con Chorus
- Chorus con Verse
- Fade-out con Transition
- Transitions con Chorus y Verse

La confusión de estribillos (etiqueta “Chorus”) con estrofas (etiqueta “Verse”) es un error grave, ya que en el estilo *pop* los estribillos y estrofas son las 2 partes principles de una canción, más allá de la intro o de las transiciones, pero como no se ha introducido a la red infomación armónica puede ser normal dicha confusión ya que lo que marca al estribillo es su repetición durante la canción, y hay infinidad de canciones en las cuales las estrofas a nivel de estructura se repiten en la canción y de ahí su posible confusión con estribillos, que además, es un fallo que también comenten los humanos al analizar música.

Capítulo IV

Detección de Transiciones

En este capítulo se describe el segundo caso de estudio realizado en este trabajo correspondiente a la detección de transiciones en la estructura formal de piezas musicales.

4.1. Proceso de Desarrollo

El objetivo de este capítulo es la detección de las transiciones entre las distintas partes de la estructura formal de piezas musicales. Actualmente, existen modelos tanto de algoritmos no supervisados como de redes neuronales [28] que resuelvan este problema de etiquetado de la estructura de piezas musicales aunque con una precisión no muy alta.

El proceso seguido en este capítulo es el mismo que el seguido en el capítulo III (figura 24). Para llevar a cabo el proceso descrito anteriormente se han programado los siguientes *scripts* de creación propia que son:

- `extract_labels_from_txt.py` (descritos en el capítulo III).
- `create_histogram.py` (descritos en el capítulo III).
- `songs_to_lagmatrix.py`: conversión de las canciones .mp3 a *Lag Matrix*.
- `data.py`: creación del *Dataloader* en Pytorch que contiene la función de conversión de las etiquetas a vectores de gaussianas.
- `model_CNN.py`: modelo de la red neuronal CNN.
- `model_CNN_LSTM.py`: modelo de la red neuronal CNN con LSTM.
- `train.py`: entrenamiento y validación del modelo.
- `mir_evaluation.py`: obtención del parámetro δ a partir del algoritmo de evaluación del MIR.
- `test.py`: test del modelo en base a los criterios del MIR (ver capítulo IV).
- `song_results.py`: evaluación de los resultados del modelo en una canción y gráfico de la misma.

4.2. Conceptos Teóricos. Recurrence Plot y Lag Matrix.

Otro método de procesar un archivo de audio para obtener la estructura musical del mismo, que es el utilizado en este trabajo es mediante el cálculo del denominado como Matriz de Recurrencia o ***Recurrence Plot* ($R_{i,j}$)**. Un *Recurrence Plot*, introducido en 1987 por Eckmann [34], es una representación gráfica de la comparación de los estados de un sistema dinámico en los instantes i y j [35]. Si los estados son similares se indican con un uno en la matriz ($R_{i,j} = 1$), y si son distintos con un cero ($R_{i,j} = 0$), de tal manera que se representan en ambos ejes horizontal y vertical la duración total del audio.

A partir del *Recurrence Plot* se puede construir la ***Lag Matrix* ($L_{i,j}$)** [36], matrices simétricas que se forman calculando la similaridad entre vectores chroma o PCPs [37]. En una *Lag Matix* en el eje horizontal se representan las secciones de la estructura del audio a lo largo de la duración de la canción (una línea horizontal significa que una determinada sección se está repitiendo durante el tiempo que dura esa línea) y en el eje vertical la similaridad entre secciones en el tiempo (ver figura 28).

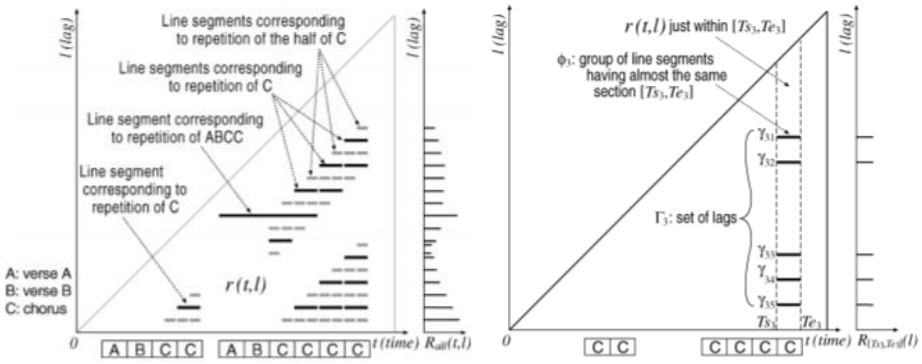


Figura 36. Estructura de una *Lag Matrix* [36]

En este trabajo, para construir los *Recurrence Plots* y *Lag Matrixes* que se introducirán como entrada a la red se ha utilizado el método propuesto por Joan Serrà, Meinard Müller, Peter Grosche, and Josep Ll. Arcos [38]. Primero se extraen los PCPs de la señal de audio mediante la librería de `librosa` [39] disponible para Python. Para ello se han escogido unos valores de frecuencia de muestreo o *sampling rate* de 44.100Hz, un tamaño de ventana de 186ms*4 (8192*4 *frames*) y un salto o *hop length* de 139ms*4 (6144*4 frames).

Para calcular la Matriz de Recurrencia $R_{i,j}$ se parte de vector chroma X con dimensiones 12xN', con N' el número de frames totales de la canción y 12 el número de notas totales que se hacen corresponder a una clase:

$$X = [x_1 \dots x_{N'}] \tag{18}$$

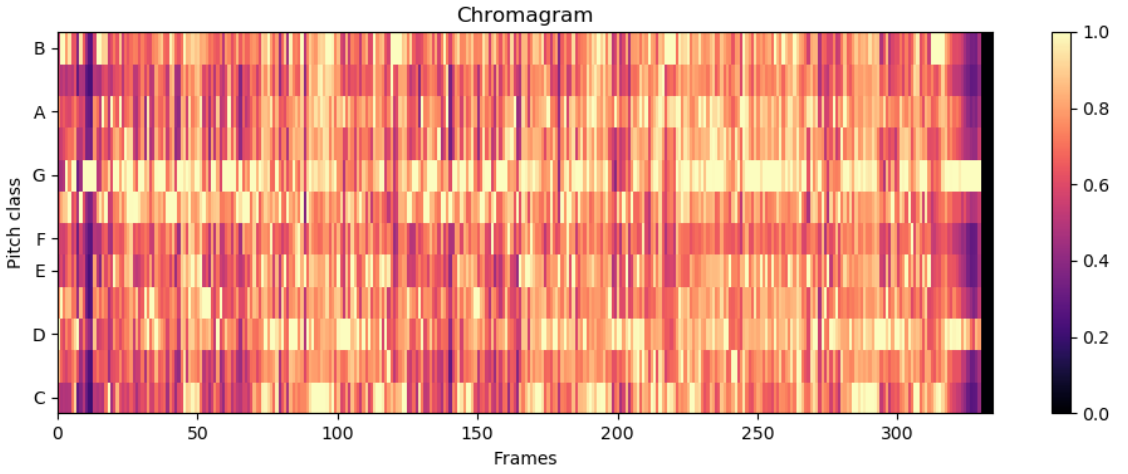


Figura 37. Vector chroma para la canción 984 del dataset SALAMI obtenida con `librosa`

Después, se construyen los vectores $\hat{x}_i$ que son vectores columna cada uno calculado a partir de una fila de la matriz chroma, y que tienen cada uno $j = i, i - \tau, \dots, i - (m-1) * \tau$ columnas. El parámetro m es el desplazamiento de cada fila de la matriz chroma respecto de la anterior y se fija a un valor de 5 segundos (9 *frames*), y τ corresponde a un delay de un segundo, por tanto, para 12 filas de una matriz chroma se tienen 12*m (12*9 = 108) vectores columna $\hat{x}_i$. Finalmente, para realizar la concatenación de estos vectores columna $\hat{x}_i$ se comienza a concatenar no por $i = 1$ sino por $i = w + 1$ dando como resultado el vector $\hat{X}$ que tiene N columnas $i = w + 1, \dots, N'$

que corresponden a los N' *frames* totales de la canción menos w, $N = N' - w$, siendo $w = (m-1) * \tau$.

$$\hat{x}_i = \left[{x_i}^T \; {x_{i-\tau}}^T \; \dots \; {x_{i-(m-1)\tau}}^T\right] \tag{19}$$

$$\hat{X} = [\hat{x}_1 \; \dots \; \hat{x}_N] \tag{20}$$

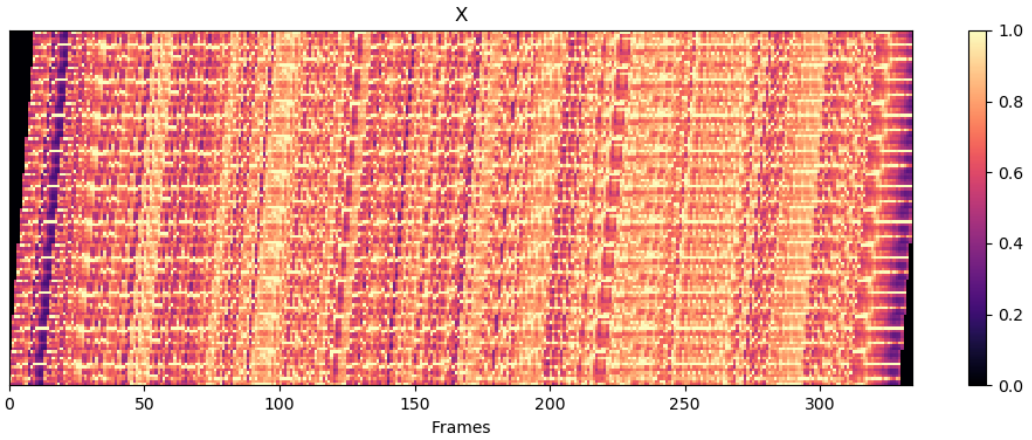


Figura 38. Vector $\hat{X}$ para la canción 6 del dataset SALAMI

Una vez obtenido el vector $\hat{X}$ se puede obtener la Matriz de Recurrencia aplicando el método de los k vecinos más cercanos o ***k-nearest neighbors (knn algorithm)*** [40], un algoritmo de clasificación supervisada donde se clasifican los datos dependiendo de la distancia euclidiana entre ellos, es decir, para un valor de $k = 13$ vecinos, se calcularán las distancias euclidianas entre todos los elementos del vectores $\hat{X}$ y se seleccionarán para cada valor aquellos 13 elementos del vector cuya distancia sea menor, de tal forma que la Matriz de Recurrencia $R_{i,j} = 1$ si $\hat{x}_i$ es vecino de $\hat{x}_j$ y $\hat{x}_j$ es vecino de $\hat{x}_i$, y $R_{i,j} = 0$ en caso contrario.

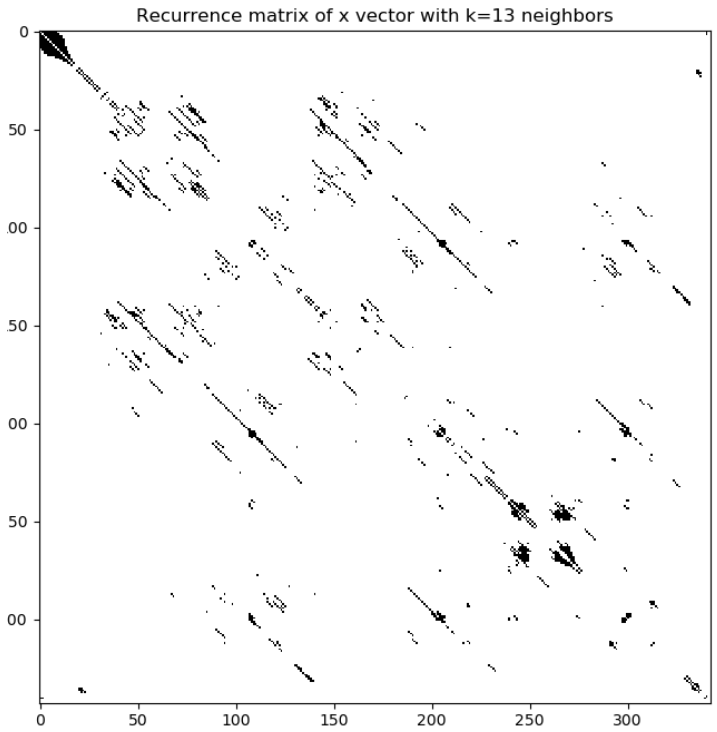


Figura 39. Matriz de Recurrencia $R_{i,j}$ para la canción 6 del dataset SALAMI

Después de obtener la Matriz de Recurrencia se calcula la *Lag Matrix* utilizando la función `recurrence_to_lag` de `librosa` que calcula $L_{i,j}$ mediante la expresión:

$$L_{i,j} = R_{k+1,j} \quad \text{con} \quad i,j = 1,\dots,N \quad \text{donde} \quad k = i + j - 2 \bmod N \tag{21}$$

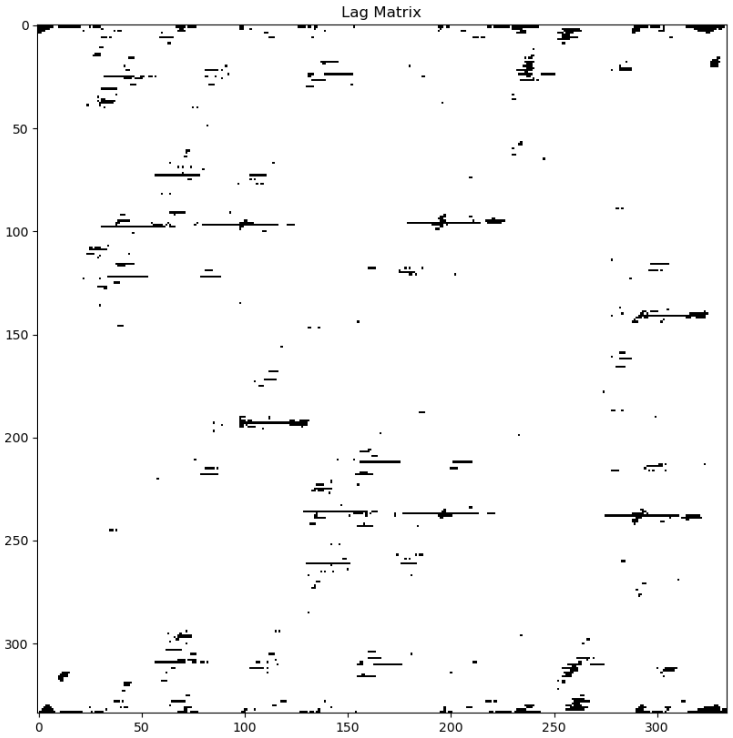


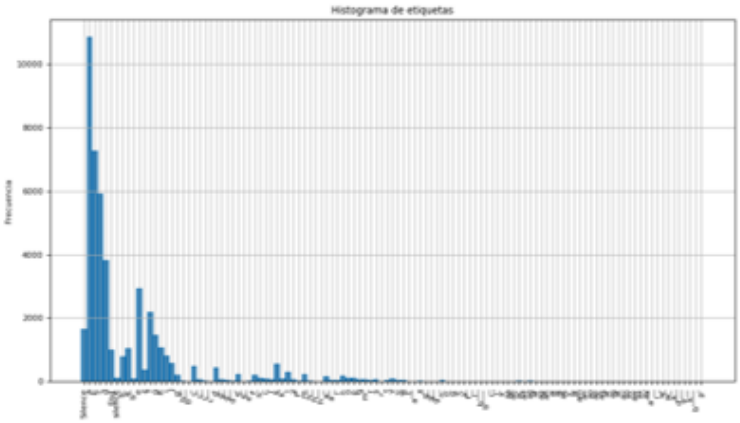
Figura 40. Matriz de Recurrencia $R_{i,j}$ para la canción 6 del dataset SALAMI

Esta *Lag Matrix* $L_{i,j}$ será calculada para cada canción del dataset y será la entrada de la red neuronal, que tendrá como objetivo la detección de transiciones en la estructura de las canciones mediante estas imágenes, las etiquetas correspondientes a las mismas y el entrenamiento de la red.

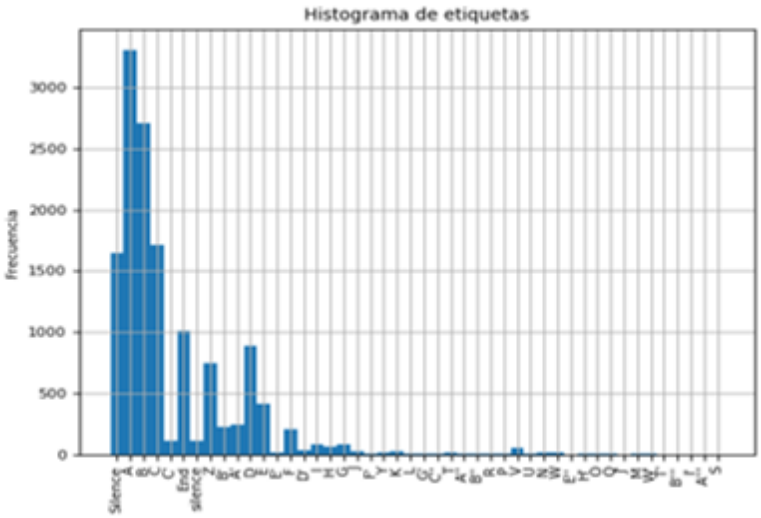
4.3. Base de Datos

La base de datos a utilizar es SALAMI, la misma que en el capítulo III. Para el caso del trabajo se ha incrementado la base de datos consiguiendo obtener 1006 canciones para las cuales el tipo de etiquetas utilizado se ha seleccionado escribiendo un script de creación propia en Python que lee todos los archivos y muestra el histograma de etiquetas totales de la base de datos. Como no todas las canciones están anotadas por los dos anotadores se han cogido las etiquetas correspondientes al anotador 2 y al anotador 1 para aquellas canciones que no estan anotadas por el anotador 2.

(a)



(b)



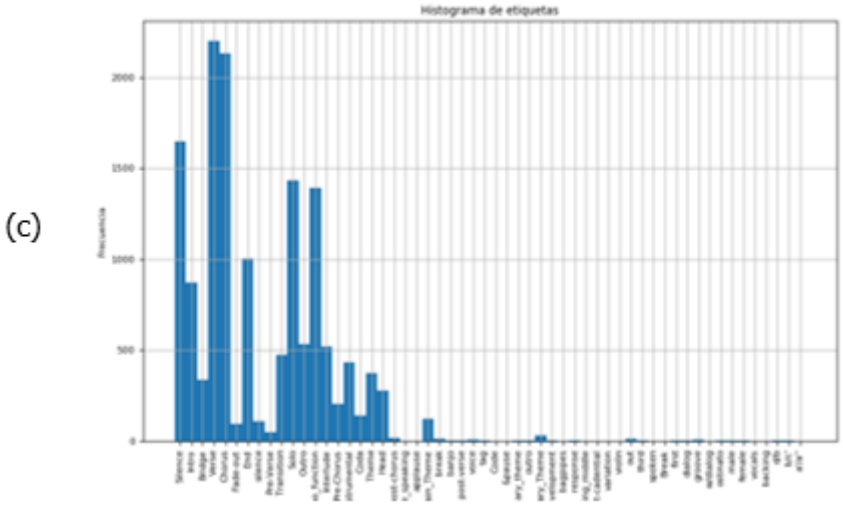


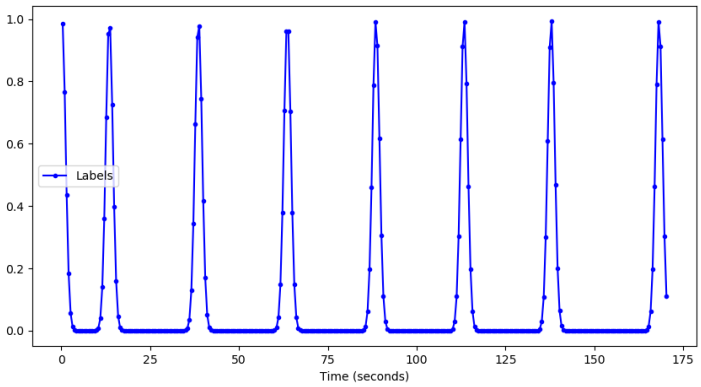
Figura 41. Histogramas de etiquetas relativos a los tres tipos de etiquetado según los ficheros (a) *lower case* (b) *upper case* (c) *functions*

Para la elección del tipo fichero de etiquetas a utilizar se ha escogido, como en el capítulo III, el último histograma mostrado anteriormente (figura 41c correspondiente al fichero *functions*) ya que es el más homogéneo y representa claramente las partes de la estructura de las piezas de la base de datos.

4.4. Desarrollo del Modelo RRNN

4.4.1. Preparación del Dataloader. Etiquetas

Una vez obtenida la base de datos tanto las imágenes como los ficheros de anotaciones es necesario convertir las etiquetas en un formato que la red sea capaz de entender. Para la creación de las etiquetas se ha escrito una función de creación propia en Python que lee los datos numéricos en segundos del fichero de etiquetas y los transforma en frames (unidad en la que están creadas las imágenes), poniendo campanas gaussianas en las transiciones para dejar un margen a la red para que se entere de la transición (ver figura 42). La primera etiqueta de cada archivo correspondiente con el inicio del archivo .mp3 en 0.0 segundos ha sido eliminada puesto que no aporta ningún tipo de información para la red, así como la última que indica el final del archivo .mp3.



Time	Label
0.0	Silence
0.1992517	Intro
13.501904761	Verse
38.61430839	Chorus
63.621836734	Verse
88.55430839	Chorus
113.373015873	Solo
137.900226757	Chorus
168.23941043	Silence
170.596916099	End

Figura 42. Etiquetas de la canción 6 de SALAMI (a) vector de gaussianas (b) fichero *functions* correspondiente

4.4.2. Estructura de la Red

En este trabajo se han realizado dos modelos de redes neuronales. El primero consiste en una CNN de 6 capas convolucionales donde la última capa actúa como si fuese una capa lineal. Se puede observar la estructura del modelo en la figura 43. El segundo modelo (figura 44) consiste en una red CNN a la que se le ha añadido una LSTM después, siendo la CNN igual que en el modelo anterior salvo la última convolución que se sustituye por una capa LSTM y después de la misma una capa lineal. Las funciones aplicadas para estructurar el modelo en Pytorch son las mismas que las descritas en el capítulo III y que se encuentran en el módulo nn de Torch [33].

- **Convolución**: Conv2d (input_channels, output_channels[4], kernel_size, stride, padding).
- ***Pooling***: MaxPool2d (num_kernel_size, stride, padding).
- **LeakyReLU**: leaky_relu (ouput_previous_conv[5]).
- **Dropout**: Dropout2d (percentage).
- **Batch Normalization**: BatchNorm1d
- **LSTM**: LSTM (sequence_len, batch, input_size)

[4] número de *kernels* a aplicar en la entrada = número de mapas de características deseado a la salida.

[5] vector de dimensiones [N, C, H, W]

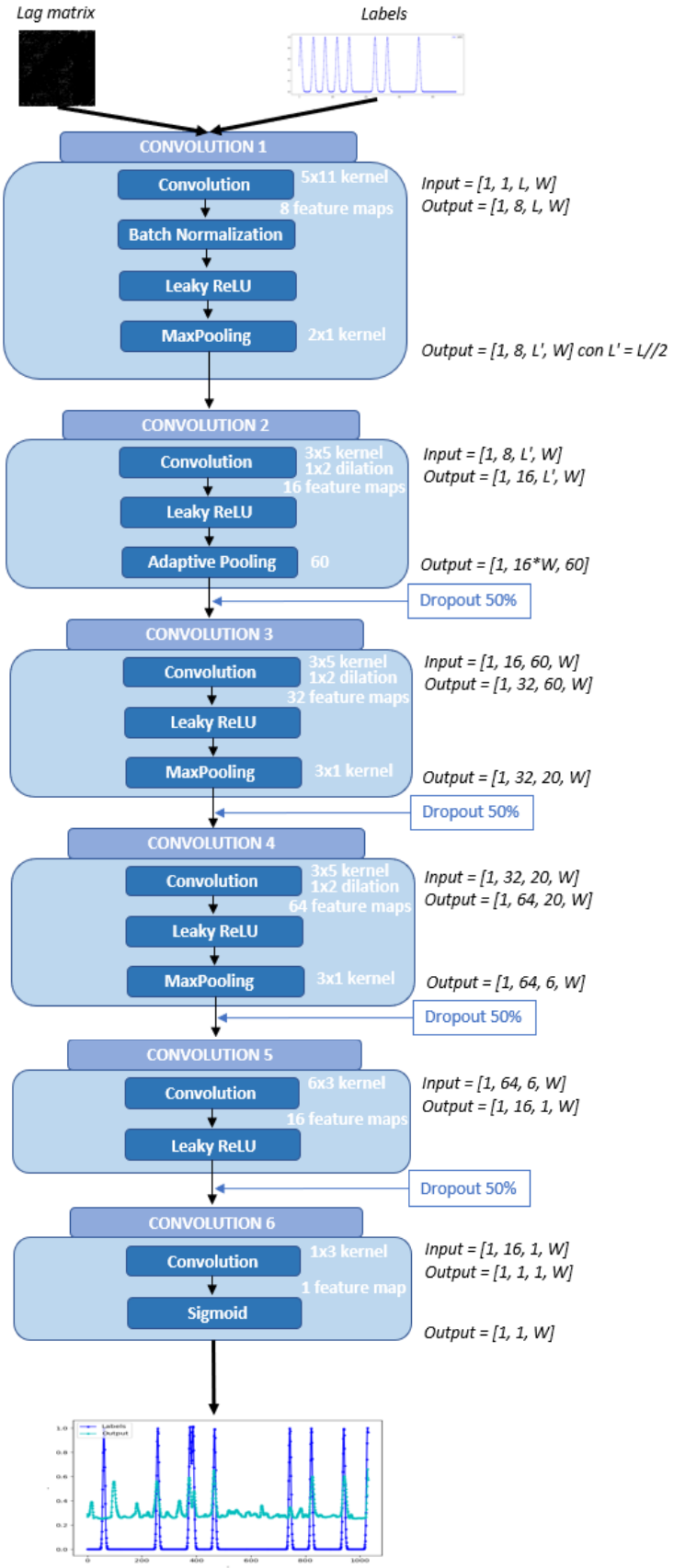


Figura 43. Estructura del modelo CNN desarrollado

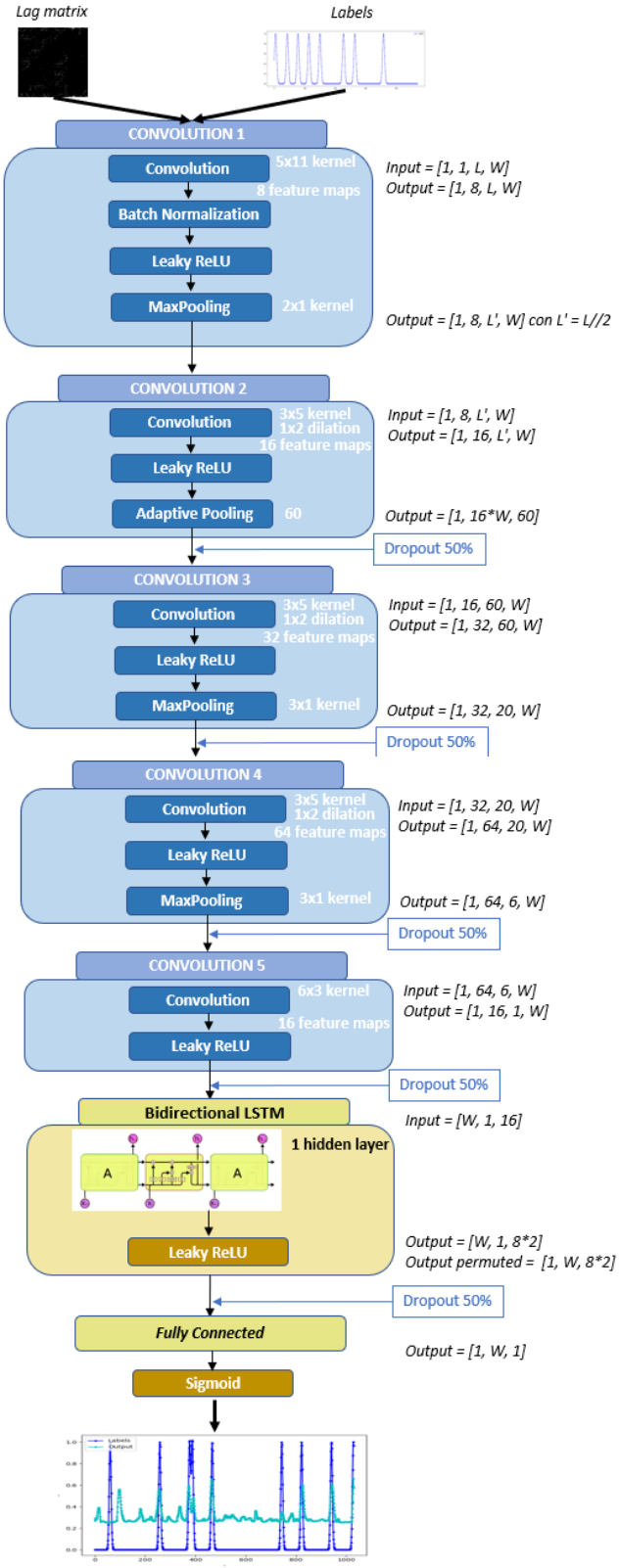


Figura 44. Estructura del modelo CNN con LSTM desarrollado

4.4.3. Entrenamiento del Modelo. Optimización de Parámetros

Para entrenar la red y obtener los resultados que mejor se adapten al problema planteado se ha dividido el dataset de 1006 canciones en un 65% para entrenamiento (654 canciones), 15% para validación (151 canciones) y un 20% para test (201 canciones).

El ***learning rate*** se ha estimado realizando entrenamientos de un pequeño número de épocas y viendo cómo evoluciona el error de entrenamiento y validación. El valor final se ha fijado en 0.001 para ambos modelos.

Como **optimizador de la función de coste** se ha escogido Adam, que es un optimizador que combina dos extensiones del algoritmo de gradiente estocástico descendente: el AdaGrad y el RMSProp.

El **número de épocas** se ha determinado lanzando un entrenamiento más largo y viendo cuándo convergen las curvas de pérdida de test y entrenamiento. El número de épocas final es 60 para el modelo CNN y 200 para el modelo CNN con LSTM. La figura 45 se ha obtenido con TensorboardX [41] donde se puede observar que el modelo CNN con LSTM sobreentrena pero, al contrario que en el modelo CNN, se han fijado aún así 200 épocas de entrenamiento debido a que la curva de precisión F_1 (figura 48c) crece tanto en entrenamiento como en validación hasta esa época a partir de la cual se hace constante. En la figura 45 la curva naranja corresponde al dataset de entrenamiento y la azul al de validación.

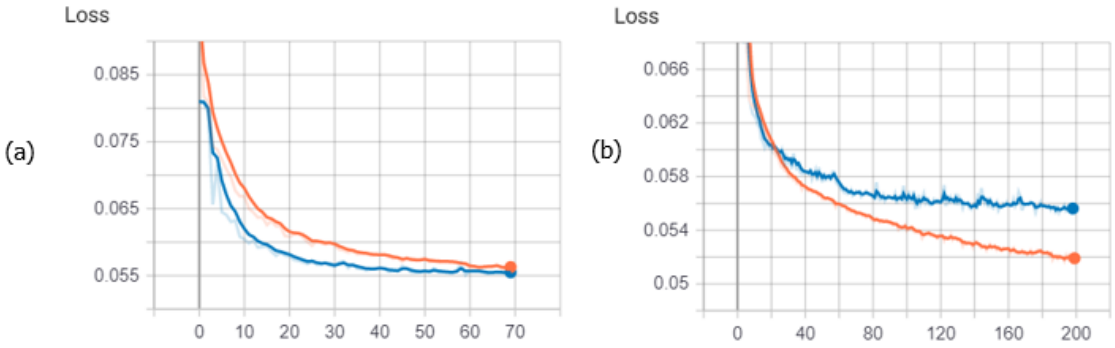


Figura 45. Función de pérdida en función del número de épocas (a) modelo CNN (b) modelo CNN con LSTM

La **función de pérdida** o *loss function* se ha determinado a partir de los resultados de entrenamiento habiendo realizado entrenamientos con distintas funciones de pérdida. A continuación se muestran los resultados de entrenamiento para las funciones de pérdidas BCE y MSE de Pytorch, siendo la **MSE** la función de pérdida escogida introducida en el capítulo anterior, aunque los resultados de entrenamiento no distan mucho entre las dos funciones. Aún previendo que entrenando los modelos con la función de pérdida BCE se obtendrán peores valores de precisión, en las tablas 1 y 2 a continuación se muestran los resultados de precisión para ambas funciones de pérdida, MSE y BCE.

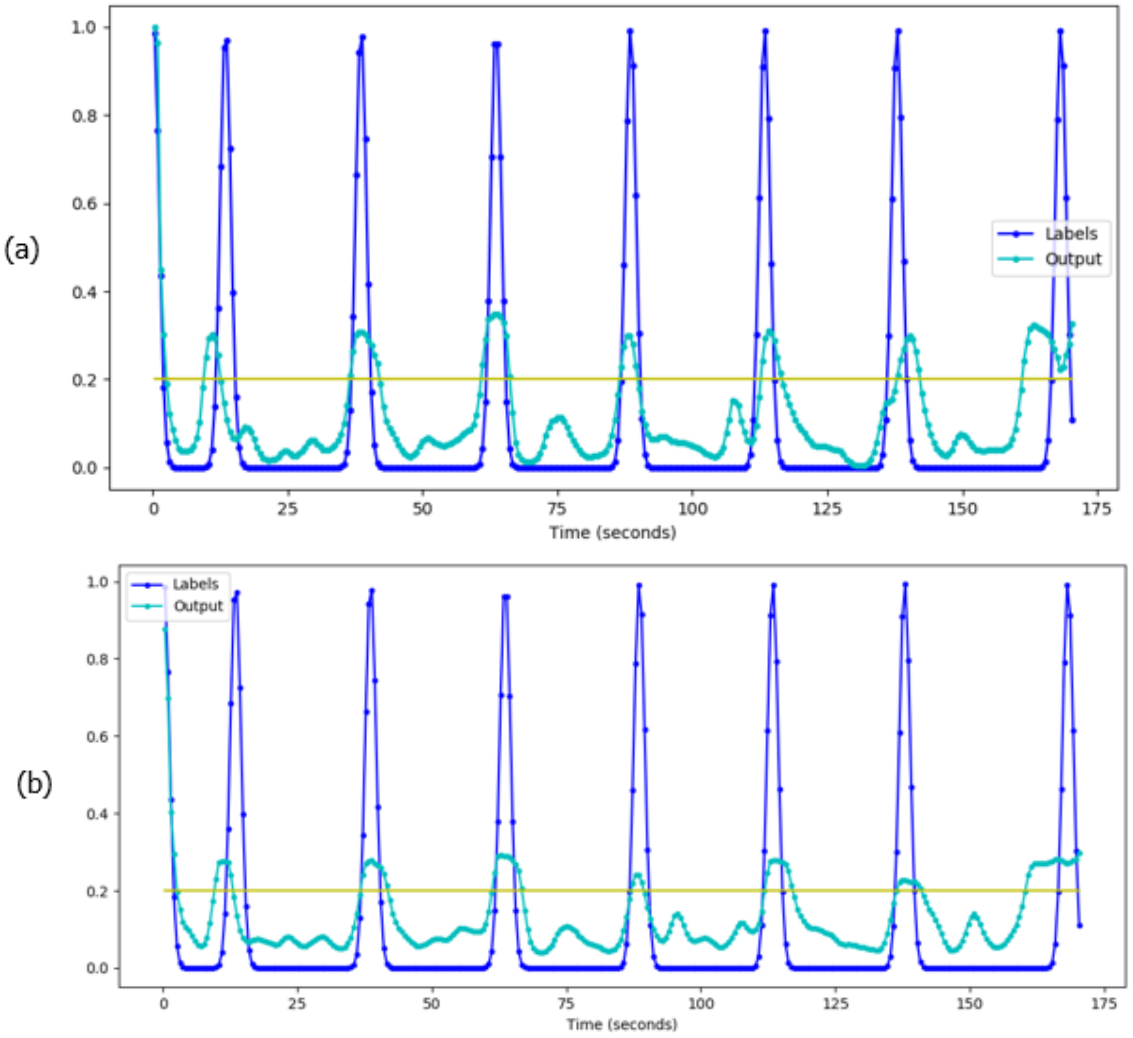


Figura 46. Resultados de entrenamiento de 60 épocas según las funciones de pérdida (a) BCE (b) MSE para la canción 6 de SALAMI (dataset de validación) con el modelo CNN de la figura 43

4.5. Análisis de los Resultados

Para evaluar los resultados de entrenamiento obtenidos, al igual que Thomas Grill y Jan Schlüter [11] se ha utilizado el método de evaluación propuesto por el MIREX (Music Information Retrieval Evaluation Exchange) [42]. El método de evaluación consiste en utilizar la función `mir_eval.segment.detection` [43], que toma como entrada dos parámetros, β y la ventana o tolerancia en segundos. Los valores de estos los parámetros se fijarán siguiendo estudios prelliminares a este trabajo [11], siendo entonces el parámetro β por defecto $\beta = 1$ y se evaluará también para el valor $\beta = 0.58$ que según estudios precedentes [44] es un valor que mejora el valor del parámetro F_1 (el subíndice 1 hacer referencia al parámetro β), y la tolerancia se evaluará para los valores de ±0.5 y ±3 segundos.

La función `mir_eval.segment.detection` retorna los parámetros *Recall* (R), Precisión (P) y *F-measure* (F_1) que miden la precisión de los resultados. El parámetro

F_1 es el que mide la capacidad del modelo de detectar las transiciones y se calcula de acuerdo a las siguientes expresiones [45]:

$$P = \frac{Verdaderos\ Positivos^{6}}{Verdaderos\ Positivos + Falsos\ Positivos^{7}} \tag{22}$$

$$R = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos + Falsos\ Negativos^{8}} \tag{23}$$

$$F_\beta = \frac{(2 * P * R) * (1 + \beta^2)}{\beta^2 * P + R} \tag{24}$$

Cuando se tiene ya el modelo entrenado, es necesario seleccionar un umbral $\delta\ \epsilon$ [0,1] en el eje vertical correspondiente a los valores de pico entre cero y uno, a partir del cual se seleccionarán los picos que se tomarán como las transiciones predichas por el modelo. Otro parámetro a fijar será λ y que se fijará en un segundo $\lambda = 1$ sg que corresponde a la distancia horizontal (eje de tiempo) mínima entre dos picos consecutivos. La función de Python mediante la cual se seleccionarán los picos será `scipy.signal.find_peaks` a la cual se le pasarán los parámetros δ (*height*) y λ (*distance*).

El parámetro δ se ha determinado en función del valor de F_1 para el modelo ya entrenado en el dataset de validación. Como se muestra en la figura 47a, para el modelo CNN se obtinene un valor máximo de F_1 cuando δ = 0.2, y para el modelo CNN con LSTM se tomará el mismo valor siguiendo los resultados de la figura figura 47b.

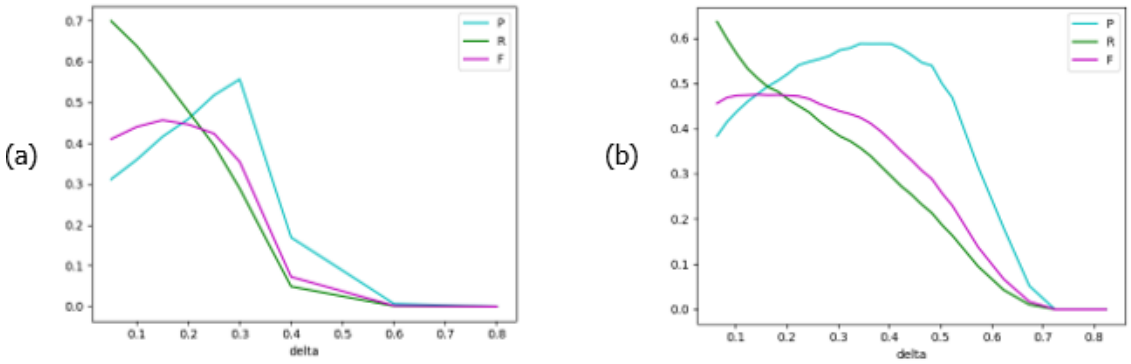


Figura 47. $F_1, R\ y\ P$ de la media del dataset de validación en función del parámetro δ (a) modelo CNN (b) modelo CNN con LSTM

[6] Un verdadero positivo es un resultado en el que el modelo predice correctamente los picos de la señal en este caso
[7] Un falso positivo es un pico predicho que no corresponde a ninguna etiqueta
[8] Un falso negativo es un resultado en el que el modelo debería predecir un pico según la etiqueta pero no lo predice

Durante el entrenamiento del modelo se han realizado los gráficos de la evaluación de los parámetros R, P y F_1 frente al número de épocas. En la figura 48 se muestran los gráficos para el modelo CNN con LSTM, δ = 0.2 y una tolerancia de ±3 segundos.

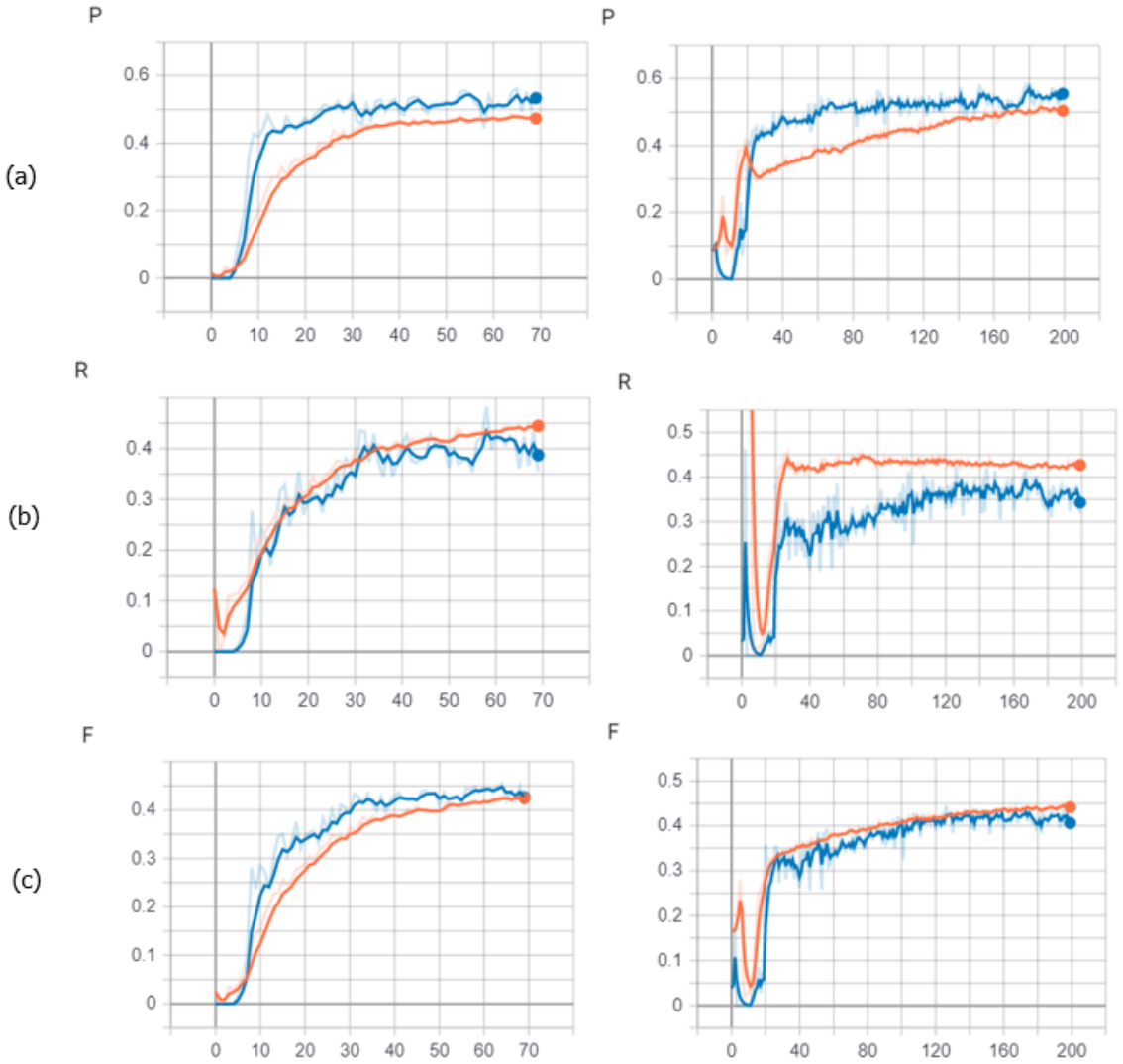


Figura 48. (a) P (b) R y (c) F_1 para los modelos CNN y CNN con LSTM de la media del dataset de entrenamiento (naranja) y validación (azul) para el modelo CNN con δ = 0.2 y ±3 segundos de tolerancia

Una vez determinado el valor del parámetro δ = 0.2 obtenido a partir de la figura 47, se evalúan los resultados en el dataset de test para una tolerancia de ±0.5 segundos.

Modelo (*Loss Function*)	**F_1**	**$F_{0.58}$**	**R**	**P**
CNN (BCE)	0.143	0.149	0.141	0.162
CNN (MSE)	0.144	0.145	0.152	0.15
CNN con LSTM (BCE)	0.138	0.143	0.138	0.154
CNN con LSTM (MSE)	0.139	0.144	0.135	0.154

Tabla 1. Reconocimiento de transiciones con una tolerancia de ±0.5 segundos para el dataset de test

Para el mismo valor de δ (δ = 0.2), con una tolerancia de ±3 segundos se tiene:

Modelo (*Loss Function*)	**F_1**	**$F_{0.58}$**	**R**	**P**
CNN (BCE)	0.449	0.473	0.438	0.524
CNN (MSE)	0.481	0.482	0.512	0.496
CNN con LSTM (BCE)	0.492	0.511	0.485	0.547
CNN con LSTM (MSE)	0.487	0.511	0.466	0.552

Tabla 2. Reconocimiento de transiciones con una tolerancia de ±3 segundos para el dataset de test

A la vista de los resultados se obtiene una mayor precisión con β = 0.58 y con una tolerancia de ±3 segundos. Esto es debido a que las imágenes que se forman a partir de la *Lag Matrix* son de una resolución mucho menor que en [11], por tanto sería natural obtener mejores resultados para una tolerancia mayor. Dado que el análisis musical no es un procedimiento con unos patrones tan fijados como en otras disciplinas cabe destacar que según [11], la precisión de los anotadores es de F_1 = 0.74, por lo que un hipotético valor de F_1 = 1 en los modelos que sigan estas anotaciones seguirían sin ser perfectos, como tampoco lo es el análisis musical como disciplina ya que existen divergencias en las formas de interpretar la estructura de ciertas obras y géneros.

En lo que concierne a la función de pérdida, se puede comprobar que no existen prácticamente diferencias entre los resultados obtenidos con ambas funciones.

A su vez, se puede afirmar que el modelo LSTM que es más sofisticado acaba sobreentrenando y tanto el error obtenido durante el entrenamiento como el valor F_1 obtenido para el dataset de test no distan prácticamente del obtenido con el modelo CNN por lo que se puede asumir también como válido el modelo con solo CNNs.

Para la figura 46b correspondiente a la canción 6 de SALAMI en el dataset de test con el modelo CNN entrenado 60 épocas y *Loss Function* MSE se muestra un ejemplo de cálculo de los parámetros P, R y F_1 para los que se tienen los siguientes valores:

$$P = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos + Falsos\ Positivos} = \frac{7}{7+1} = 0.89$$

$$R = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos + Falsos\ Negativos} = \frac{7}{7+0} = 1$$

$$F_1 = \frac{(2 * P * R) * (1+\beta^2)}{\beta^2 * P + R} = \frac{2 * 0.89 * 1}{0.89 + 1} = 0.941$$

Por último, se muestra el *Recurrence Plot* de la canción 6 de SALAMI con las transiciones reales y las predichas.

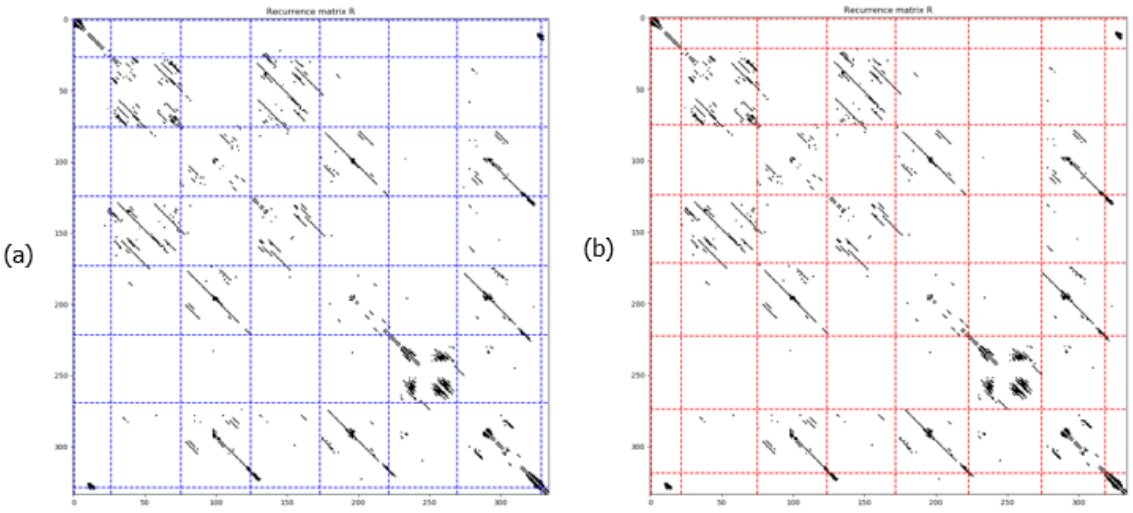


Figura 49. *Recurrence Plot* de la canción 6 de SALAMI con (a) etiquetas de los anotadores functions.txt (azul) (b) etiquetas predichas por el modelo CNN (rojo)

Capítulo V

Comparación de los Resultados

En este capítulo se realiza un a comparación de los resultados obtenidos en los capítulos III y IV con el estado del arte.

5.1. Etiquetado de Estructura

En el caso de etiquetado de la estructura de las partes depiezas musicales, no hay estudios preliminares que realicen esta función mediante redes neuronales. Los estudios precedentes [26] utilizan Modelos Ocultos de Markov y métodos estadísticos [25], y obtienen resultados superiores a los obtenidos en el capítulo III de este trabajo pero no con la misma base de datos, por lo que los resultados no son comparables.

5.2. Detección de Transiciones

En lo que concierne al capítulo IV de detección de transiciones que es donde hay un estado del arte con el que comparar resultados de una forma más directa, se comparan a continuación los resultados obtenidos en este trabajo para ver a qué punto se ha conseguido llegar.

La tabla 3 muestra un resumen de los resultados de [28] considerando múltiples anotaciones para cada canción, los resultados de [46] con una anotación por canción, y los de este trabajo también con una anotación por canción. Cabe recordar que el modelo de referencia [28] utiliza como entradas un espectograma de MEL (MLS en la tabla 3) más dos *Self-Similarity Lag Matrixes* (SSLM en la tabla 3) referentes a la vecindad a partir de la cual se forma la SSLM (14 segundos para la cercana o SSLM-*near* y 88 segundos para la lejana o SSLM-*far*). En este trabajo por el contrario se ha utilizado solo una *Lag Matrix* como entrada a la red.

Tolerancia	**Modelo (Entrada Red)**	**F_1**	**$F_{0.58}$**	**R**	**P**
	Máximo anotadores (estimado en [9])	0.74	0.74	-	-
0,5 sg	CNN [46]	0.523	0.596	0.484	0.646
	CNN (MLS+SSLM) [9]	0.508	0.529	0.502	0.572
	CNN (MLS+SSLM) [9]	0.496	0.506	0.509	0.536
	CNN (MLS+SSLM) [41]	0.469	0.466	0.504	0.475
3 sg	**CNN con LSTM (Lag)**	**0.492**	**0.511**	**0.485**	**0.552**
	Base (estimado en [9])	0.15	0.21	-	-

Tabla 3. Reconocimiento de transiciones con una tolerancia de ±0.5 segundos y ±3 segundos (fila en negrita del modelo CNN con LSTM implementado en este trabajo en el capítulo IV) para el dataset de test de SALAMI 2.0

En [46], el modelo de referencia de [28], que es el punto de partida de este trabajo, F_1 = 0.469, y en [28] se consigue un resultado de F_1 = 0.508 para el mejor modelo, estos resultados con una ventana de ±0.5 segundos. Para este trabajo los resultados mejores salen para una ventana de ±3 segundos al contrario que en [28] donde empeoran con una ventana más ancha. Esto puede ser debido a que en [28] el modelo particularice más pero sea menos genérico, por ello faltaría comprobar si para imágenes con la resolución que se usa en [28], el modelo propuesto en este trabajo mejoraría aún más los resultados siendo un modelo más general que mejorase el estado del arte.

Capítulo VI

Conclusiones y Líneas Futuras

En este capítulo se exponen las conclusiones del trabajo realizado y las líneas futuras que quedan por resolver.

6.1. Conclusiones

En este trabajo se ha realizado una estimación del etiquetado de las partes de piezas de música atendiendo su estructura formal y de la detección de las transiciones en la estructura de piezas musicales de un dataset de obras musicales de diversos géneros tomando como referencia un modelo [28] realizado para el mismo propósito.

Para el caso de estudio de etiquetado de las diferentes partes de la estructura musical de las piezas de música, a la vista de los resultados obtenidos en el capítulo III se puede confirmar que el modelo implementado en el capítulo III no es suficiente para realizar el propósito marcado como objetivo, pero cabría analizar a fondo con expertos en materia de análisis musical si, como se ha comentado en el último párrafo del apartado anterior las confusiones del modelo son debidas son confusiones que podría cometer un músico a la hora de analizar una canción, o es sobre todo debido a la información suministrada a la red, que quizás no sea suficiente y hay a que suministrar además información armónica. De todas formas, cabe mencionar que en el caso de etiquetado de las partes de las obras de música del capítulo III sería necesario de cara a futuras investigaciones comprobar el modelo con la base de datos de SALAMI al completo como en el capítulo IV e introducir como entrada las *Lag Matrixes* utilizadas en el trabajo y no las obtenidas mediante la *Toolbox* de Matlab como se ha hecho en este anexo.

En cambio, en el caso de la detección de transiciones, se han utilizado menos recursos que en el estado del arte, ya que se introduce solo una imagen de entrada (*Lag Matrix*) frente a 2 imágenes y el espectograma de MEL en [28], lo que lleva a la conclusión de que se han conseguido unos resultados similares pero para distintos márgenes de error dada la resolución de las imágenes de entrada utilizadas como entrada a la red en este trabajo, y por tanto quedaría por comprobar si, obteniendo las imágenes con las que se ha entrenado el modelo a más resolución e implementando la entrada del modelo como en la referencia [28] se obtendrían mejores resultados mejorando así el estado del arte.

6.2. Líneas Futuras

Antes de plantear futuras líneas de investigación relacionadas con el análisis y composición musical mediante Redes Neuronales convendría plantear una mejora del detector de transiciones del capítulo IV y a partir de ahí remodelar el caso de estudio de etiquetado de las partes de las piezas de música del capítulo III y abordarlo de otra forma con el fin de mejorar los resultados obtenidos.

Las líneas de investigación acerca del procesamiento de audio y análisis musical a través de la Inteligencia Artificial siguen siendo muy amplias. Desde la detección e identificación automática de acordes, detección de transiciones como se ha realizado en este trabajo o etiquetado de la estructura propia de una obra musica, hasta el sintetizado de audio o composición musical son campos de investigación donde todavía queda mucho por recorrer. En lo que concierne a este trabajo, las líneas de investigación más cercanas son la mejora de la detección de transiciones que posteriormente podrían ayudar a conseguir modelos que analizasen la estructura formal de una pieza de música, y su combinación con otras técnicas como la detección e identificación armónica podría

utilizarse para una posterior composición musical que hasta la fecha no se ha conseguido de una forma estructurada.

Más allá de la composición musical, línea de investigación principal de este trabajo, hay muchos enfoques distintos dentro de la música y la ciencia, como la interacción humano-máquina en detección de emociones provocadas por distintos estilos musicales, etc.

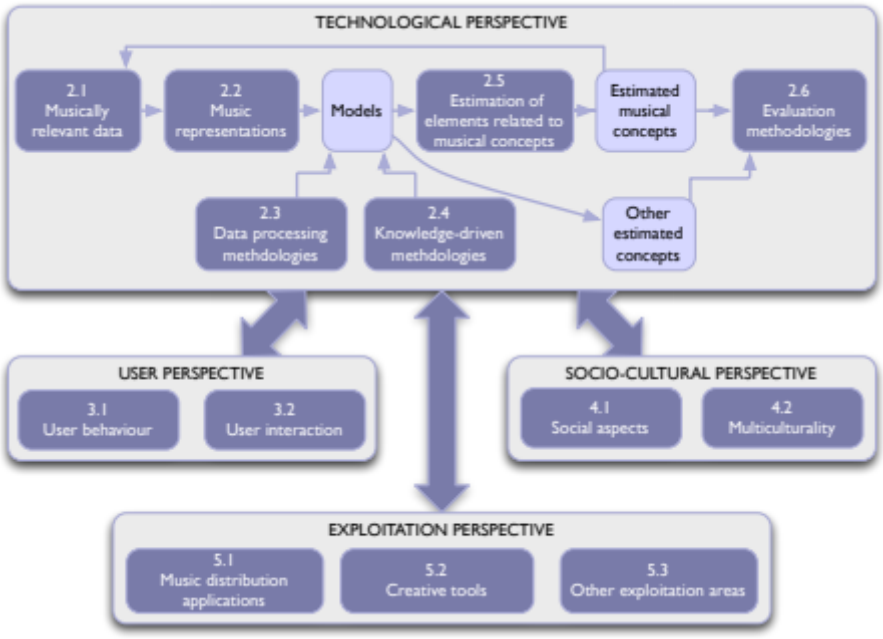


Figura 50. Perspectivas futuras de enfoque en el MIR (*Music Information Retrieval*) [47]

Referencias

[1] RESTELLI, Marcello. Machine Learning course. s.l. : *Politecnico di Milano*, 2019.

[2] http://musicaenbejar.blogspot.com/2012/01/analisis-musical-acercamiento-los.html (*último acceso, noviembre 2019*).

[3] https://es.slideshare.net/jazabril/analisis-fugal-de-la-fuga-ii-bwv-847-del-libro-i-del-clave-bien-temperado-de-johann-sebastian-bach (*último acceso, noviembre 2019*).

[4] RUSSELL, Stuart J.; NORVIG, Peter. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[5] MCCULLOCH, Warren S.; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biology*, 1990, vol. 52, no 1-2, p. 99-115.

[6] https://noeliagorod.com/2019/06/26/deep-learning-timeline/ (*último acceso, noviembre 2019*).

[7] HINTON, Geoffrey E.; OSINDERO, Simon; TEH, Yee-Whye. A fast learning algorithm for deep belief nets. *Neural computation*, 2006, vol. 18, no 7, p. 1527-1554.

[8] https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/ (*último acceso, noviembre 2019*).

[9] MATTEUCCI, Matteo. Soft Computing course. s.l. : *Politecnico di Milano*. 2019.

[10] RUMELHART, David E., et al. Learning representations by back-propagating errors. *Cognitive modeling*, 1988, vol. 5, no 3, p. 1.

[11] LECUN, Yann, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998, vol. 86, no 11, p. 2278-2324.

[12] BISHOP, Christopher M. *Pattern recognition and machine learning*. springer, 2006.

[13] WANG, Jinjiang, et al. Deep learning for smart manufacturing: Methods and applications. *Journal of Manufacturing Systems*, 2018, vol. 48, p. 144-156.

[14] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 (*último acceso, noviembre 2019*).

[15] https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/ (*último acceso, noviembre 2019*).

[16] https://es.switch-case.com/52278319 (*último acceso, noviembre 2019*).

[17] https://www.mdpi.com/1424-8220/19/7/1693/htm (*último acceso, noviembre 2019*).

[18] https://pytorch.org/docs/stable/nn.html#maxpool2d (*último acceso, noviembre 2019*).

[19] https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-4-full-connection (*último acceso, noviembre 2019*).

[20] SRIVASTAVA, Nitish, et al. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 2014, vol. 15, no 1, p. 1929-1958.

[21] IOFFE, Sergey; SZEGEDY, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[22] HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. *Neural computation*, 1997, vol. 9, no 8, p. 1735-1780.

[23] GOODFELLOW, I., BENJIO, Y., COURVILLE A. *Deep Learning* (Chapter 10). MIT Press. 2016.

[24] https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (*último acceso, noviembre 2019*).

[25] PAULUS, Jouni; KLAPURI, Anssi. Music structure analysis using a probabilistic fitness measure and a greedy search algorithm. *IEEE Transactions on Audio, Speech, and Language Processing*, 2009, vol. 17, no 6, p. 1159-1170.

[26] LEVY, Mark; SANDLER, Mark. Structural segmentation of musical audio by constrained clustering. *IEEE transactions on audio, speech, and language processing*, 2008, vol. 16, no 2, p. 318-326.

[27] HADJERES, Gaëtan; PACHET, François; NIELSEN, Frank. Deepbach: a steerable model for bach chorales generation. En *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017. p. 1362-1371.

[28] GRILL, Thomas; SCHLÜTER, Jan. Music Boundary Detection Using Neural Networks on Combined Features and Two-Level Annotations. En *ISMIR*. 2015. p. 531-537.

[29] FOOTE, Jonathan. Visualizing music and audio using self-similarity. En *ACM Multimedia (1)*. 1999. p. 77-80.

[30] MÜLLER, Meinard. *Fundamentals of music processing: Audio, analysis, algorithms, applications*. Springer, 2015.

[31] Jordan B. L. Smith, J. Ashley Burgoyne, Ichiro Fujinaga, David De Roure, and J. Stephen Downie. 2011. Design and creation of a large-scale database of structural annotations. *Proceedings of the International Society for Music Information Retrieval Conference*. Miami, FL. 555-60.

[32] https://www.audiolabs-erlangen.de/resources/MIR/SMtoolbox/ (*último acceso, noviembre 2019).*

[33] https://pytorch.org/docs/stable/nn.html (*último acceso, noviembre 2019*).

[34] ECKMANN, J. P., et al. Recurrence plots of dynamical systems. *World Scientific Series on Nonlinear Science Series A*, 1995, vol. 16, p. 441-446.

[35] N. Marwan,M. C. Romano, M. Thiel, and J. Kurths, "Recurrence plots for the analysis of complex systems," *Phys. Rep.*, vol. 438, no. 5–6, pp. 237–329, 2007.

[36] GOTO, Masataka. A chorus section detection method for musical audio signals and its application to a music listening station. *IEEE Transactions on Audio, Speech, and Language Processing*, 2006, vol. 14, no 5, p. 1783-1794.

[37] FUJISHIMA, Takuya. Real-time chord recognition of musical sound: A system using common lisp music. *Proc. ICMC, Oct. 1999*, 1999, p. 464-467.

[38] Joan Serrà, Meinard Müller, Peter Grosche, and Josep Ll. Arcos. Unsupervised music structure annotation by time series structure features and segment similarity. In IEEE Transactions on Multimedia, 16(5):1229–1240, 2014.

[39] MCFEE, Brian, et al. librosa: Audio and music signal analysis in python. En *Proceedings of the 14th python in science conference*. 2015.

[40] COVER, Thomas; HART, Peter. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 1967, vol. 13, no 1, p. 21-27.

[41] https://github.com/lanpa/tensorboardX (*último acceso, noviembre 2019*).

[42] RAFFEL, Colin, et al. mir_eval: A transparent implementation of common MIR metrics. En *In Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR*. 2014.

[43] https://craffel.github.io/mir_eval/ (*último acceso, noviembre 2019*).

[44] NIETO, Oriol, et al. Perceptual analysis of the f-measure for evaluating section boundaries in music. En *Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR 2014)*. 2014. p. 265-270.

[45] https://www.music-ir.org/mirex/wiki/2019:Audio_Onset_Detection_(*último acceso, noviembre 2019*).

[46] GRILL, Thomas; SCHLUTER, Jan. Music boundary detection using neural networks on spectrograms and self-similarity lag matrices. En *2015 23rd European Signal Processing Conference (EUSIPCO)*. IEEE, 2015. p. 1296-1300.

[47] SERRA, Xavier, et al. Roadmap for music information research. 2013.