Universidad Zaragoza
1542

Trabajo Fin de Máster

Sistema de transcripción multitímbrica basado en redes neuronales profundas

Multitimbral transcription system based on deep neural networks

Autor

Carlos Hernández López

Director

José Ramón Beltrán Blázquez

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2021

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Carlos Hernández López,

con nº de DNI 73213482C en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Máster, (Título del Trabajo) Sistema de transcripción multitímbrica basado en redes neuronales profundas,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 30 de Mayo del 2021

Fdo: Carlos Hernández López

RESUMEN

El objetivo de este trabajo es conseguir una transcripción MIDI a partir de un archivo WAV en melodías polifónicas (varias notas sonando al mismo tiempo) y multitímbrica (para diferentes instrumentos) , haciendo uso del *deep learning*. Para ello, se entrenan dos modelos, uno que detecta las notas completas (*frames*) y otro que se centra únicamente en los inicios de notas (*onsets*), finalizando con un algoritmo de *note tracking* que unirá ambos modelos. El *deep learning* es una herramienta que ha evolucionado en gran medida estos últimos años gracias al avance de la tecnología. La implementación se realiza mediante Python en el entorno de Anaconda, sobre todo, con los programas de Spyder y Jupyter notebook. Además, para facilitar el trabajo con las redes neuronales se utilizará Tensorflow.

ABSTRACT

The aim of this work is to achieve a MIDI transcription from a WAV file in polyphonic (several notes sounding at the same time) and multitimbral (for different instruments) melodies, making use of deep learning. To do this, two models are trained, one that detects whole notes (frames) and another that focuses only on the beginnings of notes (onsets), ending with a note tracking algorithm that will unite both models. Deep learning is a tool that has evolved greatly in recent years thanks to advances in technology. It is implemented using Python in the Anaconda environment, mainly with the Spyder and Jupyter notebook programs. In addition, Tensorflow will be used to facilitate the work with neural networks.

Índice

Capítulo 1

Introducción y Objetivos

1.1. Motivación

En los años 50, empezaron los primeros pasos en la Inteligencia Artificial (IA). En sus comienzos, esta rama del conocimiento se aplicaba mayoritariamente en laboratorios e investigación. Con la ayuda del gran impulso tecnológico que ha habido en los últimos años y la creciente necesidad de procesar grandes cantidad de datos, este campo se encuentra en una constante mejora. En la figura 1.1, se puede observar la evolución de inversión privada en *start-ups* relacionadas con la IA.

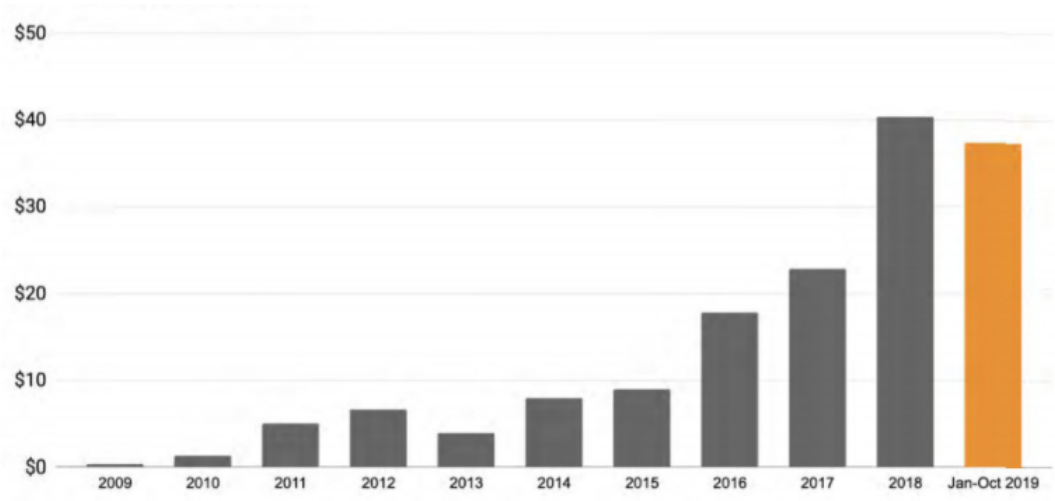

Figura 1.1: Crecimiento de la inversión privada en inteligencia artificial (en miles de millones de dólares) [1]

Además, según el *AI Index Report* de 2019 [2], el número de artículos relacionados con la inteligencia artificial ha crecido en un 300 % en 2018 con respecto a 1998.

En los últimos años el crecimiento y el interés por el mundo del *Machine Learning* ha sido exponencial, en gran parte apoyado por la gran mejora en la computación. En este trabajo, nos vamos a centrar en la aplicación del *Deep Learning* al ámbito musical, concretamente en el campo de la transcripción automática de música polifónica.

1.2. Objeto

La transcripción musical es una parte muy importante en el análisis de señales musicales, ya que contribuye a la mejora de aplicaciones dentro de esta industria. Con ella, se consigue una aceleración del desarrollo de la música comercial y se facilita la educación musical. Sin embargo, este proceso tiene como requisito tener unos grandes conocimientos en este ámbito y experiencia en el sector de la música.

Este trabajo se centra en el desarrollo una red neuronal (ver capítulo 2) con Keras [3] y Tensorflow [4] en el ámbito del análisis musical. La tarea a resolver por parte de la red será transcribir música polifónica y multitímbrica. La música polifónica es aquella en la que pueden estar sonando simultáneamente varias notas musicales y multitímbrica es que servirá para varios instrumentos. El lenguaje de programación utilizado es Python, haciendo uso las herramientas de Spyder y Jupyter notebook.

Se hará uso de tres bases de datos (*datasets*): Slakh2100 con la que entrenaremos la red; y MedleyDB y Bach10, con los que haremos un test para poder compararlo con el estado del arte.

El desarrollo software del proyecto se compone de diferentes módulos:

- **Módulos de lectura de los dataset**: Se encuentran las funciones necesarias para poder hacer una lectura correcta de las bases de datos.
- **Módulo de procesamiento de entrada**: Almacena las funciones que nos permitirán procesar las señales de audio. En nuestro caso, se hará un Transformad de Q constante (CQT) [5] (ver capítulo 2.8) de la señal y se apilarán varios armónicos en una misma entrada.
- **Módulo de modelos**: En este módulo se definirán los modelos a utilizar y las funciones de pérdida (ver capítulo 2.4.4).
- **Módulo de entrenamiento**: En el que se encuentran las funciones para entrenar las red, así como la visualización de las salidas y del resultado de la funcion de pérdida a tiempo real. Debido a la disposición de los datos, será necesario hacer el cálculo de las CQTs dentro del bucle de entrenamiento.
- **Módulo de test**: Nos permitirá hacer la valoración final del comportamiento de la red en los diferentes *datasets*.

El objetivo del trabajo es definir una arquitectura que sea capaz de transcribir de manera correcta una melodía multitímbrica y que lo haga para diferentes instrumentos (multitímbrica). De tal manera, que a partir de un archivo WAV se extraiga un MIDI.

El resultado de este trabajo, se ve reflejado en la publicación de un artículo en la revista MDPI [6].

Esta memoria esta estructurada se estructura de siete capítulos. Este capítulo de introducción donde se explica cual es la motivación, objetivos y estructura del desarrollo del software.

El capítulo 2 es una introducción a la transcripción automática de música donde se hablará de ella, así como de conceptos relacionados como la inteligencia artificial y el procesamiento de la señal de audio.

En el capítulo 3 se expone el estado del arte donde se incluye este trabajo. Se hablará sobre las diferentes arquitecturas de modelos neuronales que existen y sus características.

En el capitulo 4 se da a conocer la arquitectura del modelo desarrollado en este trabajo y sus características. Para posteriormente, en el capitulo 5, explicar los experimentos realizados y sus resultados. En el capítulo 6, se explica como se ha implementado en Python.

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

Capítulo 2

Introducción a la transcripción automática de música

El objetivo de la transcripción musical es crear una representación en un formato de notación musical a partir de un archivo de audio, en nuestro caso la representación será en formato MIDI (ver apartado 2.7). Este tipo de representación facilita la tarea al compositor a la hora de buscar nuevos elementos musicales y ayuda a reducir el espacio de almacenamiento de las melodías.

2.1. Transcripción automática de música (AMT)

La transcripción automática de música [7] consiste en el diseño de algoritmos que consigan convertir señales de audio en notación musical mediante el procesamiento de señales y, en nuestro caso, la inteligencia artificial. Dentro de la transcripción existen varias tareas como la detección de notas, el reconocimiento de instrumentos, el ritmo de una canción o la composición de partituras. Debido al gran número de tareas que existen, la transcripción automática se encuentra como uno de los problemas más importantes de la recuperación de información musical (MIR).

El procedimiento comienza con una señal de audio (figura 2.1(a)), se realiza un procesado de esa señal en una representación tiempo/frecuencia (figura 2.1(b)), con estos datos se hace una representación de las notas con respecto al tiempo (*pianoroll*, figura 2.1(c)) y en algunos casos se llega a la representación de partituras (figura 2.1(d)).

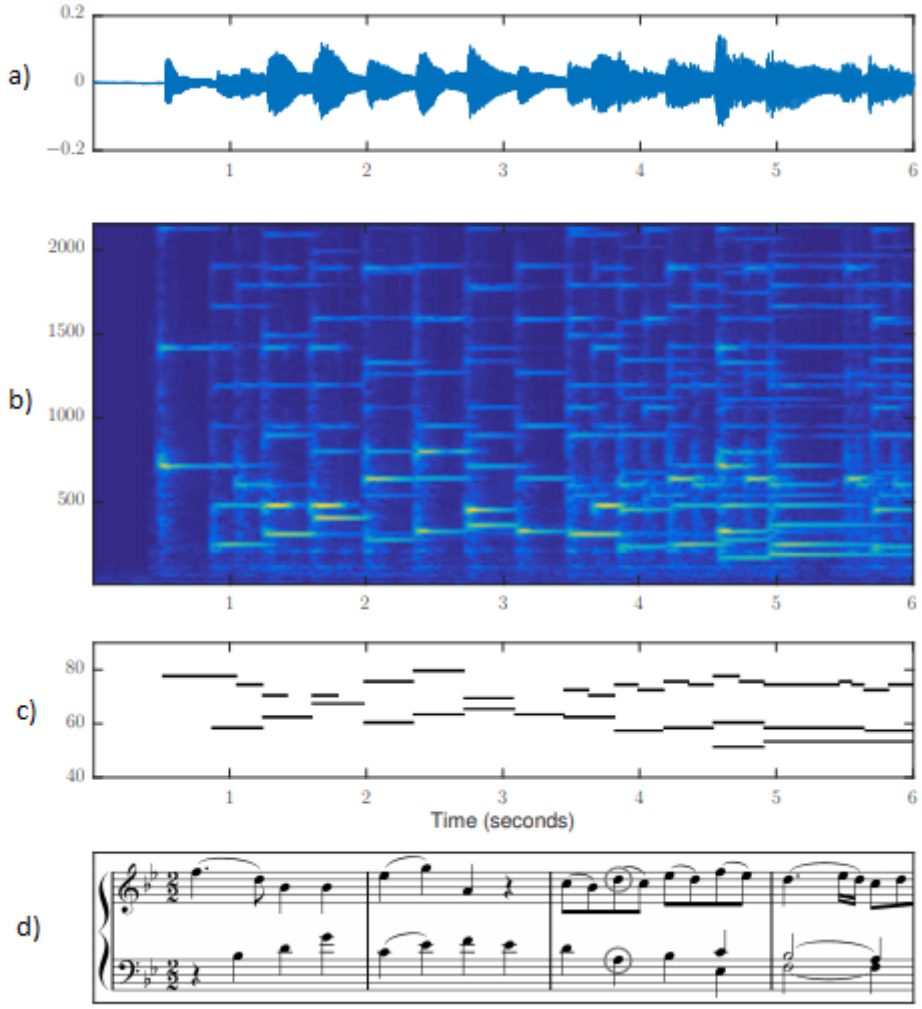

Figura 2.1: Representación de los datos que se tienen en un sistema de transcripción de música automática [7]

También existen métodos para detectar instrumentos que no están afinados (como un tambor u otros instrumentos de percusión), es decir, que no pueden hacer sonar una nota musical. Estos métodos tienen como objetivo la detección de golpes de sonido con saltos de intensidad muy grandes en el momento del inicio del sonido.

Existen varios factores que provocan que la transcripción de música sea especialmente difícil si la comparamos con otras ramas del campo del procesamiento de señales musicales [8].

- **Polifonía musical**: Las melodías musicales pueden contener varias notas sonando al mismo tiempo. Debido a la mezcla de armónicos que existen entre ellas provoca que la señal sea realmente complicada de separar.

- **Eventos sonoros superpuestos**: Las notas presentan similitudes armónicas entre sí.

- **Estructura regular**: Los músicos a la hora de componer presentan mucha atención en la sincronización de los inicios de las notas y los finales.

- **Anotar música polifónica es muy costoso**: Saber las notas que están sonando en una melodía requiere de mucho tiempo y de mucha experiencia para hacer esta tarea correctamente. Esta anotación es muy importante ya que con ella se pueden entrenar algoritmos supervisados basados en redes neuronales.

Dependiendo del objetivo de transcripción que se persiga en cada caso, se pueden diferenciar varios tipos de categorías o niveles: *frame-level*, *note-level*, *stream-level* y *notation-level*, que pasamos a describir a continuación.

- ***Frame-level***: En este nivel se obtiene la nota que está sonando en un instante de tiempo. Las melodías se analizan en ventanas temporales (*frames*) independientes unas de otras.

- ***Note-level***: Este nivel es superior al anterior, y ya no se hace una valoración independiente de cada *frame*. En este caso se intenta obtener el tono de una nota musical, el instante en *frames* de inicio (*onset*) y el instante de finalización (*offset*). Se consigue así una mejor identificación de la melodía.

- ***Stream-level***: Se estiman las notas y se hacen grupos en los que cada uno de ellos pertenecerá a un instrumento.

- ***Notation-level***: En este último nivel, se transcribe audio a una partitura (en un pentagrama). Esta transcripción requiere de un entendimiento mucho mayor de las estructuras musicales.

Merece la pena hacer un especial énfasis en explicar la diferencia entre la música monofónica y la polifónica. En la música monofónica es mucho más fácil detectar las diferentes notas musicales ya que en un instante de tiempo solo está sonando una nota de un instrumento en concreto. A diferencia de la música polifónica en que existen varias notas sonando al mismo tiempo y será el principal objetivo de este trabajo.

2.2. Inteligencia artificial

La Inteligencia Artificial (AI) es un campo que engloba al campo del *Machine Learning* (ML). Las redes neuronales (NN) es un subcampo del ML, y a su vez, el *Deep Learning* (DL) es un subcampo de las redes neuronales. Toda esta estructura se puede ver en la figura 2.2.

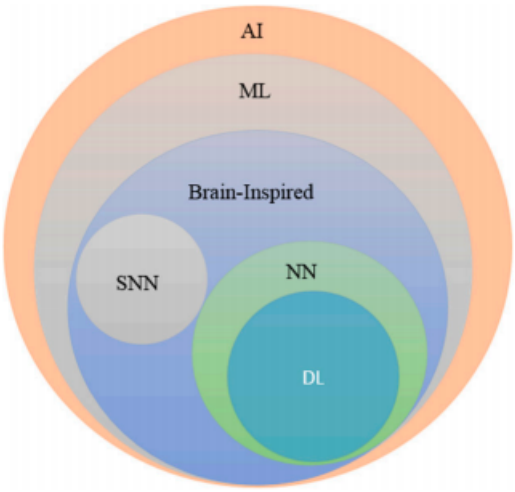

Figura 2.2: Taxonomía de la inteligencia artificial. SSN: *Spiking Neural Networks*; NN: *Neural Networks*; DL: *Deep Learning* [9];

El DL consiste en el uso de arquitecturas profundas que se empezó a desarrollar en gran medida a partir del 2006. El aprendizaje es un procedimiento que consiste en estimar los parámetros de un modelo con el objetivo de realizar una tarea en concreto. Estos parámetros en el caso de las redes neuronales artificiales (ANN) son matrices de pesos. En el caso del DL, las redes tienen muchas capas entre la entrada y la salida, lo que permite mejorar la detección de patrones en el conjunto de datos (*dataset*) [9]. En este momento, el DL se está convirtiendo en un enfoque de aprendizaje que permite solucionar diferentes tipos de problemas en innumerables campos del conocimiento [10].

2.3. Enfoques del Deep Learning

Los enfoques del *deep learning* se puede categorizar por: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje reforzado (RL). Esta clasificación se puede ver en la figura 2.3.

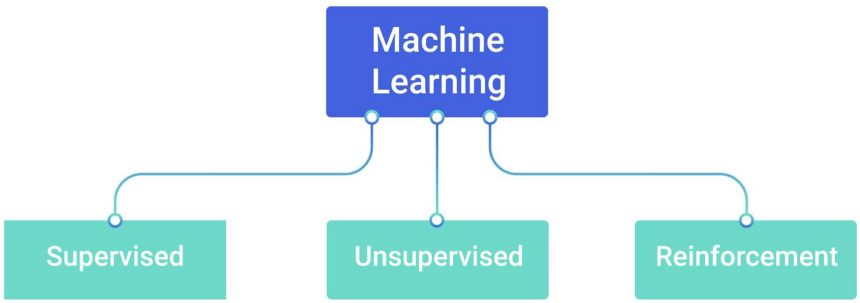

Figura 2.3: Enfoques del Deep Learning

2.3.1. Aprendizaje supervisado

Este tipo de aprendizaje es el que utiliza datos que están etiquetados. Es decir, utilizan un conjunto de datos formado por el conjunto de entradas y sus correspondientes salidas. El modelo de aprendizaje tendrá que ser capaz de estimar la salida a partir de una entrada y recibirá el error que ha tenido comparándolo con la salida real (*ground truth*). El error nos lo reporta una función de pérdida (ver sección 2.4.4), la cual se utilizará para ajustar los parámetros del modelo y mejorar cada vez más su comportamiento. En el enfoque del aprendizaje supervisado se incluyen diferentes tipos de redes como las neuronales profundas, convolucionales profundas, recurrentes, Long Short Term Memory (LSTM [11]) o Gated Recurrent Units (GRU [12]).

2.3.2. Aprendizaje no supervisado

Al contrario que el enfoque anterior, en el aprendizaje no supervisado se utilizan conjuntos de datos que no están etiquetados. En este caso, el agente inteligente aprenderá las posibles relaciones que existen en los datos de entrada. Las tareas suelen ser de agrupamientos, reducción de la dimensionalidad o técnicas generativas. Algunas de las estructuras para realizar estas tareas son los Auto-Encoders [13] o las Generative Adversarial Networks (GANs) [14].

2.3.3. Aprendizaje reforzado

El aprendizaje reforzado es una técnica que se aplica en entornos desconocidos, sus inicios son en el 2013 con Google Deep Mind [15]. Un ejemplo de aprendizaje reforzado sería el siguiente: el entorno le hace una pregunta al modelo, este le da una respuesta y el entorno le devuelve una puntuación. Con esto se consigue que mediante el entrenamiento, el modelo aprenderá a dar una mejor respuesta al entorno. La principal diferencia que existe entre el aprendizaje reforzado y el supervisado es que, en primer lugar, no se tiene acceso total a una función para calcular el error, sino que el agente debe hacer continuamente preguntas al entorno (prueba y error) y, en segundo lugar, la entrada depende de las acciones realizadas anteriormente por el modelo.

2.4. Introducción a las Redes Neuronales Artificiales

Las redes neuronales artificiales surgen a raíz de la idea de imitar al cerebro humano y con el objetivo de resolver múltiples tareas. Con la información que se tiene sobre las neuronas, sus conexiones y la interacción que existe entre ellas, se realizó un modelo que se puede representar de manera matemática. Esta idea es muy sencilla, pero se puede escalar para

hacer modelos más complejos.

2.4.1. Distribución del conjunto de datos

La red toma los datos de un conjunto de entrenamiento que se divide en tres grupos: entrenamiento, validación y *test*. El primer grupo son los datos con los que el modelo aprenderá a detectar los patrones, el grupo de validación se utiliza para ver si durante el entrenamiento está generalizando correctamente; y por último, el grupo de *test* se utiliza una vez terminado el entrenamiento. Se dice que un modelo está generalizando cuando está aprendiendo a encontrar los patrones del *dataset* o por el contrario está memorizando los datos (*overfitting*). Además, existe el concepto de *underfitting* que significa que la red necesita más entrenamiento ya sea por una falta de datos o por la falta de tiempo. En la figura 2.4, se puede ver un resumen de estos conceptos.

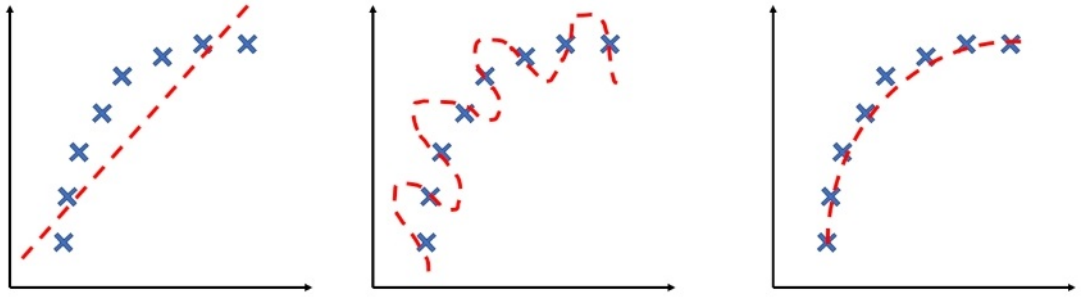

Figura 2.4: Resultados de un entrenamiento. Izquierda: *underfitting*; Centro: *overfitting*; Derecha: Entrenamiento correcto

El porcentaje de distribución de cada uno de estos grupos puede ser muy variado ya sea un 80/20/20 (entrenamiento/validación/*test*), 90/10/10 o incluso para conjuntos de datos muy grandes 99/1/1. Lo verdaderamente importante es que la distribución de los datos sea la misma en cada uno de los grupos. Si esto no sucede, el modelo podría no ser entrenado con un tipo de dato, o podríamos estar haciendo un *test* sin tener en cuenta una parte de nuestro conjunto de datos.

2.4.2. Modelo de una neurona artificial

Como se puede ver en la figura 2.5, el modelo de una neurona artificial se compone de sumas y multiplicaciones, en el que a partir de una o varias entradas, el modelo nos entrega una salida. A la salida de la neurona, se le añade una función de activación con el objetivo de hacer la neurona no-lineal y poder modelar funciones más complejas.

Figura 2.5: Neurona artificial con 5 entradas [16]

En la ecuación 2.1, se ve la expresión matemática que tiene una neurona, siendo y la salida, x_i las entradas, w_i los pesos, b el bias que corresponde a un valor entero y g la función de activación.

$$y = g(b + \sum_{i=1}^{N} x_i w_i) \tag{2.1}$$

Normalmente, se crean redes añadiendo varias neuronas en una misma capa y el nombre que recibe es una capa totalmente conectada (*fully connected o FC)*. En esta capa, todas las neuronas están conectadas con todas las anteriores como se puede ver en la figura 2.6.

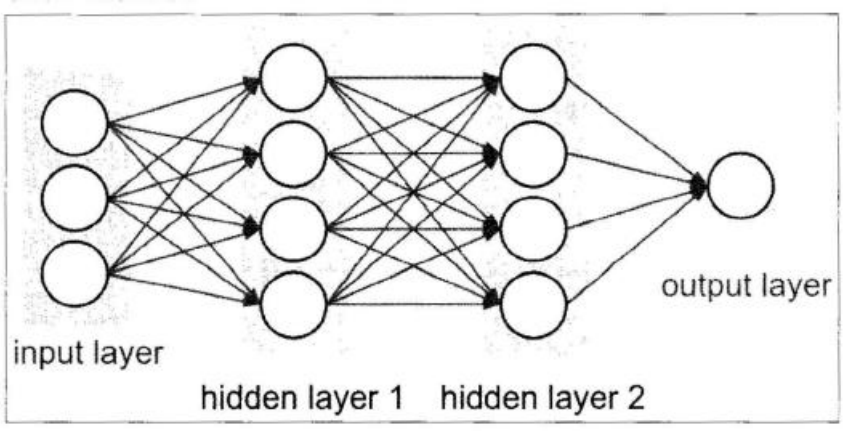

Figura 2.6: Capa totalmente conectada [17]

2.4.3. Tipos de funciones de activación

La función de activación se encarga de convertir la salida de la neurona en una función no-lineal [18]. Además, se buscan funciones en las que la derivada sea simple, para minimizar el coste computacional. Algunas de las funciones más utilizadas son las siguientes:

Sigmoide

Transforma los valores de la neurona a una escala [0,1] eliminando los datos negativos. Tiene el inconveniente de que en los extremos el gradiente se hace muy pequeño, lo que provoca que el modelo converja en un mínimo más lentamente. Este tipo de función, se suele utilizar comúnmente en la ultima capa. Su expresión se puede ver en la ecuación 2.2:

$$f(x) = \frac{1}{1 - e^{-x}} \tag{2.2}$$

Tangente hiperbólica

Al igual que la sigmoide, tiene el problema de que los gradientes en los extremos se hacen 0, con la diferencia de que el rango de salida es [-1,1]. Se utiliza bastante en las redes recurrentes. La expresión corresponde a la ecuación 2.3.

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.3}$$

ReLU

La *Rectified Lineal Unit* (ReLU) es una de las funciones más utilizadas en las redes convolucionales ya que se anulan los valores negativos y no pierde el gradiente en la parte positiva. Es una función por partes y se que puede ver en la ecuación 2.4.

$$f(x) = max(0, x) = \begin{cases} 0 & , \text{ si } x < 0 \\ x & , \text{ si } x \geq 0 \end{cases} \tag{2.4}$$

Softmax

La función Softmax transforma todas las salidas en forma de probabilidades, de tal manera que la suma de todas las salidas de 1. Se utiliza sobre todo para tareas de clasificación cuando hay varias clases (ecuación 2.5), Siendo z el vector de entrada a la función, el índice j la posición del vector de entrada, z_j el valor del vector de entrada en la posición j y K el

número de dimensiones del vector.

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.5}$$

2.4.4. Función de pérdida

La función de pérdida o de coste, es aquella que nos indica "lo bien"que lo está haciendo nuestra red neuronal con respecto a los datos de entrenamiento y la salida esperada. Esta función tiene como parámetros la predicción de la red y el resultado real, con ello calcula un error que se utilizará para optimizar nuestro modelo. Existen muchos tipos de funciones de coste e incluso se pueden crear funciones personalizadas para ajustarnos mejor al problema que se quiere resolver. Las funciones de coste más típicas son la *binary cross entropy* (ecuación 2.6) o la *Mean Squared ºError* (MSE, ecuación 2.7). En ambas ecuaciones y es el resultado real, p es la predicción de nuestro modelo, L es el error y N el tamaño del *batch*.

$$L = -\frac{1}{N}\sum_{i=1}^{N} y \log(p) + (1-y)\log(1-p) \tag{2.6}$$

$$L = \frac{1}{N}\sum_{i=1}^{N}(p-y)^2 \tag{2.7}$$

2.4.5. Optimizadores

El objetivo de entrenar un modelo es minimizar las funciones de coste que se han presentado en el apartado anterior. Para lograr este objetivo, se utilizan algoritmos de optimización, los cuales actúan de manera iterativa en busca de estos valores óptimos. Esta optimización se hace con la ayuda de un método llamado *backpropagation*, que calcula la derivada de los parámetros desde la salida hasta la entrada. Este método comienza su cálculo con el error reportado por la función de coste. Algunos optimizadores son:

- ***Stochastic Gradient Descent* (SGD)**: El cálculo del nuevo valor de los parámetros se realiza con la derivada de los parámetros calculada por el *backpropagation* y es proporcional al *learning rate* (un valor escalar). Siempre se avanza en la dirección de mayor pendiente.
- ***Adaptive Gradient Algorithm* (AdaGrad)**: Es un optimizador adaptativo, en el que el nuevo valor ya no dependerá únicamente de una iteración, sino que tendrá en cuenta las anteriores. El objetivo es mejorar el SGD modificando la dirección de descenso, ya que la mayor pendiente no siempre apuntará directamente al mínimo global. Aunque tiene el innconveniente de que la tasa de aprendizaje es cada vez menor.

– ***Root Mean Squeare Propagation*** **(RMSprop)**: Utiliza una media móvil de los gradientes al cuadrado para normalizar el gradiente. Esta normalización provoca un aumento del paso (impulso) en los gradientes pequeños para evitar que desaparezcan. Aunque puede aumentar demasiado el gradiente, dando problemas al optimizar.

– ***Adaptive moment estimation*** **(Adam)**: Es una combinación del AdaGrad y el RMSprop, siendo este el mejor en la mayoría de situaciones [19]. Los parámetros de este optimizador son el *learning rate*, β_1 , β_2 y ϵ.

El optimizador Adam es el que se utiliza en este trabajo. En las ecuaciones 2.8 y 2.9, se puede ver la influencia de los parámetros β_1 y β_2 para la optimización. Siendo g_t los gradientes de los parámetros en la iteración t. Estas ecuaciones representan dos medias móviles una con g_t y la otra con g_t^2.

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{2.8}$$

$$v_t = \beta_2 \cdot m_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{2.9}$$

Posteriormente, se hace una corrección de ambas medias móviles como se puede ver en las ecuaciones 2.10 y 2.11.

$$\hat{m_t} = \frac{m_t}{(1 - \beta_1^t)} \tag{2.10}$$

$$\hat{v_t} = \frac{v_t}{(1 - \beta_2^t)} \tag{2.11}$$

Por último, las medias móviles corregidas junto con el *learning rate* (α) se utilizan en la ecuación 2.12 para calcular el nuevo valor de los parámetros (θ) en la siguiente iteración. ϵ es un valor que se utiliza para evitar el 0 en el denominador.

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{2.12}$$

2.4.6. Métricas

La evaluación de un modelo propuesto para resolver una tarea en concreto se realiza a través de métricas. Las métricas de este trabajo son las que corresponden a una tarea de clasificación, en nuestro caso, interesa saber el nivel de acierto del modelo propuesto cuando predice el valor y el instante temporal de una nota musical. Las métricas se han hecho a través de la librería mir_eval [20].

Las métricas se calculan a partir de los positivos verdaderos (TP), negativos verdaderos (TN), positivos falsos (FP) y negativos falsos (FN).

- ***Precision***: Es el porcentaje de veces que el modelo ha predicho una nota y ha acertado, frente a las notas que ha predicho y ha fallado. En la ecuación 2.13, se puede ver la expresión utilizada para calcular este parámetro.

$$Precision = \frac{TP}{TP + FP} \tag{2.13}$$

- ***Recall***: Es el porcentaje de veces que el modelo ha predicho una nota y ha acertado, frente al número de notas que existen y no ha predicho. En la ecuación 2.14, se ve el cálculo necesario para obtener esta métrica.

$$Recall = \frac{TP}{TP + FN} \tag{2.14}$$

- ***F-score***: Es una relación entre las dos, es la métrica de la cual nos vamos a fijar más. Se puede ver su expresión en la ecuación 2.15.

$$L = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.15}$$

- ***Accuracy***: Mide el porcentaje de casos en los que el modelo ha acertado, tanto cuando acierta en las notas, como cuando predice que no hay nota y no la hay, como se ve en la ecuación 2.16.

$$Recall = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.16}$$

- ***Chroma Accuracy***: Es una métrica un poco especial que se aplica únicamente en la música, donde antes de calcular la *acurracy* con la ecuación 2.16 se mapean todas las notas a una misma octava.

2.4.7. Conceptos relacionados con el entrenamiento

Hay algunos conceptos que se utilizan en el entrenamiento que son necesarios nombrar para seguir mejor este trabajo:

- ***Epoch***: Es el hiperparámetro que define el número de veces que el algoritmo ha recorrido el conjunto de entrenamiento.

- ***Batch size***: La cantidad de muestras de nuestro conjunto de entrenamiento que se utiliza en una iteración. Los parámetros de una red se actualizan en cada iteración.

– ***Batch normalization***: Se utiliza con el objetivo de que la red converja en un mínimo más rápidamente en el mínimo global más rápidamente.

– ***Dropout***: Se utiliza para evitar el *overfitting* durante el entrenamiento, y consiste en congelar de manera aleatoria un cierto número de neuronas en cada *batch size*.

A continuación se van a describir dos de las redes neuronales que se han utilizado en este trabajo. Una descripción completa de todas las RNN que se emplean hoy en día está fuera de la limitación de espacio de este trabajo.

2.5. Redes neuronales convolucionales (CNN)

Las redes neuronales convolucionales [21] son conocidas sobre todo por su aplicación en imágenes, aunque en este trabajo las aplicaremos en señales de audio. En señales de audio, las imágenes corresponden a representaciones de tiempo y frecuencia como la transformada de Q constante (CQT) [5] o el *Mel_Spetrogram* [22]. Estas redes se componen de tres etapas, una etapa de convolución, otra de pooling y otra de activación.

Estas redes se basan principalmente en el uso de capas convolucionales que aplican filtros en imágenes con el objetivo de obtener una serie de características de ellas.

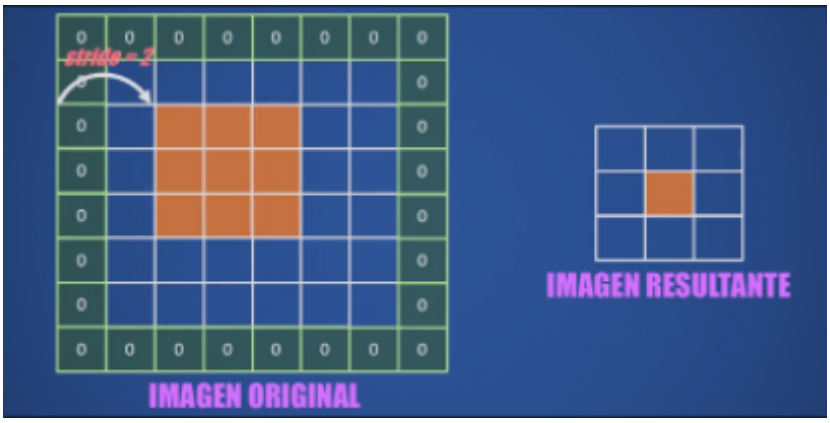

Figura 2.7: Funcionamiento de una convolución

Una convolución consiste en la multiplicación de un filtro por una parte de la imagen de entrada sobre la que actúa el filtro y tiene diferentes parámetros que se pueden modificar (figura 2.7). Estos parámetros son los siguientes :

– **Tamaño del filtro**: Son las dimensiones que va a tener el filtro que se va a aplicar en toda la imagen, en la figura el tamaño es de 3x3.

– ***Stride***: El *stride* es el desplazamiento en píxeles que hace el filtro en la imágenes de entrada. En el caso anterior es de 2.

– ***Padding***: Se utiliza para mantener las dimensiones de la imagen de entrada, en el caso anterior hay un padding de 1. Se añaden ceros alrededor de la matriz de entrada.

– **Número de filtros**: En una capa convolucional se pueden utilizar varios filtros.

Además, de la etapa convolucional este tipo de redes se caracterizan tambien por utilizar etapas de *pooling*, en las que se reduce las dimensiones de la etapa de entrada con el objetivo de extraer la información más representativa de la entrada. Un ejemplo de *pooling*, es el *Max-pooling* donde se extrae el valor máximo que se encuentra dentro de un filtro, como se puede ver en la figura 2.8.

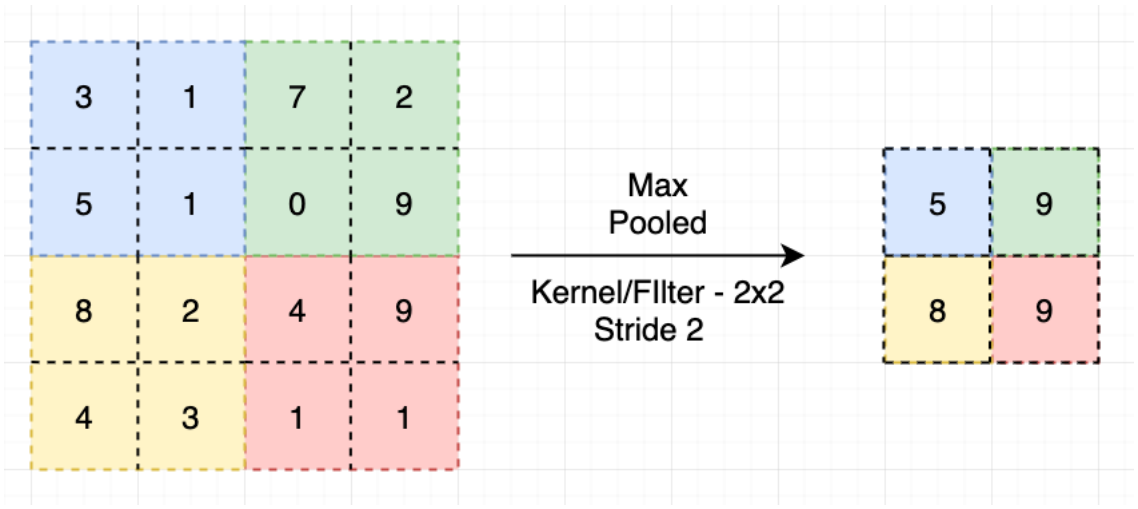

Figura 2.8: Etapa de *pooling* [23]

Al igual que las redes neuronales artificiales, se utiliza a la salida una etapa no-lineal mediante funciones de activación. La función ReLU, que ya hemos nombrado anteriormente es la más común en este tipo de redes.

2.6. Long Short Term Memory (LSTM)

Las redes LSTM son un tipo de redes recurrentes que tienen memoria para aprender conceptos a corto y largo plazo. Este tipo de redes se propusieron por primera vez en 1997 por Hochreiter y Schmidhuber [24] con el objetivo de recordar la información durante más tiempo.

Estas redes se estructuran en una cadena en el que hay un módulo de repetición, como se puede ver en la figura 2.9.

Este módulo se compone de tres capas que se comunican con una celda de estado (figura 2.10). Las tres capas de las que se compone son: una capa que olvida información (*forget gate*), otra que almacena nueva información (*input gate*) y una capa de salida (*output gate*).

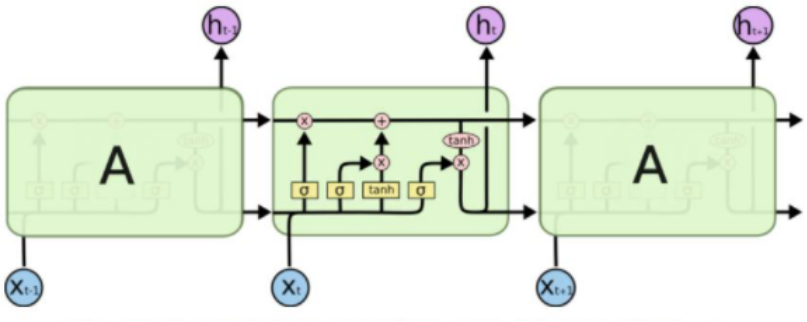

Figura 2.9: Cadena de módulos LSTM [25]

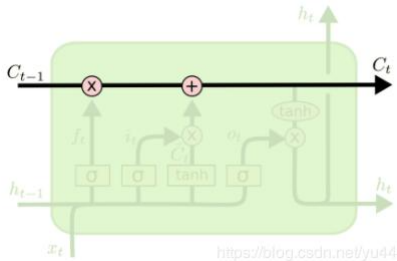

Figura 2.10: Celda estado de una LSTM [25]

En primer lugar, la capa *forget gate* (figura 2.11) se compone de una red que tiene como función de activación una función sigmoidea. La salida de la red dará unos valores entre 0 y 1, siendo los valores más cercanos a 1 los que mantendrá en la celda de estado.

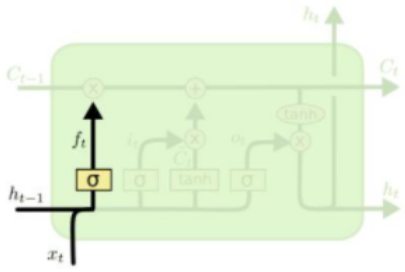

Figura 2.11: *Forget gate* de una LSTM [25]

La segunda capa es la que aportará nueva información a la celda de estado. Se compone de dos partes, en primer lugar hay una capa con una sigmoide que determina los valores que se actualizarán. En segundo lugar, hay otra capa con una tangente hiperbólica que determinará que candidatos de los nuevos valores se agregarán al nuevo estado. Ambas partes se combinan para actualizar la celda de estado, como se puede ver en la figura 2.12.

Finalmente, nos encontramos con la *output gate* que reportará el valor de la salida del módulo. En primer lugar, se ejecuta otra función sigmoidea que determina las partes la de celda que se quieren generar. Después, a la información que hay en la celda de estado se le aplica una tangente hiperbólica. Combinando ambas da la salida como se ve en la figura 2.13.

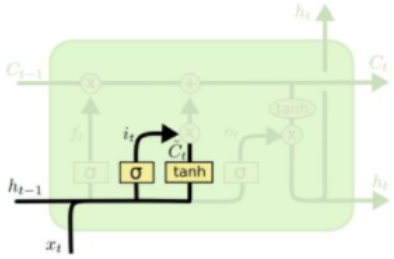

Figura 2.12: *Input gate* de una LSTM [25]

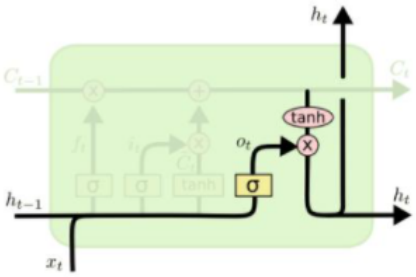

Figura 2.13: *Output gate* de una LSTM [25]

2.7. Musical Instrument Digital Interface (MIDI)

Las siglas de MIDI corresponden a *Musical Instrument Digital Interface* y es un lenguaje que permite a ordenadores, sintetizadores e instrumentos musicales comunicarse entre sí.

Este lenguaje se desarrolló en 1981 de la mano de Ikaturo Kakehashi aunque el estándar MIDI no se publicó hasta 1982. Este lenguaje se compone de instrucciones que se envían al un instrumento musical digital (sintetizador, ordenador, etc.) y hace sonar una nota (también llamado evento). Las principales instrucciones que se envían son las siguientes:

- ***Onset* y *Offset***: Es el inicio y el fin de una nota.
- ***Pitch***: Es el tono que tiene la nota musical.
- **Velocidad**: Es la fuerza con la que se pulsa la tecla.
- **Tempo**: Se mide en BPM (*Beat por Minute* o golpes por minuto) y establece la rapidez que tiene la melodía. Por ejemplo, en una melodía con 60 golpes por minuto, el tiempo de una negra equivaldrá de un segundo. Por el contrario, en una melodía de 120 BPM, la negra valdrá 0,5 segundos.

Los datos en formato MIDI los utilizaremos para realizar una representación de la música en un formato *pianoroll* que será útil para el entrenamiento de nuestra red. Un *pianoroll* es una matriz en la que en el eje Y se encuentran las notas musicales, y en el eje X el tiempo.

La nota que está sonando tiene un valor equivalente a la velocidad y las notas que no están sonando su valor es 0. En la figura 2.14 se puede ver un ejemplo de *pianoroll*.

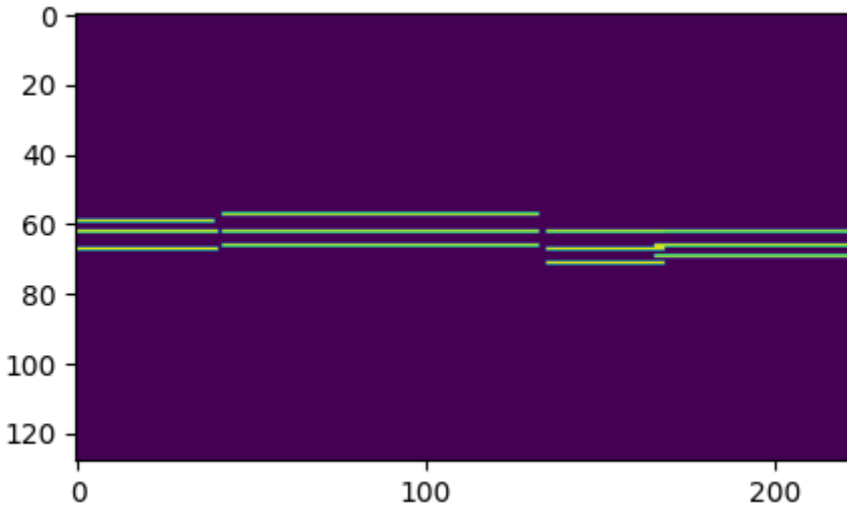

Figura 2.14: Ejemplo de un *pianoroll* utilizado para entrenar el modelo neuronal propuesto en este trabajo

2.8. Transformada de Q constante (CQT)

La *Constant-Q Transform* (CQT) o transformada de Q constante [5] es una representación tiempo-frecuencia de una señal en la que se utiliza un banco de filtros equiespaciados en escala logarítmica. La representación logarítmica en frecuencia resulta muy adecuada en la descripción de una señal musical ya que las posiciones relativas de los armónicos serán constantes. Como los armónicos de una señal periódica son f, 2f, 3f, ... la distancia relativa dibujada entre el primer y segundo armónico en escala logarítmica es $\log 2$, entre el segundo y el tercero, $\log(2/3)$, y así sucesivamente.

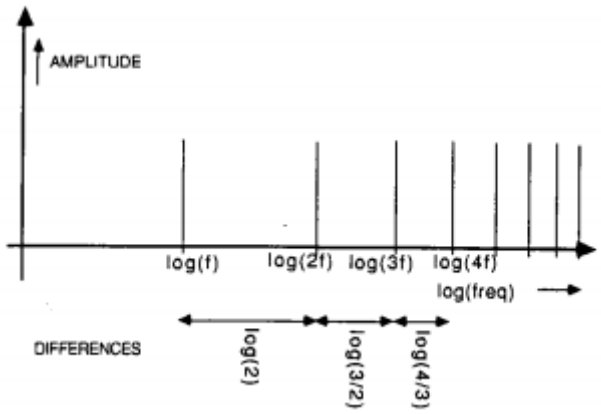

Figura 2.15: Transformada de Fourier representada en relación con el logaritmo de la frecuencia [5]

Esta representación igualitaria para cualquier frecuencia fundamental (F0), nos ayuda a resolver mejor los problemas de identificación de instrumentos o de detección de la F0. Con la transformada de Fourier convencional al ser estas distancias lineales con la frecuencia provoca que estas tareas sean más difíciles de realizar.

Uno de los problemas que aparecen a la hora de hacer la transformada corta de Fourier (STFT) en un dominio del logaritmo de las frecuencias es que para las bajas frecuencias hay muy poca información y en altas frecuencias hay demasiada. Si, por ejemplo, se toma una ventana fija de 1024 muestras y suponemos que la frecuencia de muestreo es de 32000 muestras por segundo, la resolución es de 32000/1024 = 31,3Hz. Para el caso de la escala baja del violín que es de 196Hz, este valor supone un 16 % de la frecuencia (mucho mayor que el 6 %, que es la distancia que hay entre dos notas adyacentes, por lo que esta resolución frecuencial no sería suficiente). En el caso contrario, en el rango alto de un piano que sería de 4186Hz, esta resolución frecuencial de 31,3Hz supone un 0,7 % y, por tanto, sería demasiado grande.

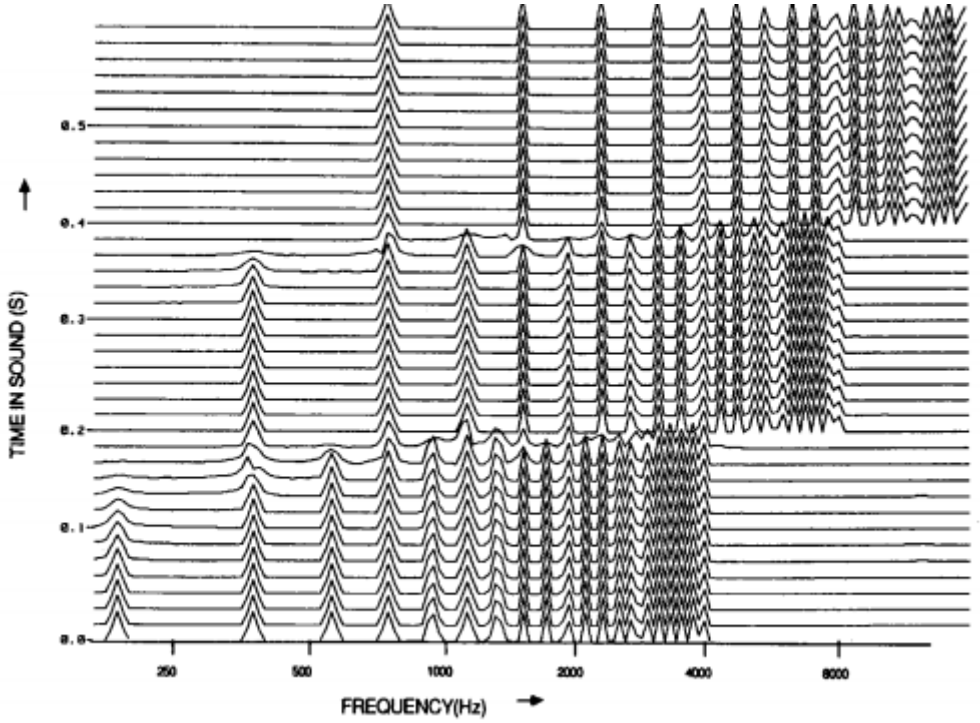

Figura 2.16: CQT de G3 (196Hz), G4 (392Hz) y G5 (784Hz) [5]

Para que la resolución sea suficiente, y se pueda diferenciar correctamente un semitono, la resolución en esas frecuencias tendrá que suponer aproximadamente un 3 % de la frecuencia de la nota que se esté analizando. Este valor corresponde a la mitad de 6 %, $2^{\frac{1}{12}} - 1 \approx 0{,}06$. El ratio que relaciona la frecuencia con la resolución se llama Q (factor de calidad) y para el ejemplo anterior valdría f/0.03f = 34. Este ratio permanecerá constante durante el calculo.

De este modo, es necesario modificar el tamaño de la ventana en relación a la frecuencia para que se mantenga este factor de calidad, o lo que es lo mismo, que se calcule la frecuencia con el mismo número de periodos para cualquier valor de nota. La ecuación 2.17, permite calcular el tamaño de la ventana N[k], donde f_s es la frecuencia de muestreo, Q es el factor de calidad (34 es el ideal para diferenciar correctamente semitonos) y f_k es la frecuencia que queremos detectar.

$$N[k] = \frac{f_s}{f_k} Q \tag{2.17}$$

Así, para el calculo de la CQT, en primer lugar se calcula la *Fast Fourier Transform* (FFT) y a partir de ella se realizan convoluciones con filtros paso banda cuya frecuencia central se corresponde con la frecuencia que queremos calcular y que tendrán un tamaño de ventana calculado con la ecuación 2.17. El número de filtros que se va a utilizar dependerá del número de *bins* totales que se quieran obtener, es decir el número de octavas que se quiera procesar y el número de divisiones frecuenciales por octava. En nuestro caso queremos calcular 6 octavas y cada una se dividirá 60 *bins*, por lo que tendremos 360 filtros.

Además, el cálculo de CQTs se hace con una distancia de salto temporal de la señal de audio (*hop lenght*). El resultado será un valor por cada una de las frecuencias y por cada ventana (*frame*). En la figura 2.17, se pueden ver visualmente estos conceptos.

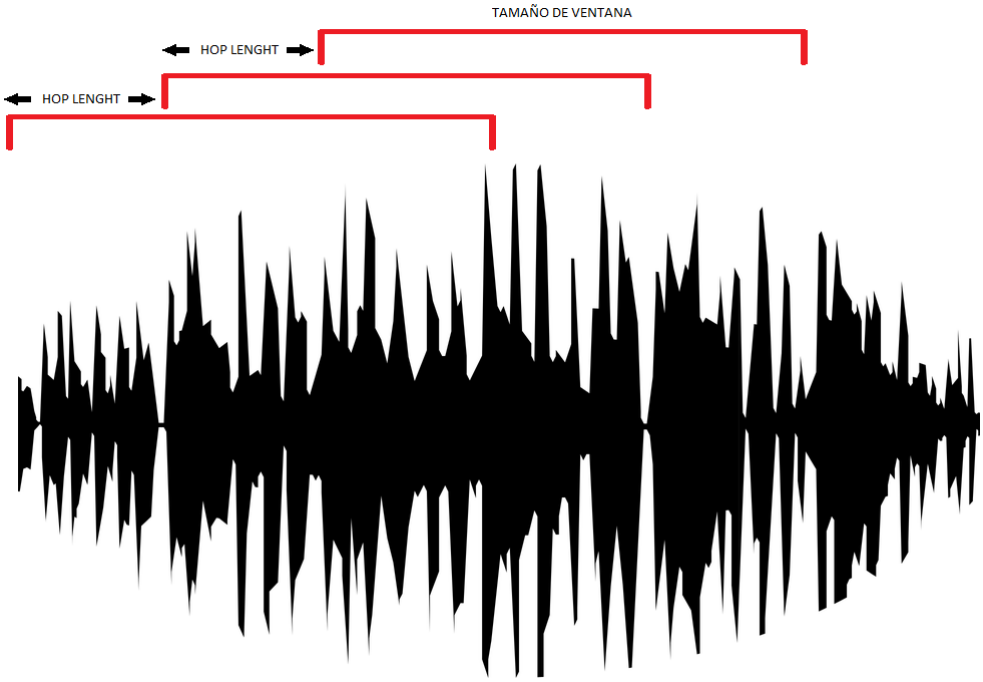

Figura 2.17: Relación entre el tamaño de la ventana y el *hop lenght*

En este trabajo se ha utilizado la librería de Python librosa [26] para hacer el cálculo de la CQT.

Capítulo 3

Estado del Arte

En este capítulo se van a repasar las aportaciones fundamentales que soportan el desarrollo de este trabajo.

Para realizar la transcripción automática de música (AMT), existen diferentes soluciones que utilizan métodos probabilísticos como pYIN [27], o redes convolucionales como las empleadas en Onsets&Frames (OaF) [28], CREPE [29], SPICE [30] o Deep Salience [31]. Esta última solución es en la que está basado este trabajo.

3.1. Estimación de la frecuencia fundamental (F0)

Existen bastantes métodos de detección de la frecuencia fundamental, basados tanto en métodos estadísticos como en aprendizaje automático (en nuestro caso Deep Learning). El principal problema que presentan estos métodos es que sólo permiten la extracción de una nota en un *frame*, con lo que sólo son útiles para detectar notas en melodías monofónicas.

3.1.1. Métodos tradicionales

pYIN

El pYIN desarrollado por Mauch y Dixon [27] es una algoritmo probabilístico para estimar frecuencias fundamentales basado en el algoritmo YIN [32], el cual tiene algunos problemas asociados. El principal inconveniente que presenta es que hace una única estimación por cada *frame*, lo que genera un resultado que cambia de manera abrupta. Para solucionar este problema y generar una estimación más suavizada se propuso una modificación en la que se disponen múltiples salidas con varios candidatos asociados a una probabilidad. Estas probabilidades se utilizan en un modelo oculto de Markov (HMM), que decodificado por un decodificador Viterbi es el que produce la estimacion del pitch, es decir, la nota musical. Posteriormente, se realizará un procesado de los resultados obteniendo una información del tono (F0) más suavizada que la estimación del algoritmo YIN.

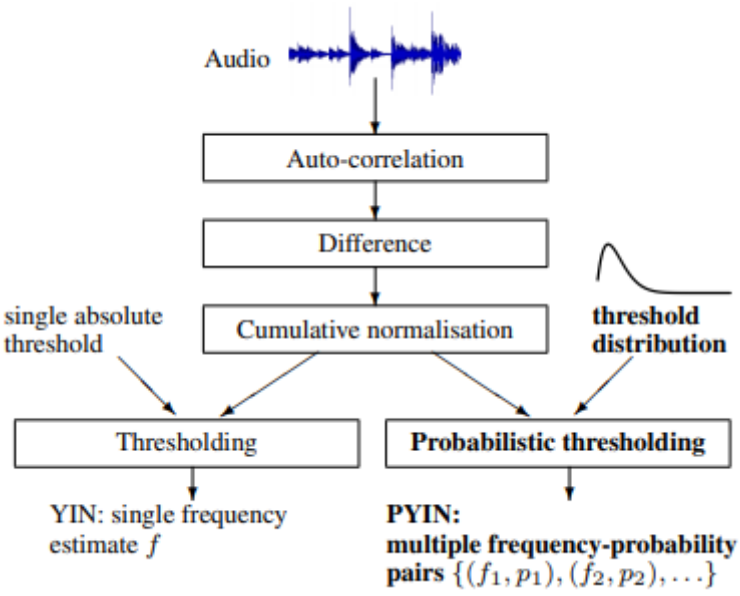

Figura 3.1: Comparación entre la estructura de YIN y pYIN [27]

Este método se divide en dos etapas: la extracción de varios candidatos por *frame* y el seguimiento (*tracking*) de estos candidatos para seleccionar únicamente una frecuencia.

Para extraer varios candidatos pYIN se basa en el mismo concepto que YIN. Si una señal x_i, $i = 1, ..., W$, siendo W el tamaño de la ventana a analizar, es prácticamente periódica, la diferencia de la ecuación 3.1 será muy pequeña siendo $\tau = \frac{1}{F_0}$ el periodo.

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2 \tag{3.1}$$

Esta función se puede calcular mediante la función de correlación:

$$r_t(\tau) = \sum_{j=t+1}^{t+W} x_j x_{j+\tau} \tag{3.2}$$

Con la ecuación 3.1 y 3.2, se obtiene que:

$$d_t(\tau) = r_t(0) + r_{t+\tau}(0) - 2r_t(\tau) \tag{3.3}$$

Una vez calculada la correlación y la diferencia, el algoritmo YIN normaliza la diferencia y se obtiene la función media de las diferencias acumuladas, $d'(\tau)$. La normalización utiliza métodos heurísticos para compensar los valores bajos que aparecen en frecuencias altas provocados por las resonancias de los formantes (o frecuencias predominantes en un sonido afinado).

El último paso que se realiza en el algoritmo de YIN, es la detección del periodo más pequeño en el que $d'(\tau)$ tenga un mínimo local menor a un cierto umbral (s, normalmente s =

0.1 o s = 0.15). En el caso de no exista, se propone el mínimo absoluto de $d'(\tau)$, $argmin_\tau d'(\tau)$. La salida del algoritmo de YIN se anota como Y(x_t, s). En pYIN se propone no definir un umbral fijo y en su lugar utilizar una distribución $P(s_i)$, donde s_i son los umbrales posibles.

Con esta distribución y la probabilidad previa de utilizar la estrategia de mínimo absoluto, podemos definir la probabilidad de que τ sea el periodo fundamental (τ_0) como:

$$P(\tau = \tau_0 | S, x_t) = \sum_{i=1}^{N} a(s_i, \tau) P(s_i) [Y(x_t, s_i) = \tau] \tag{3.4}$$

donde,

$$a(s_i, \tau) = \begin{cases} 1, & d'(\tau) < s_i \\ p_a, & d'(\tau) \geq s_i \end{cases} \tag{3.5}$$

siendo $p_a = 0.01$.

La siguiente etapa se encarga de seleccionar únicamente una frecuencia. Para ello se divide un espacio de 4 octavas (de los 55Hz a los 880Hz) en 480 bins frecuenciales. Los bins se modelan como estados ocultos del modelo de Markov y con las probabilidades de los candidatos ya obtenidas previamente se les asigna la frecuencia más cercana.

Se ha hecho comparaciones de este modelo con el YIN original, y una modificación del modelo original más suavizado (YIN + S) utilizando las métricas de *Recall*, *Precision* y *F-score*.

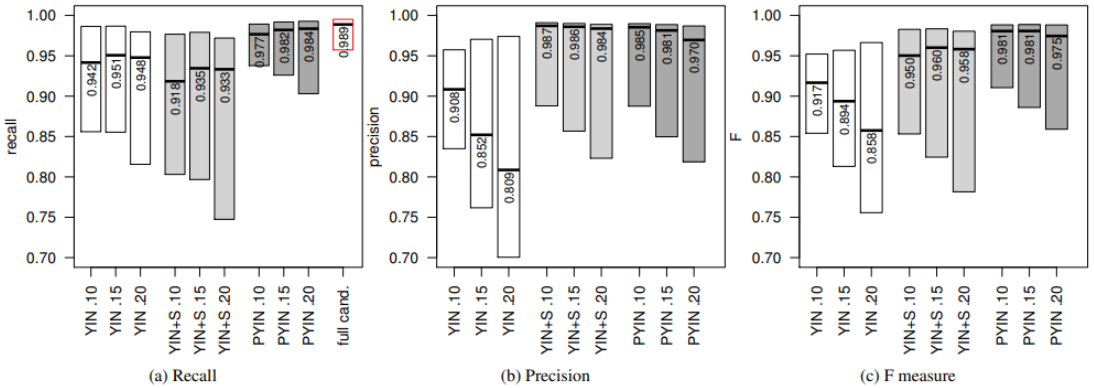

Figura 3.2: Comparación de las métricas de Recall, Precision y F-score entre YIN y pYIN , con una s = 0.1, 0.15, 0.2 [27].

Como se puede observar en la figura 3.2 el F-score del pYIN es mejor que los otros casos. El método YIN+S tiene mejor precisión, pero como se observa en los datos de Recall no reconoce tantos *frames* con notas. El F-score nos reporta una información más global ya que se calcula conjuntamente con la Precision y el Recall.

3.1.2. Métodos basados en Deep Learning

CREPE

El objetivo que persigue el algoritmo CREPE [29] es el de conseguir un método para detectar notas en melodías monofónicas (no tiene notas tocadas de manera simultánea), superando los resultados que se obtienen con enfoques heurísticos como pYIN. En este caso, se consigue una precisión del 90 % con un umbral muy pequeño (10 cents). En la ecuación 3.6 se puede ver la definición de un *cent* que es una unidad que representa los intervalos relativos a un tono de referencia f_{ref} en Hz, y se define en función de la frecuencia en hercios.

$$c(f) = 1200 \cdot \log_2 \frac{f}{f_{ref}} \tag{3.6}$$

La arquitectura de CREPE es una red convolucional que opera directamente en el dominio del tiempo de una señal de audio, con el objetivo de conseguir una estimación de la nota que produce en cada instante de tiempo (*frame*). La entrada de la red son 1024 muestras, con un frecuencia de muestreo de 16kHz. La arquitectura de la red es la presentada en la figura 3.3.

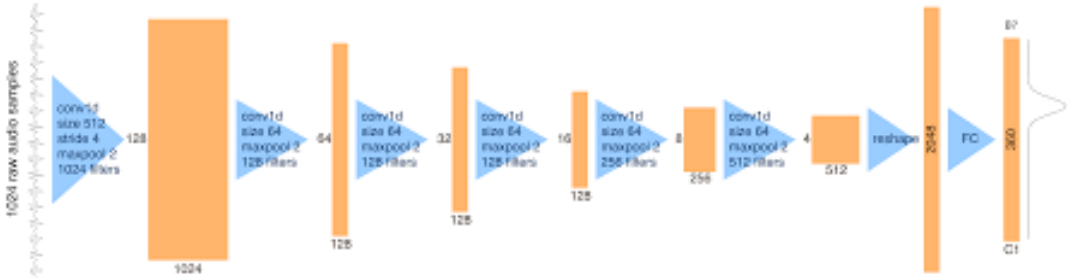

Figura 3.3: Arquitectura de CREPE

Como se puede ver en la figura 3.3, la salida de la red son 360 neuronas que representan las notas de C1 a B7, es decir, de la frecuencia 32,70Hz a la 1975,5 Hz.

Esta red ha sido entrenada con dos *datasets*, el primero se llama RWC-synth que contiene 6.16 horas de audio sintetizado. El sintetizado se ha hecho usando una suma fija de un pequeño número de sinusoides, con el que se consigue un conjunto de datos muy homogéneo en timbre.

El segundo son 230 pistas monofónicas que pertenecen al dataset MedleyDB, pero sintetizada de tal manera que la anotación de F0 mantiene el timbre y la dinámica de la pista original. Los detalles se pueden encontrar en [33]. Este *dataset* lo han denominado MDB-stem-synth y en total son 15.56 horas con 25 instrumentos diferentes.

Para realizar el entrenamiento de los modelos se ha dividido el dataset en tres conjuntos: entrenamiento, validación y *test*, con una división proporcional de 60/20/20, respectivamente. Además, se intenta evitar utilizar el mismo artista para el entrenamiento y el test, para que no aumente la precisión de manera artificial.

Para la evaluación se utiliza la librería mir_eval [20], y las métricas que se utilizan son las de *raw pitch accuracy* (RPA), que corresponde a la métrica de *accuracy*, y *chroma accuracy* (RCA) (Ambas métricas explicadas en la sección 2.4.6) con 50 cents de *threshold* (umbral) [34]. El objetivo es hacer una comparación con los métodos heurísticos pYIN. En este trabajo se realizó comparación con pYIN y SWIPE. SWIPE es un estimador del tono que se inspira en la forma de onda de diente de sierra cuya descripción puede verse en [35].

Dataset	Metric	CREPE	pYIN	SWIPE
RWC-synth	RPA	**0.999±0.002**	0.990±0.006	0.963±0.023
	RCA	**0.999±0.002**	0.990±0.006	0.966±0.020
MDB-stem-synth	RPA	**0.967±0.091**	0.919±0.129	0.925±0.116
	RCA	**0.970±0.084**	0.936±0.092	0.936±0.100

Figura 3.4: Comparación entre CREPE, pYIN y SWIPE, de las medias de RPA y RCA con un *threshold* de 50 cents en los datasets de MDB y RWC [29].

Dataset	Threshold	CREPE	pYIN	SWIPE
RWC-synth	50 cents	**0.999±0.002**	0.990±0.006	0.963±0.023
	25 cents	**0.999±0.003**	0.972±0.012	0.949±0.026
	10 cents	**0.995±0.004**	0.908±0.032	0.833±0.055
MDB-stem-synth	50 cents	**0.967±0.091**	0.919±0.129	0.925±0.116
	25 cents	**0.953±0.103**	0.890±0.134	0.897±0.127
	10 cents	**0.909±0.126**	0.826±0.150	0.816±0.165

Figura 3.5: Comparación entre CREPE, pYIN y SWIPE, de las medias de RPA variando el *threshold* entre 50, 25 y 10 en los *datasets* de MDB y RWC [29].

SPICE

Al igual que CREPE, SPICE [30] es un detector de notas en melodías monofónicas que permite estimar la frecuencia fundamental. Este modelo parte de la idea de que el oído humano diferencia mejor el intervalo de frecuencia entre dos notas, que el valor absoluto de una nota en concreto. La red tiene como entradas dos CQTs. La primera entrada a la red es la CQT de una señal de audio y la segunda entrada es la CQT anterior pero haciendo un desplazamiento en el tono como se puede ver en la figura 3.6. Este desplazamiento en el tono es un parámetro que se puede controlar a la entrada del algoritmo.

La red se compone de dos encoders que se alimentan con las dos CQTs que devolverá cada uno un valor escalar. Un encoder se compone de varias capas convolucionales y *poolings* que comprimen la entrada reduciendo la dimensionalidad de la misma. Posteriormente, con la ayuda de una función de pérdida se fuerza a que la diferencia de las salidas de los dos encoders sea proporcional al desplazamiento del tono que hay entre los dos CQT.

Para calcular el tono absoluto de la nota que está sonando, se utiliza únicamente la salida del primer enconder. Este encoder es alimentado con melodías con un tono conocido para poder estimar la relación que hay entre la salida y el tono real. Cuando se obtiene esta relación, ya se puede saber el tono de manera absoluta.

Otro característica de este modelo es que esta red puede determinar si una señal musical es con voz o sin voz mediante una capa neuronal totalmente conectada (*fully connected*) que recibe la penúltima capa del encoder.

Como se puede ver, esta red se diferencia de otras porque al entrenarla no se necesita un *dataset* etiquetado ya que se hace mediante la diferencia entre la onda de sonido y la misma onda de sonido modificada.

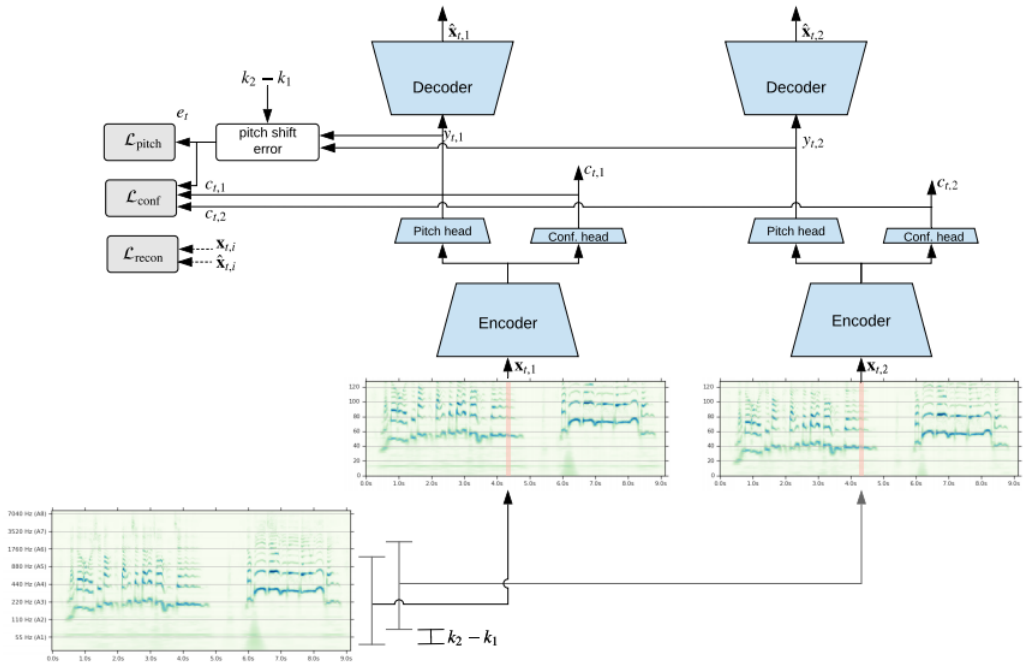

Figura 3.6: Arquitectura de SPICE [30]

Los experimentos realizados en Tensorflow se han hecho con una frecuencia de muestreo de 16 kHz. Para realizar la CQT se hacen 24 bins por octava (cada octava se ha dividido en 24 frecuencias) y un *hop-lenght* de 512, lo que da lugar que cada frame de la CQT sea de 32ms.

Las entradas de los encoder son de 128 dimensiones que corresponden a las frecuencias de la CQT y la salida son dos escalares: el escalar para detectar el tono (*pitch*) y la confianza que es para detectar si una pista tiene voz o no.

El enconder se compone de 6 capas convolucionales con filtros de 3x3 y un *stride* de 1, el numero de filtros es de 64, 128, 256, 512, 512 y 512 (el decoder tiene la misma estructura pero con la mitad de filtros). Entre las convoluciones hay una capa de *batch normalization* y

una ReLU, así como un *Max-pooling* de tamaño 3x3 con un *stride* de 2.

La salida de los encoders van a dos redes: la que estima la nota son dos *fully connected* de 48 y 1 neurona, y la que estima la confianza es una *fully connected* de 1 unidad. El modelo se entrena con el optimizador Adam, un *learning rate* de 0.0001 y un *batch size* de 64.

Los *datasets* empleados para entrenar esta red son el MIR-1k que contiene 1000 pistas de audio de canciones de pop chino y el MDB-stem-synth que musica resintetizada monofónica con varios instrumentos.

Para hacer la evaluación, se utiliza la RPA (raw pitch accuracy) con un umbral de desviación de 0.5 semitonos. Algunos de los resultados, comparados con CREPE fueron los siguientes:

			MIR-1k		*MDB-stem-synth*
Model	# params	Trained on	RPA (CI 95%)	VRR	RPA (CI 95%)
SWIPE	-	-	86.6%	-	90.7%
CREPE tiny	487k	many	90.7%	88.9%	93.1%
CREPE full	22.2M	many	90.1%	84.6%	92.7%
SPICE	2.38M	*SingingVoices*	$90.6\% \pm 0.1\%$	86.8%	$89.1\% \pm 0.4\%$
SPICE	180k	*SingingVoices*	$90.4\% \pm 0.1\%$	90.5%	$87.9\% \pm 0.9\%$

Figura 3.7: Comparación entre CREPE , SPICE Y SWIPE, con la métrica RPA en los datasets MIR-1k y MDB-stem-synth [30]

3.1.3. Estimación de la Multi-F0

A diferencia de la estimación de la frecuencia fundamental, al estimar múltiples frecuencias fundamentales (multi-F0) es posible detectar en un mismo *frame* varias notas que están sonando simultáneamente. Es una tarea realmente difícil, ya que como hemos dicho anteriormente, los armónicos de las notas se pueden solapar lo que hace muy complicada la extracción de las mismas.

Deep Salience

El objetivo de Deep Salience [31] es estimar las notas en musica polifónica, aumentando la dificultad con respecto a los modelos nombrados anteriormente.

Para este modelo se ha utilizado un *dataset* de 240 canciones. De las cuales 108 corresponden al *dataset* MedleyDB [36] que contiene canciones de diferentes estilos musicales. Las 132 canciones restantes corresponde a música pop occidental desde los años 80 hasta la actualidad.

Las señales de audio se procesan mediante una CQT implementada con librosa [26], con una normalización de los valores para que se encuentren en un intervalo [0,1]. En el cálculo

de la CQT, como ya se ha comentado anteriormente, se hace una división frecuencial en bins. La frecuencia de cada bin (f_k), nos lo da la expresión $f_k = f_{min} \cdot 2^{k/B}$.

La entrada de la red no será una única CQT, sino 6 armónicos apilados. La frecuencia de los armónicos (h) de cada bin corresponden a $h \cdot f_k$ y, por consiguiente, tendrá la expresión $f_k = h \cdot f_{min} \cdot 2^{k/B}$. Se tendrá un armónico (h) que será el fundamental (h = 1), 4 armónicos por encima del fundamental (h = 2,3,4,5) y un subarmónico (h = 0.5). El tiempo de cada *frame* de la CQT será de $\approx 11ms$ y se utilizarán 60 bins por octava en 6 octavas. Para el primer armónico la frecuencia mínima es de 32,7 Hz (C1) y en cada armónico irá variando.

La salida de la red corresponderá a una representación tiempo-frecuencia del mismo tamaño que cada uno de los armónicos de entrada (figura 3.8). El intervalo de frecuencias será el mismo que tenía el primer armónico de entrada. Para hacer el *ground truth* se ponen a 1 los valores más cercanos a la nota en una representación tiempo-frecuencia.

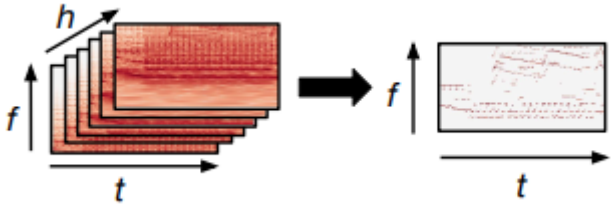

Figura 3.8: Input y Output de Deep Salience[31]

La arquitectura del modelo presenta 5 capas convolucionales (ver figura 3.9). Las dos primeras capas tienen 128 y 64 filtros de 5x5, con el objetivo de cubrir 1 semitono. Las siguientes tienen 64 filtros de 3x3 y la ultima capa son 64 filtros de 70x3 que abarcan 14 semitonos (2 semitonos más que los de una octava).

En toda la red se mantiene el mismo tamaño de frecuencia y tiempo (360, 50) para evitar eliminar los pequeños cambios que puede haber en la frecuencia. Como se puede ver en la figura 3.9.

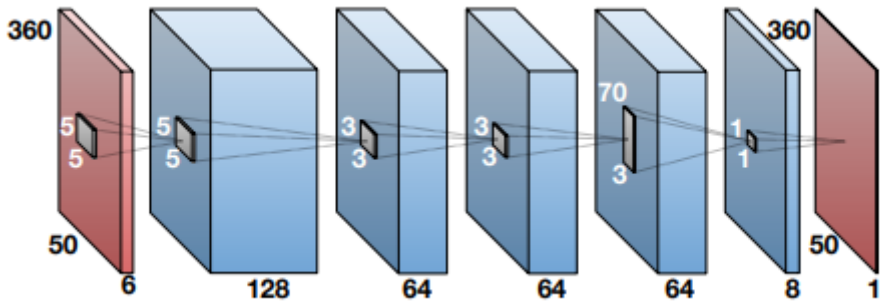

Figura 3.9: Arquitectura de Deep Salience[31]

El modelo utiliza el optimizador Adam y la función para minimizar la *cross entropy*, que

se puede ver la ecuación 3.7.

$$L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log (1 - \hat{y}) \tag{3.7}$$

donde y es es la salida real , $\hat{y}$ es la predicción de la red

Para el entrenamiento, se utiliza un *dataset* a partir de anotaciones hechas por máquinas y humanos, en los que se anotan las frecuencias fundamentales. El *dataset* consiste en 240 canciones sacadas a partir de combinar el MedleyDB (108) y 132 canciones de música pop de los 80. Las 240 se dividen en tres conjuntos (entrenamiento, validación y test).

Para un posterior análisis se utilizan dos conjuntos de datos más, el Bach10 [37] y el Su [38]. El Bach10 que contiene 10 grabaciones de 30 segundos de un cuarteto de Bach y las pistas separadas por instrumento; y el Su que se compone de 10 extractos de grabaciones clásicas de solos de piano, quintetos de piano y sonatas de violín.

Para obtener los resultados es necesario seleccionar los valores de frecuencia más importantes, por lo que se sacan todos los máximos relativos y se aplica un umbral (0.3 en este caso), con el objetivo de filtrar la salida de la red.

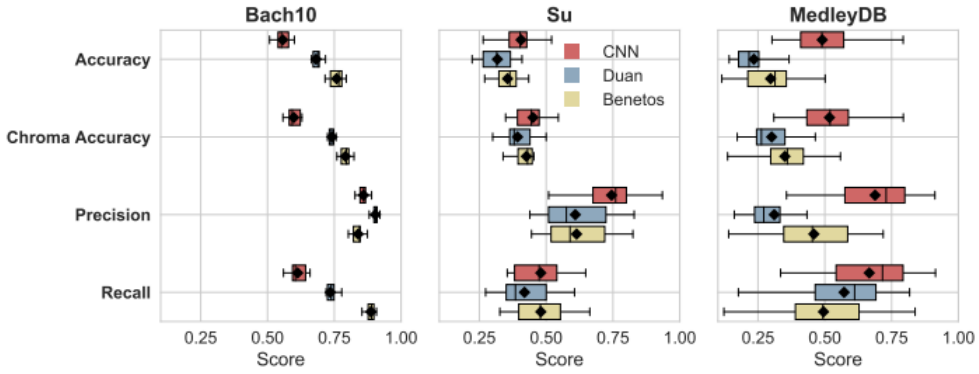

Figura 3.10: Comparación de Deep Salience (rojo) con Benetos (amarillo) y Duan (azul) para los *datasets* de Bach10, Su y MedleyDB; empleando las métricas *Accuracy*, *Chroma Accuracy*, *Precision* y *Recall* [31]

Se hacen tests para tres *datasets* distintos, el MedleyDB, Bach10 y Su; y así poder hacer una comparación con los modelos de referencia Benetos [39] y Duan [40]. La comparación se realiza mediante las métricas de *Accuracy*, *Chroma Accuracy*, *Precision* y *Recall*. Como se observa en la figura 3.10, los resultados de Deep Salience para los *datasets* Su y MedleyDB mejoran con respecto a los otros dos modelos.

Onsets & Frames

Onsets & Frames (OaF) [28] es una red que se incluye en el proyecto Magenta de Google y tiene la particularidad de que esta red proporciona como salida las notas en formato MIDI, es

decir, el *note-level*, en lugar de la frecuencia fundamental (F0) como en los casos anteriores. En todos los modelos nombrados anteriormente la salida era *frame-level*, es decir, las frecuencias que están sonando en una ventana de tiempo pequeña (*frame*).

OaF Utiliza una red convolucional profunda y recurrente que está entrenada para detectar los *onsets* (inicio de una nota musical) y la duración de la misma, es decir, los *frames* que esa nota esta activa. A la hora de predecir, se restringe el detector de *frames*, es decir, sólo actuará cuando el detector de *onsets* indique que hay una nota. Ambos detectores se entrenan en conjunto.

La red ha sido entrenada únicamente para piano y la base de datos que utilizan se llama MAPS [41] en la que se incluye tanto el audio como las anotaciones de notas aisladas, acordes y melodías completas para este instrumento. En este *dataset* se incluyen tanto piezas renderizadas por sintetizadores como piezas interpretadas por un piano Yamaha Disklavier.

La evaluación se realiza mediante las métricas (*Precision*, *Recall* y *F-Score*) calculadas a partir de la librería mir_eval [20]. Se calculan dos tipos de métricas: una en la que se validan los *onsets* con una margen de ±50ms obviando el offset de la nota, y otra en la que se tienen en cuenta los *offsets*.

La arquitectura del detector de *onsets* se compone de un modelo acústico basado en la arquitectura presentada en [42] con alguna modificaciones, seguida de 128 unidades LSTM bidireccionales y por último una capa totalmente conectada con una sigmoidea de 88 neuronas que corresponden a cada una de las 88 notas del piano.

La arquitectura del detector de frames es el modelo acústico, seguido de una capa totalmente conectada con una sigmoidea de 88 salidas. Estas salidas se concatenan con la salida del detector de *onsets* y le siguen 128 unidades LSTM bidireccionales. Por último, hay otra capa totalmente conectada con una sigmoidea de 88 neuronas. Para determinar si el detector de *onsets* esta activo, utilizan un umbral o *threshold* de 0.5. La estructura de OaF se puede ve en la figuras 3.11.

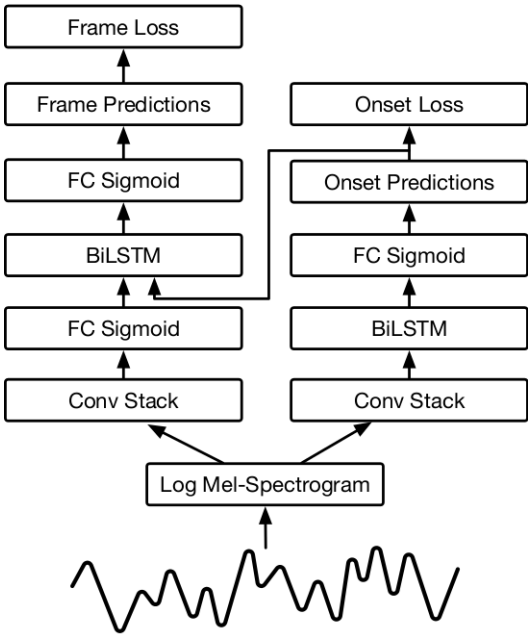

Figura 3.11: Esquema de Onsets&Frames [28]

Para realizar los experimentos se utilizó un tamaño de batch de 8, un learning rate de 0.0006. El optimizador utilizado fue el Adam y se entrenó durante 50000 iteraciones en un tiempo de 5 horas en tres GPUs P100.

Para la presentación de los resultados, los autores hacen una comparación con los modelos descritos en [42] y en [43]. Así como, en el software Melodyne versión 4.1.1.011 [44].

	Frame			Note			Note w/ offset			Note w/ offset & velocity		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Sigtia et al., 2016 [18]	71.99	**73.32**	72.22	44.97	49.55	46.58	17.64	19.71	18.38	—	—	—
Kelz et al., 2016 [13]	81.18	65.07	71.60	44.27	61.29	50.94	20.13	27.80	23.14	—	—	—
Melodyne (decay mode)	71.85	50.39	58.57	62.08	48.53	54.02	21.09	16.56	18.40	10.43	8.15	9.08
Onsets and Frames	**88.53**	70.89	**78.30**	**84.24**	**80.67**	**82.29**	**51.32**	**49.31**	**50.22**	**35.52**	**30.80**	**35.39**

Figura 3.12: Comparación de Onsets and Frames con [42], [43] y el software Melodyne [28]

Como se ve, hay una gran variedad de métodos que se utilizan para la transcripción de música. Modelos probabilísticos como pYIN que fueron los primeros métodos en utilizarse en este ámbito y, posteriormente, la aparición de redes neuronales como CREPE o SPICE. Estos modelos mencionados solo pueden transcribir música monofónica, así que con el objetivo de transcribir música polifónica aparecieron modelos como Deep Salience o Onsets&Frames. Debido a que el objetivo de este trabajo era transcribir música polifónica y multitímbrica, y Onsets&Frames solo había sido entrenada con un único instrumento (Piano), se decidió utilizar Deep Salience.

Capítulo 4

Modelo Convolucional para la Transcripción Automática de Música

El modelo propuesto en este trabajo se engloba en el marco de la transcripción automática de música, cuyo objetivo es detectar las notas que suenan en una melodía polifónica (varias notas sonando de manera simultánea) y multitímbrica (varios instrumentos diferentes). Como se ha comentado en el capítulo anterior, la red que mejor se ajusta a nuestro objetivo es Deep Salience [31], por lo que en este trabajo nos basaremos principalmente en dicha arquitectura. Aunque, con la diferencia de que se ha añadido una capa convolucional en la salida que reduce las dimensiones a (72,50) para facilitar el entrenamiento con archivos MIDI.

La entrada del modelo propuesto tiene unas dimensiones de (360, 50, 6). Estas dimensiones representan lo siguiente:

- 360: La dimensión de frecuencia (6 octavas, divididas en 60 partes o *bins* por octava)
- 50: el número de *frames* (cada uno representa $\approx 11ms$ de la canción)
- 6: Se apilarán 6 armónicos, el fundamental (C1 a C7) , un subarmónico y 4 armónicos superiores.

Las capas convolucionales intermedias son 6. Las dos primeras tienen 128 y 64 filtros de tamaño 5x5, estas capas cubren un semitono (recordamos que cada octava se divide en 60 *bins*, por lo que cada semitono son 5 *bins*). Las dos siguientes tienen 64 filtros de un tamaño de 3x3. La penúltima capa utiliza un filtro muy grande de 70x3, con el objetivo de cubrir más de una octava (14 semitonos). Por último, hay una capa convolucional que tiene el objetivo de comprimir estas frecuencias, tiene un filtro de 5x1, y un stride es de 5x1. Con ello, se consigue hacer una convolución de las 5 frecuencias más cercanas a las notas reales.

La salida del modelo propuesto tiene unas dimensiones de (72, 50). Es decir, se consiguen 12 semitonos por octava, que corresponden a las notas reales en el intervalo del armónico

fundamental (C1 a C7, o lo que es lo mismo de 32,7Hz a 2093Hz). El hecho de que la salida tenga estas dimensiones nos facilita la tarea de entrenar nuestra red únicamente con archivos MIDI, ya que en un MIDI únicamente se proporcionan las notas. La arquitectura de la red se puede ver en la figura 4.1.

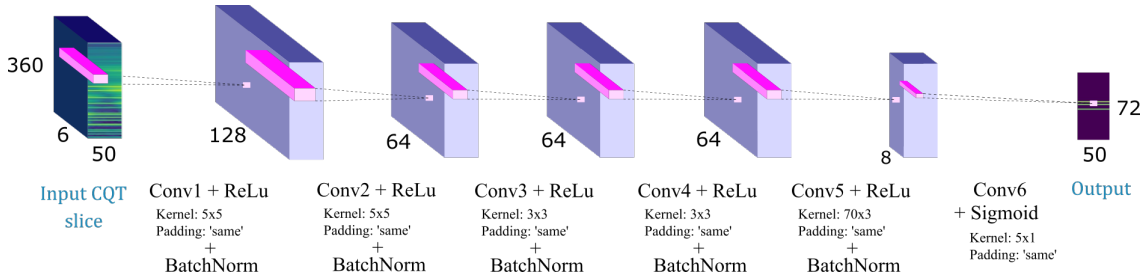

Figura 4.1: Arquitectura de nuestro modelo [6]

Se ha utilizado un optimizador Adam (ADaptative Moment Estimation) con un *learning rate* de 0.001, β_1 de 0.9, β_2 de 0.999 y ϵ de 10^{-7}, éste será el encargado de modificar los parámetros para reducir el error. La función de pérdida es la de reducción de la entropía cruzada (ecuación 4.1). Donde y_{pred} es la predicción del modelo, y_{real} es el *ground truth* y L es el error que devuelve la función.

$$L(y_{real}, y_{pred}) = -y_{real} \log (y_{pred}) - (1 - y_{real}) \log (1 - y_{pred}) \tag{4.1}$$

La red se ha entrenado con el *dataset* Slakh2100 [45]. Este dataset se compone de 2100 canciones y sus correspondientes archivos MIDI. Se ha dividido en 1500, 375 y 225 canciones entre entrenamiento, validación y *test*. Esta base de datos también proporciona todas las melodías separadas por pistas de cada instrumento y sus correspondientes MIDIs. Dentro de cada canción existe un archivo con formato *yaml* llamado *metadata*, que proporciona información acerca de cada una de las pistas. Este archivo es imprescindible para automatizar la búsqueda de instrumentos que queremos utilizar para entrenar la red.

4.1. Entrenamiento del modelo

Para calcular la CQT a partir de archivos WAV del *dataset* Slakh2100 se ha utilizado la librería *librosa* [26]. Se utiliza una frecuencia de muestro de 44100Hz ya que utilizar otra frecuencia haría necesario hacer un remuestreo y esto ralentizaría el entrenamiento (la frecuencia estándar de muestreo de un archivo *.wav* es de 44100Hz). Se define un *hop lenght* de 512, lo que supondrá que cada *frame* tendrá una duración de $\frac{512}{44100} \approx 11ms$.

A la hora de hacer el cálculo de las CQTs de una ventana se tomarán 1024 *frames* donde la ventana de análisis se sitúe en el centro, ya que si se hace el cálculo del CQT de la pista de audio completa el entrenamiento se ralentizaría mucho. Este margen se ha calculado mediante

prueba y error, siendo 1024 el mínimo valor para el cual se consigue un resultado similar al de calcular la misma ventana con la canción entera.

Para calcular el número óptimo de *frames*, se ha realizado una estudio empírico, variando el número de *frames* y comparando el resultado obtenido de la CQT para toda la canción con respecto al cálculo de la CQT para un número determinado de *frames*. En la figura 4.2, se observan diferencias en las frecuencias bajas entre la CQT calculada con toda la canción frente al calculado con 256 *frames*. Por el contrario, en la figura 4.3 no se observa ninguna diferencia, siendo éste el tamaño de ventana que utilizaremos para el entrenamiento.

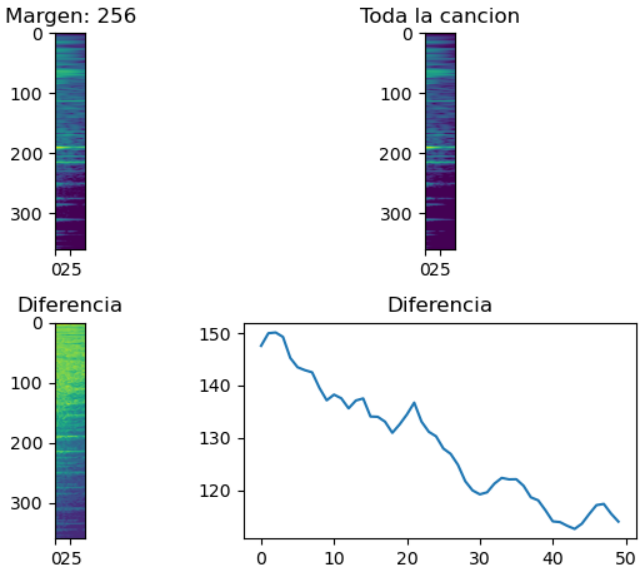

Figura 4.2: Comparativa del cálculo de un CQT con 256 *frames* y otro obtenido con la canción entera. Arriba izquierda: cálculo de la CQT con 256 *frames* de una ventana; Arriba derecha: cálculo del CQT con toda la canción de la misma ventana; Abajo izquierda: diferencia entre las dos CQTs; Abajo derecha: Suma en el eje x de la diferencia

Este hecho está provocado por la insuficiencia de muestras, y se acentúa en las frecuencias bajas donde el periodo de esas frecuencias es mayor y, por ello, se necesita un mayor numero muestras para conseguir el mismo factor de calidad (Q). Al no tener el número de muestras necesario en las frecuencias bajas, el factor de calidad disminuye en ellas. Este hecho no ocurre en la figura 4.3 donde no hay diferencias entre el CQT calculado con toda la canción y el CQT que limitamos el número de *frames*.

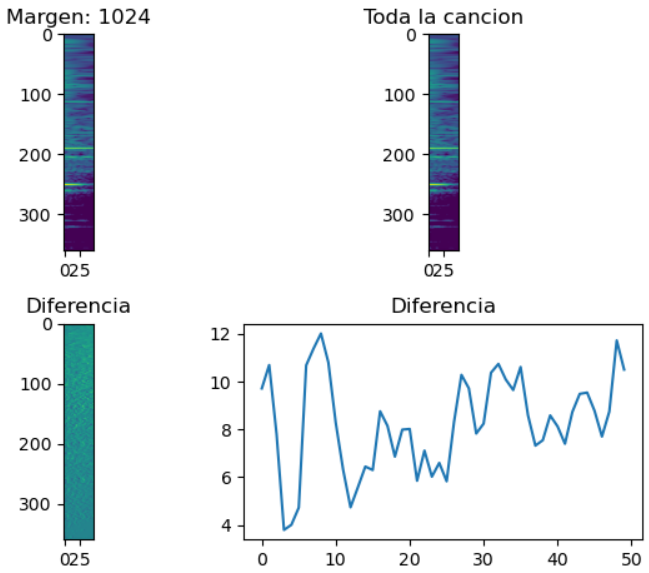

Figura 4.3: Comparativa del cálculo de una CQT con 1024 *frames* y otra obtenido con la canción entera. Arriba izquierda: cálculo de la CQT con 1024 *frames* de una ventana; Arriba derecha: cálculo de la CQT con toda la canción de la misma ventana; Abajo izquierda: diferencia entre los dos CQTs; Abajo derecha: suma en el eje x de la diferencia

Este procedimiento hay que realizarlo tanto para el subarmónico (0.5) como para los 4 que están por encima del fundamental (2, 3, 4, 5). Posteriormente, se apilan para darnos una matriz de 360x50x6 la cual tiene las dimensiones de entrada de nuestra red, en la que se refleja un intervalo de $\approx 0{,}58s$ de una canción.

Para la lectura de los MIDIs y conseguir de ellos los *pianorolls* se ha hecho uso de la librería de PrettyMIDI [46]. Esta librería facilita la tarea de leer un archivo MIDI y de crear el *pianoroll* a partir de ese archivo. En la figura 4.4 se puede ver la CQT del armónico fundamental de una ventana y el *pianoroll* que se utilizará para entrenar la red. Ambos corresponden exactamente al mismo intervalo temporal de una canción.

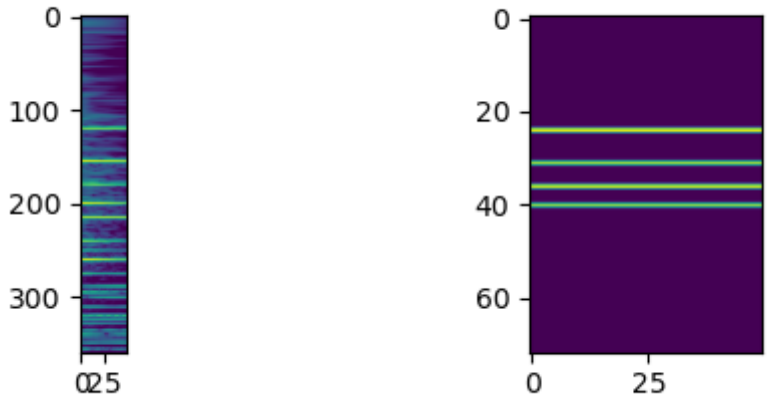

Figura 4.4: A la izquierda la CQT del armónico fundamental de una ventana de 50 *frames* y a la derecha su pianoroll

Para el entrenamiento de la red, no se han utilizado todos los instrumentos de la Slakh2100 ya que hay algunos, como los tambores y otros instrumentos de percusión, no están afinados (no pueden hacer sonar notas musicales). Los instrumentos que se han utilizado han sido: piano, bajo, instrumentos de metal, instrumentos de percusión cromática, guitarra, órgano, flauta, instrumentos de cuerda frotada e instrumentos de lengüeta.

El procedimiento para el entrenamiento ha sido, en primer lugar, captar todas las rutas de las pistas de sonido y sus archivos MIDI correspondientes a partir del archivo *metadata.yaml*. Este archivo guarda la información del instrumento presente en una pista en concreto. En la figura 4.5, se puede ver la distribución de los instrumentos en el conjunto de entrenamiento.

Instrumentos	Número de pistas
Piano	2458
Bajo	1578
Guitarra	3831
Cuerda frotada	2381
De lengüeta	550
De metal	627
Percusión cromática	317
Flauta	514
Órgano	526
Total	12782

Figura 4.5: Distribución de las pistas de los instrumentos que se van a utilizar para entrenar el modelo propuesto

Para entrenar el modelo se define un *batch size* de 32. Para montar un *batch* se toman aleatoriamente las rutas de 32 pistas y se leen con la librería *librosa*. Posteriormente, se selecciona una ventana aleatoria de cada una de las canciones. Además, se ha añadido un límite de ventanas sin notas musicales que puede haber dentro de un *batch*.

En nuestro caso cada *batch* no contiene más de un 10 % de ventanas vacías con un tamaño de 32 trozos por *batch*. De tal manera que la entrada del *batch* es una matriz de 360x50x6x32 en el que están los 6 armónicos de la CQT de 32 ventanas tomados de manera aleatoria entre todas las pistas y comenzando por un *frame* aleatorio, y otra matriz de 72x50x32 que corresponde a sus *pianoroll* en la misma parte de pista.

4.2. Filtrado y Análisis

Una vez que se ha entrenado la red, es necesario realizar un filtrado, ya que la red arroja un resultado borroso y como se puede ver en la figura 4.6.

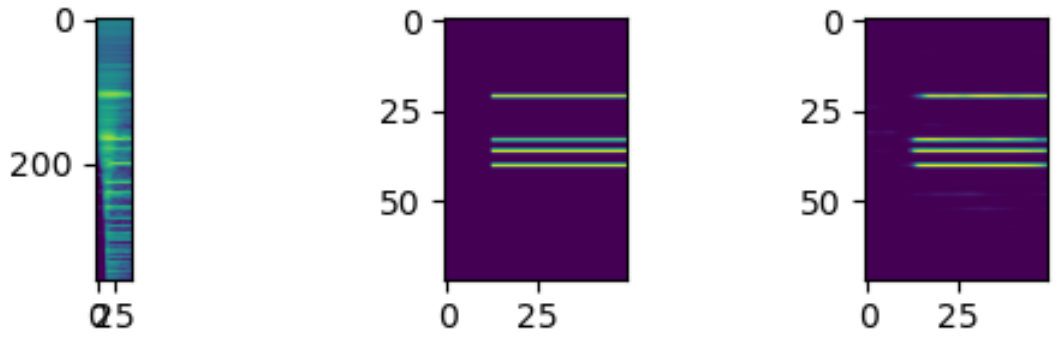

Figura 4.6: A la izquierda la CQT del armónico fundamental de una ventana de 50 frames, en medio su *pianoroll* y a la derecha se muestra la predicción de la red

Para que la red proporcione un resultado como el de la figura del *pianoroll* se hace una búsqueda de los máximos relativos en el eje de la frecuencia y se ponen a 1 todos aquellos que superen un cierto umbral o *threshold*, en caso contrario el valor es 0. En nuestro caso este umbral es de 0.44 y se puede ver el resultado en la figura 4.7.

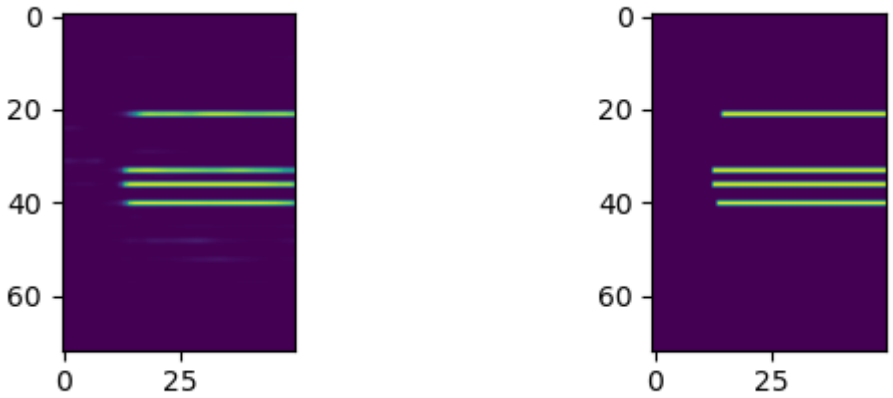

Figura 4.7: A la izquierda la salida de la red y a la derecha tras haber realizado el filtrado

El cálculo de este umbral se ha hecho mediante el análisis completo de 100 pistas diferentes.

Se han analizado por separado y mediante la librería mir_eval [20] se han obtenido las métricas de *Precision* y *Recall*. Con estas métricas se calcula el *F-score* para cada valor de umbral.

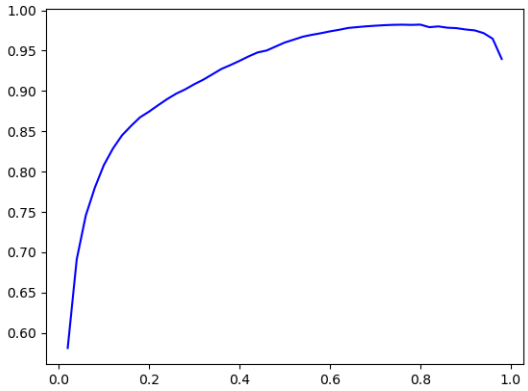

Figura 4.8: Evolución de la *Precision* en función del valor del umbral

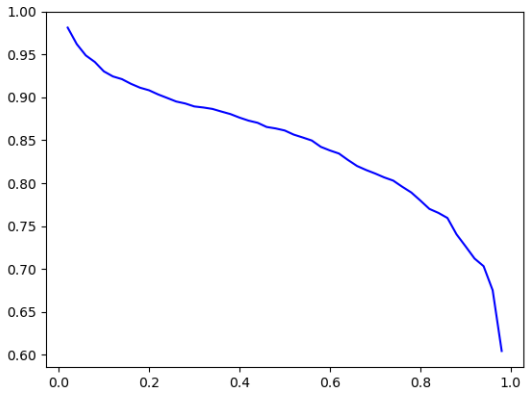

Figura 4.9: Evolución del *Recall* en una canción en función del valor del umbral

En la figura 4.8 y 4.9 se puede ver la evolución de estas métricas en función del umbral. Como se puede ver el *Recall* va disminuyendo a medida que aumentamos el umbral, ya que se eliminan más notas. A su vez, las notas que quedan son las notas que la red estaba más segura por lo que la precisión aumenta. En la figura 4.10 se puede ver el efecto en la salida de la representación en *pianoroll* al filtrar con dos valores distintos del umbral.

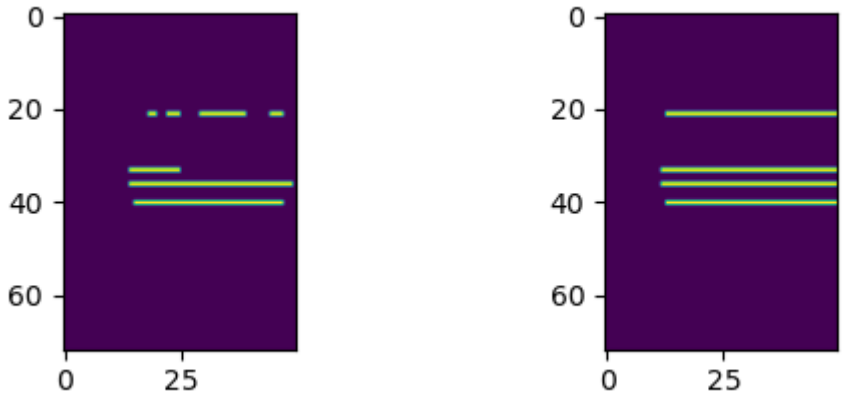

Figura 4.10: Comparativa entre la salida filtrada con un umbral alto (Izquierda) y un umbral bajo (Derecha)

Como ambas, métricas son tan importantes al mismo tiempo se utiliza el *F-score* como métrica para determinar el comportamiento idóneo de la red ($Fscore = 2 \cdot \frac{precision \cdot recall}{precision + recall}$). En la mayoría de pistas la evolución de esta métrica tiene una forma de campana invertida como se puede ver en la figura 4.11.

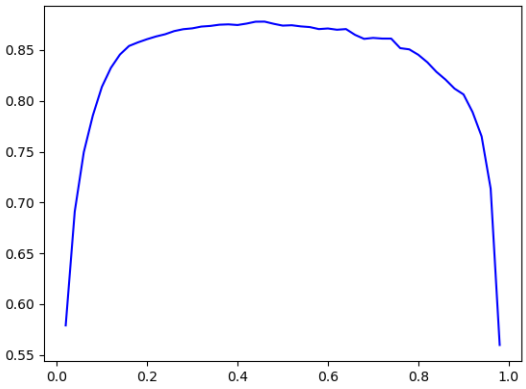

Figura 4.11: Evolución del *F-score* en una canción en función del valor del umbral

Este procedimiento se realiza con las 100 canciones, y se almacena el valor del umbral del pico máximo. Una vez que se tiene los valores de las 100 canciones se realiza una media, dándonos como resultado 0.44.

4.3. Diagrama de flujo del entrenamiento del modelo de transcripción

En la figura 4.12 se ha representado el diagrama de flujo del procedimiento de entrenamiento de la red desarrollada.

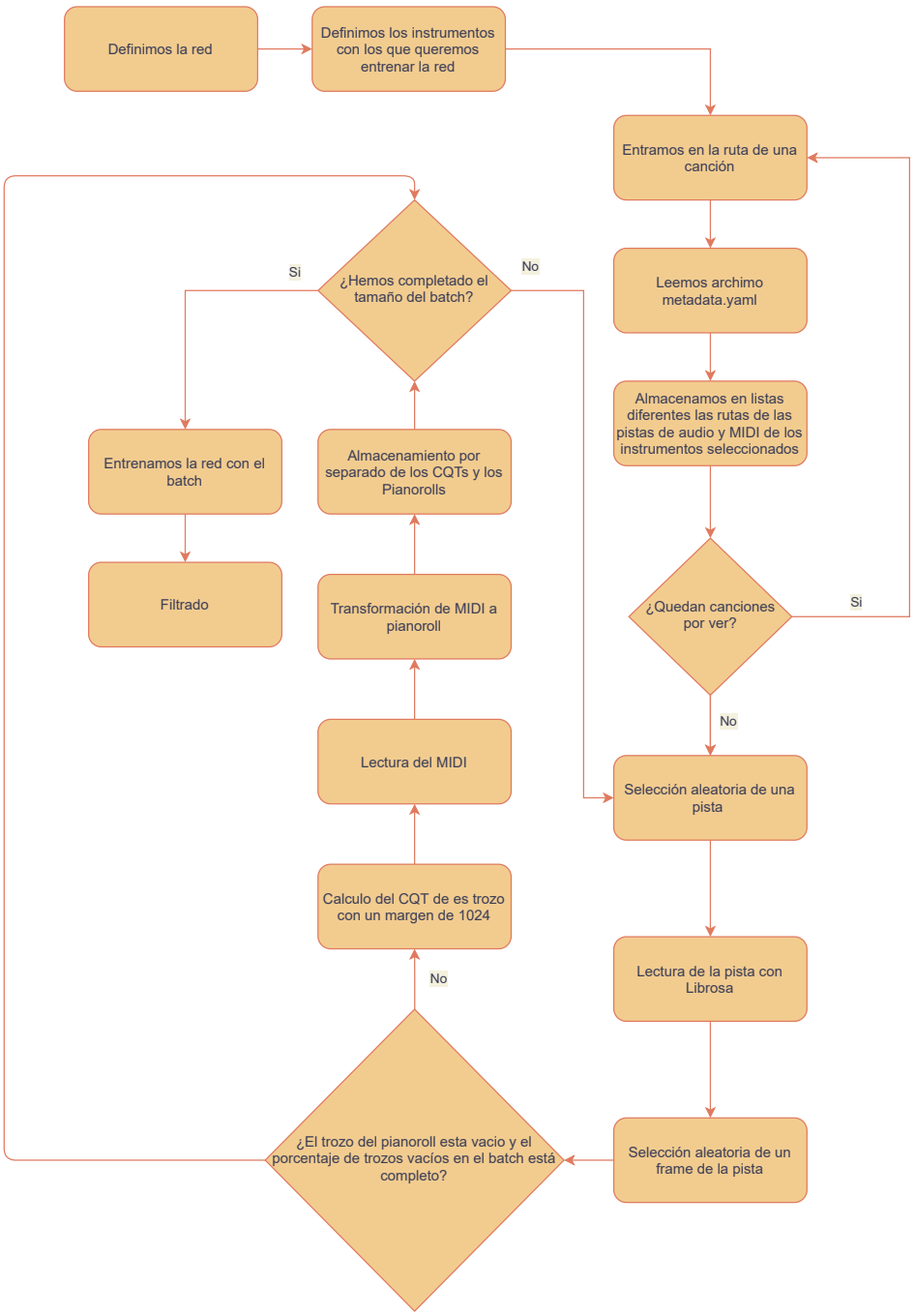

Figura 4.12: Diagrama del procedimiento de entrenamiento del modelo de transcripción

Antes de comenzar a entrenar la red, es necesario definir la arquitectura que va a tener,

se ha hecho mediante la librería de Tensorflow [4]. Además, se definen los instrumentos del *dataset* de Slakh2100 que se utilizarán para entrenar la red.

El siguiente paso es almacenar todas las rutas de los archivos MIDI y WAV de la base de datos en dos lista separadas. Esta tarea se lleva a cabo leyendo el archivo que se encuentra en cada canción: metadata.yaml donde se indica a qué instrumento pertenece cada pista de la canción.

Una vez obtenidas las listas comienza el proceso de entrenamiento. Para ello, se escogerá una pista aleatoria de la lista, se leerá con librosa [26] obteniendo así una lista de todas las muestras del archivo de audio. De manera aleatoria se seleccionará un trozo de la canción y se realizará el procesamiento de la señal de audio, mediante el cálculo de la CQT. El trozo de la canción corresponderá a 1024 *frames* como ya se ha hablado anteriormente, del que posteriormente se sacarán los 50 *frames* que corresponden a la entrada de la red. Además, se hace un conteo de los trozos que están vacíos, con el objetivo de no exceder el entrenamiento con trozos con escasa información.

El siguiente paso es obtener el *pianoroll* real que corresponde al trozo de entrada, obteniendo así el *ground truth* de ese trozo en concreto. Todo este procedimiento se realiza tantas veces como hayamos definido el tamaño del *batch*, en nuestro caso 32. Cuando ya tenemos montado el batch, solo queda entrenar la red con este lote, y repetir este proceso hasta que la red esté entrenada.

Cuando la red ya esta entrenada, se hace la predicción de una canción y se aplica el filtrado.

Capítulo 5

Experimentos y Resultados

Los experimentos que se han realizado se dividen en dos categorías principales: los entrenamientos realizados para detectar las notas en un *frame* y los entrenamientos que se centran únicamente en detectar los *onsets* (inicios de la notas). Por último, se combinará un modelo especializado en detectar *frames* del piano con otro especializado en detectar *onsets* que junto a un algoritmo de *note tracking* (seguimiento de notas) nos reportará los mejores resultados para este instrumento.

5.1. Entrenamiento basado en frames

Se ha entrenado una red de *frames*, con todos los instrumentos afinados de Slack2100 y posteriormente se ha hecho una evaluación con las métricas multi-F0. En la Tabla 5.1, se pueden ver los resultados para cada uno de los instrumentos de Slack2100, así como de un *testeo* con los *datasets* Bach10 y MedleyDB. El umbral utilizado ha sido de 0.44. Las métricas que se valoran son *Chromma accuracy* (CAcc), *Accuracy* (Acc), *Precision* (P), *Recall* (R) y *F-score* (F).

Tabla 5.1: Métricas multi-F0 del modelo propuesto para Slack2100, Bach10 y MedleyDB

Base de datos	Instrumento	CAcc	Acc	R	P	F
Slakh2100	Piano	80.42	78.83	**88.39**	88.16	87.28
	Bajo	**85.78**	76.08	83.14	84.66	83.36
	Guitarra	70.85	67.50	80.07	80.34	78.67
	Cuerda frotada	82.94	**82.01**	85.68	**91.96**	**87.96**
	Instrumentos de lengüeta	75.88	75.42	82.46	88.24	84.67
	Instrumentos de metal	72.55	50.66	54.72	65.61	58.44
	Percusión cromática	21.15	15.05	21.44	30.75	23.82
	Flauta	78.80	77.04	84.95	88.27	86.49
	Órgano	68.43	55.32	70.84	65.60	67.00
Bach10	Bach10single	82.37	70.86	84.06	81.00	82.48
	Bach10multi	69.71	67.39	73.93	88.39	80.47
MedleyDB	Todos	51.89	47.18	68.73	56.89	61.01

En la figura 5.1, se pueden ver los diagramas de cajas de las pistas de piano del Slack2100, el Bach10multi (que corresponde a las melodías con varios instrumentos al mismo tiempo), el Bach10single (pistas con solo un instrumento) y el MedleyDB. Se puede hacer una comparativa con el estado del arte en la figura 3.10. Los valores en negrita de la tabla 5.1 corresponden a las puntuación más altas

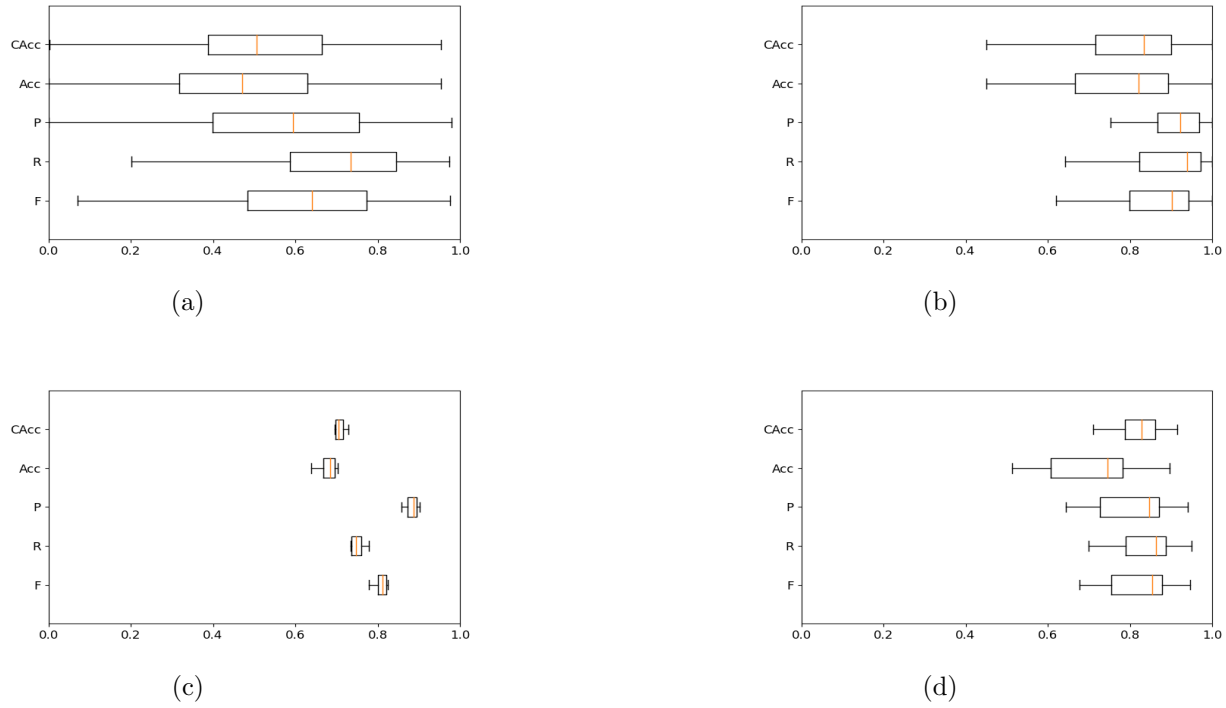

Figura 5.1: Métricas Multi-F0 para: a) MedleyDB, b) Pianos de Slakh2100, c) Bach10 multi and d) Bach10 single

Posteriormente, con la red de *frames* ya entrenada, se ha hecho una evalución con las métricas de transcripción para poder hacer una comparación con OaF. Para ello, se han utilizado las métricas de transcripción con la librería *mir_eval*. En esta librería se evalúa si el inicio (*onset*), final de la nota (*offset*) y el tono son los correctos. Aunque también se ha hecho una evaluación sin tener en cuenta los *offsets*, ya que los finales de la nota pueden sufrir muchos cambios en función del tipo de instrumento utilizado, al tener tiempos de decaimiento del sonido variables.

Tabla 5.2: Métricas de transcripción de nuestro modelo (CNN) y Onsets & Frames (OaF) en el conjunto de datos Slakh2100.

Instrumento	Umbral	Nota sin offset			Nota con offset		
		P	R	F	P	R	F
Piano (CNN)	0.44	65.42	78.72	67.07	30.91	39.06	33.45
Piano (OaF [28])	-	**88.47**	**94.73**	**90.68**	**58.48**	**63.93**	**60.49**
Bajo (CNN)	0.44	**68.65**	**85.96**	**74.81**	53.87	65.02	57.48
Bajo (OaF [28])	-	65.98	78.52	71.02	**62.39**	**73.85**	**66.94**
Guitarra (CNN)	0.44	**55.36**	73.25	59.73	36.77	45.85	40.19
Guitarra (OaF [28])	-	55.20	**84.95**	**64.32**	**39.92**	**60.63**	**46.43**
Cuerda frotada (CNN)	0.44	34.24	62.32	41.70	**17.91**	26.27	**20.29**
Cuerda frotada (OaF [28])	-	**40.16**	**81.03**	**48.88**	15.78	**30.25**	19.72
Inst. de lengüeta (CNN)	0.44	**66.12**	**75.84**	**69.23**	**51.76**	**57.68**	**53.57**
Inst. de lengüeta (OaF [28])	-	42.91	75.69	52.95	24.99	43.91	30.91
Inst. de metal (CNN)	0.44	**49.32**	58.64	**49.98**	**39.35**	41.52	39.25
Inst. de metal (OaF [28])	-	45.11	**68.53**	48.99	37.28	**52.42**	**39.59**
Perc. cromática (CNN)	0.44	25.77	26.07	22.97	4.99	3.98	4.04
Perc. cromática (OaF [28])	-	**50.31**	**68.79**	**56.27**	7.84	11.38	9.11
Flauta (CNN)	0.44	**45.55**	53.51	**47.86**	**30.89**	**34.36**	**31.88**
Flauta (OaF [28])	-	20.66	**59.56**	29.40	4.35	11.25	6.00
Órgano (CNN)	0.44	19.58	54.47	25.31	9.77	19.01	11.39
Órgano (OaF [28])	-	**20.59**	**57.47**	**27.67**	15.92	**30.61**	19.98

Además, se han entrenado distintas redes empleando únicamente un único instrumento para observar si hay una mejora en cada uno de ellos, y si la red se adapta mejor a las características tímbricas de cada instrumento. Los resultados se encuentran en la Tabla 5.3. Para cada uno de los instrumentos, se ha recalculado el umbral óptimo.

Tabla 5.3: Métricas de transcripción para los modelos entrenados con un único instrumento en comparación con Onset & Frames. Los valores en negrita corresponden a las puntuación más altas

Instrumento	Umbral	Base de datos	Nota sin offset			Nota con offset		
			P	R	F	P	R	F
Bajo (CNN)	0.407	Slakh2100	**92.91**	**90.78**	**91.54**	**85.54**	**85.41**	**84.99**
Bajo (OaF [28])	-	Slakh2100	65.98	78.52	71.02	62.39	73.85	66.94
Metal (CNN)	0.416	Slakh2100	**57.26**	**76.05**	**61.07**	**42.76**	50.32	**44.10**
Metal (OaF [28])	-	Slakh2100	45.11	68.53	48.99	37.28	**52.42**	39.59
Lengüeta (CNN)	0.302	Slakh2100	**76.34**	**84.26**	**79.02**	**61.12**	**65.12**	**62.31**
Lengüeta (OaF [28])	-	Slakh2100	42.91	75.69	52.95	24.99	43.91	30.91
Flauta (CNN)	0.334	Slakh2100	**64.98**	**74.47**	**66.58**	**48.20**	**51.78**	**48.02**
Flauta (OaF [28])	-	Slakh2100	20.66	59.56	29.40	4.35	11.25	6.00

5.2. Entrenamiento basado en onsets

El siguiente experimento se ha realizado con la intención de especializar a la red para detectar únicamente los *onsets* y el *pitch* (tono). Se puede observar en la figura 5.2 la entrada de la red y el resultado que queremos que arroje nuestra red (*ground truth*).

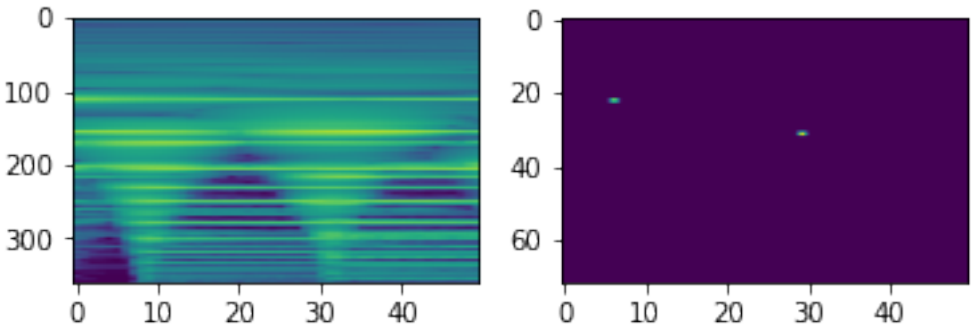

Figura 5.2: A la izquierda, la entrada del primer armónico de la CQT y a la derecha, la representación de los onsets y el tono en forma de pianoroll.

Se ha entrenado una red con los *onsets* de todos los instrumentos de Slack2100 y se han obtenido las métricas evaluando el *onset* y el tono, y otra evaluando únicamente el *onset*. Los resultados se ven reflejados en la tabla 5.4 y el *threshold* utilizado es de 0.13.

Tabla 5.4: Evaluación del *onset* y el tono, y únicamente el *onset* del modelo especializado en entrenar *onsets*. Los valores en negrita corresponden a las puntuación más altas

Instrument	Nota onset y pitch			Nota solo onset		
	P	R	F	P	R	F
Piano	82.34	84.91	82.84	87.51	88.04	86.57
Bass	**88.45**	**85.49**	**86.77**	91.81	**88.84**	**90.10**
Guitar	75.22	80.80	76.85	80.14	86.14	81.79
Strings	81.23	34.24	45.30	84.38	35.42	46.92
Reed	78.30	65.92	68.03	84.56	71.75	73.94
Brass	66.98	59.38	60.72	80.07	69.78	71.96
Chromatic Perc.	45.13	38.02	37.91	62.60	43.11	44.66
Pipe	87.33	41.27	53.94	**91.91**	43.66	56.93
Organ	47.24	45.46	41.71	70.96	63.54	60.54

Además, se realizaron otras pruebas cambiando la entrada de la red para ver si se conseguían mejorar los resultados. En lugar de la CQT, se utilizó el *Mel_Spetrogram* [22] en una red únicamente entrenada para Piano y se comparó con otra red únicamente entrenada para Piano con CQTs. En la figura 5.3 se puede ver la entrada de esta red y su salida.

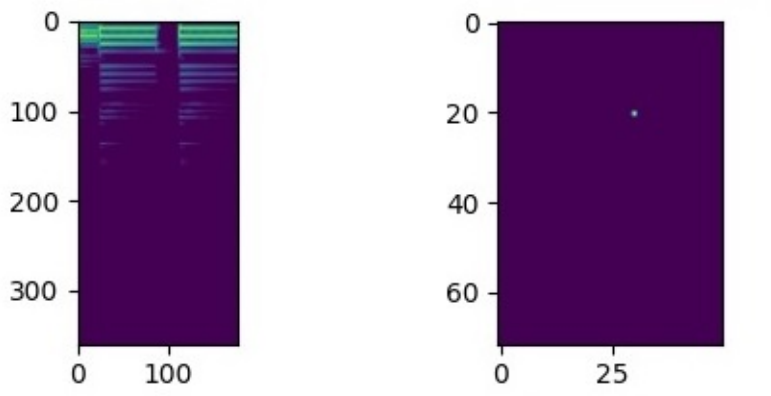

Figura 5.3: A la izquierda, la entrada del primer armónico del espectrograma de Mel y a la derecha, la representación de los onsets y el tono en forma de pianoroll.

Los resultados como se pueden ver en la tabla 5.5, no mejoraron el comportamiento en comparación al CQT.

Tabla 5.5: Comparación entre red entrenada únicamente con Piano para detectar onsets con entrada CQT y espectrograma de Mel. Los valores en negrita corresponden a las puntuación más altas

Instrumento	Nota onset y pitch			Nota solo onset		
	P	R	F	P	R	F
Piano (CQT)	**95.16**	**87.32**	**90.16**	**96.09**	**88.11**	**91.01**
Piano (Mel)	70.97	58.32	61.56	84.80	66.88	71.24

5.3. Algoritmo de note tracking

Cuando se quiere generar un MIDI con los resultados de la red entrenada con *frames*, pese a que los resultados multi-F0 sean notables, el sonido obtenido a partir de esos datos no se escucha bien. La razón es que se crean huecos entre las notas y se activan notas en zonas donde no debería haberlas. Para resolverlo se necesita incluir un algoritmo para que los resultados se puedan escuchar correctamente.

El algoritmo de *note tracking* se ha implementado únicamente para el instrumento de piano. Para la aplicación de este algoritmo es necesario entrenar por separado una red con *onsets* y otra red de *frames*. Aprovechando la salida de cada una de las redes y mediante el algoritmo de *note tracking* se consigue un *pianoroll* que mejora al de la red de *frames*. En la figura 5.4, se puede ver los resultados de estas dos redes superpuestas y el filtrado que realiza el note tracking.

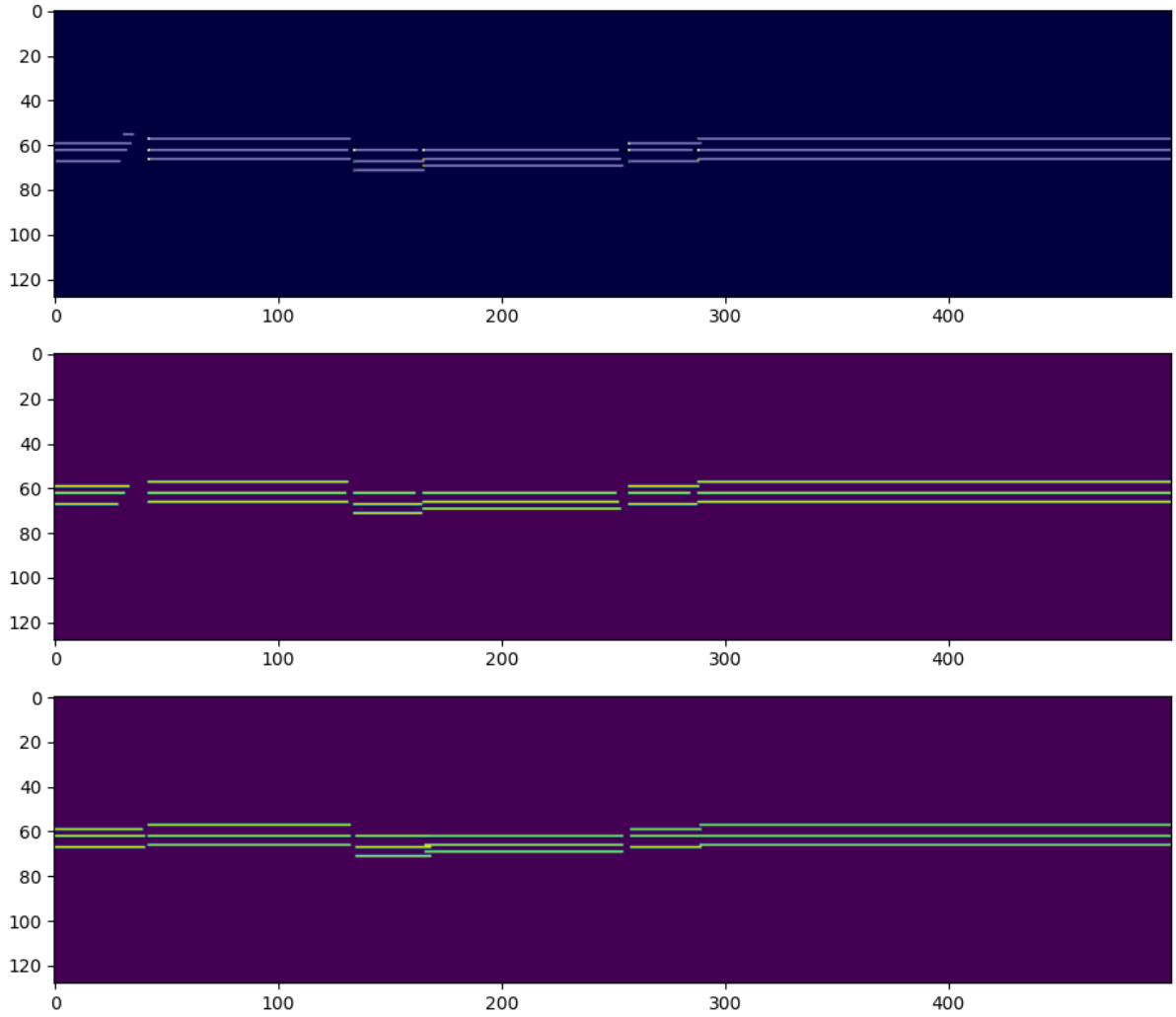

Figura 5.4: Arriba, los onsets y los frames superpuestos que corresponde a la salida de cada una de las redes. En el medio, la salida del *note tracking*. Abajo, el *pianoroll* de la canción real.

El algoritmo de *note tracking* se compone de tres partes distintas que consiguen mejorar la salida de la red de *frames*, utilizando la información que proporciona la red de *onsets*. Esto es debido a que la red de *onsets* tiene mejores resultados a la hora de adivinar los inicios de nota, ya que está especializa en ellos. Las tres partes son las siguientes:

- **Eliminar onsets vacíos**: Se eliminan los *onsets* que predice la red especializada en ellos en los que en el detector de *frames* no detectado nada en un cierto intervalo temporal.

- **Crear onsets**: Si el detector de *onsets* detecta un inicio de nota que se encuentra dentro de una nota detectada por el detector de *frames*, se genera un hueco de dos *frames* detrás de la predicción del onset. Esto se hace para que a la hora de pasar de *pianoroll* a MIDI, se genere un *onset* donde lo ha detectado el detector de *onsets*.

– **Rellenar huecos**: Esta función rellena huecos que se crean en las predicciones de la red de *frames*, siempre y cuando no exista una predicción de *onset* de la red de *onsets* en esos huecos. Se hace para evitar que se generen *onsets* falsos.

En la tabla 5.6, se hace una comparación con las métricas de transcripción entre la red de frames y las dos redes combinadas con el algoritmo de *note tracking*. Se han vuelto a calcular las métricas teniendo en cuenta los *offsets* y sin tenerlos en cuenta. Además, se hace una comparativa con Onsets&Frames que tiene una estructura simular, ya que se compone de dos redes unidas, una de *onsets* y otra de *frames*.

Tabla 5.6: Resultados del algoritmo de note tracking

Instrumento	Nota sin offset			Nota con offset		
	P	R	F	P	R	F
Piano (Onsets + frames)	**89.91**	77.60	80.63	57.23	49.91	52.26
Piano (Frames)	68.21	86.80	59.54	44.57	52.11	40.10
Piano (OaF)	88.47	**94.73**	**90.68**	**58.48**	**63.93**	**60.49**

Como se puede observar en la tabla 5.6, la mejora con respecto al uso de una red única de *frames* en las métricas de transcripción son muy notorias. Al hacer la comparativa con una red similar como es la del proyecto de Google Magenta (OaF), esta es superior a nuestro modelo. La diferencia puede ser dada por el hecho de que ambas redes se entrenan de manera conjunta.

5.4. Discusión de resultados

En el primer experimento de *frames*, donde se han utilizado todos los instrumentos afinados de Slakh2100, se puede observar que hay una gran diferencia en los resultados de detección de F0 para cada instrumento. Por ejemplo, basándonos en las métricas de multi-F0, el F-score de los instrumentos de cuerda frotada es de 87.96 frente a solo un 23.82 de los de percusión cromática. Por lo tanto, se puede llegar a la conclusión de que el timbre es muy importante a la hora de realizar una transcripción.

Por este motivo, se decidió entrenar redes que se especializaran en un único instrumento (tabla 5.3). Como se ha podido observar se obtiene una mejora clara en algunos de los instrumentos como en el Bajo, que su F-score en métricas sin *offset* aumenta de 74.81 a 91.54. En instrumentos de metal la mejora es clara, de un 49.98 a 61.07, pero todavía no da unos resultados que se podrían considerar buenos.

También se puede observar que existe una clara diferencia entre las métricas multi-F0 y las métricas de transcripción. Como se observa en la tabla 5.2, estos valores son más bajos que los de la tabla 5.1. Esta situación es provocada por la aparición de falsos inicios y finales de nota, ya que la red hace una estimación *frame* a *frame*.

Por esta razón se decidió hacer un entrenamiento que se especializara únicamente en la detección de *onsets* (tabla 5.4). Esta red mejoraba por mucho las métricas con respecto a la red de *frames*. Además, se hizo una prueba cambiando la forma de procesar la señal de audio con un espectrograma de Mel, pero como se puede ver en la tabla 5.5, los resultados seguían siendo mejores utilizando como procesado de entrada la CQT.

Debido al gran éxito en la red de *onsets* y con ayuda de un algoritmo de *note tracking*, se combinaron los dos tipos de redes. Como se puede ver en la tabla 5.6, los resultados para el piano se ven claramente mejorados. Además, los resultados en el sonido son abismales, la transcripción es mucho más nítida pudiendo así escuchar una melodía muy similar a la original. Esto hace que un algoritmo como el planteado en este trabajo, aunque presenta un cierto nivel de imprecisión, permite reproducir la música original con bastante precisión.

Desde un punto de vista práctico también se puede ver que para mejorar la precisión es mejor entrenar redes especializadas en timbre, más que redes que permitan transcribir todos los instrumentos.

Capítulo 6

Implementación

La implementación del modelo se ha hecho en el lenguaje de programación Python. Los entornos utilizados son Jupyter notebook para el módulo de entrenamiento y Spyder para los módulos de lectura de los *datasets*, procesamiento de entrada, modelos y *test*.

6.1. Librerías utilizadas

Para facilitar la tarea a la hora de programar se han utilizado algunas librerías como podrían ser:

- **Keras [3], Tensorflow [4] y Pescador [47]** : Para tareas relacionadas con redes neuronales
- **Librosa [26]**: Para leer y procesar archivos WAV
- **Pretty_midi [46]**: Para leer archivos MIDI y transformarlos en un formato de pianoroll
- **Os**: Utilizado principalmente para facilitar la tarea de almacenar las rutas de los archivos
- **Numpy [48]**: Librería que nos ayuda a trabajar de una manera cómoda con matrices y otras funciones matemáticas
- **Pandas [49]**: Utilizada únicamente para almacenar métricas en archivos CSV
- **Matplotlib [50]**: Librería muy útil para hacer gráficas que nos ayudará a representar resultados de una manera visual y observar la evolución del entrenamiento
- **YAML [50]**: Utilizada para leer los archivos metadata que se encuentra en formato yaml
- **Mir_eval [51]**: Utilizada para calcular las métricas de calidad relacionadas con tareas de identificación musical.

6.2. Definición de la arquitectura

La arquietctura se define en la función CrossEntropy implementada del siguiente modo:

```
def CrossEntropy(y_true, y_pred):
    y_true = K.clip(y_true, K.epsilon(), 1.0 - K.epsilon())
    y_pred = K.clip(y_pred, K.epsilon(), 1.0 - K.epsilon())
    return K.mean(K.mean(
        -1.0*y_true* K.log(y_pred) - (1.0 - y_true)
        * K.log(1.0 - y_pred), axis=-1), axis=-1)


def model3_def():
    input_shape = (360,50,6)
    inputs = Input(shape=input_shape)

    y0 = BatchNormalization()(inputs)


    y1 = Conv2D(128, (5, 5), activation='relu',
    name='Semitono1', padding="same")(y0)
    y1a = BatchNormalization()(y1)



    y2 = Conv2D(64, (5, 5), activation='relu',
    name='Semitono2', padding="same")(y1a)
    y2a = BatchNormalization()(y2)



    y3 = Conv2D(64, (3, 3), activation='relu',
    name='Detalles1', padding="same")(y2a)
    y3a = BatchNormalization()(y3)



    y4 = Conv2D(64, (3, 3), activation='relu',
    name='Detalles2', padding="same")(y3a)
    y4a = BatchNormalization()(y4)



    y5 = Conv2D(8, (70, 3), activation='relu',
    name='Octavas', padding="same")(y4a)
    y5a = BatchNormalization()(y5)

    y6 = Conv2D(1, (5, 1), strides= (5,1), activation='sigmoid',
    name='Comprimir', padding="same")(y5a)

    y7 = Lambda(lambda x: K.squeeze(x, axis=3))(y6)

    model = Model(inputs=inputs, outputs=y7)
    model.compile(loss=CrossEntropy, optimizer='adam')
    return model
```

Como se puede ver en el código anterior, desarrollado en Keras, la definición de las arquitecturas se hace de manera secuencial. Entre cada capa de las redes se hace una normalización de *batch* y a la salida se utiliza una función de activacion ReLU, exceptuando en la última capa en la que se emplea una sigmoide. Además, se ha mantenido activado el *padding* con el objetivo de no variar las dimensiones temporales y frecuenciales en las 5 primeras capas. En la última capa las dimensiones varían solo en la dimensión frecuencial, ya que se hace una convolución de las frecuencias en bloques de 5 en 5 (se consigue utilizando un filtro y un *stride* de 5x1). En la figura 6.1 se pueden ver los valores de las dimensiones y el número de parámetros capa a capa.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_89 (InputLayer)        [(None, 360, 50, 6)]      0
_________________________________________________________________
batch_normalization_528 (Bat (None, 360, 50, 6)        24
_________________________________________________________________
Semitono1 (Conv2D)           (None, 360, 50, 128)      19328
_________________________________________________________________
batch_normalization_529 (Bat (None, 360, 50, 128)      512
_________________________________________________________________
Semitono2 (Conv2D)           (None, 360, 50, 64)       204864
_________________________________________________________________
batch_normalization_530 (Bat (None, 360, 50, 64)       256
_________________________________________________________________
Detalles1 (Conv2D)           (None, 360, 50, 64)       36928
_________________________________________________________________
batch_normalization_531 (Bat (None, 360, 50, 64)       256
_________________________________________________________________
Detalles2 (Conv2D)           (None, 360, 50, 64)       36928
_________________________________________________________________
batch_normalization_532 (Bat (None, 360, 50, 64)       256
_________________________________________________________________
Octavas (Conv2D)             (None, 360, 50, 8)        107528
_________________________________________________________________
batch_normalization_533 (Bat (None, 360, 50, 8)        32
_________________________________________________________________
Comprimir (Conv2D)           (None, 72, 50, 1)         41
_________________________________________________________________
lambda_88 (Lambda)           (None, 72, 50)            0
=================================================================
Total params: 406,953
Trainable params: 406,285
Non-trainable params: 668
_________________________________________________________________
```

Figura 6.1: Dimensiones y número de parámetros por capa

6.3. Definición de funciones para leer el Dataset

Se define una función, que permite almacenar todas las canciones de un grupo de entrenamiento. La estructura de Slakh2100, son 3 carpetas (*train*, *validación* y *test*) y en cada una de ellas están las canciones divididas por carpetas.

```
def get_paths_songs(grupo):

    file_paths = []
    songs = os.listdir(grupo)

    for song in songs:
            file_paths.append(grupo + '/' + song)
    return file_paths, songs
```

La siguiente función devuelve todos los nombres de las pistas que pertenecen a un instrumento en concreto a partir de la ruta de la carpeta de una canción. Para ello es necesario leer el archivo metadata.yaml.

```
def busca_instrumento(song, instrumento):

    path = song + "/MIDI"
    midis = [os.path.splitext(filename)[0] for filename in os.listdir(path)]


    i_instrumentos = []
    with open(song + "/metadata.yaml") as file:

        data = yaml.load(file, Loader=yaml.FullLoader)

        for midi in midis:

            if (data['stems'][midi]['inst_class'])==instrumento:
                i_instrumentos.append(midi)

    return i_instrumentos
```

Mezclando ambas funciones anteriores, se obtiene una función que almacena en dos listas todas las rutas de los WAV y de los MIDIS de los instrumentos que se van a utilizar de un grupo en concreto (*train*, *validación* o *test*). En los bucles principales que recorren todas las canciones y todos los intrumentos hay una condición que corrije un error que se produce en algunos casos en los hay rutas que no existen pese a aparecer en el archivo metadata.yaml.

```
def get_paths_instrumentos(grupo, instrumentos):

    songs, name_songs = get_paths_songs(grupo)

    paths_instrumento_stem = []
    paths_instrumento_midi = []

    for song in songs:

        for instrumento in instrumentos:

            i_instrumentos = busca_instrumento(song, instrumento)

            for i_instrumento in i_instrumentos:

                path_stem = song + "/stems/" + i_instrumento + ".wav"
                path_midi = song + "/MIDI/" + i_instrumento + ".mid"

                if os.path.exists(path_stem) and os.path.exists(path_midi):

                    paths_instrumento_stem.append(song + "/stems/"
                    + i_instrumento + ".wav")
                    paths_instrumento_midi.append(song + "/MIDI/"
                    + i_instrumento + ".mid")


    return paths_instrumento_stem, paths_instrumento_midi
```

La siguiente función, está creada exclusivamente para obtener las rutas en el conjunto de entrenamiento. La diferencia con la anteriore es la lista de entrada 'instrumentosExtra' que aumentará la frecuencia de los instrumentos introducidos en ella a la hora de entrenar la red.

```
def get_paths_instrumentosTrain(grupo, instrumentos, instrumentosExtra):

    songs, name_songs = get_paths_songs(grupo)

    paths_instrumento_stem = []
    paths_instrumento_midi = []

    for song in songs:

        print(song)

        for instrumento in instrumentos:

            i_instrumentos = busca_instrumento(song, instrumento)

            for i_instrumento in i_instrumentos:

                path_stem = song + "/stems/" + i_instrumento + ".wav"
                path_midi = song + "/MIDI/" + i_instrumento + ".mid"

                if os.path.exists(path_stem) and os.path.exists(path_midi):

                    paths_instrumento_stem.append(song + "/stems/"
                    + i_instrumento + ".wav")
                    paths_instrumento_midi.append(song + "/MIDI/"
                    + i_instrumento + ".mid")


                    if instrumento in instrumentosExtra:
                        paths_instrumento_stem.append(song + "/stems/"
                        + i_instrumento + ".wav")
                        paths_instrumento_midi.append(song + "/MIDI/"
                        + i_instrumento + ".mid")

    return paths_instrumento_stem, paths_instrumento_midi
```

6.4. Definición de la función de cálculo de las CQTs

Se definen de manera global algunas variables: número de divisiones por octava, número de octavas que se van a procesar, la frecuencia de muestreo, la frecuencia mínima a la que se empezará a calcular el primer armónico del CQT y el *hop lenght*.

```
BINS_PER_OCTAVE = 60
N_OCTAVES = 6
HARMONICS = [0.5, 1, 2, 3, 4, 5]
SR = 44100
FMIN = 32.7
HOP_LENGTH = 512
```

```
def compute_hcqt_trozo(audio_fpath, pianoroll, margen
, n_vacios, batch_size, X_vacios):

    y, fs = librosa.load(audio_fpath, sr=SR)

    mitad = int(margen/2)
    low = margen
    high = pianoroll.shape[1] - margen


    error = False

    if len(y)==0 or low>=high:

        error = True
        log_hcqt = 0
        freq_grid = 0
        time_grid = 0
        i_trozo = 0

    else:

        i_trozo = np.random.randint(low=low,high=high)

        if n_vacios>=(X_vacios * batch_size) and
        np.all(pianoroll[:,i_trozo:i_trozo+n_samples]==0):

            error = True
            log_hcqt = 0
            freq_grid = 0
            time_grid = 0
            i_trozo = 0

        else:

            cqt_list = []
            shapes = []

            margen_lado = int((margen-n_samples)/2)

            #Saca los cqt de varias armonicos, en total son 6 cqts
            cqt = librosa.cqt(
                y[(i_trozo-margen_lado)*HOP_LENGTH:
                (i_trozo+n_samples+margen_lado)*HOP_LENGTH-1],
                sr=fs, hop_length=HOP_LENGTH,
                fmin=FMIN*np.min(HARMONICS),
                n_bins=BINS_PER_OCTAVE*10,
                bins_per_octave=BINS_PER_OCTAVE
            )

            for h in range(len(HARMONICS)):

                if h==0:
                    i=0
                else:
                    i = round((np.log2(2*h)*60))

                cqt_list.append(cqt[i:i+360,:])
                shapes.append(cqt.shape)

            shapes_equal = [s == shapes[0] for s in shapes]
            if not all(shapes_equal):
                min_time = np.min([s[1] for s in shapes])
                new_cqt_list = []
                for i in range(len(cqt_list)):
                    new_cqt_list.append(cqt_list[i][:, :min_time])
                cqt_list = new_cqt_list


            cqt_list = np.array(cqt_list)

            if np.all(cqt_list==0):

                log_hcqt = np.zeros(cqt_list.shape)

            else:
```

```
77                      log_hcqt = ((1.0/80.0) * librosa.core.amplitude_to_db(
78                          np.abs(cqt_list), ref=np.max)) + 1.0
79
80                  log_hcqt = log_hcqt[:,:,margen_lado:margen_lado+n_samples]
81
82
83                  if np.all(pianoroll[:,i_trozo:i_trozo+n_samples]==0):
84                      n_vacios = n_vacios + 1
85
86          return log_hcqt, i_trozo, error, n_vacios
```

La función anterior tiene como objetivo almacenar todos los armónicos de una ventana elegida aleatoriamente de una canción en concreto. Para ello necesita de varios elementos de entrada:

- **Ruta del archivo de audio**: Es necesaria para el cálculo del CQT.

- **Pianoroll**: Es necesario conocerlo por dos razones. En primer lugar se tiene que saber su longitud, ya que los archivos de audio son un poco más largos que su *pianoroll* debido a los silencios del final que no están en un archivo MIDI. Saber esta longitud nos ayudará a saber los límites en los que podemos elegir aleatoriamente una ventana para hacer el cálculo del CQT. La segunda razón es porque una vez calculado la ventana es necesario comprobar si el *pianoroll* está vacío o no.

- **Margen**: Este será el margen que dejamos para calcular un CQT de una ventana. Dentro del margen de cálculo, la ventana que queremos calcular estará situada justo en medio. Tras el cálculo se extraerá la ventana en cuestión.

- **Número de huecos vacíos y tamaño de batch**: Esta variable dará información de la cantidad de número de ventanas vacías que ya se encuentran en el *batch*, con ayuda del tamaño del *batch* se sabrá que porcentaje de este *batch* se encuentra ya con ventanas vacías. El objetivo es descartar ventanas vacías en el caso de que ya se haya llegado a dicho porcentaje.

Como se puede ver en la función hay algunas condiciones las cuales activan una variable llamada 'error'. En primer lugar si la lista de elementos 'y', que corresponde a las medidas de las amplitudes muestreada a 44100Hz por librosa es 0. En algunos casos, hay algunos archivos WAV corrompidos que no se leen correctamente y esto provoca errores durante el entrenamiento. Otro caso que activa esta variable, es si la ventana de análisis actual esta vacía y ya se ha llegado al límite de porcentaje deseado por *batch*.

Dentro de esta función, hay una parte que también merece ser explicada. Tras haber calculado las CQTs, debido a la forma que tiene librosa de hacerlas, los tamaños de cada uno de los armónicos no son exactamente iguales (se desvían en una unidad), por lo que se recorta para poder juntarlos.

De las CQTs calculados se elimina la fase y se coge el valor absoluto de la amplitud en ese momento, se cambia las unidades de amplitud a decibelios que nos aporta valor en un intervalo de [-80, 80], Debido a los valores de amplitud son menores a 1, el intervalo queda entre [-80, 0]. Posteriormente, se hace una reducción con un factor de 80 y se suma 1 con el objetivo que todos los valores queden entre [0, 1].

La función nos reporta el valor de los coeficientes CQT procesados, el índice de la ventana que se ha procesado, la variable error y el número de ventanas vacíos que hay actualmente en el batch.

6.5. Definición del entrenamiento

Se crea una función, que guarda de manera aleatoria en dos listas 32 rutas de las almacenadas anteriormente.

```
1    def get_paths_batch(paths_instrumento_stem, paths_instrumento_midi
2    , instrumento):
3
4        i_paths = np.random.randint(low=0,high=len(paths_instrumento_stem))
5        return paths_instrumento_stem[i_paths], paths_instrumento_midi[i_paths]
```

Se define una función para crear un *batch*, que tiene como objetivo devolver dos matrices de los coeficientes de la CQT y los *pianoroll* apilados listos para entrenar la red.

```
1    def load_batch_data(paths_instrumento_stem, paths_instrumento_midi,
2    instrumento, batch_size, margen, X_vacio):
3
4        batch_hcqt_train = []
5        batch_pianoroll_train = []
6
7        n_vacios = 0
8
9        while (len(batch_hcqt_train) < batch_size):
10
11           path_batch_stem, path_batch_midi
12           = get_paths_batch(paths_instrumento_stem
13           , paths_instrumento_midi, instrumento)
14
15           pm = pretty_midi.PrettyMIDI(path_batch_midi)
16           nota_min = 24
17           nota_max = 95
18           pianoroll = pm.get_piano_roll(SR/HOP_LENGTH)[nota_min:(nota_max+1), :]
19
20           hcqt, i_trozo, error, n_vacios = compute_hcqt_trozo(path_batch_stem
21           , pianoroll, margen, n_vacios, batch_size, X_vacio)
22           trozo_pianoroll = pianoroll[:,i_trozo:i_trozo+50]
23
24           if not error:
25               batch_hcqt_train.append(hcqt.transpose(1,2,0))
26
27               if trozo_pianoroll.size == 0:
28                   trozo_pianoroll = np.zeros((nota_max-nota_min+1,50))
29
30               batch_pianoroll_train.append(trozo_pianoroll)
31
32
33       print("Vacios:_" + str(n_vacios))
34       batch_hcqt_train = np.stack(batch_hcqt_train)
35       batch_pianoroll_train = np.stack(batch_pianoroll_train)
36
37       return (batch_hcqt_train, batch_pianoroll_train)
```

La siguiente función define el filtro a utilizar tras la salida de la red. El filtro guarda todos los máximos relativos y en aquellos valores que supera un umbral, coloca un 127. El valor de 127, es debido a que en un MIDI la velocidad (fuerza de la nota) se mueve en un intervalo [0, 255] y 127 es la mitad.

```
    def filtro(hcqt_pred, threshold):

        peaks = scipy.signal.argrelmax(hcqt_pred, axis=0)
        pred_fil = np.zeros(hcqt_pred.shape)
        pred_fil[peaks] = hcqt_pred[peaks]


        imidx = np.where(pred_fil >= threshold)
        relleno = np.ones(hcqt_pred.shape)*127


        pianoroll_pred = np.zeros(hcqt_pred.shape)
        pianoroll_pred[imidx] = relleno[imidx]

        return pianoroll_pred
```

La función de entrenamiento, simplemente consiste en leer un *batch* y entrenarlo mediante las funciones de Keras. Al tiempo que se van almacenando los resultados de la función de pérdida del entrenamiento y del conjunto de validación. En nuestro entrenamiento cada *epoch* equivale al entrenamiento de un *batch*.

```
    def training(model, epochs, paths_instrumento_stem, paths_instrumento_midi
    , paths_instrumento_stemV, paths_instrumento_midiV, instrumento
    , batch_size, margen,X_vacio, n):

    train_loss = []
    e_train = []


    val_loss = []
    e_val = []

    for epoch in range (1, epochs+1):


        print("<<<<<<<Epoch:_" + str(epoch) + "\\" + str(epochs) + ">>>>>>>" )
        # Leo datos para un batch


        batch_hcqt_train, batch_pianoroll_train = load_batch_data(
        paths_instrumento_stem, paths_instrumento_midi, instrumento
        , batch_size, margen, X_vacio)
        print("Load_train_DONE")

        logs = model.train_on_batch(batch_hcqt_train, batch_pianoroll_train)
        train_loss.append(logs)
        e_train.append(epoch)
        print("Train_DONE")


        if epoch == 1 or epoch  % 20 == 0:

            batch_hcqt_val, batch_pianoroll_val = load_batch_data(
            paths_instrumento_stemV, paths_instrumento_midiV, instrumento
            , batch_size, margen, 1)
            print("Load_val_DONE")
            clear_output()

            logs2 = model.test_on_batch(batch_hcqt_val, batch_pianoroll_val)
            val_loss.append(logs2)
            e_val.append(epoch)


            print("Val_DONE")

            fig = plt.figure()
            fig.set_figwidth(10)
            plt.plot( e_train, train_loss, 'b')
            plt.plot( e_val, val_loss, 'r')
            plt.ylabel('Loss_function')
            plt.xlabel('Epoch')
            plt.savefig('C:/Users/Charlie/Desktop/Nueva_carpeta/PianoFinalV3'
            + str(n) + '.jpg')
            plt.show()

            print("Loss_function_train:_" + str(logs)
            + "_//_Loss_function_val:_" + str(logs2))
            print("Epoch:_" + str(epoch) + "\\" + str(epochs) )


            batch_pianoroll_pred = model.predict(batch_hcqt_val)

            for i in range(10):

                plt.figure(epoch + i)
                plt.subplot(221)
                plt.imshow(batch_hcqt_val[i,:, :, 1])
                plt.subplot(222)
                plt.imshow(batch_pianoroll_pred[i, :, :])
```

```
                plt.subplot(223)
                plt.imshow(batch_pianoroll_val[i, :, :])
                plt.subplot(224)
                filtrada = filtro(batch_pianoroll_pred[i, :, :], 0.7)
                plt.imshow(filtrada)
                plt.show()



                suma_pred = batch_pianoroll_pred[i, :, :].sum(axis=1)
                suma_test = batch_pianoroll_val[i, :, :].sum(axis=1)
                plt.figure(epoch + i + 1)
                plt.subplot(221)
                plt.imshow(batch_pianoroll_pred[i, :, :])
                plt.subplot(222)
                plt.plot(suma_pred)
                plt.subplot(223)
                plt.imshow(batch_pianoroll_val[i, :, :])
                plt.subplot(224)
                plt.plot(suma_test)
                plt.show()


        model.save_weights(
        'C:/Users/Charlie/Desktop/Nueva_carpeta/PianoFinalV3' + str(n) + '.h5'
        )
```

Cada 20 *epochs* se mostrará una serie de gráficas como la evolución de la función de pérdida en el conjunto de entrenamiento y validación (figura 6.2) o diferentes ventanas del *batch* (figura 6.3).

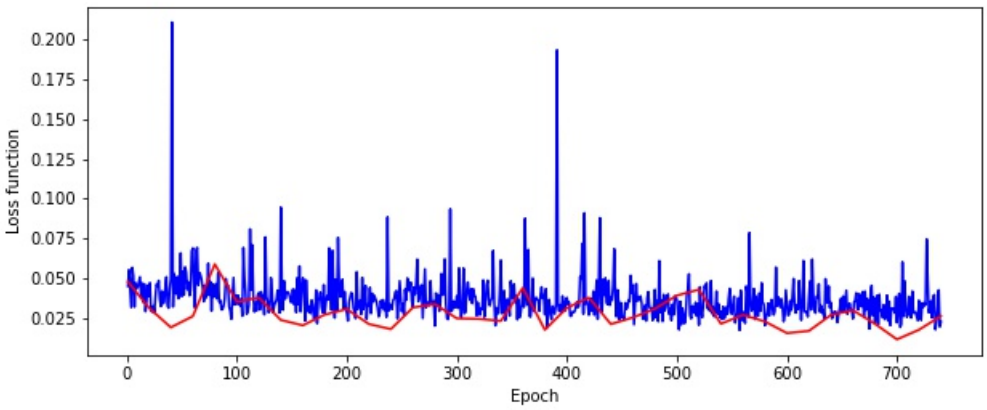

Figura 6.2: Evolución de la función de pérdida durante en el entrenamiento del conjunto de entrenamiento (Azul) y del conjunto de validación (Rojo)

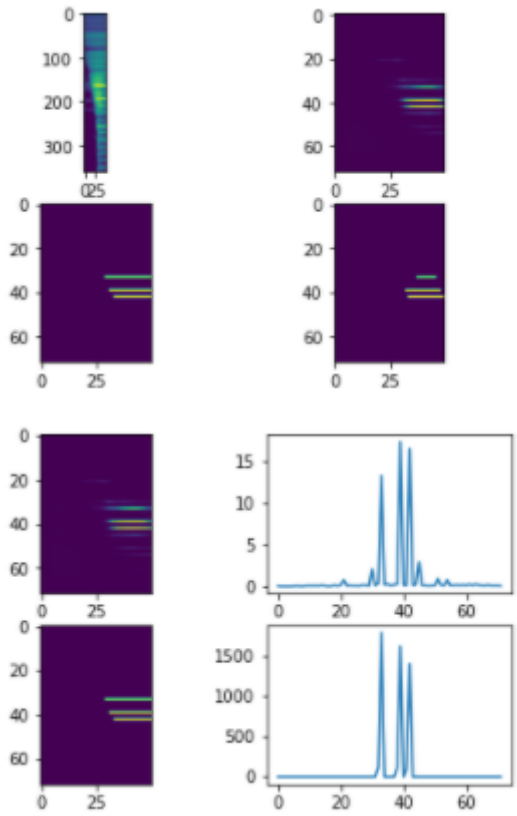

Figura 6.3: Evolución de las ventanas durante el entrenamiento. Primera fila: Primer armónico de entrada y predicción; Segunda fila: Pianoroll real y predicción filtrada; Tercera fila: Predicción y suma en el eje x de la predicción; Cuarta fila: Pianoroll real y suma en el eje x del pianoroll real

El objetivo Figura 6.3 era observar cada cierto tiempo la mejora en el entrenamiento y ver visualmente si había una mejora en la estimación de las notas musicales. La suma en el eje x de la predicción y el *pianoroll* real tenia como objetivo ver si el perfil de frecuencias se asemejaban.

6.6. Cálculo del umbral óptimo

Se definen una serie de variables que se van a utilizar como el *batch size*, la nota mínima y máxima del primer armónico, el número de intentos que corresponde al número de canciones que se utilizarán para el calculo del umbral óptimo. Se cargan los pesos del modelo a analizar y las rutas del *test* de los instrumentos que queremos comprobar.

```
\UseRawInputEncoding
    batch_size = 32
    nota_min = 24 #incluida
    nota_max = 95 #incluida
    intentos = 100
    paso = 0.02
    limite = 1

    save_path = "C:/Users/Charlie/Desktop/graficas"
    test = "E:/DATASETS/Slakh2100_wav/slakh2100_flac/test"

    weights = "C:/Users/Charlie/Desktop/Nueva_Carpeta/PianoFinalV311.h5"
```

```
13        model = model3_def()
14        model.load_weights(weights)
15
16        paths_instrumento_stemT, paths_instrumento_midiT
17        = get_paths_instrumentos(test, instrumento)
```

Se inicializan las diferentes listas donde guardamos los resultado y se crea una array de canciones aleatorias del conjunto de test con las cuales se va a hacer el cálculo.

```
1         f_scores_max = []
2         th_fscores_max = []
3
4         th_precisiones_max = []
5         precisiones_max = []
6
7         recalls_max = []
8         th_recalls_max = []
9
10        accuracys_max = []
11        th_accuracys_max = []
12
13
14        accs_chroma_max = []
15        th_accs_chroma_max = []
16
17
18        indices = np.random.randint(low=0, high=len(paths_instrumento_stemT)
19        , size=intentos)
```

En las líneas siguientes se lee el hcqt y el pianoroll de toda la canción. La función compute_hcqt_all es muy similar a la función compute_hcqt_trozo, pero analizando toda la canción, al igual que get_midi que guarda el pianoroll de una canción entera.

```
1         hcqt, _ , _ = compute_hcqt_all(song_path)
2         pianoroll = get_midi(pianoroll_path, nota_min, nota_max)
```

La función get_pred tiene como objetivo, dado un modelo de red neuronal y un HCQT (harmonics CQT que reúne todos los armónico de un CQT), devolver la predicción de cada trozo, se divide la canción entera en trozos de 50 frames y se van pasando por la red. Posteriormente, se juntan todos.

```
1         def get_pred(model, hcqt):
2
3             x = 0
4             n_trozo = 0
5             l_trozo = 50
6             prediccion1 = []
7
8             while (hcqt.shape[2] > x):
9
10                if hcqt[:,:,x:x+l_trozo].shape[2] == 50:
11
12                    trozo_pred = model.predict(
13                    hcqt[np.newaxis, :,:,x:x+l_trozo].transpose(0, 2, 3, 1)
14                    )
15
16
17                    trozo_pred = np.squeeze(trozo_pred)
18                    prediccion1.append(trozo_pred)
19
20                n_trozo = n_trozo + 1
21                x = x + l_trozo
22
23            hcqt_pred1 = np.hstack(prediccion1)
24
25            return hcqt_pred1
26
27
28        pianoroll_pred = get_pred(model, hcqt)
```

Una vez que se tiene todas las predicciones, se hace un análisis en el que se obtienen todas las métricas con la librería mir_eval. Pero esta librería necesita los datos como dos listas de

las frecuencias que están sonando y otra del tiempo al que corresponde. Esto es lo que hace la función piano_to_freqs.

Por último hay que ir analizando para cada umbral y almacenando los datos de las métricas que nos interesan (Precisión, Recall, F-score, Accuracy, Chroma Accuracy).

```
    ref_time, ref_freqs = piano_to_freqs(pianoroll)

    # Calculamos el F-score, el Recall y la Precision
    f_scores = []
    precisiones = []
    recalls = []
    accuracys = []
    accs_chroma = []

    for th in arange(paso, limite , paso):

        filtrada = filtro(pianoroll_pred, th)
        est_time, est_freqs = piano_to_freqs(filtrada)

        if len(est_time)>1:
            precision, recall, accuracy ,_,_,_,_,_, acc_chroma, _,_,_,_ =
            = mir_eval.multipitch.metrics(ref_time, ref_freqs
            , est_time, est_freqs)
        else:
            precision = 0
            recall = 0
            accuracy = 0
            acc_chroma = 0

        if precision + recall !=0:
            fscore = 2 * ((precision*recall)/(precision + recall))
        else:
            fscore = 0

        f_scores.append(fscore)
        precisiones.append(precision)
        recalls.append(recall)
        accuracys.append(accuracy)
        accs_chroma.append(acc_chroma)
```

Tras hacer el análisis completo se crean unas gráficas para ver la evolución durante toda la canción como se puede ver en las siguientes figuras.

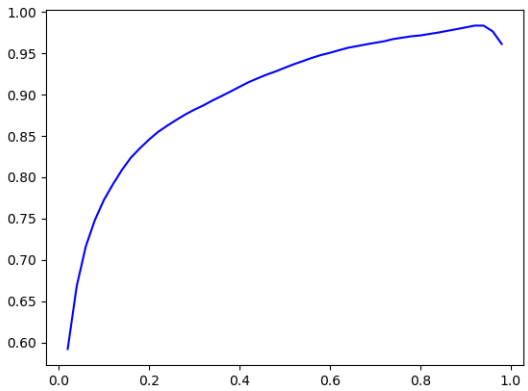

Figura 6.4: Evolución de la precisión al variar el umbral de 0 a 1 con un paso de 0.02

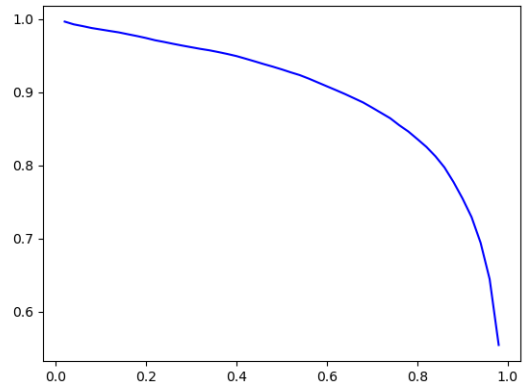

Figura 6.5: Evolución de la recall al variar el umbral de 0 a 1 con un paso de 0.02

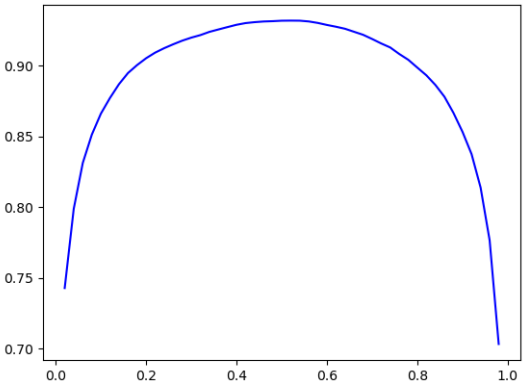

Figura 6.6: Evolución de la F-score al variar el umbral de 0 a 1 con un paso de 0.02

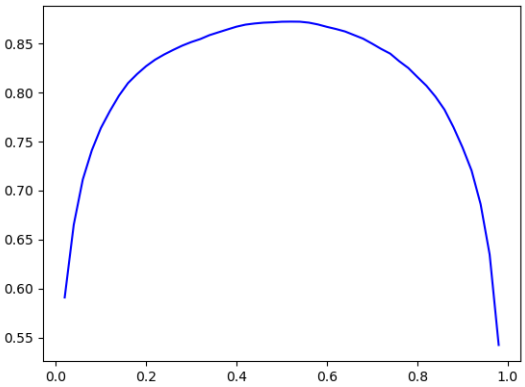

Figura 6.7: Evolución de la accuracy al variar el umbral de 0 a 1 con un paso de 0.02

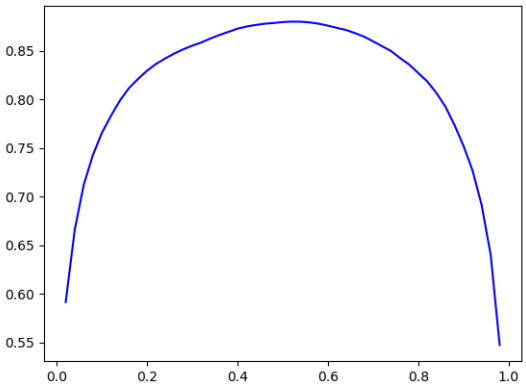

Figura 6.8: Evolución de la chroma accuracy al variar el umbral de 0 a 1 con un paso de 0.02

```
Threshold F-score máx: 0.52

F-score máx: 0.9318

Threshold Recall máx: 0.02

Recall máx: 0.9963

Threshold Precision máx: 0.92

Precision máx: 0.9834

Threshold Accuracy máx: 0.52

Accuracy máx: 0.8723

Threshold Chroma Accuracy máx: 0.52

Chroma Accuracy máx: 0.8800
```

Figura 6.9: Datos que se guardan en un fichero TXT tras finalizar el analisis de una canción

Tras hacer un análisis de 100 canciones (el tiempo de duración media esta en torno a las 2 horas), se hace una media de todas las métricas y guardamos un archivo TXT como se puede ver en la figura 6.10.

```
Threshold F-score máx: 0.44380000000000003
F-score máx: 0.8447
Threshold Recall máx: 0.029599999999999994
Recall máx: 0.9117
Threshold Precision máx: 0.7437999999999999
Precision máx: 0.8998
Threshold Accuracy máx: 0.9400000000000001
Accuracy máx: 0.8365
Threshold Chroma Accuracy máx: 0.9400000000000001
Chroma Accuracy máx: 0.8365
```

Figura 6.10: Datos que se guardan en un fichero TXT tras finalizar el análisis de todas las canciones

El dato que se utiliza como umbral y que consideramos el mejor, es aquel que maximiza el *F-score*, ya que tiene encuentra tanto la *Precision* como el *Recall*.

6.7. Tests utilizando el umbral óptimo

El procedimiento es similar al cálculo del umbral óptimo (0.44 en nuestro caso), pero en este caso la predicción solo se realiza para el valor de este umbral.

Los datos se recogen en un archivo CSV y mediante diagramas de cajas. El número de canciones que se van a analizar en el caso de un test general es de 100 canciones, pero también se van a realizar pruebas para cada instrumento en el que solo se utilizarán 30 pistas. En la figura 6.11 se puede ver un análisis de un test general:

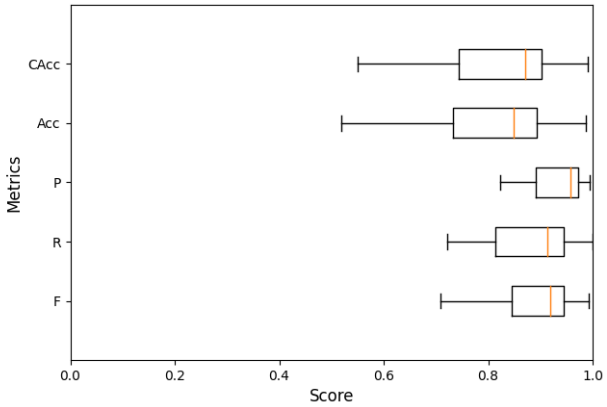

Figura 6.11: Diagrama de cajas de un test general de 100 canciones en Slakh2100

Algunos ejemplos de test para instrumentos se puede ver en las figura 6.12 (Piano) o en la 6.13 (Bajo).

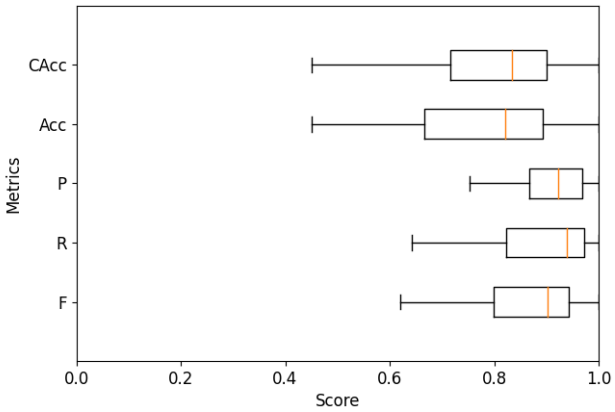

Figura 6.12: Diagrama de cajas de un test de 30 canciones de piano en Slakh2100

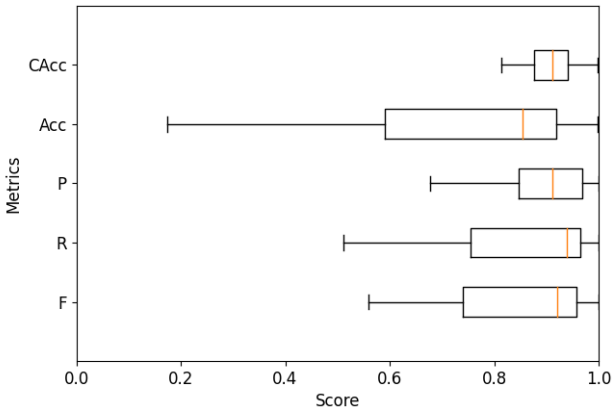

Figura 6.13: Diagrama de cajas de un test de 30 canciones de bajo en Slakh2100

En la figura 6.14, se puede observar el aspecto de este archivo. En este documento se almacena toda la información del *test*, el cual se puede utilizar para hacer gráficas sin tener que volver a realizar el procedimiento completo.

```
Nombre,Fscore,Recall,Precision,Accuracy,Chroma Accuracy
Track02060,1.0,1.0,1.0,1.0,1.0
Track02061,0.9481224384300114,0.9909694350108058,0.9088270687336306,0.9013619769727604,0.9013619769727604
Track01908,0.7763009295456519,0.978215654077723,0.643479513213797,0.6343887547919921,0.6343887547919921
Track02026,0.7794708725515137,0.738355141087737,0.8254357371838044,0.6386335688530399,0.7222343921139102
Track01928,0.8925606641123883,0.8253210648034247,0.9717281733387405,0.8059679976935274,0.8124125958431788
Track02100,0.6813882687233611,0.5233196159122085,0.9762707921367919,0.516746706070681,0.5169334975369458
Track01957,0.877908830092445,0.91571072319202,0.8431042400122455,0.7823863636363636,0.8075482569864593
Track01915,0.6273949099227909,0.7815920720253903,0.5240142435296161,0.45708333333333334,0.4867048001855144
Track01948,0.6206551915602443,0.957676490747087,0.4590931493346476,0.4499637710329281,0.4511320602691161
Track01965,0.8909343409876305,0.8230204666149532,0.9710644888763729,0.8033197437390798,0.8357147091954704
Track01910,0.766257225433526,0.7339100346020762,0.8015873015873016,0.6210834553440703,0.7136666150750658
Track02055,0.8008364800108371,0.6863897596656218,0.9610851452956716,0.6678292548609837,0.6679234625432465
Track01995,0.9430090454195535,0.9632623974050228,0.92358984025258,0.8921637799576667,0.8938016811316887
Track01968,0.7988324576765907,0.8495157685622051,0.75385632437197,0.6650466562986003,0.6655323286339329
Track01942,0.9430939226519337,0.9578740630887009,0.9287629707356885,0.8923157344485102,0.894827360589236
Track02092,0.9038589399218105,0.9457895029776813,0.8654884384231435,0.8245826863354038,0.8338820201429025
Track02069,0.925434019542738,0.9682382133995037,0.8862542018715363,0.8612165621965215,0.8838352247341614
Track01926,0.939855858154219,0.9603336664204718,0.920233139050791,0.8865359162555649,0.8918473233318586
Track02045,0.9863738051657515,0.9971217105263158,0.9758551307847082,0.9731139646869984,0.9731139646869984
Track01892,0.8423519175820374,0.8144968547446512,0.8721796814217272,0.7276407488384805,0.7626259105642884
Track02001,0.8952979493259173,0.883242462977939,0.9076870833407803,0.8104429142496948,0.8309394315962751
Track01954,0.9206727803594195,0.882250945775536,0.9625935162094763,0.8530061723691229,0.8537124561670987
Track02024,0.9003837298541827,0.8402506714413608,0.9697871461045671,0.8188163037409268,0.8230788065339816
Track01945,0.9654627067004675,0.972405715624268,0.9586181415111767,0.9332314194505643,0.9351455961114361
Track02098,0.9727319062181448,0.9720713073005094,0.9733934036042162,0.9469114363681469,0.9506213753106877
Track01903,0.9768353528153956,0.9810307802433786,0.9726756564939674,0.9547196098920236,0.9795414462081129
Track01932,0.7497682843133786,0.6432482305665829,0.8985686666892786,0.599703458891706,0.6345962574884926
Track02032,0.9417664233576643,0.9330353914464645,0.9506624029237094,0.8899419222226821,0.9267280781942198
Track02098,0.9426405378343584,0.9990903577926016,0.892228540481993,0.891504329004329,0.8920162381596752
Track01919,0.9746630727762803,0.9993367234136635,0.9511784511784511,0.9505783385909569,0.9505783385909569
```

Figura 6.14: Archivo CSV de un test de 30 canciones de piano en Slakh2100

La creación de estos datos se ha hecho mediante la librería pandas para los archivos CSV y matplotlib para los diagramas de cajas.

```
1   #CREAMOS EL CSV
2   Metrics = {'Nombre': nombres,
3              'Fscore': f_scores,
4              'Recall': recalls,
5              'Precision': precisiones,
6              'Accuracy': accuracys,
7              'Chroma_Accuracy': accs_chroma,
8            }
9
10  df = DataFrame(Metrics)
11  df.to_csv (save_path + "/" + "Metrics.csv", index = False, header=True)
12
13
14  #DIAGRAMA DE CAJAS
15  fig = plt.figure()
16  plt.axis([0, 1, 0, 6])
17  plt.boxplot(
18  (f_scores, recalls, precisiones, accuracys, accs_chroma)
19  , showfliers=False, vert=False
20  , labels=["Fscore", "Recall", "Precision", "Accuracy", "Chroma_Accuracy"]
21  )
22
23  fig.set_figwidth(15)
24  plt.rcParams.update({'font.size': 8})
25  plt.savefig("C:/Users/Charlie/Desktop/graficas/Boxplot.png")
26  plt.show()
```

6.8. Funciones para entrenar con onsets

Para detectar los *onsets* en lugar del pianoroll completo, con el objetivo de entrenar una red que se centre en los inicios de las notas y aumentar la precisión en esta tarea se utiliza la siguiente función.

```
    def piano_onset(path, fs):

    midi = pretty_midi.PrettyMIDI(path)
    instrumentos = midi.instruments


    for instrumento in instrumentos:
        notes=instrumento.notes
        onsets = [n.start for n in notes]
        pitch = [n.pitch for n in notes]


    pianoroll = np.zeros((128, int(midi.get_end_time()*fs)))
    for on,pi in zip(onsets,pitch):
        pianoroll[pi][int(on*fs)-1] = 1

    return pianoroll
```

Además, el filtro que se utiliza a la hora de entrenar la red con *onsets* también cambia y es el siguiente:

```
def filtro_onsets(hcqt_pred, threshold, velocity = 1):

    peaks0 = scipy.signal.argrelmax(hcqt_pred, axis=0)
    pred_fil0 = np.zeros(hcqt_pred.shape)
    pred_fil0[peaks0] = hcqt_pred[peaks0]

    peaks = scipy.signal.argrelmax(pred_fil0, axis=1)
    pred_fil = np.zeros(hcqt_pred.shape)
    pred_fil[peaks] = hcqt_pred[peaks]

    imidx = np.where(pred_fil >= threshold)
    relleno = np.ones(hcqt_pred.shape)*velocity


    pianoroll_pred = np.zeros(hcqt_pred.shape)
    pianoroll_pred[imidx] = relleno[imidx]

    return pianoroll_pred
```

6.9. Note tracking

El algoritmo de *note tracking*, se basa en métodos simples con reglas basadas en condiciones. Este algoritmo se aprovecha de la información aportada por la red de *frames* y *onsets*. Se divide en las 3 partes ya comentadas en el apartado 5.3:

- **Eliminar onsets vacíos**:

```
    def elimina_onsets_vacios(onsets, pianoroll):


        onset_mal = []


        for i in range(len(onsets)):

                n_frames = 0

                if onsets[i][1] > 5:

                    for k in range(onsets[i][1], onsets[i][1]+15):

                        if pianoroll[onsets[i][0]][k]==1:
                            n_frames += 1


                    if n_frames<4:

                        onset_mal.append(i)


        for ind in sorted(onset_mal, reverse=True):

            onsets.pop(ind)


        return onsets
```

– **Crear onsets**:

```
1
2
3           def AbreHuecoOnsets(pianoroll, notas):
4
5             for i in range(len(notas)):
6
7                   if notas[i][1] > 2:
8
9                       pianoroll[notas[i][0]][notas[i][1]-1] = 0
10                      pianoroll[notas[i][0]][notas[i][1]-2] = 0
11
12
13            return pianoroll
```

– **Rellenar huecos**:

```
1         def rellenaHuecos(pianoroll, notas):
2
3             notas_completas = []
4
5             for pitch in range(128):
6
7                 mismo_pitch = []
8
9                 for i in range(len(notas)-1):
10
11
12                    if pitch == notas[i][0]:
13                        mismo_pitch.append(notas[i])
14
15
16
17                for k in range(len(mismo_pitch)-1):
18
19                    offset = get_offset_pitch(mismo_pitch[k][1], mismo_pitch[k+1][1], pitch, pianoroll)
20                    notas_completas.append((mismo_pitch[k][1], offset, mismo_pitch[k][0]))
21
22
23            return notas_completas
```

Uniendo las funciones anteriores, se obtiene la función total del algoritmo:

```
1   def note_tracking(pianoroll, pianorollOnsets):
2
3       notas = get_onsets_pianoroll2(pianorollOnsets)
4       notas = elimina_onsets_vacios(notas, pianoroll)
5       pianorollOnsets = actualiza_pianorollOnsets(pianorollOnsets, notas)
6
7       pianoroll = AbreHuecoOnsets(pianoroll, notas)
8
9       notas_enteras = rellenaHuecos(pianoroll, notas)
10
11
12      return pianoroll, pianorollOnsets, notas_enteras
```

Capítulo 7

Conclusiones y Líneas futuras

7.1. Conclusiones

El objetivo de este trabajo era crear un red neuronal profunda que permita transcribir música polifónica para varios instrumentos a la vez (multitímbrica). Para ello, partiendo del estudio del estado del arte se ha implementado desde cero una red neuronal basada en Deep Salience a la que se la han introducido una serie de modificaciones:

- **Adición de una última capa**: Nuestro modelo tiene una capa que reduce las dimensiones de (360,50) a (72,50), con el objetivo de hacer un entrenamiento para archivos MIDI en un intervalo de notas de C1 a C7 (72 notas).
- **Entrenamiento con Slakh2100**: Esta red se ha entrenado con un *dataset* distinto, el Slakh2100. En el caso de Deep Salience, la base de datos se componía de 108 canciones de MedleyDB y 132 canciones de música pop occidental.
- **Algoritmo de *note tracking***: En nuestro modelo se entrenan dos redes con la misma arquitectura, pero una con el *pianoroll* de las notas completas y otra que solo detecta *onsets*. Las predicciones de ambas redes son utilizadas por un algoritmo de *note tracking* que es el que nos reporta los mejores resultados.

Se han realizado los siguientes experimentos:

- **Entrenamiento con todos los intrumentos afinados de Slakh2100**: Un entrenamiento general con las notas completas de todos los instrumentos que pueden hacer sonar notas musicales, un ejemplo de instrumento que no esta afinado sería el tambor.
- **Entrenamiento de redes con instrumentos separados**: Entrenamiento de redes con notas completas para instrumentos individuales, con el objetivo de ver si existe una mejora cuando se especializa una red a un instrumento en concreto.

- **Entrenamiento de una red de *onsets* para los instrumentos afinados de Slakh2100**: Al igual que en el primer experimento se ha entrenado una red con todos los instrumentos, pero con la salvedad que se entrena únicamente para los inicios de notas.

- **Entrenamiento de una red de *onsets* para piano con un CQT y un espectrograma de Mel**: Se ha entrenado una red de *onsets* únicamente para piano en el que la entrada es un CQT y otra red utilizando otra forma de procesamiento de audio como sería el espectrograma de Mel. Este experimento sirve para ver que forma de procesar el audio es la mejor.

- **Algoritmo de note tracking**: Entrenamiento de una red de *frames* especializada en piano y otra red de *onsets* de piano. Posteriormente, se procesan los resultados de ambas redes con un algoritmo de *note tracking*.

Los resultados obtenidos son los siguientes:

- **Gran diferencia entre tipos de timbre**: En el primer experimento se ve que para algunos instrumentos como los de percusión cromática o instrumentos de metal, las métricas multi-F0 son muy bajas. Por el contrario, para instrumentos como el bajo y piano, las métricas son mucho mejores.

- **Mejora en la especialización de una red por instrumento**: Al hacer el entrenamiento para redes entrenadas con un instrumento, las mejoras de estos mejoraron considerablemente.

- **Mejora de la detección de *onsets***: Existe una gran diferencia en las métricas de inicios de nota, cuando se entrena una red que se especializa únicamente con *onsets*.

- **La CQT es la mejor forma para procesar el audio de entrada**: Como se ve en el cuarto experimento, la CQT sigue siendo la mejor forma para procesar el audio.

- **Mejora al aplicar el algoritmo de *note tracking***: En el último experimento, la mejora de las métricas de transcripción con respecto a una red entrenada únicamente con los *frames* es abismal. Ya no solo se puede ver la mejora en las métricas, sino que de manera auditiva la mejora es clara.

Al hacer una comparación con el modelo de Deep Salience, observamos que nuestro modelo tiene una gran mejora en el *dataset* Bach10. Aunque en el MedleyDB sigue siendo mejor en su caso, pero esta diferencia también puede ser provocada debido a que su modelo ha sido entrenado con pistas de esta misma base de datos.

Al observar que la red dependía mucho del timbre, se decidió entrenar redes para cada instrumento y se observó que hubo mejoría con respecto a la red que se entrenó con todos los instrumentos al mismo tiempo. Debido a que las métricas de transcripción no eran los suficientemente buenas, se tuvo que entrenar una red especializada en inicios de notas. Como resultado final, se entrenaron dos redes especializadas en Piano, pero una detecta *frames* y la otra *onsets*. Junto con un algoritmo de *note tracking* se consiguieron los mejores resultados, produciendo archivos MIDI a partir de un WAV que se asemeja mucho a los originales.

Se ha desarrollado un modelo que se puede generalizar para otro tipo de instrumentos musicales. La calidad obtenida es similar en el piano al estado del arte y se ha puesto de manifiesto la dependencia con el timbre de este tipo de redes profundas para la transcripción automática. Este último hecho es el que ha dado lugar a la publicación [6].

7.1.1. Líneas Futuras

Un posibilidad de continuación de este trabajo, es entrenar las mismas redes desarrolladas para otros instrumentos que no sean el piano y analizar su funcionamiento. Como próximo objetivo, hay que profundizar en la posibilidad de generar una única red que se entrene con más de un instrumento (o con, al menos) una familia de instrumentos y obtenga unos niveles aceptables de calidad de las transcripciones.

Para mejorar esta red, se podría plantear una red que concatene las dos redes (*frames* y *onsets*) y se entrenen al mismo tiempo sin necesidad de utilizar un algoritmo de *note tracking*, como la estrategia que utiliza OaF de Magenta. Debido a la relación temporal que existe en la música este nuevo modelo podría estar formado de módulos LSTM, además de capas convolucionales.

Para un futuro más lejano, una meta es crear una red que a partir de una melodía consiga transcribir por separado las pistas de cada uno de los instrumentos que están sonando. Para ello, sería necesario una red que separe las pistas y posteriormente se transcriban.

Capítulo 8

Bibliografía

[1] Este es el estado actual de la inteligencia artificial a nivel mundial, según el ai index report 2019. https://www.xataka.com/inteligencia-artificial /este-estado-actual-inteligencia-artificial-a-nivel-mundial-ai-index-report-2019.

[2] The 2019 ai index report. https://hai.stanford.edu/ai-index-2019.

[3] François Chollet et al. Keras. `https://keras.io`, 2015.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] Judith Brown. Calculation of a constant q spectral transform. *Journal of the Acoustical Society of America*, 89(1):425–434, January 1991.

[6] Carlos Hernandez-Olivan, Ignacio Zay Pinilla, Carlos Hernandez-Lopez, and Jose R. Beltran. A comparison of deep learning methods for timbre analysis in polyphonic automatic music transcription. *Electronics*, 10(7), 2021.

[7] Dixon S. Giannoulis D. et al. Benetos, E. Automatic music transcription: challenges and future directions. *J Intell Inf Sys*, 41:407–434, 2013.

[8] E. Benetos, S. Dixon, Z. Duan, and S. Ewert. Automatic music transcription: An overview. *IEEE Signal Processing Magazine*, 36(1):20–30, 2019.

[9] Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C. Van Essen, Abdul A. S. Awwal, and Vijayan K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019.

[10] Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[12] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014. cite arxiv:1412.3555Comment: Presented in NIPS 2014 Deep Learning and Representation Learning Workshop.

[13] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. Transforming auto-encoders. In *ICANN*, 2011.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[16] Premio turing 2018: La revolución de las redes neuronales artificiales. https://users.dcc.uchile.cl/ jperez/difusion/turing-award-2018.html.

[17] Aprendizaje profundo fácil de entender. https://programmerclick.com /article/66021436698/.

[18] Función de activación - redes neuronales. https://www.diegocalvo.es /funcion-de-activacion-redes-neuronales/.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[20] Colin Raffel, Brian McFee, Eric J. Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, and Daniel P. W. Ellis. Mir eval: A transparent implementation of common mir metrics. In Hsin-Min Wang, Yi-Hsuan Yang, and Jin Ha Lee, editors, *Proceedings of the 15th International Society for Music Information Retrieval Conference*, pages 367–372, 2014.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Beth Logan. Mel frequency cepstral coefficients for music modeling. In *In International Symposium on Music Information Retrieval*, 2000.

[23] Pooling layer — short and simple. https://ai.plainenglish.io/pooling-layer-beginner-to-intermediate -fa0dbdce80eb.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[25] Introduccion al modelo lstm. https://programmerclick.com/article/1367398979/.

[26] Brian McFee, Matt McVicar, Stefan Balke, Carl Thomé, Colin Raffel, Dana Lee, Oriol Nieto, Eric Battenberg, Dan Ellis, Ryuichi Yamamoto, Josh Moore, Rachel Bittner, Keunwoo Choi, Pius Friesch, Fabian-Robert Stöter, Vincent Lostanlen, Siddhartha Kumar, Simon Waloschek, Seth Kranzler, Rimvydas Naktinis, Douglas Repetto, Curtis "Fjord" Hawthorne, CJ Carr, Waldir Pimenta, Petr Viktorin, Paul Brossier, Jo ao Felipe Santos, Jackie Wu, Erik Peterson, and Adrian Holovaty. librosa/librosa: 0.6.1, May 2018.

[27] Matthias Mauch and Simon Dixon. Pyin: A fundamental frequency estimator using probabilistic threshold distributions. In *ICASSP*, pages 659–663. IEEE, May 2014.

[28] Curtis Hawthorne, Erich Elsen, Jialin Song, Adam Roberts, Ian Simon, Colin Raffel, Jesse Engel, Sageev Oore, and Douglas Eck. Onsets and frames: Dual-objective piano transcription. In *Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018, Paris, France, 2018*, 2018.

[29] Jong Wook Kim, Justin Salamon, Peter Li, and Juan Pablo Bello. Crepe: A convolutional representation for pitch estimation. *CoRR*, abs/1802.06182, 2018.

[30] Abiodun Ogunyemi, David Lamas, Jan Stage, and Marta Lárusdóttir. Assessment model for HCI practice maturity in small and medium sized software development companies. In Antònia Mas, Antoni Lluís Mesquida, Rory V. O'Connor, Terry Rout, and Alec Dorling, editors, *Software Process Improvement and Capability Determination - 17th International Conference, SPICE 2017, Palma de Mallorca, Spain, October 4-5, 2017, Proceedings*, volume 770 of *Communications in Computer and Information Science*, pages 55–69. Springer, 2017.

[31] Rachel M. Bittner, Brian McFee, Justin Salamon, Peter Li, and Juan P. Bello. Deep salience representations for f0 estimation in polyphonic music. In Sally Jo

Cunningham, Zhiyao Duan, Xiao Hu, and Douglas Turnbull, editors, *Proceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017*, Proceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017, pages 63–70. International Society for Music Information Retrieval, 2017. 18th International Society for Music Information Retrieval Conference, ISMIR 2017 ; Conference date: 23-10-2017 Through 27-10-2017.

[32] Alain Cheveigné and Hideki Kawahara. Yin, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111:1917–30, 05 2002.

[33] Justin Salamon, Rachel Bittner, J. Bonada, Juan José Bosch, Emilia Gómez, and Juan Pablo Bello. An analysis/synthesis framework for automatic f0 annotation of multitrack datasets. In *18th International Society for Music Information Retrieval Conference*, Suzhou, China, 23/10/2017 2017.

[34] Matthias Mauch, Chris Cannam, Rachel Bittner, George Fazekas, Justin Salamon, Jiajie Dai, Juan Bello, and Simon Dixon. Computer-aided melody note transcription using the tony software: Accuracy and efficiency. 05 2015.

[35] Parker Lamb, Alexander Millar, and Ramon Fuentes. Swipe dynamics as a means of authentication: results from a bayesian unsupervised approach, 2020.

[36] Rachel Bittner, Justin Salamon, Mike Tierney, Matthias Mauch, Chris Cannam, and Juan Bello. Medleydb: A multitrack dataset for annotation-intensive mir research. In *in Proc. the 15th International Society for Music Information Retrieval Conference (ISMIR*, 2014.

[37] Zhiyao Duan, Bryan Pardo, and Changshui Zhang. Multiple fundamental frequency estimation by modeling spectral peaks and non-peak regions. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(8):2121–2133, 2010.

[38] Li Su and yi-hsuan Yang. Escaping from the abyss of manual annotation: New methodology of building polyphonic datasets for automatic music transcription. volume 9617, pages 309–321, 09 2016.

[39] M. Mueller and F. Wiering, editors. *An efficient temporally-constrained probabilistic model for multiple-instrument music transcription*, Malaga, Spain, October 2015. ISMIR.

[40] Zhiyao Duan, B. Pardo, and C. Zhang. Multiple fundamental frequency estimation by modeling spectral peaks and non-peak regions. *IEEE Transactions on Audio, Speech, and Language Processing*, 18:2121–2133, 2010.

[41] Valentin Emiya, Nancy Bertin, Bertrand David, and Roland Badeau. Maps - a piano database for multipitch estimation and automatic transcription of music, 07 2010.

[42] Rainer Kelz, Matthias Dorfer, Filip Korzeniowski, Sebastian Böck, Andreas Arzt, and Gerhard Widmer. On the potential of simple framewise approaches to piano transcription. *CoRR*, abs/1612.05153, 2016.

[43] S. Sigtia, E. Benetos, and S. Dixon. An end-to-end neural network for polyphonic piano music transcription. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(5):927–939, 2016.

[44] Melodyne. https://www.celemony.com/es/melodyne/what-is-melodyne.

[45] Ethan Manilow, Gordon Wichern, Prem Seetharaman, and Jonathan Le Roux. Cutting music source separation some Slakh: A dataset to study the impact of training data quality and quantity. In *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*. IEEE, 2019.

[46] Colin Raffel and Daniel P. W. Ellis. Intuitive analysis, creation and manipulation of midi data with pretty_midi, 2014.

[47] Brian McFee, Christopher Jacoby, and Eric Humphrey. pescador, March 2017.

[48] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed ¡today¿].

[49] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[50] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[51] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[52] Melody extraction from polyphonic music signals using pitch contour characteristics. *IEEE Transactions on Audio, Speech and Language Processing*, 20:1759–1770, 08/2012 2012.

[53] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[54] Jean-Louis Durrieu, Alexey Ozerov, Cédric Févotte, Gaël Richard, and Bertrand David. Main instrument separation from stereophonic audio signals using a source/filter model. 01 2009.

[55] A. Ozerov, P. Philippe, F. Bimbot, and R. Gribonval. Adaptation of bayesian models for single-channel source separation and its application to voice/music separation in popular songs. *Trans. Audio, Speech and Lang. Proc.*, 15(5):1564–1578, July 2007.

[56] Jean-Pierre Briot, Gaëtan Hadjeres, and Francois Pachet. Deep learning techniques for music generation - a survey. 09 2017.

Lista de Figuras

Lista de Tablas