Escuela Politécnica Superior de Linares

Grado en Ingeniería de Tecnologías de Telecomunicación

UNIVERSIDAD DE JAÉN

Escuela Politécnica Superior de Linares

Trabajo Fin de Grado

VOCAL BOOST: SOFTWARE PARA PC PARA AJUSTAR EL VOLUMEN DE LOS DIÁLOGOS EN TIEMPO REAL USANDO INTELIGENCIA ARTIFICIAL

Alumno: Almagro Martos, Bernardo

Tutor: Prof. D. VERA CANDEAS, PEDRO

Depto.: Ing. Telecomunicación

Tutor: Prof. D. CABAÑAS MOLERO, PABLO

Depto.: Ing. Telecomunicación

Septiembre,2024

UNIVERSIDAD DE JAÉN

Escuela Politécnica Superior de Linares

Grado en Ingeniería de Tecnologías de la Telecomunicación

Trabajo Fin de Grado

VOCALBOOST: SOFTWARE PARA PC PARA AJUSTAR EL VOLUMEN DE LOS DIÁLOGOS EN TIEMPO REAL

Alumno: Almagro Martos, Bernardo

Tutor: Prof. D. VERA CANDEAS, PEDRO

Depto.: Ing. Telecomunicación

Tutor: Prof. D. CABAÑAS MOLERO, PABLO

Depto.: Ing. Telecomunicación

CABAÑAS MOLERO PABLO ANTONIO - 77342099E

Firmado digitalmente por CABAÑAS MOLERO PABLO ANTONIO - 77342099E
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-77342099E, givenName=PABLO ANTONIO, sn=CABAÑAS MOLERO, cn=CABAÑAS MOLERO PABLO ANTONIO - 77342099E
Fecha: 2024.09.06 12:59:35 +02'00'

ALMAGRO MARTOS BERNARDO - 53911256S

Firmado digitalmente por ALMAGRO MARTOS BERNARDO - 53911256S
Fecha: 2024.09.06 13:23:07 +02'00'

VERA CANDEAS PEDRO - 26228400M

Digitally signed by VERA CANDEAS PEDRO - 26228400M
Date: 2024.09.06 12:56:16 +02'00'

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

DEFINICIONES Y ACRÓNIMOS

DEFINICIONES

- Inferencia: La inferencia en el contexto de los modelos de aprendizaje profundo se refiere al proceso de utilizar un modelo entrenado para hacer predicciones o generar resultados a partir de nuevos datos.

- Época: se refiere a un ciclo completo a través del conjunto de datos de entrenamiento. Es decir, una época ocurre cuando el algoritmo de entrenamiento ha pasado por todos los ejemplos en el conjunto de datos de entrenamiento una vez.

- Stems: pistas individuales que componen una mezcla de audio completa. Cada stem representa una fuente de audio aislada, como la voz, la guitarra, el bajo, o la batería, que cuando combinan forman la mezcla completa de una canción.

- Overhead: costos extras o esfuerzo adicional que se necesita para realizar una tarea más allá de las operaciones básicas requeridas (administración de memoria, sincronización, control de flujo, paralelización, cálculos adicionales).

- Batch: se refiere cuando el modelo empleado procesa un solo ejemplo de entrenamiento a la vez.

GLOSARIO DE ACRÓNIMOS

TFG	*Final Degree Project*	(Trabajo Fin de Grado)
SDR	*Signal-to-Distortion Ratio*	(Relación Señal a Distorsión)
SAR	*Signal-to-Artifact Ratio*	(Relación Señal a Artefacto)
SIR	*Signal-to-Interference Ratio*	(Relación Señal a Interferencia)
NMF	*Non-negative Matrix Factorization*	(Factorización de Matrices No Negativas)
DNN	*Deep Neuronal Network*	(Redes Neuronales Profundas)

ML	*Machine Learning*	(Aprendizaje Automático)
YAML	*YAML Ain't Markup Language*	(YAML No es un Lenguaje de Marcado)
GUI	*Graphical User Interface*	(Interfaz Gráfica de Usuario)
DFT	*Discrete Fourier Transform*	(Transformada de Fourier Discreta)
STFT	*Short-Time Fourier Transform*	(Transformada de Fourier en Tiempo Corto)
ISTFT	*Inverse Short-Time Fourier Transform*	(Transformada Inversa de Fourier en Tiempo Corto)
CNN	*Convolutional Neuronal Network*	(Red Neuronal Convolucional)
RNN	*Recurrent Neuronal Network*	(Red Neuronal Recurrente)
LSTM	*Long Short-Term Memory*	(Memoria a Corto-Largo Plazo)
GRU	*Gate Recurrent Units*	(Unidades Recurrentes Cerradas)
SGD	*Stochastic Gradient Descent*	(Descenso de Gradiente Estocástico)
ReLU	*Rectified Linear Unit*	(Unidad Lineal Rectificada)
GELU	*ºGaussian Error Linear Unit*	(Unidad Lineal de Error Gaussiano)
PESQ	*Perceptual Evaluation of Speech Quality*	(Evaluación Perceptual de la Calidad del Habla)
STOI	*Short-Time Objective Intelligibility*	(Inteligibilidad Objetiva a Corto Plazo)
MSS	*Musical Source Separation*	(Separación de Fuentes Musicales)
GT	*Ground Truth*	(Valor Verdadero)
IDC	*Instant Dialogue Cleaner*	(Limpiador de Diálogos Instantáneo)

DNN-DS	*Deep Neuronal Network Dialog Separation*	(Separación de Diálogos en Redes Neuronales Profundas)
PSD	*Power Spectral Density*	(Densidad Espectral de Potencia)
DL	*Deep Learning*	(Aprendizaje Profundo)
IA	*Artificial Intelligence*	(Inteligencia Artificial)
ADM	*Audio Definition Model*	(Modelo de Definición de Audio)
ADR	*Azimuth Discrimination and Resynthesis*	(Resíntesis y Discriminación de Azimut)
CPU	*Central Processing Unit*	(Unidad Central de Procesamiento)
GPU	*Graphics Processing Unit*	(Unidad de Procesamiento Gráfico)
DEMUCS	*Deep Extractor for Music Sources*	(Extractor Profundo de Fuentes Musicales)
IDE	*Integrated Development Environment*	(Entorno de Desarrollo Integrado)
RAM	*Random Access Memory*	(Memoria de Acceso Aleatorio)

1. RESUMEN

La mejora de diálogos o 'Dialog Enhancement' es uno de los principales desafíos hoy en día debido a la baja inteligibilidad de los diálogos hablados en películas o programas de televisión, y todo ello como consecuencia del elevado volumen de la música de fondo o los efectos de sonido. Este proceso ha sido objeto de investigación debido a su valor en sistemas de comunicación y dispositivos de audición. Todo ello y gracias al avance de la inteligencia artificial, y específicamente de las redes neuronales profundas, el campo ha progresado significativamente en los últimos años. Las redes neuronales permiten modelos que distinguen entre voz y ruido, mejorando la calidad del habla en diversos entornos, para que la audiencia pueda sumergirse en una mezcla de voz y otros sonidos equilibrada.

En el presente Trabajo Fin de Grado (TFG) se pretende utilizar el concepto de separación de fuentes musicales, pero exclusivamente en la extracción o separación de la voz del resto de acompañamiento. En este caso de separación de fuentes sonoras se ha empleado DEMUCS como una versión de última generación para algoritmos de redes neuronales profundas que se encarga de separar o extraer señales individuales de un archivo de audio compuesto por múltiples fuentes sonoras superpuestas como son: bajo, batería, voz y resto de acompañamiento. Sobre la separación de voz es sobre la que se va a trabajar, permitiendo realizar el procesamiento de separación de voz en tiempo real con bajo retardo que es uno de los aspectos más importantes a destacar en este TFG.

La finalidad del mismo es ofrecer al usuario, mediante una aplicación de escritorio elaborada en Python, realizar una modificación o ajuste de la voz separada del resto de sonidos, permitiendo también modificar el volumen del fondo y el volumen global, para establecer la sonorización adecuada que desee el usuario. El usuario seleccionará el flujo de audio que desea ajustar a través de un menú desplegable. Al realizar esta selección se podrá activar el procesamiento en el que se ejecutará internamente todo el enrutamiento necesario entre la interfaz gráfica y el modelo DEMUCS, el cual ha sido adaptado para procesar audio en tiempo real con baja latencia. Este proceso permitirá que el audio se reproduzca en los altavoces, facilitado por el enrutamiento y enlace interno de dispositivos de audio virtuales. Además, el modelo DEMUCS se configurará para capturar el audio de entrada con la mezcla completa y generar la salida con las fuentes de audio separadas, garantizando una experiencia de audio óptima y personalizada para el usuario. Además, se evaluarán métricas objetivas como Signal-to-Distortion Ratio (SDR), Signal-to-Artifact Ratio (SAR) y Signal-to-Interference Ratio (SIR) para medir la efectividad y calidad de la

separación de fuentes de audio lograda por el modelo híbrido original de DEMUCS y el modelo de DEMUCS con las modificaciones realizadas para su ejecución en tiempo real.

Cabe resaltar que la aplicación desarrollada está pensada para aquellas personas que presentan problemas de audición, o situaciones en los que haya ambientes ruidosos que dificulten la audición de la película, serie o vídeo que emplee el diálogo en la misma. Este TFG, no solo contribuye al desarrollo tecnológico en este campo, sino que también tiene el potencial de mejorar la calidad de vida de los usuarios al proporcionar una experiencia de audio más clara y agradable.

2. ABSTRACT

Dialog enhancement is one of today's major challenges due to the low intelligibility of spoken dialogues in movies or television programs, which is often caused by high background music volume or loud sound effects. This process has been a subject of research due to its value in communication systems and hearing devices. With the advancement of artificial intelligence, specifically deep neural networks, the field has significantly progressed in recent years. Neural networks enable models to distinguish between voice and noise, improving speech quality in various environments, allowing the audience to immerse themselves in an atmosphere with balanced background music and sound or according to their needs.

In this final degree project (TFG), the concept of musical source separation is used exclusively to extract or separate the voice from the rest of the accompaniment. In this case, DEMUCS is employed as a state-of-the-art version for deep neural network algorithms that handles the separation or extraction of individual signals from an audio file composed of multiple overlapping sound sources, such as bass, drums, and voice from the rest of the accompaniment. This project focuses on enabling real-time voice separation with low latency, which is one of the most important aspects highlighted in this TFG.

The aim of this project is to allow users, through a graphical interface developed in Python, to modify or adjust the separated voice from the rest of the accompaniment, also allowing the adjustment of background and overall volume to achieve the desired sound balance. The user will select the audio stream they wish to adjust through a dropdown menu. Upon selection, the necessary routing between the graphical interface and the DEMUCS model will be executed internally, which has been adapted to process audio in real-time with low latency. This process will allow the audio to be played through speakers, facilitated by the internal routing and linking of virtual audio devices. Additionally, the DEMUCS model will be configured to capture the input audio with the complete mix and generate the output with the separated audio sources, ensuring an optimal and personalized audio experience for the user. Additionally, computational metrics such as Signal-to-Distortion Ratio (SDR), Signal-to-Artifact Ratio (SAR), and Signal-to-Interference Ratio (SIR) will be evaluated to measure the effectiveness and quality of the audio source separation achieved by the original hybrid Demucs model and the modified DEMUCS model.

It is worth noting that this project focuses on people with hearing difficulties or situations with noisy environments that make it hard to hear the dialogue in movies, series,

or videos. This TFG not only contributes to technological development in this field but also has the potential to improve the quality of life for users by providing a clearer and more pleasant audio experience.

3. INTRODUCCIÓN

La separación de fuentes de audio ha emergido como un campo crucial en la investigación debido a sus numerosas aplicaciones prácticas. Desde mejorar la calidad del habla en entornos ruidosos hasta la producción musical y la asistencia en dispositivos auditivos, la capacidad de extraer señales de audio individuales a partir de una mezcla ha demostrado ser esencial. En este TFG, se aborda específicamente la dificultad de ajustar el volumen de los diálogos en tiempo real, una necesidad evidente en películas, series y videos donde los diálogos pueden ser ensombrecidos por música de fondo o efectos sonoros elevados.

El principal problema que este TFG pretende abordar o resolver es la falta de claridad e inteligibilidad en los diálogos dentro de medios audiovisuales, especialmente en contextos donde el sonido de fondo o la música interfieren con la inteligibilidad del habla. Este problema es especialmente relevante en películas, series y vídeos, donde la mezcla de audio no siempre permite la inteligibilidad del diálogo sobre otros elementos sonoros. La incapacidad de distinguir claramente los diálogos afecta a la experiencia del usuario, reduciendo su comprensión del contenido y, en algunos casos, su disfrute. Para abordar este problema, se busca desarrollar una herramienta que permita a los usuarios ajustar el volumen de los diálogos en tiempo real, mejorando así la claridad y la comprensión del contenido audiovisual. Esta herramienta tiene el potencial de beneficiar a una amplia audiencia, incluyendo a personas mayores, individuos con pérdida auditiva y hablantes no nativos que pueden tener dificultades para comprender los diálogos en un idioma extranjero.

La motivación principal de este trabajo es mejorar la experiencia auditiva de los usuarios mediante una solución que les permita ajustar la voz con respecto al resto de sonidos según sus necesidades en tiempo real, mientras están reproduciendo cualquier plataforma o medio audiovisual. Esta necesidad surge de la observación de que algunos programas de TV o plataformas de streaming, como por ejemplo Netflix, no priorizan la inteligibilidad del diálogo, a menudo quedando estos casi silenciados bajo capas de música y efectos sonoros (Suárez, 2023). Al proporcionar una herramienta que permita a los usuarios personalizar la mezcla de audio según sus preferencias, se espera mejorar significativamente la accesibilidad y la calidad de vida de diversos grupos de usuarios.

Además, este TFG aborda un desafío técnico significativo: la necesidad de realizar la separación de fuentes en tiempo real. Las soluciones actuales a menudo no son adecuadas para aplicaciones en vivo o interactivas debido a la alta latencia y la necesidad

de procesamiento intensivo (cantidad significativa de recursos computacionales.) Implementar una solución eficiente y efectiva en tiempo real requiere un enfoque innovador que optimice tanto el software como el hardware, asegurando que se realice de manera rápida y precisa. Desde una perspectiva técnica, este trabajo también busca explorar y extender los límites de la tecnología de las redes neuronales profundas en el campo de procesamiento de audio. Al adaptar y optimizar modelos avanzados como DEMUCS, en su versión híbrida, para el procesamiento en tiempo real, se pueden abrir nuevas posibilidades en diversas áreas, como la asistencia auditiva, la producción musical y la realidad aumentada. La capacidad de separar y ajustar dinámicamente los niveles de audio en tiempo real no solo tiende a mejorar la experiencia del usuario, sino también tiene implicaciones significativas para el desarrollo de tecnologías futuras en el ámbito del audio digital.

Históricamente, la separación de fuentes de audio ha sido un desafío técnico significativo. Los primeros intentos de abordar este problema se basaron en métodos como la factorización de matrices no negativas (NMF) y técnicas de filtrado en el dominio tiempo-frecuencia (Lee, y otros, 1999), (Smaragdis, y otros, 2003).Aunque estos métodos lograron cierto éxito, su capacidad para manejar mezclas complejas con múltiples fuentes de audio fue limitada. La aparición de las redes neuronales profundas (DNN) ha marcado un punto de inflexión en el campo de la separación de fuentes de audio. Los modelos de DNN, como los de Wave-U-Net y Spleeter, han mostrado una capacidad impresionante para separar instrumentos musicales y voces con una alta precisión (Stoller, y otros, 2018), (Hennequin, y otros, 2020). Wave-U-Net, por ejemplo, utiliza una arquitectura convolucional que procesa el audio directamente en el dominio del tiempo, permitiendo una separación detallada de las fuentes (Stoller, y otros, 2018). Por otro lado, Spleeter, desarrollado por Deezer, emplea una red de convolución profunda que ha sido entrenada específicamente para la separación de diferentes componentes en mezclas musicales, ofreciendo resultados de alta calidad.

Sin embargo, a pesar de los avances significativos que estos modelos representan, no están diseñados para funcionar en aplicaciones que necesitan tiempo real. Esto significa que procesan el audio en segmentos largos y no pueden cumplir con restricciones de baja latencia necesarias para aplicaciones en tiempo real.

Para resolver el problema de la separación de fuentes en el caso de la voz y el fondo en tiempo real, se ha optado por utilizar DEMUCS debido a su efectividad y calidad en la separación de fuentes. DEMUCS, desarrollado por Facebook Research, es un modelo de red neuronal profunda que es entrenada específicamente para la tarea de separar

componentes de audio en mezclas musicales complejas (Défossez, y otros, 2021). Su capacidad para manejar múltiples fuentes y su diseño basado en capas convoluciones y recurrentes lo hacen una opción robusta para este TFG. En concreto en este TFG se ha usado una versión híbrida de DEMUCS, que combina una rama de procesado en el tiempo y otra en la frecuencia en una misma arquitectura. Además, el uso de capas convoluciones y recurrentes facilita capturar tanto las dependencias locales como globales (‘patrones y características’) en la ventana de contexto de la señal de audio. Esta arquitectura combinada permite una separación más precisa y eficiente de las diferentes fuentes sonoras. Además, DEMUCS ha demostrado ser especialmente eficaz en la tarea de separar la voz del acompañamiento musical, lo que le convierte en la herramienta ideal para el propósito de este TFG. Para adaptar DEMUCS a las necesidades del procesamiento en tiempo real, se han implementado varias optimizaciones específicas. Estas incluyen la introducción de un retardo controlado, optimización del uso de la GPU, modificación del modelo para funcionar con la técnica del solapamiento suma en ventanas de tiempo más cortas, permitiendo que se procese el audio de manera más rápida, manteniendo la calidad de separación.

El alcance de este TFG incluye el desarrollo de una aplicación de software que permita a los usuarios ajustar el volumen de los diálogos en tiempo real, utilizando una interfaz gráfica de usuario (GUI) intuitiva. La aplicación capturará el audio desde aplicaciones específicas o desde el sistema completo, procesará la señal de audio utilizando el modelo de DEMUCS adaptado para tiempo real, y permitirá a los usuarios remezclar la voz y el fondo según sus preferencias lanzando el resultado final a los altavoces de reproducción del sistema. La interfaz gráfica ha sido desarrollada en Python, utilizando librerías como ‘Tkinter’ para la construcción de la GUI y ‘Sounddevice’ para la captura y reproducción del audio. El procesamiento de audio se realiza mediante la integración del modelo de DEMUCS, que ha sido modificado para trabajar con segmentos de audio de 1 segundo, introduciendo un solapamiento entre llamadas para controlar el retardo. Además, se han implementado técnicas como ‘Overlap-Add’ para asegurar una transición suave entre los segmentos procesados. Se utilizan buffers para manejar el flujo continuo de datos de audio. Por un lado, el modelo de DEMUCS se configura para capturar el audio de entrada con la mezcla completa y generar la salida con las fuentes de audio separadas. Además, se evalúan métricas de calidad objetivas como SDR, SAR y SIR para medir la efectividad y calidad de separación de fuentes de audio lograda por el modelo híbrido original de DEMUCS y el modelo de DEMUCS con las modificaciones realizadas.

Este TFG, no solo contribuye al desarrollo tecnológico en el campo de la separación de fuentes de audio, sino que también ayuda a mejorar significativamente la calidad de vida de los usuarios al proporcionar una experiencia auditiva más clara y personalizada. La solución propuesta combina la última tecnología en redes neuronales profundas con una implementación práctica, proporcionando una herramienta poderosa para el ajuste del volumen de los diálogos en tiempo real. En definitiva, el resultado del presente trabajo es permitir que el usuario pueda seleccionar el stream de audio que desea mejorar en cuanto a voz y fondo, enrutar ese flujo de audio mediante interfaces de audio virtual y reproducir la mejora del diálogo ‘Speech Enhancement’ a través de los altavoces del sistema, destacando como parte fundamental la separación de la voz y el fondo de sonido en tiempo real utilizando el modelo de DEMUCS adaptado para baja latencia.

4. OBJETIVOS

El objetivo principal de este TFG es desarrollar una solución avanzada para la separación de la voz del resto de fuentes sonoras utilizando DNN, específicamente DEMUCS. Este TFG, se enfocará en adaptar la red neuronal para su funcionamiento en tiempo real, proporcionando una herramienta eficiente y de alta calidad para usuarios que necesiten controlar el volumen de los diálogos de manera independiente al fondo. Para alcanzar este objetivo principal, se han establecido los siguientes objetivos específicos:

1. Diseñar y GUI que permita a los usuarios ajustar el volumen de la voz y los diálogos durante el procesamiento en tiempo real. Incluir opciones para seleccionar la latencia, adaptándose a las capacidades de cómputo del sistema empleado.

2. Implementar la captura de audio desde aplicaciones específicas o del sistema completo. Garantizar que la captura de audio sea eficiente y adecuada para facilitar la separación de fuentes de audio.

3. Integrar y adaptar la red neuronal DEMUCS para operar en tiempo real, logrando una separación efectiva de voz y fondo con baja latencia. Optimizar DEMUCS para diferentes configuraciones de latencia, adaptándose a las capacidades de cómputo del sistema utilizado.

4. Permitir remezclar la voz y el fondo según el volumen especificado por los usuarios, proporcionando una experiencia auditiva personalizada. Asegurar que el sonido remezclado se envíe a la salida correspondiente de manera fluida y en tiempo real.

5. Evaluar objetivamente el rendimiento de los modelos, comparando el DEMUCS original con el DEMUCS adaptado para tiempo real. Utilizar métricas estándar de calidad de audio, como SDR, SAR, SIR, para medir la calidad de separación de forma objetiva.

6. Analizar la eficiencia computacional y la latencia de ambos modelos, proporcionando una base sólida para futuras optimizaciones y mejoras en el procesamiento en tiempo real.

4.1 ORGANIZACIÓN DE LA MEMORIA

La estructura de la memoria se compone principalmente de cinco secciones, destinadas a describir el objetivo del proyecto y las soluciones alcanzadas. Comienza con un breve resumen al inicio del documento, seguido de una introducción que proporciona al lector contexto sobre la situación y los objetivos del proyecto. La sección de estado del arte es crucial para comprender la motivación detrás del proyecto, el origen del problema de la separación de fuentes y las diversas aproximaciones existentes mediante algoritmos de redes neuronales profundas.

Posteriormente, se detallan las especificaciones de diseño de la aplicación de escritorio, lo que conduce a la presentación de la solución implementada en el capítulo de Materiales y Métodos. Además, después de realizar la aplicación de escritorio se emplean métricas de calidad objetiva en términos de separación de fuentes de audio para comparar el modelo de DEMUCS en su versión original híbrida y el modelo adaptado de DEMUCS para baja latencia en tiempo real. Finalmente, se ofrece una conclusión general que sintetiza el trabajo realizado, junto con las perspectivas futuras y otras alternativas que podrían considerarse.

5. ESTADO DEL ARTE

Esta sección se va a centrar en conceptos que deben ser explicados para un conocimiento previo y entendimiento del trabajo. La separación de fuentes de audio es un área que continuamente está en evolución. Este campo ha ganado una relevancia significativa debido a sus aplicaciones en áreas como la mejora del habla (speech enhancement), la producción musical y la asistencia en dispositivos de audición. El avance de las redes neuronales profundas ha permitido desarrollar modelos capaces de realizar estas tareas con una precisión y calidad sin precedentes. Además, cada día se experimentan avances significativos gracias al desarrollo de tecnologías como el procesamiento de señales digitales y el aprendizaje automático. En primera instancia, se reserva una sección para abordar el estado actual del campo, con el propósito de brindar al lector un contexto y comprensión básica acerca de la separación de fuentes de audio aplicando redes neuronales profundas y su rendimiento en la tarea específica que aborda este TFG: la separación de la voz del resto de sonidos en tiempo real. Esta misma sección presenta al lector los conceptos de separación de fuentes musicales (MSS), mejora de voz (Speech Enhancement), redes neuronales profundas, su aplicación en la tarea de MSS y la separación en tiempo real, destacando ciertas aplicaciones que presentan el uso en la mejora del diálogo en programas de televisión o películas. Posteriormente, en la sección de materiales y métodos se explicará como se ha llevado a cabo la separación de fuentes en el caso de voz y fondo, todo el proceso que conlleva el enrutamiento de audio, adaptación del modelo híbrido de DEMUCS para procesar el audio en tiempo real con bajo retardo y diferentes soluciones que se han tomado para llegar a una solución final. Al optimizar el modelo para baja latencia y ajustar su funcionamiento a las necesidades específicas del usuario, se crea una herramienta poderosa y versátil que puede mejorar significativamente la calidad de la inteligibilidad de la voz en tiempo real.

5.1 ACLARACIÓN DE CONCEPTOS Y ALCANCE DEL PROYECTO

En este TFG, se enuncian diversas técnicas de procesamiento de audio que incluyen la mejora de voz, la separación de voz cantada de acompañamiento y la separación de diálogos de fondo que es el objetivo final del TFG. Es fundamental aclarar que, aunque estos conceptos están interrelacionados dentro del ámbito del procesamiento de señales de audio, son técnicamente distintos y no deben ser utilizados como sinónimos. A continuación, se presenta una descripción detallada de cada uno de estos conceptos que posteriormente serán profundizados y se delimita el alcance específico de este trabajo.

5.1.1 Mejora de voz (Speech Enhancement)

La mejora de la voz se refiere a un conjunto de técnicas que se destinan a aumentar la claridad y calidad del habla en presencia de ruido de fondo. Este proceso es esencial en aplicaciones como:

- Comunicaciones verbales: Mejora la inteligibilidad en videoconferencias y llamadas telefónicas (Honglie Chen, 2024).
- Sistemas de reconocimiento de voz: Aumenta la precisión de sistemas como asistentes virtuales y dictado por voz (Drgas, Szymon, 2023).

Las técnicas de mejora de voz incluyen la cancelación de ruido, la filtración adaptativa y el uso de algoritmos avanzados como las redes neuronales profundas (7). Estos métodos están ampliamente investigados y existen implementaciones en tiempo real y con muy bajo retardo que permiten su uso en diversas aplicaciones prácticas. Es importante subrayar que este trabajo no se enfoca en la mejora de voz, aunque se menciona y se explica en la revisión del estado del arte para proporcionar un contexto amplio sobre esta técnica de procesamiento de audio.

5.1.2 Separación de voz cantada de acompañamiento

La separación de voz cantada de acompañamiento es un área de investigación que se centra en la separación de pistas vocales de las instrumentales en una grabación musical. Esta técnica tiene aplicaciones en:

- Karaoke: Facilita la creación de pistas de karaoke eliminando la voz del cantante original (Liu, y otros, 2020).
- Producción musical: Permite a los productores editar y mezclar pistas vocales e instrumentales por separado (Donahue, y otros, 2023).

Este campo ha avanzado significativamente con el desarrollo de algoritmos de separación basados en el análisis de espectrogramas y el uso de modelos de aprendizaje profundo (Liu, y otros, 2020), (Donahue, y otros, 2023). Aunque la separación de voz cantada de acompañamiento se menciona en este documento para contextualizar el estado del arte, no constituye el objetivo principal de este trabajo.

5.1.3 Separación de diálogos de fondo

El objetivo central de este TFG es la separación de diálogos de fondo en películas, series o vídeos. Este proceso implica la identificación y aislamiento de la voz hablada o diálogos de otros sonidos presentes en el entorno, como música de fondo, efectos sonoros,

es decir, el resto de pistas que componen la mezcla de audio completa. Las aplicaciones de esta técnica incluyen:

- Mejora de la inteligibilidad en transmisiones en vivo: Fundamental para garantizar que el diálogo sea claramente audible sobre otros sonidos.
- Doblaje y subtitulación en tiempo real: Facilita la sincronización de diálogos en múltiples idiomas durante la postproducción (Strauss, y otros, 2021).
- Postproducción de audio para cine y televisión: Permite a los ingenieros de sonido mejorar la calidad de los diálogos y ajustar el balance de sonido en las producciones audiovisuales (Strauss, y otros, 2021).

Las técnicas utilizadas para la separación de diálogos de fondo incluyen el análisis espectral, el filtrado espacial y el uso de modelos de aprendizaje automático que pueden distinguir entre diferentes fuentes de sonido. Este TFG se centra en el desarrollo y aplicación de estas técnicas para ajustar el volumen de los diálogos en tiempo real, mejorando la experiencia auditiva del espectador.

Este TFG se dedica exclusivamente al ajuste del volumen de los diálogos en tiempo real en películas, vídeos o series. Las menciones a la mejora de voz y a la separación de voz cantada de acompañamiento se incluyen únicamente para proporcionar un contexto más amplio sobre las técnicas de procesamiento de audio.

5.2 INTRODUCCIÓN A LA SEPARACIÓN DE FUENTES DE AUDIO

La separación de fuentes de audio es un campo de investigación y desarrollo que se centra en la capacidad de extraer señales de audio individuales a partir de una mezcla (mixture) de varias fuentes sonoras. Una fuente sonora permite reproducir sonidos desde una localización que posteriormente pueden ser captados por un equipo receptor como puede ser un micrófono. Por consiguiente, se consideran fuentes sonoras instrumentos musicales, la voz como parte fundamental de este proyecto, o cualquier fuente de ruido. Existe un amplio interés científico en la separación de la voz y otros sonidos en señales complejas, con el objetivo de mejorar la calidad de la señal de voz. Sin embargo, el propósito de este TFG es separar la señal de voz del resto de sonidos presentes en una escena sonora, ya sean estos instrumentos musicales u otros tipos de sonidos (Rodríguez Serrano, 2014).

En la actualidad se está trabajando en la separación de señales musicales, cuyo objetivo anteriormente mencionado es la extracción de la señal objetivo desde una mezcla. Sus principales aplicaciones son extraer instrumentos musicales y recuperar información musical. Hay una gran cantidad de técnicas para realizar la separación de fuentes, como

son la factorización de matrices no negativas y el uso de redes neuronales. La mente humana tiene la capacidad innata de resolver este problema, es capaz de discernir y separar las fuentes que escucha. Sin embargo, esta tarea presenta un desafío considerable para el procesamiento computacional. Para poder resolver este problema computacionalmente existen algoritmos creados para hacerlo. Cada algoritmo presenta sus ventajas y desventajas en cuanto a cómo realiza la separación de las señales que componen la mezcla, el cual depende de su diseño específico. En el presente TFG, se opta por emplear la técnica de aprendizaje profundo, conocida como “deep learning”, para resolver esta problemática.

La separación de fuentes de audio se puede complicar por diversos motivos: puede haber muchos instrumentos musicales y voces en una grabación de uno o varios canales, y las fuentes a menudo se procesan añadiendo filtros, reverberación u otros procesos incluso no lineales en el proceso de grabación y mezcla. En algunos casos, las fuentes pueden estar en movimiento, o los parámetros de producción pueden ir cambiando, lo que da como resultado que la mezcla resultante varía con el tiempo. Todas estas cuestiones hacen que la separación de fuentes sonoras sea un problema muy complicado. Si bien las fuentes sonoras poseen estructuras y ciertas propiedades que pueden ayudar a resolver estos problemas. Un ejemplo a destacar sería la estructura armónica regular de frecuencias a intervalos regulares de las fuentes sonoras armónicas, o el timbre entendido como los contornos de frecuencia característicos de cada instrumento musical.

Las fuentes sonoras pueden superponerse tanto en el espectro de frecuencia como en el dominio temporal (Cano, y otros, 2019). Por otra parte, es crucial tener en cuenta que las piezas musicales a separar pueden estar compuestas por uno (monocanal), dos (estéreo) o más canales. El número de canales que conforman la mezcla es fundamental al generar modelos de MSS, ya que determina la cantidad de información disponible. Las mezclas multicanal permiten posicionar las fuentes en el espacio, lo que proporciona información espacial que resulta útil en el proceso de separación. Cada fuente contribuye de manera distinta en cada uno de los canales disponibles, lo que reduce la superposición de fuentes. En contraste, las mezclas monocanal no poseen esta información espacial, lo que dificulta la separación de fuentes. En el presente TFG se han empleado ficheros de dos canales (estéreo).

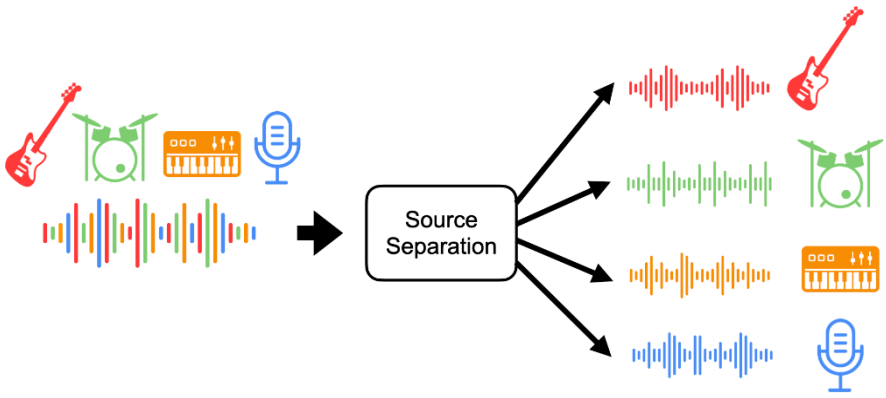


Figura 1. Separación de fuentes de audio (Manilow, y otros, 2020)

5.2.1 Desafíos y consideraciones en la separación de fuentes de audio

Las fuentes musicales están altamente correladas, lo que significa que todas las fuentes tienden a cambiar simultáneamente. Un ejemplo destacado sería en una banda de rock, si el bajo cambia de nota al inicio de un nuevo compás, es probable que los demás instrumentos también lo hagan. La música se procesa y se mezcla de una manera que no es natural y además no lineal. Las prácticas de grabación contemporáneas permiten que una fuente específica en una mezcla pueda no presentarse de forma natural en el mundo real. El filtrado, la reverberación y otras técnicas de procesamiento de señales no lineales complican la separación de la música, aunque son herramientas comunes para los ingenieros de grabación y los músicos a la hora de crear la música. Esto es problemático debido a que rara vez o nunca, se sabe qué procesamiento se ha aplicado a una fuente o a toda la mezcla (Manilow, y otros, 2020).

5.2.2 ¿Por qué separar fuentes de audio?

En el campo de la recuperación de información musical (MIR), se ha demostrado que separar las fuentes facilita procesos como la transcripción automática de música, la alineación de letras y música, la detección de instrumentos musicales, el reconocimiento de letras, la identificación automática del cantante, la detección de actividad vocal, la estimación de la frecuencia fundamental y la compresión de modelos de audio. En cuanto a la alineación de letras y música, permite un análisis más claro y directo de la voz principal sin que suponga la interferencia de otros instrumentos, pudiendo ser útil para aplicaciones de karaoke y análisis lírico. La detección de instrumentos musicales en una mezcla es crucial realizando la separación de fuentes, al poder aislar cada instrumento, los algoritmos de inteligencia artificial pueden identificar mejor ciertas características específicas de cada uno, mejorando la precisión de los sistemas de clasificación de instrumentos (Manilow, y otros, 2020).

En aplicaciones como la realizada en este TFG, es fundamental que los resultados de la separación de fuentes sean de alta calidad. Los sistemas de separación deben producir resultados que sean aceptables para el usuario final, minimizando artefactos y manteniendo la fidelidad del sonido original.

5.2.3 Conceptos básicos en separación de fuentes de audio

Para comprender ciertos aspectos que se abordan en este TFG, es necesario detallar y explicar cuáles son las características clave para la separación de las fuentes de audio. Primero de todo, se han de examinar las representaciones de entrada y salida de una fuente, facilitando la identificación y el aislamiento de las componentes individuales de una mezcla, destacando la voz y el resto de acompañamiento. Diferentes representaciones de audio pueden hacer que ciertos aspectos del sonido sean más o menos fáciles de separar. Las representaciones más comunes incluyen la representación en el dominio del tiempo, la representación tiempo-frecuencia y la representación en el dominio de la frecuencia. En el dominio del tiempo como se puede observar en la Figura 2 en la gráfica de la izquierda, es la forma más directa de visualizar el audio, mostrando cómo varía la señal a lo largo del tiempo. Sin embargo, separar fuentes en esta representación puede ser complicado porque las características frecuenciales no están explícitamente representadas, es decir, es menos intuitiva para identificar fuentes de audio con características similares. En el dominio de la frecuencia, el audio se representa como una serie de componentes sinusoidales, cada uno con una frecuencia, amplitud y fase específicas. Esta representación se obtiene mediante la transformada de Fourier, que descompone la señal en sus componentes frecuenciales como se puede apreciar en la Figura 2 en el lado derecho (Manilow, y otros, 2020).

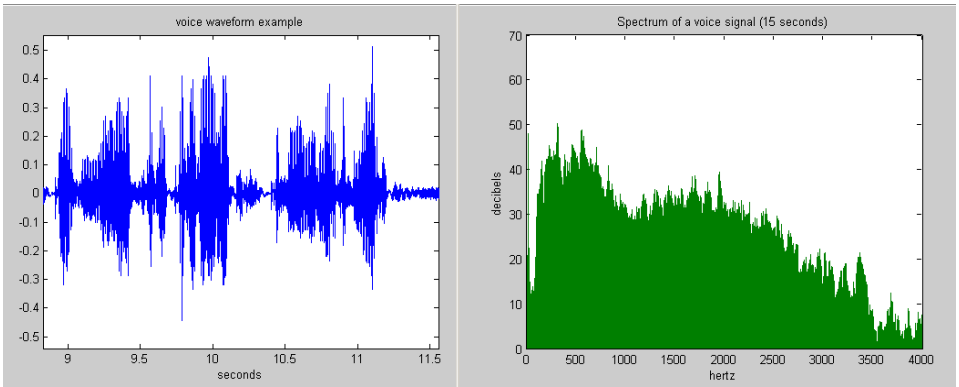


Figura 2. Representación de la voz en el dominio del tiempo y frecuencia (Wikipedia, n.d).

5.2.4 Espectrogramas

Las representaciones tiempo-frecuencia (espectrogramas) combinan los beneficios de las representaciones tanto en tiempo como en frecuencia, mostrando cómo la energía

de la señal de audio se distribuye a lo largo del tiempo y las diferentes frecuencias. Es una matriz bidimensional donde una dimensión está indexada por el tiempo y la otra por la frecuencia. Para su construcción, la señal de audio se segmenta en pequeñas ventanas mediante una función de ventana y a cada uno de los segmentos enventanados se le aplica la Transformada de Fourier de Tiempo Corto (STFT), todo esto se explica en los siguientes apartados. Esta representación resulta ser útil en el presente TFG y en tareas de separación de fuentes, ya que permite aislar e identificar las componentes individuales de una mezcla. De forma breve, la visualización del espectrograma facilita la identificación visual de patrones, características que revelan una fuente de audio como, por ejemplo: notas musicales, formantes vocales, y otros elementos.

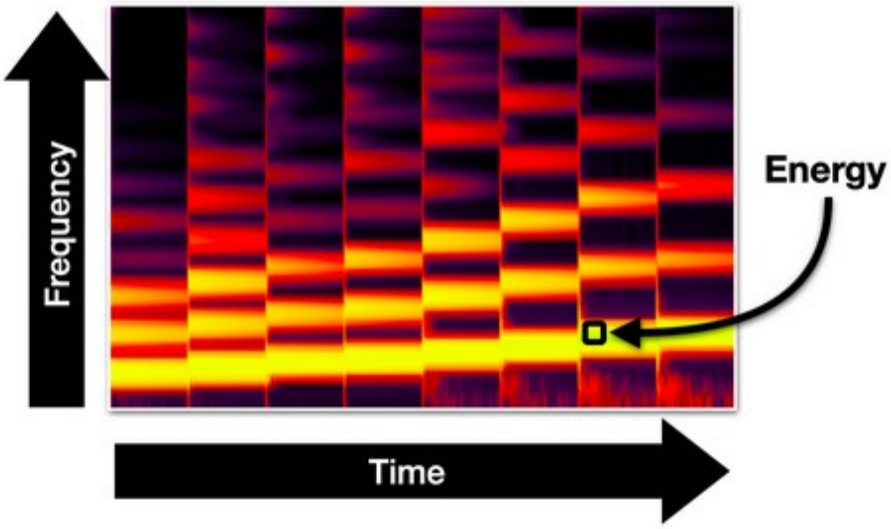


Figura 3. Representación TF "Espectrograma" (Manilow, y otros, 2020).

5.2.5 Características específicas de las fuentes de audio

Aunque a primera vista las señales musicales pueden parecer simplemente otro tipo de onda dentro del amplio espectro de señales de audio, poseen características distintivas que son aprovechadas en el desarrollo de métodos de separación de fuentes de música. Para empezar, se ha de tomar la definición de la fuente musical para referirse a un instrumento en particular, como por ejemplo un saxofón o una guitarra. Esta fuente musical es comúnmente conocida como fuente objetivo. En la separación de voces cantadas, es el caso dónde el objetivo a menudo incluye la separación tanto de la voz principal como de los coros. En términos generales, las fuentes musicales se pueden categorizar en tres tipos: predominancia armónica, predominancia percusiva o voz cantada. Las señales armónicas tienen componentes tonales claros con una frecuencia fundamental fo y una serie armónica. Además, las trayectorias temporales de la fo y sus armónicos tienden a ser muy parecidas, un fenómeno conocido como destino común. Esto se puede observar

claramente en el espectrograma de las voces en la Figura 4, donde los armónicos vocales siguen trayectorias similares a lo largo del tiempo. La intensidad relativa de estos armónicos y su evolución temporal contribuyen al timbre característico del sonido, lo que permite a los oyentes distinguir entre diferentes instrumentos. La voz cantada es una mezcla compleja de componentes armónicos, producidos por las vibraciones de las cuerdas vocales, y componentes sordos, como los sonidos consonánticos 'k' o 'p' que no involucran vibración de las cuerdas vocales. Estos componentes son filtrados por la cavidad vocal, que, al cambiar de forma, produce diferentes frecuencias. Como se muestra en la Figura 4, el canto generalmente presenta una mayor variación de tono en comparación con otros instrumentos musicales (Cano, y otros, 2019).

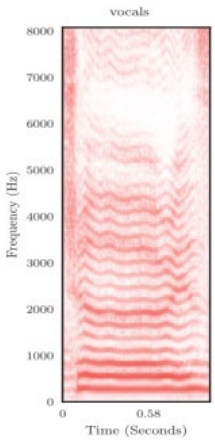


Figura 4. Espectrograma de la voz cantada (Cano, y otros, 2019).

Una característica destacada de las fuentes musicales es su escasez (sparse). Esto significa que, en la mayoría de los puntos en el tiempo o en la frecuencia, las fuentes tienen muy poca energía presente. Esta propiedad es aprovechada en los métodos de MSS para identificar y aislar las diferentes fuentes dentro de una mezcla de audio. Otra característica importante es la presencia de estructuras repetitivas en diferentes escalas de tiempo. Por ejemplo, los patrones de percusión pueden repetirse durante unos segundos, mientras que estructuras más amplias, como los patrones de verso-estribillo en las canciones pop, se repiten a lo largo de toda la pieza musical. Estas repeticiones pueden ser utilizadas en los métodos de MSS para mejorar la precisión en la separación de las fuentes.

5.2.5.1 Transformada de Fourier en Tiempo Corto (STFT)

La STFT es la técnica utilizada en este TFG para convertir las señales de audio en un espectrograma. Este proceso implica la aplicación de la Transformada de Fourier a segmentos pequeños de la señal, obtenidos mediante ventanas que se desplazan a lo largo del tiempo. Cada uno de los segmentos produce un espectro de frecuencias que se organiza en una matriz dónde el tiempo representa el eje x y la frecuencia representa el

eje y. Primero, se selecciona la ventana que se desea aplicar, por ejemplo, una ventana de hanning, esta se aplica a segmentos de audio con un tamaño pequeño (e.j: 1024 muestras) y se va desplazando desde el inicio al final de la señal con pasos uniformes 'hop length' (e.j: 512 muestras) multiplicando cada segmento por la ventana seleccionada. Por otro lado, una vez se tiene cada segmento enventanado, se calcula la STFT obteniendo el espectro de frecuencias correspondiente. La STFT es fundamental para analizar cómo las frecuencias de una señal cambian con el tiempo y es ampliamente utilizada en separación de fuentes de audio como se puede observar en la Figura 5.

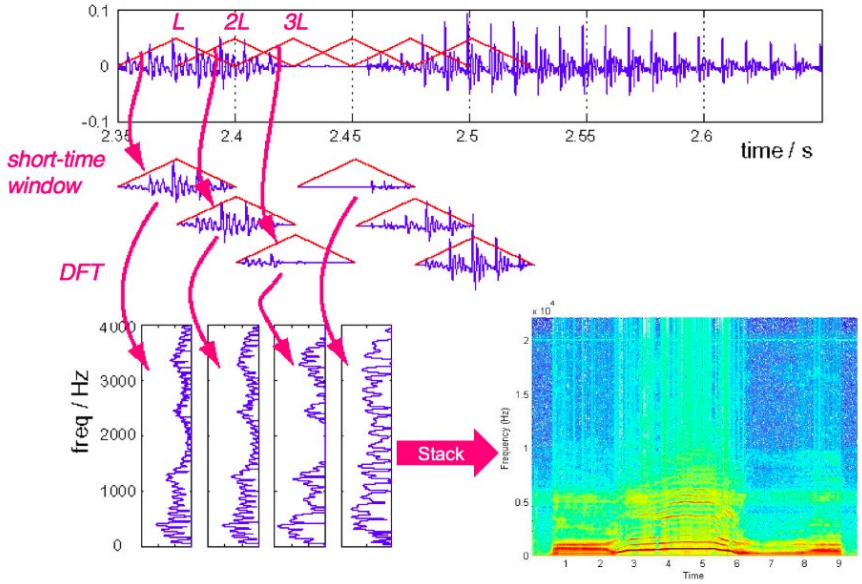


Figura 5. STFT en separación de fuentes de audio (Manilow, y otros, 2020).

5.2.5.2 Ventanas

La forma de la ventana utilizada para poder segmentar la señal de audio antes de aplicar la STFT afecta de una manera significativa a la representación resultante. Existen una variedad de ventanas, cada una con sus propias ventajas y características:

- **Ventana de Hanning:** Es una ventana muy utilizada debido a su gran capacidad para reducir fugas espectrales, mejorando la resolución en frecuencia. En el presente trabajo, se emplean para el solapamiento mitades de ventanas de hanning.
- **Ventana de Hamming:** Similares a las ventanas de hanning, pero con un enfoque distinto ya que reducen las fugas a costa de tener un menor ancho de banda de la ventana.
- **Ventana de Bartlett**: Se conoce con el nombre de ventana triangular, tiene un gran compromiso entre la resolución temporal y la reducción de fugas.

- **Ventana de Blackman:** Reduce aún más las fugas espectrales a costa de tener una mayor ampliación del lóbulo principal.
- **Ventana Rectangular:** Es la más sencilla, no aplica ningún peso a los datos y tiene una mayor probabilidad de fugas espectrales (Manilow, y otros, 2020).

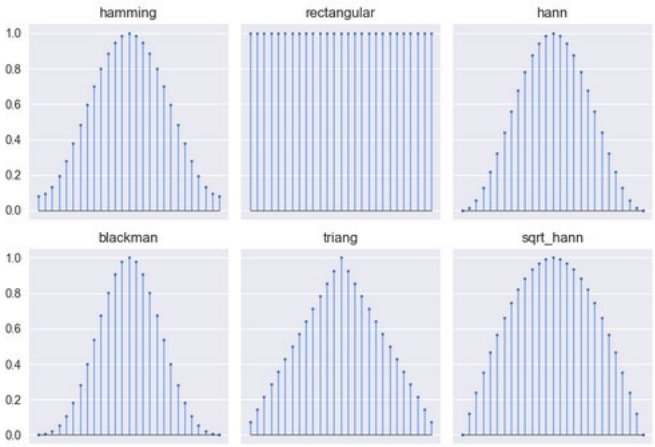


Figura 6. Tipos de ventanas utilizadas en separación de fuentes de audio (Manilow, y otros, 2020).

5.2.5.3 Longitud de ventana y salto

La longitud de la ventana es otro de los conceptos claves que hay que tener en cuenta en separación de fuentes de audio. Esta indica cuántas muestras se incluyen en cada segmento corto de ventana. Tiene un impacto directo tanto en la resolución temporal como en frecuencia de la STFT. Ventanas con un mayor tamaño proporcionan una mejor resolución en frecuencia en cambio ventanas más cortas mejoran la resolución temporal. En definitiva, la elección de la longitud de la misma se centra en conseguir el equilibrio deseado entre las dos resoluciones.

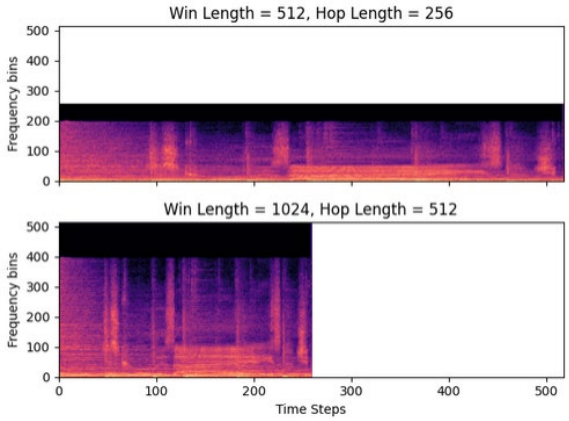


Figura 7. Espectrogramas con diferentes longitudes de ventana (Manilow, y otros, 2020).

La longitud del salto al igual que la longitud de la ventana es un concepto importante en este TFG. Se define como la distancia en muestras entre ventanas consecutivas ya que este parámetro controla cuánto se solapan las ventanas a medida que se aplica la STFT a lo largo de toda la señal de audio. Un salto pequeño, supone que cada uno de los segmentos de la señal se solapa con el anterior, lo que supone tener una mejor resolución temporal ya que cada punto en el tiempo es analizado varias veces por diferentes ventanas, a costa de aumentar la redundancia de los datos y el coste computacional. Por otro lado, un salto grande reduce el solapamiento y puede tener una representación más eficiente, pero resultando en un menor detalle temporal.

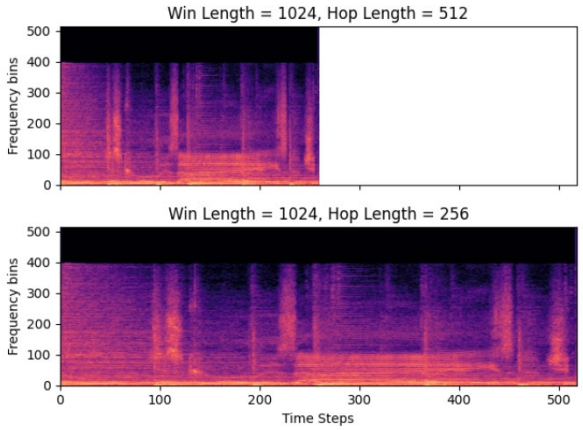


Figura 8. Espectrogramas con diferentes longitudes de salto (Manilow, y otros, 2020).

5.2.5.4 Overlap-Add

El método del solapamiento suma (Overlap-Add) es una técnica utilizada en el procesamiento de señales de audio y empleada en este trabajo para reconstruir señales segmentadas tras aplicar la STFT. Este método implica dividir la señal original en segmentos que se solapan, aplicar la ventana correspondiente para poder suavizar los bordes de cada segmento y realizar la STFT en cada uno de los segmentos. Los segmentos que han sido transformados se procesan y se extraen en el dominio de la frecuencia ciertas características que pueden servir de utilidad para la separación de fuentes y, luego, se reconstruyen al dominio temporal a través de la Transformada Inversa de Fourier de Tiempo Corto (ISTFT). Finalmente, los segmentos que han sido reconstruidos se solapan y se suman para asegurar una transición suave entre segmentos y minimizando la aparición de artefactos en la señal de audio final (Heinzel, y otros, 2002).

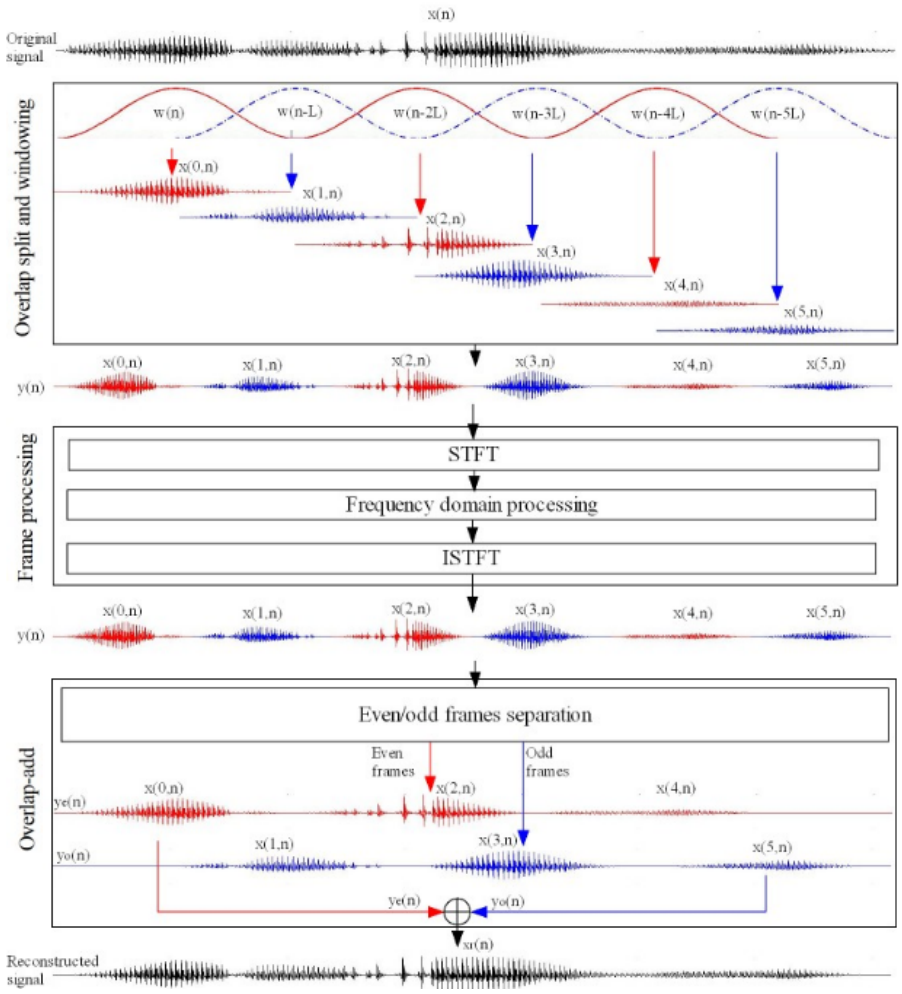


Figura 9. Método de solapamiento suma (Bahoura, 2019).

5.3 SEPARACIÓN Y MEJORA DEL HABLA

La mejora de voz o el habla ‘speech enhancement’, de manera global, tiene como objetivo aislar y limpiar el habla del ruido de fondo. Uno de los problemas de la mejora de voz puede llegarse a formular como un problema de separación de fuentes en el que una de las fuentes es siempre la voz y la otra fuente es siempre ruido, donde la fuente de “ruido” puede modelarse o ignorarse por completo.

Aunque muchos avances en la mejora del habla (Speech Enhancement) tienen el potencial de ser adaptados para la separación de fuentes musicales, esto requiere modificaciones específicas debido a las características únicas de las señales musicales (Fuchs, y otros, 2012).

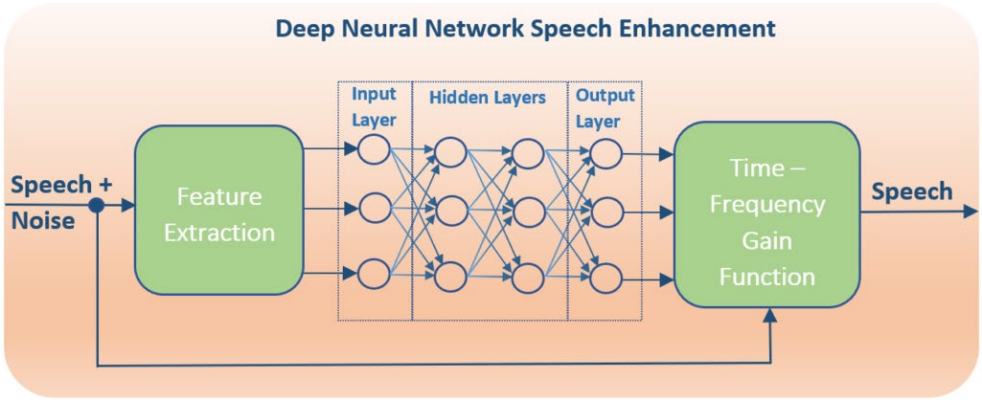


Figura 10. DNN Speech Enhancement (Vocal Technologies Ltd, n.d).

Este desafío de clasificación se presenta como un escenario idóneo para las DNN. El aprendizaje profundo ha impulsado avances significativos en tareas de aprendizaje supervisado, como el reconocimiento del habla y la identificación de imágenes. La mayoría de los principios fundamentales se pueden extrapolar a la mejora del habla. Como posteriormente se enunciará, las DNN comprenden tres componentes esenciales: la red neuronal en sí, el proceso de entrenamiento donde ocurre el aprendizaje y la extracción de características acústicas.

En la Figura 11, se puede observar un diagrama típico de mejora del habla sin incluir métodos específicos. En primer lugar, suponiendo la entrada a un sistema en el que se capta una señal de audio, como, por ejemplo, una conversación con ruido de fondo. Esta señal de audio se va segmentando en pequeños trozos y enventanando con la ventana correspondiente, como se explicó en apartados anteriores, para mejorar la continuidad y la suavidad en la señal reconstruida. Calculando la STFT de cada uno de los segmentos enventanados se obtienen las diferentes componentes frecuenciales individuales, las cuales pueden ser utilizadas para estimar la separación de fuentes de audio. De manera paralela, se realiza una estimación del ruido de cada segmento transformado y se implementa un método de mejora del habla. Más adelante se explicarán los diferentes métodos que existen en el mercado y la implementación realizada en este trabajo. Finalmente, se calcula la ISTFT y los segmentos reconstruidos se solapan y se suman para poder reconstruir la señal original asegurando una transición suave y minimizando los artefactos en la señal de audio limpia de salida.

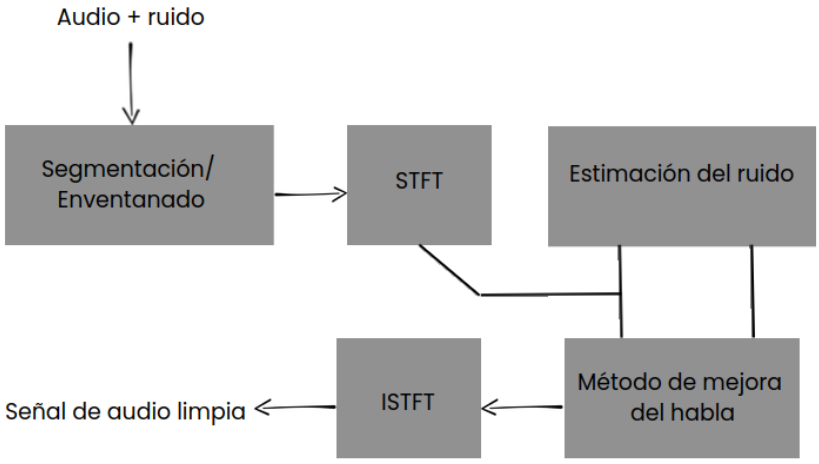


Figura 11. Diagrama de bloques de mejora del habla.

Con la finalidad de mejorar el habla, según el dominio utilizado, los algoritmos se pueden dividir en filtrado espectral y filtrado espacial. En cuanto al filtrado espectral, son algoritmos que son conocidos como mejora del habla de un solo canal debido a que procesan señales de habla ruidosa capturadas por un sólo micrófono o la salida de un algoritmo de filtrado espacial. En el filtrado espacial, se reducen los sonidos interferentes basándose en sus propiedades espaciales, mientras que se mantienen las señales provenientes de la dirección de interés. La mayoría de enfoques para mejorar las señales de un sólo canal se basan en la STFT empleada en este TFG (University of Hamburg, 2023).

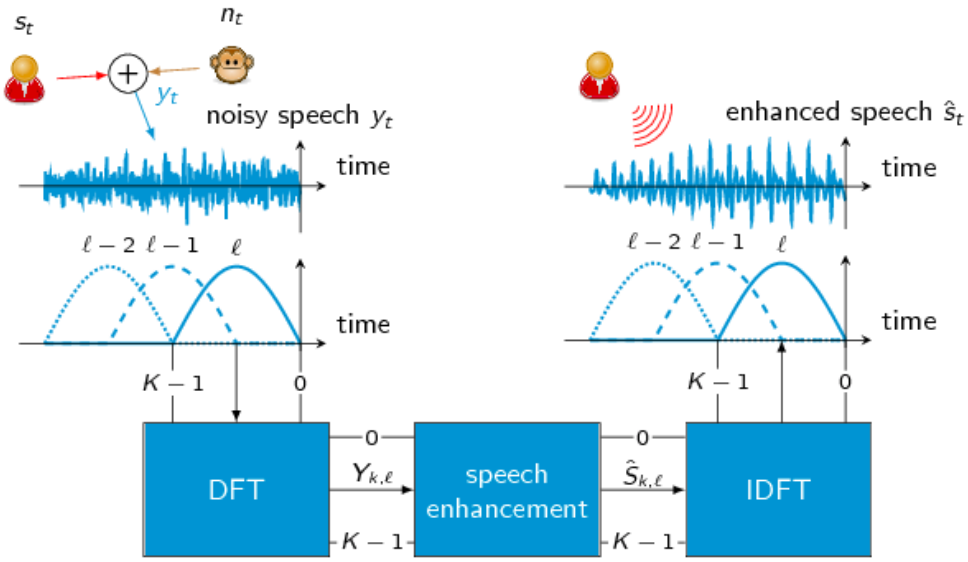


Figura 12. Diagrama de bloques del algoritmo de mejora del habla en el dominio STFT (University of Hamburg, 2023).

El diagrama de bloques básico para la mejora de voz de un sólo canal basado en STFT permite mejorar la inteligibilidad del habla. Dado que se conoce que el habla es una

señal no estacionaria, ya que varía considerablemente con el tiempo, la señal de entrada con voz y ruido se divide en segmentos de tiempo cortos y además superpuestos. Cada uno de los segmentos de la señal se transforma al dominio de la frecuencia mediante la Transformada de Fourier Discreta (DFT), resultando en la STFT. El propósito de los algoritmos de mejora del habla es estimar los coeficientes de voz limpia $\hat{S}_{k,l}$ a partir de las observaciones de ruido $Y_{k,l}$ en el dominio de la frecuencia. En la Figura 12, se puede ver que k representa el índice de frecuencia y l el índice del segmento dónde la mayoría de estimadores de voz limpia en el dominio espectral pueden expresarse de la siguiente manera dónde $G_{k,l}$ es la función de ganancia: (University of Hamburg, 2023)

$$\hat{S}_{k,l} = G_{k,l} Y_{k,l} \tag{1}$$

Generalmente, se asume que la función de ganancia es real, ya que muchos modelos estadísticos de los coeficientes de habla limpia no proporcionan información sobre la fase. Hasta hace poco, en la mejora del habla basada en STFT, el enfoque principal se centraba en modificar únicamente la magnitud de los componentes STFT. Esto se debía a la creencia general de que la mayor parte de la información sobre la estructura de la señal se podía obtener a partir de la magnitud, mientras que la componente de fase proporcionaba poca información útil. Además, ciertos algoritmos sólo estiman la magnitud $|\hat{S}_{k,l}|$ de la señal de voz limpia. En estos casos, se estima una magnitud mejorada $|\hat{A}_{k,l}|$ la cual se combina con la fase de la observación de ruido $\phi_{Y_{K,l}}$ de la siguiente manera: (University of Hamburg, 2023)

$$\hat{S}_{k,l} = |\hat{A}_{k,l}| \exp(\phi_{Y_{k,l}}) \tag{2}$$

5.3.1 Mejora de voz en el dominio de la frecuencia

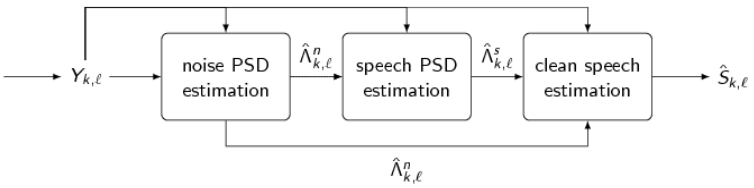


Figura 13. Diagrama de bloques del proceso de mejora espectral (University of Hamburg, 2023).

La Figura 13, ilustra el diseño general del proceso de mejora espectral del habla. La mayoría de los métodos para estimar la voz limpia se desarrollan dentro de un marco estadístico, requiriendo la estimación de la densidad espectral de potencia del habla limpia

(PSD $\Lambda_{k,l}^{s}$) y del ruido (PSD $\Lambda_{k,l}^{n}$). Estas cantidades se suelen estimar a partir de las observaciones o señal de ruido $Y_{k,l}$. Como paso inicial, se estima el "Power Spectral Density" (PSD) del ruido $\Lambda_{k,l}^{n}$ el cual se emplea después para estimar la densidad espectral de potencia del habla $\Lambda_{k,l}^{s}$. Ambas estimaciones de PSD se utilizan para determinar la función de ganancia que se aplica en la estimación de los coeficientes de voz limpia.

No obstante, para tener una estimación de la densidad espectral de potencia del habla y del ruido, se pueden emplear métodos de aprendizaje no supervisado en la que el modelo empleado aprende a partir de los datos que se utilicen en el entrenamiento. Los modelos no supervisados descubren por sí mismos los patrones subyacentes en los datos analizados, este proceso se asemeja al proceso que ocurre en el cerebro humano al aprender cosas nuevas. Se han propuesto algoritmos diseñados específicamente para esta tarea, aprovechando la suposición de que el ruido de fondo cambia más lentamente que el habla. Además, los enfoques de aprendizaje automático pueden ser empleados para obtener estas estimaciones donde las propiedades del habla y del ruido se aprenden a partir de datos representativos antes del proceso de mejora.

5.4 INTRODUCCIÓN A LAS REDES NEURONALES PROFUNDAS

Las DNN son una parte fundamental de la inteligencia artificial y el aprendizaje automático, diseñadas para imitar el funcionamiento del cerebro humano a través de múltiples capas de neuronas artificiales interconectadas. Estas redes permiten procesar y aprender patrones complejos en grandes volúmenes de datos mediante algoritmos de optimización como la retropropagación, que ajustan los pesos de las conexiones neuronales para minimizar los errores en las predicciones. A medida que las señales atraviesan estas capas, las redes neuronales profundas extraen características cada vez más abstractas y detalladas, lo que resulta en una comprensión jerárquica de los datos. En el ámbito de MSS, las DNN han demostrado ser especialmente eficaces. Estas redes no solo aprenden las características distintivas de diferentes señales musicales, sino que también pueden generalizar esta información a nuevos datos, mejorando la precisión y la calidad de la separación de las fuentes. Esto es crucial en la producción musical, donde la capacidad de aislar instrumentos y voces de una mezcla puede transformar significativamente la calidad del producto final como se presenta en este TFG.

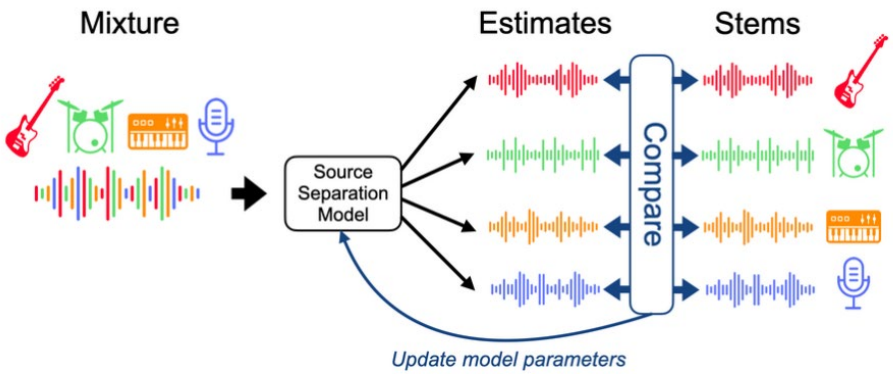


Figura 14. Bloque funcional de una red neuronal en separación de fuentes (Manilow, y otros, 2020).

5.4.1 Deep Learning

La inteligencia artificial (IA), el aprendizaje automático (Machine Learning, ML) y el aprendizaje profundo (Deep Learning, DL) son términos interrelacionados, pero con diferencias clave en su complejidad y capacidades. La IA es un campo amplio que busca desarrollar sistemas capaces de realizar tareas que requieren inteligencia humana, como el reconocimiento de voz, la toma de decisiones y la traducción de idiomas. Dentro de la IA, el ML es una subcategoría que se centra en desarrollar algoritmos que permiten a las máquinas aprender de los datos y mejorar con la experiencia, sin ser explícitamente programadas para cada tarea específica. El ML puede ser supervisado, no supervisado o por refuerzo, dependiendo de cómo se etiqueten y utilicen los datos.

El DL, por su parte, es una técnica avanzada dentro del ML que utiliza redes neuronales artificiales con múltiples capas (capas profundas) para analizar y aprender de grandes volúmenes de datos. Este enfoque permite a los algoritmos de DL identificar patrones complejos y realizar tareas avanzadas con alta precisión, como el reconocimiento de imágenes, la conducción autónoma y el diagnóstico médico. Mientras que los modelos tradicionales de ML pueden alcanzar un punto de saturación donde dejan de mejorar con más datos, los modelos de DL continúan mejorando con cantidades mayores de datos y mayor capacidad computacional, beneficiándose del uso de GPUs y grandes conjuntos de datos (DataScientest, 2024a), (DataScientest, 2024c), (DataScientest, 2024b).

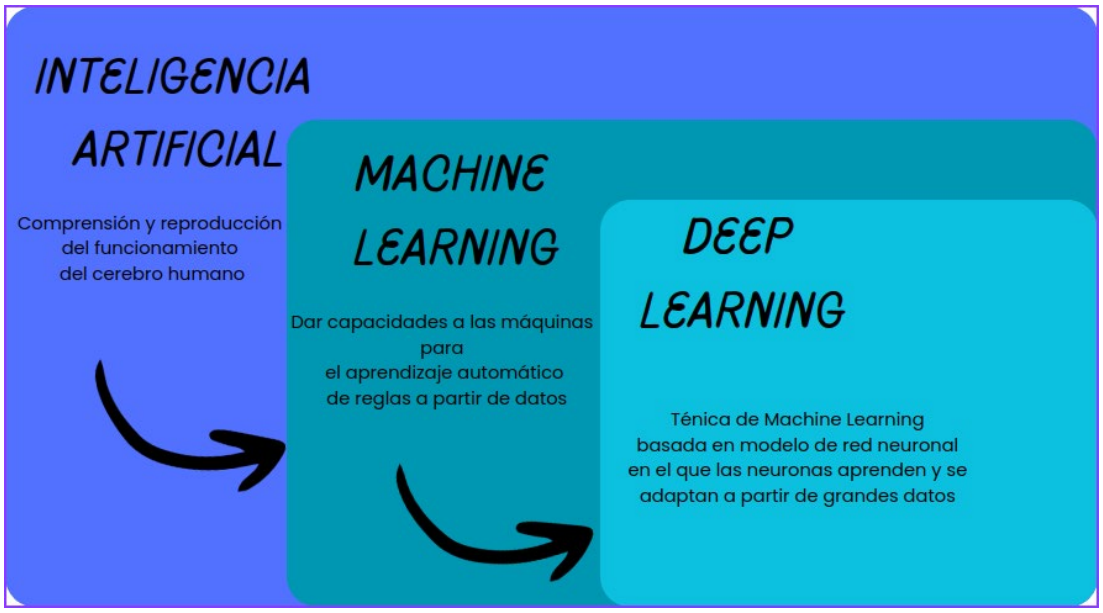


Figura 15. Proceso de evolución de IA.

5.4.1.1 Aprendizaje supervisado y no supervisado

Para poder seguir introduciendo los conocimientos de redes neuronales, es imprescindible tener en cuenta dos conceptos fundamentales en el campo del Machine Learning como se mencionó en el apartado anterior, la principal diferencia radica en el uso de etiquetas o conocimiento previo de los datos:

Aprendizaje Supervisado: El aprendizaje supervisado tiene como requisito un conocimiento previo o una etiqueta asociada que indique el valor de salida que se espera para cada una de las muestras de datos. Este valor de salida se le conoce como “ground truth” (GT). El objetivo del aprendizaje supervisado es encontrar una función que, al tomar un conjunto de datos de entrenamiento con sus respectivas etiquetas, aproxime la relación entre las variables que hay a la entrada y a la salida observadas en los datos. Sus aplicaciones más comunes son problemas de clasificación y regresión, cuyos algoritmos más comunes utilizados son: regresión logística, máquinas de soporte vectorial y redes neuronales artificiales (Stack Sánchez, 2021).

Aprendizaje no supervisado: El aprendizaje no supervisado se utiliza cuando los datos no están etiquetados. En este enfoque, el modelo busca identificar patrones o estructuras ocultas en los datos sin ninguna guía externa. Las aplicaciones más comunes de este tipo de aprendizaje son: agrupamiento, segmentación y reducción de dimensionalidad, además, los algoritmos típicos son: K-medias, análisis de componentes principales (PCA) y NMF (Stack Sánchez, 2021).

5.4.2 Arquitectura y capas de una red neuronal

Las redes neuronales artificiales tienen su origen en el perceptrón, un modelo introducido por Frank Rosenblatt en 1957. El perceptrón es una neurona artificial que procesa múltiples entradas y genera una salida basada en una función de activación. Este concepto básico se expande en redes neuronales más complejas con múltiples capas. El proceso de aprendizaje de estas redes, a través de técnicas como la retropropagación, permite ajustar los pesos y mejorar la precisión en diversas tareas. Esto hace que las redes neuronales sean herramientas versátiles y poderosas para aplicaciones como el reconocimiento de imágenes, la clasificación de textos y, en el contexto de este TFG, la separación de fuentes en el caso de la voz y el resto de sonidos (DataScientist, 2024d).

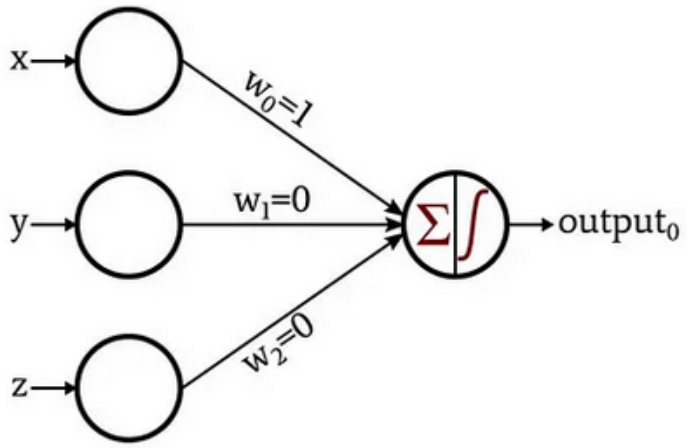


Figura 16. Estructura de un perceptrón (Robert, 2019).

En las DNNs, para poder construir una red neuronal profunda, es crucial comprender cuáles son sus capas y diversas funciones que presenta. Al hablar de capas de red, nos referimos a las pasadas hacia adelante y hacia atrás a través de la red. En la pasada hacia adelante, cada capa recibe datos de la capa anterior o de la entrada inicial, y transforma estos datos operando cada componente por un conjunto de pesos, produciendo una salida. Aunque el proceso es más complejo, esencialmente, una entrada atraviesa la red neuronal sufriendo múltiples transformaciones en cada capa.

5.4.3 Componentes que forman una red neuronal

Una red neuronal se puede definir con los siguientes elementos:

1. **Capa de entrada (Input Layer):** Pertenece a la primera capa que compone una red neuronal. Es el punto dónde se introducen cuáles son las observaciones de entrenamiento a través de cada una de las variables independientes (Dataquest, 2023).

2. **Capas ocultas (Hidden Layer):** Son las que forman las capas intermedias entre la capa de entrada y la capa de salida. Dentro de las mismas es dónde la red neuronal realiza la mayor parte de las tareas de aprendizaje y parte del procesamiento (Dataquest, 2023).
3. **Capas de salida (Out Layer):** Supone la última capa de la red neuronal, donde se recoge la salida final de todas las predicciones y clasificaciones dentro de las capas ocultas y se genera la salida de la red (Dataquest, 2023).

Además de su clasificación, las redes neuronales se pueden clasificar en dos tipos como redes neuronales simples y redes neuronales profundas. Las redes neuronales simples como se tiene en la Figura 17, tienen un número pequeño de capas dónde normalmente se componen de dos o una capa oculta. Su aplicación suele ser común en tareas simples como podría ser la clasificación o regresión. Por otro lado, las redes neuronales profundas como su propio nombre indica están compuestas por múltiples capas en la que cada capa está formada por un conjunto de neuronas que son capaces de aprender a extraer ciertas características. No obstante, ciertas capas adicionales profundas tienen la capacidad de ser más precisas, debido a que aprenden patrones mucho más complejos en relación con una red neuronal poco profunda. Se utilizan en aplicaciones complejas en tiempo real, tales como previsiones en tiempo real, procesamiento de separación de fuentes propio de este TFG, etc. Este tipo de red se puede apreciar en la Figura 18 (Dataquest, 2023).

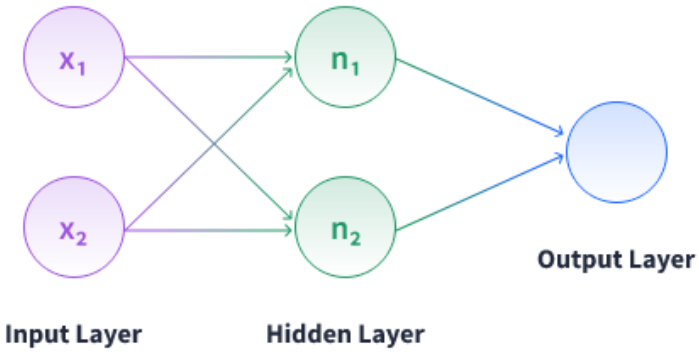


Figura 17. Arquitectura básica de red neuronal simple (Dataquest, 2023).

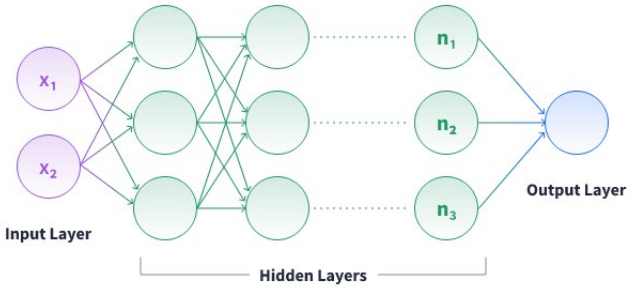


Figura 18. Arquitectura básica de una red neuronal profunda (Dataquest, 2023).

5.4.3.1 Funciones de activación

Para seguir comprendiendo los conceptos de redes neuronales profundas y su importancia que tiene en este trabajo, hay que enunciar brevemente las funciones de activación. Estas funciones introducen no linealidades, lo que es esencial para poder modelar relaciones complejas en las fuentes de audio. Se aplican sobre una neurona para permitirle aprender patrones complejos de datos, determinando qué información ha de transmitirse a la siguiente neurona (Alulema, 2022).

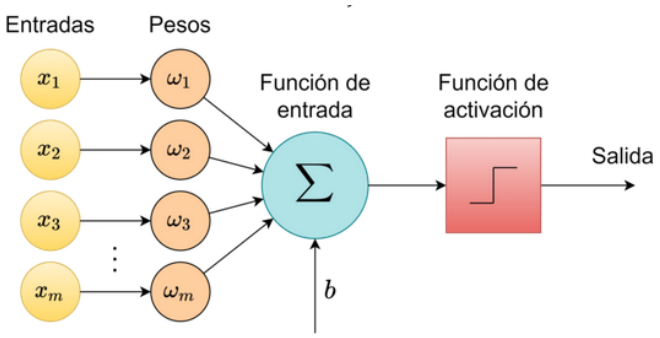


Figura 18. Arquitectura básica entrada-salida de red neuronal profunda (Alulema, 2022).

Algunas de las funciones de activación más comunes son:

- **Sigmoid.** Transforma la entrada en un valor entre 0 y 1, útil para crear máscaras en separación de fuentes (Manilow, y otros, 2020) . Se puede apreciar en la Figura 19 como los cambios son graduales y se pueden conseguir valores distintos entre los mencionados anteriormente (Alulema, 2022).

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

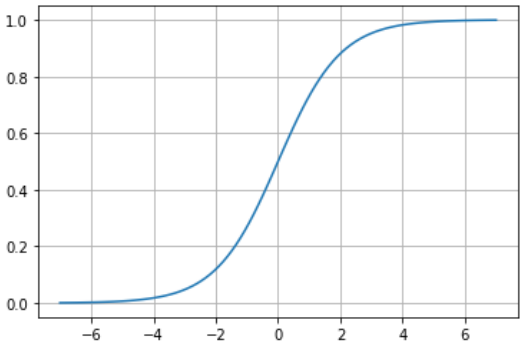


Figura 19. Función de activación. Sigmoide (Alulema, 2022).

- **Tanh**. Similar a la función Sigmoid pero en un rango de -1 a 1, proporcionando salidas centradas de datos. Presenta el mismo problema que la función sigmoide en el desvanecimiento del gradiente que es una técnica que se explicará en el siguiente apartado (Manilow, y otros, 2020), (Alulema, 2022).

$$f(x) = tanh(x)$$

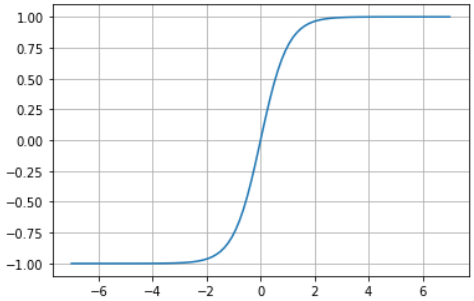


Figura 20. Función de activación. $tanh\,(x)$ (Alulema, 2022)

- **ReLU (Rectified Linear Unit).** Activa las neuronas solo si la entrada es positiva y se encarga de abordar problemas de desvanecimiento de gradiente. Además, acelera la convergencia del SGD qué es el desvanecimiento de gradiente estocástico, lo que permite a la neurona aprender más rápido. Esta ventaja presenta una debilidad, ya que esta velocidad de aprendizaje supone que en ciertos momentos los pesos de la neurona oscilen y se actualicen desde ciertos valores óptimos no llegando a activarse en ningún punto. Por ejemplo, si en una red neuronal se está procesando una imagen y una neurona en la capa oculta recibe un valor de

entrada de -3, la salida de esa neurona será 0. Si, en cambio, recibe un valor positivo como 5, la salida será 5 (Manilow, y otros, 2020), (Alulema, 2022).

$$f(x) = max(0, x)$$

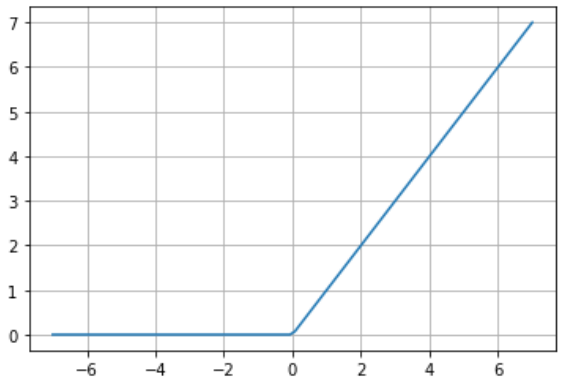


Figura 21. Función de activación. $ReLU$ (Alulema, 2022)

- **GELU (Gaussian Error Linear Unit).** GELU, permite unir las funcionalidades de distintas funciones de activación como son (ReLU, ELU) con una función que se encarga de regularizar el 'Dropout', reduciendo el sobreajuste producido en las neuronas artificiales. Al tener una distribución gaussiana, elimina ciertos problemas que pueden aparecer en el desvanecimiento en valores negativos que presenta ReLU. La importancia que presenta en este TFG, es la capacidad de proporcionar transiciones más suaves y una activación más estable, lo cual supone un beneficio para el procesamiento de señales de audio complejas en la separación de fuentes (Alulema, 2022), (Déffossez, 2021).

$$f(x) = xP(X \leq x) = x\ \Phi(x) = x \cdot \frac{1}{2}\left[1 + erf(\frac{x}{\sqrt{2}})\right]$$

$$f(x) \approx 0.5x\left(1 + tanh\left[\frac{\sqrt{2}}{\pi}\left(x + 0.044715x^3\right)\right]\right)$$

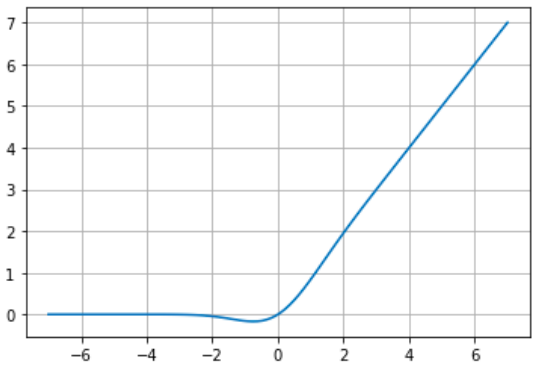


Figura 22. Función de activación. $GELU$ (Alulema, 2022)

5.4.3.2 Descenso de gradiente estocástico (SGD)

El método de SGD es una técnica de optimización muy utilizada en aprendizaje automático y redes neuronales. A diferencia del descenso de gradiente tradicional, que utiliza todo el conjunto de datos para calcular el gradiente de la función de pérdida, SGD actualiza los parámetros del modelo utilizando solo un subconjunto aleatorio de datos (mini batch) en cada iteración. Esto permite que el entrenamiento sea más rápido y eficiente, aunque introduce mayor variabilidad en las actualizaciones, lo que puede ayudar al modelo a evitar mínimos locales y encontrar soluciones más robustas. Sin embargo, esta variabilidad también puede hacer que la convergencia sea más ruidosa.

5.4.4 Tipos de redes neuronales

5.4.4.1 Redes neuronales convolucionales (CNN)

Las redes neuronales convolucionales son uno de los tipos de redes neuronales más conocidas, están compuestas por capas convolucionales que aplican filtros a la entrada para extraer características importantes. Estas capas reducen el riesgo de sobreajuste y son invariantes a la traslación, lo que las hace muy adecuadas para el procesamiento de imágenes y espectrogramas. En términos de separación de fuentes de audio, las CNNs se utilizan tanto en el dominio temporal (convoluciones 1D) como en el dominio tiempo-frecuencia (convoluciones 2D) para procesar espectrogramas y generar máscaras de separación (Manilow, y otros, 2020).

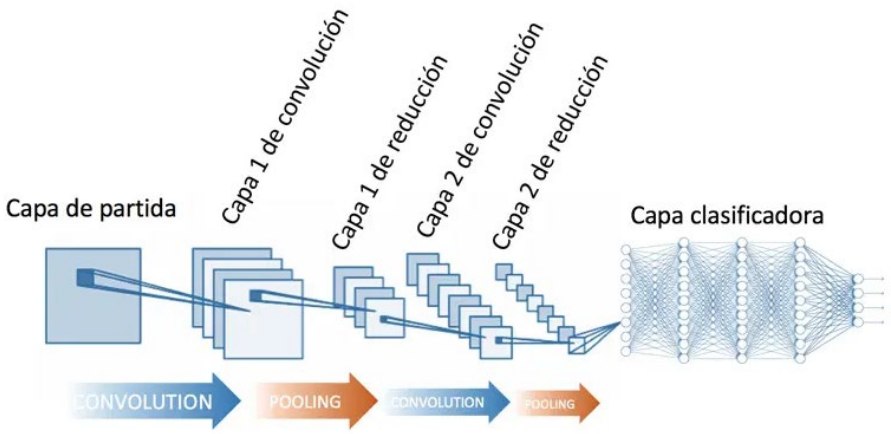


Figura 23. Arquitectura de una red neuronal convolucional (Calvo, 2017)

En la Figura 23 se observa la disposición de varias capas convolucionales, donde se aplican filtros a cada imagen de entrenamiento con diferentes resoluciones. La salida de cada imagen procesada se utiliza como entrada para la siguiente capa. Las capas superiores se encargan de la extracción de características, mientras que las capas inferiores determinan la categoría de la imagen de entrada.

Componentes principales de la CNN

En la Figura 23 se puede apreciar cual es la estructura que sigue una CNN, a continuación, se muestran y se definen cada una de las capas que sigue la estructura:

1. **Capa Convolucional:** La capa convolucional es la principal en las CNNs, donde se definen los parámetros de ajuste, filtros y funciones asociadas. Incluye una función de activación y realiza una convolución entre un filtro y el volumen de datos de entrada. Este proceso implica deslizar el filtro sobre diferentes regiones de la imagen, calculando el producto escalar entre el filtro y cada región cubierta, y sumando los valores resultantes. Esta operación permite extraer características importantes de la imagen, como bordes y texturas. (IBM, 2024a)

2. **Capa de Pooling:** reduce la dimensión de los datos disminuyendo el número de parámetros de la entrada. Similar a la capa convolucional, la operación de agrupación utiliza un filtro que se desliza sobre la entrada, pero sin pesos. En su lugar, el kernel aplica una función de agregación a los valores dentro del campo receptivo, generando la matriz de salida. Los dos tipos principales son: Max-Pooling, Average-Pooling. (IBM, 2024a)

3. **Capa clasificadora:** El nombre de la capa totalmente conectada o clasificadora describe con precisión su función: cada nodo en la capa de salida está directamente conectado a cada nodo de la capa anterior. Esta capa se encarga de la clasificación basándose en las características extraídas de las capas anteriores y sus filtros. Mientras que las capas convolucionales y de agrupación suelen utilizar funciones ReLU, las capas totalmente conectadas generalmente emplean una función de activación softmax para clasificar las entradas y generar probabilidades entre 0 y 1. (IBM, 2024a)

5.4.4.2 Redes neuronales recurrentes (RNN)

Las redes neuronales recurrentes (RNNs) son una clase de redes diseñadas para procesar datos secuenciales y temporales, como texto o series temporales. A diferencia de las redes tradicionales, las RNNs pueden mantener una "memoria" de entradas previas, utilizando esta información para influir en las salidas actuales. Esto les permite recordar información de estados anteriores, lo que es ideal para tareas como el reconocimiento de voz, la traducción automática y el análisis de sentimientos. Variantes como LSTM (Long Short-Term Memory) y GRU (Gated Recurrent Units) mejoran la capacidad de las RNNs

para capturar dependencias a largo plazo y abordar problemas como el desvanecimiento del gradiente (IBM, 2024b).

Funcionamiento y características

Las RNNs pueden recordar información previa y utilizarla para predecir las salidas actuales, lo que es fundamental para entender el contexto en datos secuenciales. A diferencia de las redes de propagación hacia delante, las redes recurrentes comparten los mismos pesos a través de diferentes pasos temporales, permitiendo una consistencia en el procesamiento de secuencias. Emplean el algoritmo de retropropagación a través del tiempo (BPTT) para poder ajustar los pesos, calculando los gradientes en cada paso temporal y sumándolos para actualizar los parámetros del modelo. Los problemas más comunes que denotan son que los gradientes pueden llegar a volverse muy pequeños o muy grandes, dificultando el aprendizaje. Esto se puede resolver, reduciendo las capas ocultas o utilizando variantes como LSTM y GRU.

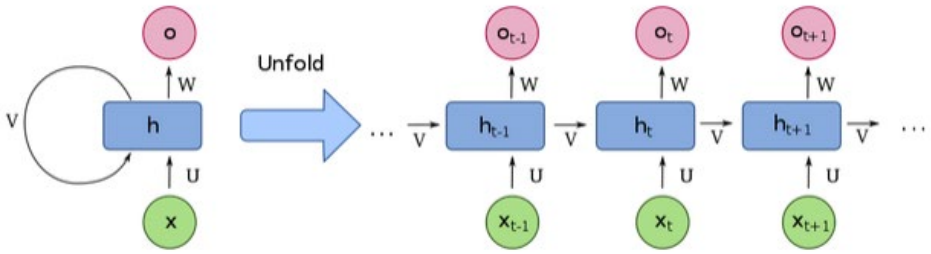


Figura 24. Arquitectura de una red neuronal recurrente (Manilow, y otros, 2020).

Memoria a corto-largo plazo (LSTM)

La memoria a corto-largo plazo (LSTM) es una arquitectura de RNN desarrollada por Sepp Hochreiter y Juergen Schmidhuber para abordar el problema del gradiente desvaneciente. Este modelo (Hochreiter, y otros, 1997) se diseñó para manejar dependencias a largo plazo, permitiendo a la red recordar información relevante durante intervalos prolongados. Las LSTM incluyen celdas de memoria con tres puertas: de entrada, de salida y de olvido, que controlan el flujo de información esencial para la predicción de la salida de la red. Estas puertas permiten al modelo retener o descartar información según sea necesario para mejorar la precisión en tareas secuenciales complejas (IBM, 2024b).

Las LSTM son especialmente útiles porque pueden mantener la información a lo largo de secuencias largas, lo que es esencial para tareas que requieren la memoria de eventos pasados. Las capas recurrentes pueden ser unidireccionales o bidireccionales, lo

que permite a la red procesar la información en ambas direcciones temporales (IBM, 2024b), (Manilow, y otros, 2020).

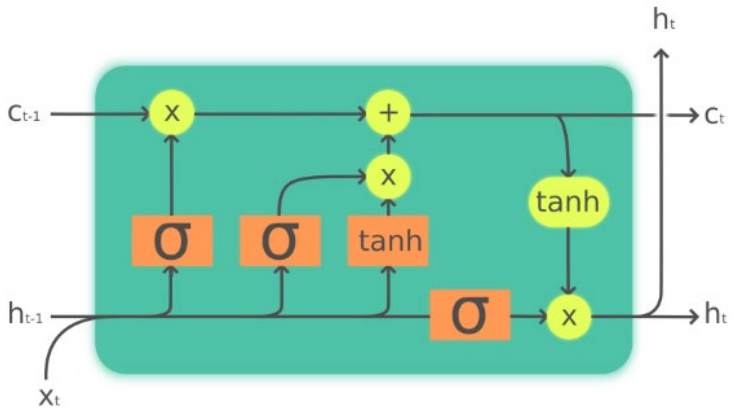
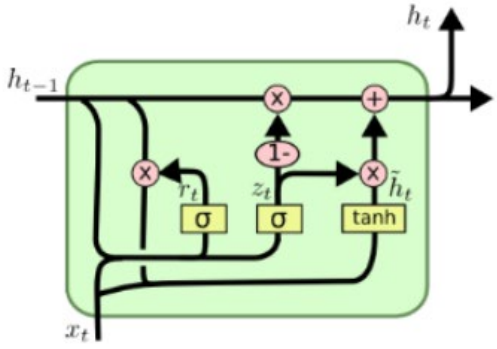


Figura 25. Celda LSTM (Chevalier, 2018).

Unidades recurrentes cerradas (GRU)

Las GRU son una variante de las RNN, similar a las LSTM, diseñadas para resolver el problema de la memoria a corto plazo en los modelos RNN. A diferencia de las LSTM, que utilizan un "estado de celda" para gestionar la información, las GRU utilizan únicamente estados ocultos y poseen dos puertas en lugar de tres: una puerta de restablecimiento y una puerta de actualización. Estas puertas controlan la cantidad y el tipo de información que se debe retener o descartar, mejorando la capacidad de la red para manejar dependencias a largo plazo (The Neural Perspective, 2018).



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figura 26. Celda GRU (The Neural Perspective, 2018)

Las GRU están formadas por una puerta de restablecimiento que controla la cantidad de información pasada del estado anterior a la nueva puerta de estado. Si se activa, permite que se mantenga más información de los pasos anteriores. Por otro lado, está la puerta de actualización que decide cuánta información del estado anterior se retiene en el nuevo estado. Combinada con el estado actual candidato, actualiza el estado oculto

de manera controlada. Las unidades recurrentes cerradas, simplifican el procesamiento y requieren menos parámetros que las LSTM, lo que facilita su entrenamiento y reduce la complejidad computacional. A pesar de esta simplificación, las GRU suelen lograr un rendimiento comparable al de las LSTM en muchas tareas (The Neural Perspective, 2018).

5.5 ALGORITMOS DE REDES NEURONALES PROFUNDAS EN SEPARACIÓN DE DIÁLOGOS.

La mejora de los diálogos se centra en la optimización de la claridad, la inteligibilidad y la naturalidad del habla capturada, ya sea en entornos ruidosos, transmisiones en vivo o material multimedia. Se emplean técnicas avanzadas de procesamiento de señales y aprendizaje automático para mitigar efectos adversos como ruido de fondo, efectos de sonido, y música. A través de algoritmos sofisticados, se busca no solo mejorar la calidad perceptual del habla, sino también adaptar el diálogo a diferentes contextos y usuarios.

Las aplicaciones de mejora del diálogo son diversas y van desde material multimedia (TV, vídeos, películas), permitiendo al usuario ajustar el volumen de los diálogos y fondo de forma independiente según preferencia personal, hasta sistemas de conferencia y telepresencia, donde la claridad del habla es crucial para la comunicación efectiva. Aunque la separación de fuentes de audio ha sido un tema de investigación importante desde hace muchos años, las técnicas especialmente pensadas para contenido de difusión no tan fácilmente disponibles. Entre otras razones, se pueden identificar factores como la necesidad de tener una alta calidad perceptual del habla y tasa de muestreo (generalmente 44-48 kHz, mientras que muchos algoritmos de voz están entrenados en 8-16 kHz) o la escasa cantidad de contenido de difusión real con señales de referencia limpias que puedan usarse para el entrenamiento (Strauss, y otros, 2021). En esta sección, se explicarán los avances recientes en tecnologías de mejora del diálogo, así como sus aplicaciones prácticas y desafíos actuales en la creación de experiencias auditivas más claras y efectivas.

5.5.1 Dialog +

Dialog + es una tecnología que ha sido desarrollada por el IIS Fraunhofer. Dialog + incorpora una red neuronal profunda para la separación de los diálogos. Esta red al igual que en el presente TFG se entrena mediante aprendizaje supervisado de regresión y utiliza una estructura completamente convolucional en el dominio de la STFT, esta estructura convolucional permite un alto rendimiento mientras se mantiene eficiente gracias al relativamente pequeño número de parámetros entrenables. Debido a que, en las pruebas de campo, tenía aproximadamente 370.000 parámetros entrenables, en comparación con

alrededor de 10 millones de parámetros entrenables en ‘Open Unmix’’ el cuál es un algoritmo para separación de fuentes musicales basado en aprendizaje profundo (Torcoli, y otros, 2021).

5.5.1.1 Funcionalidad de Dialog +

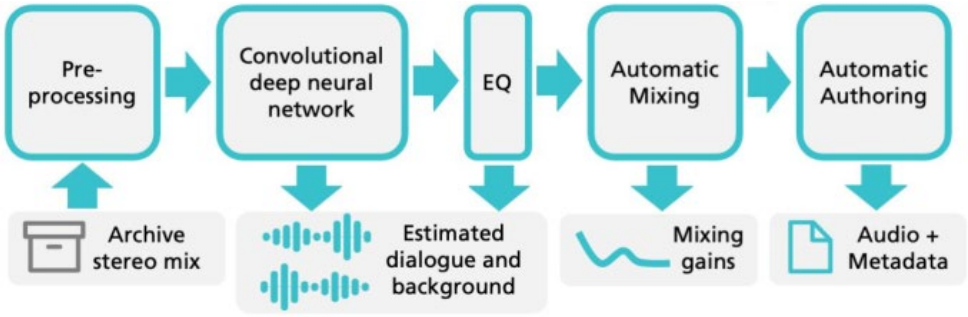


Figura 27. Bloques funcionales de Dialog + (Torcoli, y otros, 2021).

Para los datos de entrenamiento se utiliza contenido de transmisión del mundo real. El idioma predominante era el alemán, aunque se incluyeron otros como el inglés o el francés. Los audios sin procesar eran limpiados manualmente para excluir ejemplos con sonidos no relacionados con el habla en el diálogo o con cualquier habla en el fondo. Esto fue crucial para obtener resultados de alta calidad dando como salida la retención de sólo aproximadamente el 15 % del material original. El modelo empleado en la prueba en línea de una cadena de radio alemana se entrenó con aproximadamente 15 horas de contenido de audio estéreo a una frecuencia de muestreo de 48 kHz (Torcoli, y otros, 2021).

Se pudo observar que la calidad perceptual del habla separada del fondo se beneficia de un ligero aumento espectral constante en las principales frecuencias del habla de 1 a 4 kHz en el componente del diálogo. Este aumento se compensa atenuando el fondo en dicha banda, preservando la misma amplitud original en la mezcla, y no comprometiendo la mezcla predeterminada y al mismo tiempo mejorando la experiencia auditiva en condiciones de remezcla. Dialog + combina la separación de diálogos con la remezcla automática, permitiendo la atenuación del fondo de manera global y variable en el tiempo. Esto puede ser útil para reducir o eliminar el fondo, pero no siempre mejora la inteligibilidad del habla. Una solución es atenuar el fondo solo cuando el diálogo está activo. En las pruebas se usó una combinación de atenuación global leve y mayor durante la actividad del diálogo.

Finalmente, se pueden generar las siguientes salidas: 1) un archivo ADM (‘Audio Definition Model’), listo para su uso en flujos de trabajo basados en objetos, y 2) una pista de audio tradicional basada en canales con un nivel de diálogo mejorado, utilizada en las pruebas de campo documentadas.

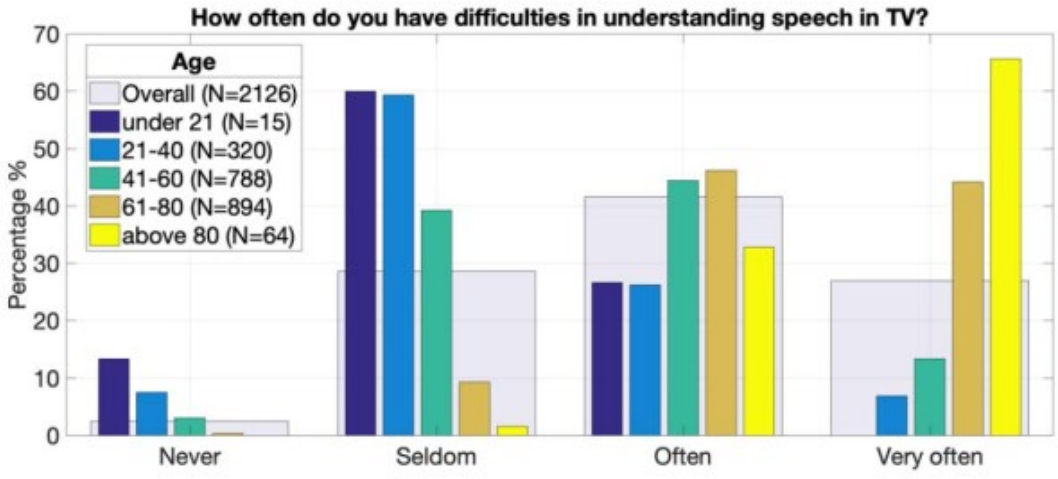


Figura 28. Problemas de comprensión del habla en TV (Torcoli, y otros, 2021).

Al realizar las pruebas de campo oportunas en diversos ámbitos como son un programa sobre fútbol y un episodio de una popular serie dramática policial. Se pudo observar cómo el 68 % de los participantes tenía problemas para comprender el habla en la televisión con frecuencia o muy a menudo, no obstante, el porcentaje aumentaba con la edad.

5.5.2 Redes de separación de música aplicadas a separación de diálogos.

El paper (Strauss, y otros, 2021) ofrece una comparación detallada de tres modelos de MSS de última generación aplicados a la separación del diálogo. Se emplean modelos avanzados de DNN inicialmente desarrollados para la separación de fuentes musicales, adaptándose para cada tarea específica de distinguir entre diálogo y otros sonidos de fondo como programas de televisión y películas. Este enfoque permite ajustar individualmente el volumen del diálogo y los sonidos de fondo como se realiza en el presente TFG, mejorando significativamente la inteligibilidad del habla y la experiencia auditiva del usuario. A través de técnicas de aprendizaje por transferencia y un ajuste con datos reales de radiodifusión, esta aplicación logra un rendimiento superior en la separación de diálogos, superando las limitaciones de los métodos tradicionales y ofreciendo una solución eficiente para la mejora de la calidad del audio en medios de transmisión. La implementación de esta tecnología utiliza bibliotecas de código abierto como Pytorch, permitiendo la ejecución eficiente de las redes neuronales en las GPU (Unidades de Procesamiento Gráfico). El uso de GPUs acelera el proceso de entrenamiento y permite la ejecución en tiempo real de la separación de los diálogos. Por otra parte, CUDA, una plataforma de cómputo paralelo desarrollada por NVIDIA y que se integra con Pytorch para aprovechar al máximo la capacidad de procesamiento de las tarjetas gráficas (Strauss, y otros, 2021).

Es de resaltar, que la aplicación no sólo separa el diálogo del fondo, sino que permite también una remasterización automática del audio. Esto incluye la atenuación global del fondo y ajustes dinámicos basados en la actividad del diálogo. La atenuación global reduce el nivel del fondo en toda la señal, mientras que la atenuación dinámica baja el nivel del fondo cuando solo hay actividad de diálogo, optimizando la inteligibilidad del habla sin comprometer otros aspectos del sonido (Strauss, y otros, 2021).

5.5.2.1 Formulación del Problema

Se trabaja con una mezcla de audio estéreo y, compuesta por una componente de diálogo x y una componente de fondo b:

$$y = x + b \tag{5}$$

El objetivo es extraer las componentes individuales separadas, $\hat{x}$ y $\hat{b}$, y mezclarlas con un nivel de fondo ajustado:

$$\hat{y} = \hat{x} + \mu\hat{b} \tag{6}$$

Se introduce un factor de atenuación que se encarga de reducir el nivel del fondo para que el diálogo sea más audible. Por ejemplo, un valor de <1 atenúa el fondo, haciendo que el diálogo se destaque más en la mezcla (Strauss, y otros, 2021).

5.5.2.2 Métodos de separación

Open-Unmix (UMX): Este modelo utiliza la STFT y está compuesto por una red LSTM bidireccional de tres capas, con capas de codificación y decodificación totalmente conectadas. Se entrena con el conjunto de datos MUSDB18-HQ y aplica un filtro de Wiener multi-canal durante la inferencia. (Stöter, y otros, 2019)

Spleeter: Spleeter es una red de tipo U-Net que también opera en el dominio STFT, utilizando 12 capas para estimar una máscara suave para cada fuente. Se entrenó con un conjunto de datos de Deezer y también usa filtrado Wiener en la inferencia. (Hennequin, y otros, 2020)

Conv-TasNet: (Stöter, y otros, 2019)Conv-TasNet es una versión multicanal del modelo Conv-TasNet utilizado para la separación de hablantes. Opera en el dominio temporal y aprende máscaras suaves para cada fuente. Se entrenó con 150 canciones adicionales para este estudio. (Défossez, y otros, 2021)

Además, se emplean sistemas específicos para la separación de los diálogos como son:

1. **Audiodinamix Instant Dialogue Cleaner (IDC):** Un plugin comercial que utiliza DNNs y opera en tiempo real. En este caso, fue obtenido para sacar el máximo rendimiento a la atenuación del fondo (Gear4Music, 2024).

2. **DNN Dialog Separation (DNN-DS):** Un sistema en desarrollo que utiliza una estructura convolucional en el dominio STFT. Considerando 1 segundo de información de la señal, lo que ayuda en la separación precisa de los componentes de audio (Strauss, y otros, 2021).

5.5.2.3 Estrategia de entrenamiento y métricas computacionales

Para asegurar una comparación justa entre los modelos, se implementaron varias estrategias de entrenamiento como son la selección aleatoria de muestras de entrenamiento de cada clip de audio disponible, esto permite que el modelo se exponga a una amplia variedad de datos durante el entrenamiento, mejorando su capacidad de generalización (Strauss, y otros, 2021).

La duración de los datos de entrada en cada red neuronal es diferente: UMX-6 segundos, Conv-TasNet-2.8 segundos y Splitter-12 segundos. Para asegurar que cada modelo se entrenara con una cantidad similar de datos por época, los elementos de UMX y Conv-TasNet se muestrearon dos y cuatro veces, respectivamente, para igualar la duración de los datos de entrada de Spleeter. Además, se incluye un decaimiento de la tasa de aprendizaje por un factor de 0.3 para ayudar al modelo a que realice ajustes más finos en sus pesos y mejorar su precisión al acercarse a la solución óptima. El número máximo de épocas se limita a 100 para evitar un entrenamiento excesivamente prolongado y el riesgo de sobreajuste (Strauss, y otros, 2021).

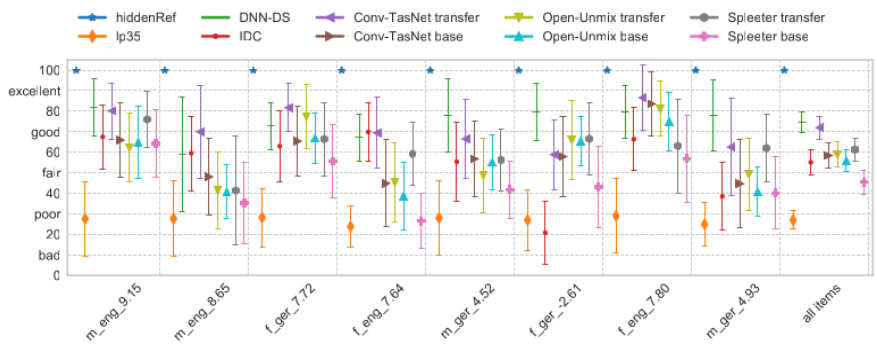


Figura 29. Problemas de comprensión del diálogo (Strauss, y otros, 2021)

Para la evaluación de los modelos se utilizaron tanto métricas objetivas como subjetivas. En particular, en la Figura 29 se presentan los resultados de una prueba de escucha subjetiva, donde los oyentes evaluaron la calidad percibida de las fuentes separadas por distintos modelos. En estas pruebas, DNN-DS obtuvo la mejor puntuación en general, destacando en condiciones de menor relación señal a interferencia de entrada, mientras que Conv-TasNet también mostró un rendimiento competitivo, especialmente en condiciones de mayor relación señal a interferencia.

5.6 SEPARACIÓN EN TIEMPO REAL CON REDES NEURONALES PROFUNDAS

5.6.1 Separación en tiempo real

Uno de los mayores desafíos para la mejora de los diálogos es poder realizar la separación en tiempo real con baja latencia. Sin embargo, se ha prestado poca atención a cómo estas redes neuronales pueden adaptarse para aplicaciones en tiempo real con baja latencia. Las soluciones más avanzadas de la industria y el mercado que se encargan de emplear optimizaciones en el procesamiento de la señal y modelos de aprendizaje profundo están diseñadas para funcionar de manera eficiente en entornos con restricciones temporales. Uno de los objetivos principales que se han abordado en este trabajo y en la industria, es llegar a un equilibrio entre la complejidad computacional que puede presentar un modelo de red neuronal con la capacidad de poder reducir el ruido de fondo y mejorar el habla. Todo ello, debido a que los dispositivos comúnmente utilizados tienen limitaciones computacionales o tienen recursos limitados (Venkatesh, y otros, 2024).

Antes del avance del aprendizaje profundo, se introdujeron técnicas como la Discriminación de Azimut (ADR) en la que se utilizan múltiples micrófonos para captar las señales de audio desde diferentes posiciones y resíntesis, la cual se realiza seguida a ADR, y consiste en reconstruir las señales de audio originales a partir de la información azimutal obtenida en el paso anterior, para la separación de fuentes en tiempo real. Para los algoritmos basados en STFT destinados a la mejora del habla, se suele observar una latencia de 32 ms debido al tamaño necesario de la ventana. Por otro lado, el aprendizaje profundo de extremo a extremo, aplicado directamente sobre audio en crudo ‘raw audio’ para la separación de diálogos, ha logrado resultados impresionantes con latencias tan bajas como 2 ms (Venkatesh, y otros, 2024). Los retos y propuestas actuales para la separación en tiempo real se basan en las arquitecturas de redes neuronales, las cuales se pueden dividir en tres enfoques principales: basadas en espectrogramas, basadas en forma de onda y enfoques híbridos. Las arquitecturas de espectrograma usualmente utilizan la magnitud y descartan la fase, es decir, predicen una máscara de magnitud y

aplican la fase de la mezcla de entrada para obtener la salida final. Las arquitecturas de forma de onda, operan directamente en el dominio temporal, separando el audio crudo sin descartar la fase. Por último, la arquitectura con un enfoque híbrido, combina las ventajas tanto del dominio espectral como del dominio temporal (Venkatesh, y otros, 2024).

5.6.2 Restricciones de baja latencia

Para desarrollar un sistema de desmezcla o 'unmixing' en tiempo real es esencial considerar varios factores. El primer aspecto es la latencia algorítmica, que indica cuánto contexto se necesita para generar una trama o 'frame' de audio. Esta latencia puede ser influenciada por operaciones como el solapamiento suma ('Overlap-Add'), que permite introducir un bajo retardo, o el enventanado de anticipación ('look-ahead-window'), que analiza un fragmento de audio por adelantado para prever el contenido y ajustar el procesamiento de la señal de audio en consecuencia. Esto proporciona información adicional al algoritmo para mejorar la precisión del procesamiento. El segundo aspecto importante es la eficiencia computacional, es decir, el tiempo que tarda un algoritmo en procesar un segmento de audio. Por ejemplo, si el algoritmo utiliza muchas ventanas de anticipación o la red neuronal tiene muchos bloques que requieren un procesamiento secuencial, se debe considerar que el algoritmo no es eficiente computacionalmente (Venkatesh, y otros, 2024).

En el paper (Venkatesh, y otros, 2024), para asegurar que el modelo de red neuronal aplicable en este caso fuera viable en un entorno de tiempo real, se tomaron en cuenta ciertos parámetros de diseño. Para un tamaño de salto 'Hop Length' específico, el tiempo promedio de *inferencia* en una unidad central de procesamiento (CPU) de 4 núcleos debe ser menor del 50% del tamaño del salto, permitiendo un margen razonable para la actualización de bloques de audio.

5.6.3 Adaptaciones de baja latencia

Modelo X-UMX: es un modelo que se basa en espectrogramas y cuya salida pasa por un filtro de Wiener para mejorar la separación y minimizar las posibles interferencias entre fuentes. Para la implementación de este modelo, se reemplazan los LSTM bidireccionales por LSTM unidireccionales causales con la finalidad de reducir la latencia. El tamaño de la ventana pasa de ser 4096 muestras a 1024 muestras, con un tamaño de salto de 1024 muestras a 512 muestras. Como la implementación se desea en tiempo real, no se utiliza filtrado de wiener debido a que este algoritmo de maximización de expectativas requiere ventanas de anticipación para un rendimiento óptimo y esto resulta no ser nada interesante para una implementación en tiempo real (Venkatesh, y otros, 2024).

En el contexto de MSS, el modelo X-UMX ha mostrado una caída significativa en SDR al excluir el filtro de Wiener y cambiar los LSTM bidireccionales por unidireccionales. Con una latencia de 93 ms, el SDR general se redujo a 4.57, considerablemente menor que el valor original de 5.79. Al disminuir la latencia a 23 ms, el SDR cayó aún más a 3.93. Estos resultados ponen de manifiesto la importancia del contexto futuro y los tamaños de ventana más grandes en la separación de fuentes, aspectos que no son viables en el procesamiento en tiempo real (Venkatesh, y otros, 2024).

Modelo TasNet: El modelo TasNet se diseñó para mejorar su rendimiento en un entorno causal lo cual quiere decir que las salidas del modelo dependen de las entradas actuales y pasadas, pero no de las futuras. Estudios previos al modelo sugerían que tamaños de ventana más pequeños conducían a un mejor rendimiento, pero en este caso los experimentos iniciales demostraron que un tamaño de 1024 muestras (aproximadamente 5 ms a una tasa de muestreo de 44100 Hz) era más efectivo, proporcionando un mayor contexto de audio en cada paso temporal. En cuanto a LSTM, se estableció el número de unidades ocultas en los LSTM unidireccionales a 1000, y se añadió una conexión de salto de identidad entre dos capas de LSTM. La capa final de TasNet es una convolución transpuesta que emplea filtros aprendidos. Para evitar ciertas discontinuidades en la salida de audio, se aplicó una ventana de hanning a cada uno de los filtros durante la convolución transpuesta (Venkatesh, y otros, 2024).

El modelo TasNet logró un mejor SDR que el X-UMX con una latencia de 23 ms. Sin embargo, se observó que el modelo tuvo dificultades para generar altas frecuencias en el espectro. Este problema está relacionado con la función de pérdida utilizada, ya que al aplicar L1, que permite prevenir el sobreajuste, directamente en el dominio de la forma de onda, se corre el riesgo de sobreponderar las frecuencias bajas. Estudios sobre la separación de voz han demostrado que la relación señal a ruido independiente de la escala (SI-SNR) y el SDR dependiente de la escala ofrecen un rendimiento superior comparado con L1/L2 en los modelos TasNet. No obstante, en los experimentos realizados con métricas de evaluación como MSS, SI-SNR y SD-SDR resultaron en un peor rendimiento y una convergencia más lenta. Esta situación podría deberse a la mayor correlación entre las fuentes o a la ampliación de datos utilizada en los experimentos.

Además, se exploró el uso de la pérdida multidominio, que calcula la pérdida tanto en los dominios de frecuencia como de tiempo. Este enfoque permitió que la red generara frecuencias más altas, aunque la salida mostró un mayor filtrado entre las fuentes y no mejoró globalmente al usar L1.

En la Figura 30, se observan los diferentes tipos de modelos que implementan la separación de fuentes de audio proporcionando una comparación detallada de varias implementaciones en relación con últimos modelos de separación. En la Figura 30, se enfoca el rendimiento en términos de 'SDR', latencia, y el tiempo de procesamiento utilizando diferentes configuraciones de hardware como son la CPU y la GPU. La comparación de los modelos X-UMX, TasNet y HS-TasNet en condiciones de baja latencia revela diferencias significativas en rendimiento y eficiencia computacional. El modelo X-UMX mostró una notable disminución en el SDR cuando se implementaron ajustes para baja latencia, lo que resalta la necesidad de contexto futuro y tamaños de ventana mayores para un mejor rendimiento. TasNet, aunque obtuvo un mejor SDR que X-UMX a 23 ms de latencia, tuvo dificultades para generar frecuencias altas de manera efectiva, debido a las limitaciones de la función de pérdida L1. Por otro lado, el modelo HS-TasNet, que es una arquitectura híbrida que combina enfoques basados en espectrogramas y formas de onda, demostró ser superior en términos de rendimiento general y capacidad para manejar frecuencias altas, con menos filtraciones entre fuentes, especialmente cuando se utilizó con datos adicionales. Además, HS-TasNet-Small proporcionó una buena combinación de rendimiento y eficiencia computacional, mostrando ser el más eficiente en términos de tiempo de procesamiento en varias configuraciones de hardware. Estos resultados subrayan la importancia de elegir la arquitectura del modelo y ajustar adecuadamente las configuraciones de datos para lograr una separación de fuentes eficaz y precisa en tiempo real (Venkatesh, y otros, 2024).

	Architecture	All	Vocals	Drums	Bass	Other	Extra?	Latency (ms)	Param.	1-core (ms)	4-core (ms)	GPU (ms)
Non-causal models	Conv-TasNet [9]	5.73	6.43	6.02	6.20	4.27	✗	2,000	9 M	**52.29**	**46.50**	**16.77**
	X-UMX [16]	5.79	6.61	6.47	5.43	4.64	✗	8,000	36 M	-	-	-
	DemucsV2 [9]	6.28	6.84	6.86	7.01	4.42	✗	8,000	288 M	-	-	-
	DemucsV4 [23]	**7.52**	**7.93**	**7.94**	**8.48**	**5.72**	✗	7,800	41 M	-	-	-
Non-causal models with extra data	Conv-TasNet [9]	6.32	6.74	7.11	7.00	4.44	150	2,000	9 M	52.29	46.50	16.77
	X-UMX [32]	6.52	7.57	7.39	6.28	4.83	1505	8,000	36 M	-	-	-
	TasNet [22]	6.52	7.34	7.68	7.04	4.04	2500	15,000	29 M	**11.75**	**4.20**	**2.38**
	DemucsV2 [9]	6.79	7.29	7.58	7.60	4.69	150	8,000	288 M	-	-	-
	DemucsV4 [23]	**9.20**	**9.37**	**10.83**	**10.47**	**6.41**	800	7,800	41 M	-	-	-
Real-time Low-Latency models	X-UMX†	3.93	4.65	4.36	3.79	2.92	✗	23	31 M	9.85	4.06	**1.80**
	TasNet†	4.40	5.02	4.38	**4.73**	3.48	✗	23	51 M	9.81	4.45	1.90
	HS-TasNet†	**4.65**	5.13	**5.22**	4.59	**3.64**	✗	23	42 M	9.10	4.26	3.90
	HS-TasNet-Small†	4.48	**5.21**	4.89	4.42	3.42	✗	23	16 M	**3.98**	**1.83**	2.10
Real-time Low-Latency models with extra data	X-UMX†	4.10	4.74	4.62	3.76	3.28	150	23	31 M	9.85	4.06	**1.80**
	TasNet†	4.92	5.22	5.54	5.13	3.78	150	23	51 M	9.81	4.45	1.90
	HS-TasNet†	**5.55**	**5.97**	**6.34**	**5.62**	**4.28**	150	23	42 M	9.10	4.26	3.90
	HS-TasNet-Small†	5.01	5.51	5.75	4.93	3.86	150	23	16 M	**3.98**	**1.83**	2.10

Figura 30. Comparación de procesos de baja latencia (Venkatesh, y otros, 2024).

5.6.3.1 Esquema de almacenamiento en Búfer en tiempo real

El almacenamiento en búfer en tiempo real es otro de los aspectos clave para el procesamiento del audio en tiempo real. En el procesamiento fuera de línea u 'offline', las señales de audio se solapan en grandes ventanas de contexto y se concatenan antes de la reproducción. Sin embargo, para un entorno en tiempo real, debe salir un flujo de audio

constante de audio procesado y los segmentos de salida consecutivos deben ser continuos y de pequeño tamaño. El método que se muestra en la Figura 32, aborda este problema, con la recomendación de un solapamiento de un 75 % lo cual significa que, en cualquier instante temporal, 4 segmentos de análisis están contribuyendo activamente a la trama de salida actual, de esta forma se podría extrapolar a la forma en la que se implementa el buffer de entrada y de salida en el presente trabajo. Además, en la Figura 31 se puede apreciar como la señal de audio de entrada se ha dividido en segmentos solapados de longitud N. Para poder emitir un segmento procesado, se deben procesar y solapar los 4 segmentos completos. Esto conlleva a una latencia considerable desde el momento en el que se realizan cambios en los parámetros hasta que se producen los efectos resultantes en la salida. Dado que el tamaño del salto de síntesis Rs se fija igual al tamaño del salto de análisis Ra, se puede cargar y procesar una sola trama de longitud N, emitir una cuarta parte de la trama y retener el resto en el búfer para poder superponerlo con el audio en las tramas de salida sucesivas. Para ello, se requiere un búfer de longitud N, para colocar la trama procesada actualmente con la ventana correspondiente. Adicionalmente, se necesitan tres búferes más de longitud N para almacenar los segmentos restantes de las tres tramas procesadas previamente. Cada una de las tramas de salida de longitud N se genera sumando muestras de estos cuatro búferes (Barry, 2019).

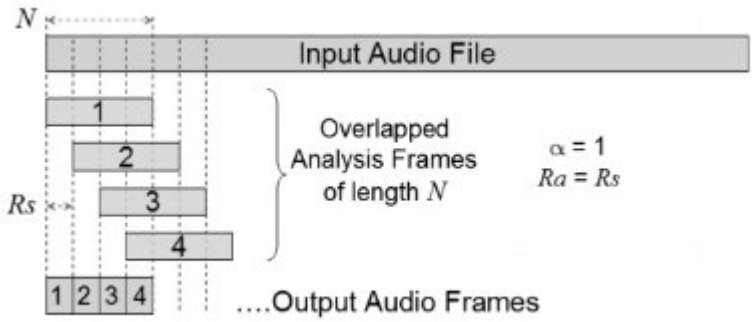


Figura 31. Almacenamiento en búfer offline (Barry, 2019).

En cada iteración de la Figura 32, se procesa una trama completa Fu de longitud N y se coloca en el búfer 1. Las muestras restantes de las tres tramas anteriores ocupan los búferes 2, 3 y 4. La trama o segmento de salida deseado de longitud N, Su, se genera sumando las primeras muestras de cada búfer, según se define en la ecuación x. Una vez generada y emitida la salida, las primeras muestras de cada búfer se van descartan, y los datos en todos los búferes se desplazan para prepararse para la siguiente iteración. El orden de desplazamiento es esencial en los buffers, el búfer 4 se llena con las muestras restantes del búfer 3, el búfer 3 se llena con las muestras restantes del búfer 2, y finalmente,

el búfer 2 se llena con las muestras restantes del búfer 1. Por lo que, el búfer 1 queda vacío y listo para recibir el siguiente segmento procesado de longitud N.

$$S^u(n) = F^u(n) + F^{u-1}\left(n + \frac{N}{4}\right) + F^{u-2}\left(n + \frac{N}{2}\right) + F^{u-3}\left(n + \frac{3N}{4}\right) \quad (3)$$

$$\forall\, n \quad 1 \leq n \leq \frac{N}{4} \quad (4)$$

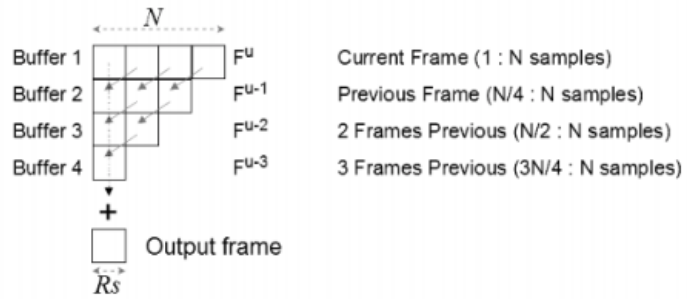


Figura 32. Almacenamiento en búfer en tiempo real (Barry, 2019).

Este esquema permite emitir una cuarta parte de una trama procesada en intervalos de Rs, igual a 1024 muestras. Con un tamaño de trama o segmento de 4096 muestras, la salida se actualizará cada 1024 muestras, lo que equivale aproximadamente a 23.2 milisegundos. Aunque el audio se procesa con parámetros recién actualizados cada 23.2 milisegundos, la latencia total será mayor y dependerá del tiempo necesario para acceder y escribir en los búferes de hardware de la interfaz de audio. En general, es posible lograr latencias menores a 40 ms, asegurando que el procesamiento de audio en tiempo real sea eficiente y continuo, manteniendo la calidad y minimizando la latencia.

5.7 DEMUCS

DEMUCS (Deep Extractor for Music Sources) es un modelo avanzado de separación de fuentes cuyo objetivo es la separación de fuentes musicales desarrollado por Facebook AI Research. Fue presentado en el artículo "Hybrid Spectrogram and Waveform Source Separation". Ha marcado un hito en la separación de fuentes musicales, ofreciendo una solución robusta y eficaz que opera en el dominio de la forma de onda (en sus primeras versiones). En el presente TFG, se ha empleado la versión de DEMUCS híbrida con ciertas modificaciones relativamente sencillas para procesar audio en tiempo real con bajo retardo, lo cual constituye el pilar fundamental de la aplicación que se explicará posteriormente. DEMUCS emplea una arquitectura de red neuronal convolucional de tipo U-Net la cual fue introducida por primera vez para la segmentación de imágenes biomédicas, es particularmente efectiva en tareas de separación de audio debido a su gran

capacidad para capturar características a múltiples escalas y combinar información de diferentes niveles de red. Además, presenta un LSTM bidireccional entre el codificador y decodificador. Su estructura de codificación y decodificación con 'Conexiones U-Net' entre capas correspondientes permite una detallada reconstrucción de la salida. DEMUCS se destaca por su capacidad de superar a las arquitecturas existentes en términos de SDR, especialmente con un aumento adecuado de datos (Défossez, y otros, 2021).

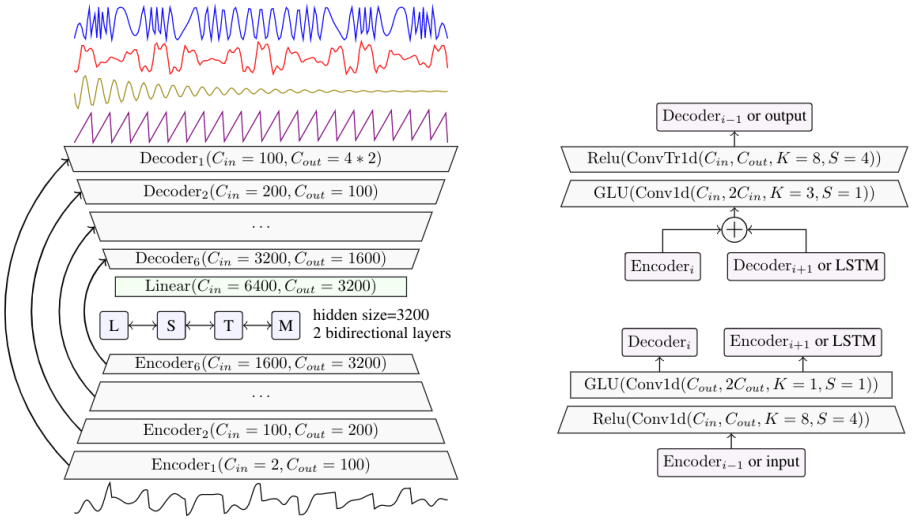


Figura 33. Arquitectura Original de DEMUCS (Défossez, y otros, 2021).

5.7.1.1 Arquitectura híbrida y metodología

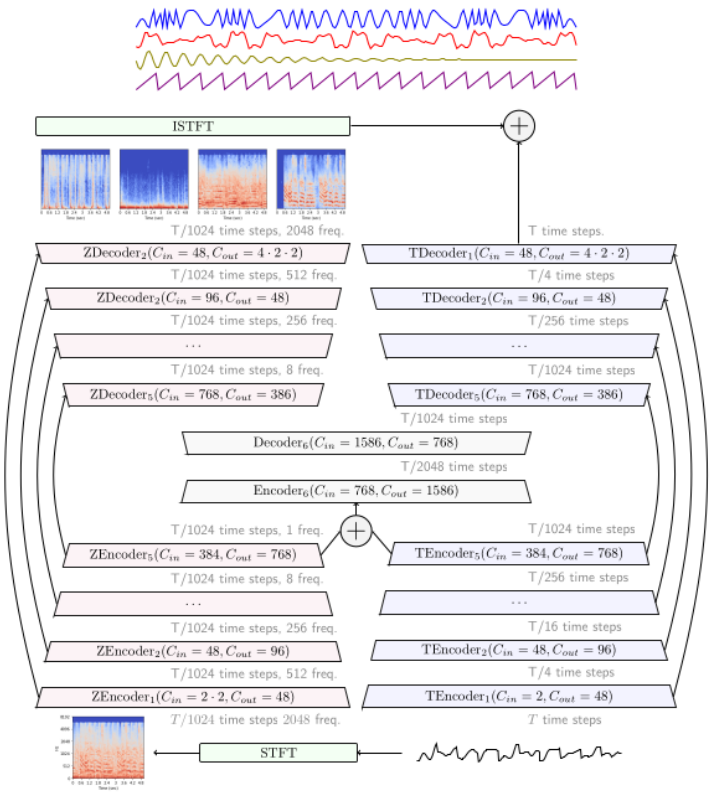


Figura 34. Arquitectura Hybrid Demucs (Défossez, y otros, 2021)

La arquitectura DEMUCS se basa en un enfoque de red neuronal convolucional, extendido con características avanzadas para mejorar su rendimiento en la separación de fuentes de audio la utilizada en este TFG, DEMUCS (v3). Esta arquitectura, conocida como DEMUCS híbrido, está compuesta por capas convolucionales para la codificación (Encoder) y decodificación (Decoder) de la señal de audio. Al trabajar directamente con la forma de onda, DEMUCS puede captar y separar mejor las componentes temporales y frecuenciales de la señal de audio. Una característica clave de U-Net son las conexiones de salto (“skip connections”) entre las capas correspondientes del encoder y decoder. Estas conexiones permiten combinar características de diferentes niveles de resolución, mejorando la precisión y la calidad de la salida, es decir, en el contexto de este TFG para la separación de fuentes de audio, permite al modelo capturar tanto las características locales como globales de la señal de audio, lo que resulta en una separación más efectiva en las diferentes fuentes (Défossez, y otros, 2021).

Esta versión del modelo DEMUCS tiene mejoras en su arquitectura con respecto a versiones anteriores: ramas residuales comprimidas, que utilizan convoluciones dilatadas, y capas LSTM bidireccionales para capturar contextos a largo plazo. Presenta un

mecanismo de atención local que permite al modelo enfocarse en segmentos específicos de la señal.

5.7.1.2 Datos de entrada y salida

Los datos de entrada al modelo de DEMUCS híbrido son señales de audio, que se pueden representar en el dominio temporal o espectral. En forma de onda, la señal de audio en su forma original puede ser procesada directamente en la rama temporal, mientras que, con el espectrograma de la señal, que se obtiene realizando la STFT, se descompone la señal en sus componentes frecuenciales.

Los datos de salida del modelo son cada una de las señales de audio separadas correspondientes a cada fuente de audio en la mezcla. En este TFG, es muy importante la salida ya que proporciona la voz y el resto de sonidos que forman el fondo.

En la Figura 34, se puede observar cómo tanto el codificador como el decodificador tienen una estructura simétrica. Cabe destacar que los datos de entrada en el dominio del tiempo y de la frecuencia, pasan por cada codificador con un determinado número de canales el cual hace uso de kernels 3D en las capas convolucionales, aumentado el número de canales debido al uso del mismo, ese aumento de canales puede suponer lo grande o compleja que puede ser la red neuronal por dentro. A medida que el espectrograma pasa por las capas del codificador espectral, se aplican convoluciones a lo largo de la dimensión de frecuencia. Cada capa del codificador reduce progresivamente el número de frecuencias hasta que solo queda una frecuencia restante. Así, el número de canales y la frecuencia de muestreo convergen en la última capa para coincidir con los de la salida de la rama temporal. Una vez que ambas ramas (temporal y espectral) han procesado la señal de audio hasta un punto equivalente (con una frecuencia restante en la rama espectral y la correspondiente resolución en la rama temporal), sus representaciones se combinan. Esta combinación se realiza antes de pasar a una capa codificador/decodificador compartida. La representación combinada se pasa por una capa que actúa tanto como codificador adicional como decodificador, preparando la señal combinada para la decodificación final. La salida de la rama espectral en la última capa del decodificador, que está en el dominio del espectrograma, se convierte de nuevo a una forma de onda mediante la ISTFT. Esta señal en forma de onda se suma a la salida de la rama temporal, obteniendo así la predicción final del modelo. Esta combinación permite que el modelo utilice la información procesada en ambos dominios (temporal y espectral) para mejorar la calidad de la separación (Défossez, y otros, 2021).

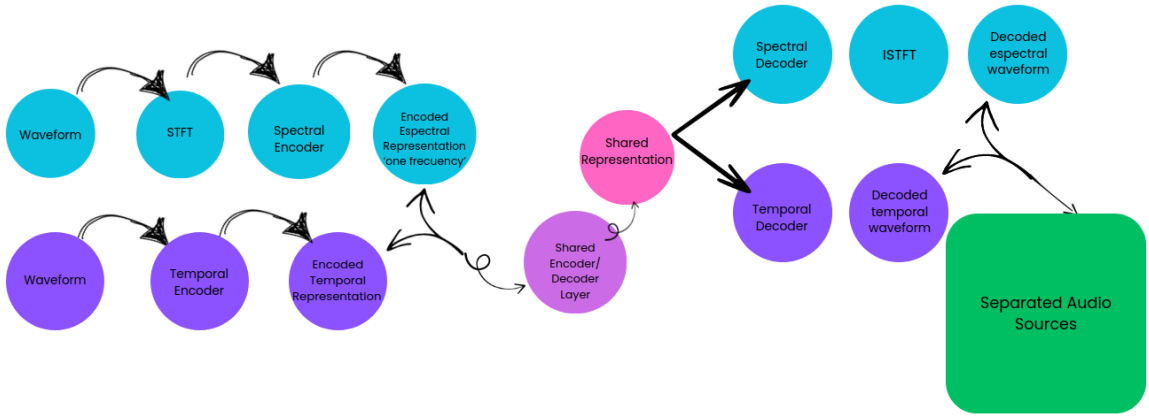


Figura 35. Representación lógica del funcionamiento de Hybrid Demucs.

5.7.1.3 Muestras utilizadas

Tanto para el entrenamiento, como la evaluación del modelo, se utiliza un conjunto de datos MUSDB-HQ que contiene 150 pistas de música con los correspondientes stems (componentes individuales de la mezcla). Este conjunto se divide en 86 pistas para entrenamiento, 14 para validación y 50 para prueba. Además, se puede emplear un conjunto de datos interno adicional y realizar técnicas de mezcla realista para crear más ejemplos de entrenamiento.

5.7.1.4 Entrenamiento y modelos pre-entrenados

En los parámetros de entrenamiento, como dispositivo de entrenamiento se utiliza GPU (CUDA) cuando está disponible, de lo contrario, se utiliza CPU. Generalmente, los modelos se entrenan durante 600 épocas. Inicialmente el modelo se entrena en MUSDB HQ, utilizando técnicas como la normalización por capas y regularización de valores singulares para estabilizar el entrenamiento. Este entrenamiento se realiza utilizando la biblioteca de PyTorch, aprovechando la capacidad de procesamiento en tarjeta gráfica para manejar el gran volumen de datos y la complejidad del modelo. Posteriormente, el modelo se afina en un conjunto de datos mixtos creado a partir de stems combinados de diferentes pistas, ajustados para la compatibilidad de ritmo y tono.

- **DEMUCS v2:** versión original del modelo DEMUCS, diseñada para separar fuentes de audio en cuatro componentes: voces, percusión, bajo y otros.

- **Hybrid DEMUCS:** Una versión mejorada que combina análisis en el dominio temporal y espectral. Este modelo fue desarrollado para mejorar la calidad de separación al usar ambos dominios. Ideal para aplicaciones que requieren alta calidad.

- **MDX-Net:** Un modelo desarrollado específicamente para la competencia Music Demixing Challenge (MDX). Combina enfoques de DEMUCS con técnicas adicionales para mejorar la separación. Sólo entrenado en MUSDB HA. Utilizado principalmente en contextos de investigación, ofreciendo rendimiento de vanguardia en la separación de música.

6. MATERIALES Y MÉTODOS

El presente TFG se centra en la adaptación, implementación y evaluación de un modelo de red neuronal, Demucs v3, para la separación en tiempo real de los diálogos respecto del fondo. El objetivo principal del mismo es la elaboración de una aplicación de escritorio mediante el lenguaje de programación Python para mejorar la inteligibilidad del habla para personas con dificultades auditivas o que encuentran perturbadores los ruidos de fondo en medios audiovisuales, como puede ser una película o un programa de televisión. Para poder alcanzar tal fin, se han seguido una serie de pautas metodológicas. En primer lugar, se ha realizado una adaptación del modelo híbrido de DEMUCS versión 3 para la separación de las fuentes de audio en el caso de la voz y fondo. Además, se ha realizado una configuración determinista para reducir el retardo en el procesamiento de audio en tiempo real. Esta adaptación ha implicado la sustitución del propio solapamiento-suma original de DEMUCS por uno optimizado que permite un menor retardo, que es uno de los objetivos en este trabajo. A continuación, se realizaron pruebas en un entorno de tiempo real utilizando la tarjeta gráfica del ordenador usando un modelo entrenado de menor complejidad (menor número de canales) que el ofrecido por DEMUCS (el cual puede ser muy complejo para ciertas GPUs). Este paso ha sido fundamental para verificar la eficiencia del modelo en condiciones reales. El entrenamiento dado a la red neuronal ha sido realizado por el profesor-tutor Pablo Antonio Cabañas Molero. Posteriormente, se desarrolla un sistema capaz de capturar el audio proveniente de aplicaciones específicas, procesarlo y capturarlo mediante el modelo para separar el diálogo del fondo y reproducirlo a través de los altavoces. Finalmente, se ha desarrollado una interfaz gráfica de usuario mediante el lenguaje de programación Python, que permite seleccionar al usuario el volumen de la voz y del fondo de forma independiente en tiempo real.

Asimismo, esta sección se ha configurado para explicar detalladamente los materiales y métodos que han sido utilizados en la separación de la voz del resto de sonidos, junto con el procesado en tiempo real con bajo retardo como característica relevante, y los diferentes elementos empleados para el desarrollo de la aplicación de usuario. Todo este desarrollo se fundamenta en un exhaustivo análisis del estado del arte, que incluye conceptos y algoritmos relacionados con la separación de fuentes de audio, la separación en tiempo real, conceptos de ‘speech enhancement’ y algoritmos de redes neuronales aplicados a esta tarea. Este análisis ha permitido identificar las mejores prácticas, técnicas y modelos para alcanzar los objetivos propuestos.

6.1 ENFOQUE GENERAL

En la sección actual, se pretende aclarar al lector cual ha sido la evolución para el desarrollo de las soluciones adoptadas y aplicadas para alcanzar los diferentes objetivos del proyecto. Se proporciona una visión general de las herramientas utilizadas, y la manera en que se han aplicado, para abordar la problemática planteada en este TFG:

1. Esta solución se ha fundamentado en el uso del sistema operativo Linux debido a su naturaleza de código abierto, lo cual lo convierte en una opción preferida en el ámbito de la inteligencia artificial. Linux proporciona una amplia gama de herramientas y paquetes gratuitos que son extensamente utilizados por la comunidad de desarrollo de IA, facilitando así la instalación y el flujo de trabajo. Para poder llevar a cabo las modificaciones y adaptaciones del código del modelo híbrido de DEMUCS para la separación en tiempo real, así como el desarrollo de la aplicación de escritorio, se ha utilizado Anaconda como distribución de python de código abierto. A través de Anaconda se ha creado un entorno para trabajar y tener los paquetes necesarios para la realización del TFG aislados de los demás proyectos, así las dependencias del presente trabajo no afectan a otros proyectos. Este entorno de trabajo ha asegurado una configuración óptima para la experimentación y el desarrollo en el ámbito de la inteligencia artificial, permitiendo alcanzar los objetivos propuestos de manera eficiente.

2. En este TFG, hemos empleado un fichero de pesos pre-entrenados para DEMUCS con un número de canales inferior al de los ficheros de pesos proporcionados por el repositorio oficial. DEMUCS utiliza por defecto 48 canales en su primera capa convolucional, resultando en una red demasiado compleja para ciertos equipos si se pretende procesamiento en tiempo real. Hemos reducido este número de canales a 32, lo cual nos ha obligado a entrenar nuestro propio fichero de pesos. El entrenamiento del modelo fue llevado a cabo por uno de los tutores de este TFG, sobre el conjunto de datos de entrenamiento de MUSDB18-HQ durante aproximadamente 300 épocas. Este modelo obtiene peores resultados de separación que los modelos publicados por DEMUCS debido a su menor número de parámetros. Así mismo, hay que destacar que los modelos públicos de DEMUCS fueron entrenados en un conjunto de canciones más amplio que el de MUSDB18-HQ, con canciones extra no disponibles públicamente. En la sección de resultados se detalla la calidad de la separación y el tiempo de ejecución alcanzados por nuestro fichero de pesos.

3. El tercer paso implica la adaptación del modelo de red neuronal DEMUCS para habilitar el procesamiento de audio en tiempo real con baja latencia. Una de las modificaciones clave es la creación de un script en Python para definir el procesamiento en tiempo real denominado *realtime.py*, en el cual se definen diversos parámetros para la separación de la voz y el fondo, tales como el tamaño del segmento en segundos, salto en muestras (latencia) y el dispositivo empleado para el procesamiento dando la posibilidad de utilizar CUDA ó CPU. Se implementa una técnica de solapamiento-suma diferente a la predeterminada en DEMUCS, utilizando mitades de ventanas de Hanning, debido a que al solapamiento original del modelo no permitía seleccionar saltos muy pequeños, lo que imposibilita un bajo retardo necesario para nuestro propósito. Esta modificación implica, sin embargo, un aumento considerable de la complejidad computacional, mayor cuanto menor sea la latencia que se pretende alcanzar. Se llevan a cabo diversas pruebas de rendimiento en la tarjeta gráfica NVIDIA GeForce GTX 1650 del ordenador empleado en el presente trabajo. Usando el modelo indicado en el punto anterior, se consigue la ejecución en tiempo real en dicha GPU con un bajo retardo.

4. Se emplea el paquete de Python sounddevice para realizar la lectura y reproducción de audio desde nuestra aplicación. Una de las utilidades más importantes instaladas es PulseAudio, esencial para la gestión y configuración de audio en el sistema. Para habilitar el enrutamiento de audio, se crean inicialmente dos dispositivos de audio virtual. Se configura una interfaz de entrada de audio de tipo monitor, denominada “Monitor of Loopback Analog Stereo” que permite inspeccionar todo el flujo enviado a la interfaz de salida “Loopback Analog Stereo”. La aplicación que se encarga de generar audio envía el mismo a través de la interfaz de salida “Loopback Analog Stereo”, mientras que el modelo modificado de Demucs, encargado de la separación de la voz y el fondo en tiempo real, captura el flujo de audio para realizar la segmentación y separación de la voz del resto de acompañamiento a través de la interfaz de entrada “Monitor of Loopback Analog Stereo” sacando el resultado de la separación a través de los altavoces del sistema. Toda esta configuración se realizó mediante comandos de Linux y el enrutamiento de audio se realizó manualmente utilizando la herramienta de control de audio de Linux, Pavucontrol, para poder establecer las entradas y salidas de audio virtual. Esto

garantizó que la clase RealTimeModel en el script de Python *realtime.py* se alimentara con datos capturados desde la fuente de audio.

5. Con el objetivo de ofrecer una solución efectiva para personas con problemas auditivos frente a la presencia de ruido de fondo en diversas situaciones audiovisuales, se ha desarrollado una interfaz gráfica, es decir, una aplicación de escritorio integrando el modelo modificado de DEMUCS para la separación en tiempo real de la voz y el fondo. Esta aplicación permite al usuario interactuar con la mejora del diálogo a través de barras de volumen independientes para ajustar el volumen de la voz y el fondo. Además, se ha implementado una barra de retardo (delay) que permite al usuario ajustar el retardo según las capacidades computacionales de su sistema, eligiendo el retardo óptimo que proporciona una reproducción en tiempo real y una sincronización coordinada entre audio y video. La aplicación es capaz de capturar el audio procedente de una plataforma específica cuyos diálogos se desean separar, e incluso del sistema completo, permitiendo enrutar el audio de manera automática para obtener el resultado de la separación de fuentes en el caso de la voz y el fondo a través de los altavoces.

6. Para poder evaluar la separación lograda con la modificación del modelo de DEMUCS en comparación con el original, se ha llevado a cabo un análisis utilizando métricas computacionales estándar como son SDR, SIR y SAR. Estas métricas se calculan mediante una función en Python llamada *metricas_calidad.py* y se detalla en el capítulo de *Resultados y Discusión* .

6.2 IMPLEMENTACIÓN DE LA SOLUCIÓN

Para poder abordar los diferentes objetivos de este TFG y desarrollar una solución eficaz para la mejora de la inteligibilidad del habla, se ha llevado a cabo una serie de pasos. A continuación, se describen las herramientas utilizadas, el proceso de adaptación de DEMUCS para su ejecución en tiempo real, y la configuración necesaria para capturar y reproducir audio de manera eficiente.

6.2.1 Equipo de desarrollo

Para realizar esta propuesta de TFG se ha utilizado un portátil ASUS-TUF GAMING F15 con sistema operativo Linux, específicamente la distribución Ubuntu 23.10. En la Figura 36 se puede observar información de esta distribución.

```
bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 23.10
Release:        23.10
Codename:       mantic
```

Figura 36. Sistema operativo y distribución del ordenador portátil empleado.

El hardware utilizado para acelerar el procesamiento en tiempo real del modelo DEMUCS es una tarjeta gráfica NVIDIA GeForce GTX 1650 cuyas características técnicas se muestran en la Figura 37. Por su parte, las características de la CPU del sistema se aprecian en la Figura 38.

```
bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$ nvidia-smi
Wed Jul  3 12:10:50 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.171.04             Driver Version: 535.171.04   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce GTX 1650        Off | 00000000:01:00.0  On |                  N/A |
| N/A   46C    P8              5W /  50W |     54MiB /  4096MiB |     22%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      2087      G   /usr/lib/xorg/Xorg                            53MiB |
+---------------------------------------------------------------------------------------+
```

Figura 37. Características técnicas de la GPU del sistema.

```
bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$ lscpu
Arquitectura:                         x86_64
  modo(s) de operación de las CPUs:   32-bit, 64-bit
  Address sizes:                      39 bits physical, 48 bits v
                                      irtual
  Orden de los bytes:                 Little Endian
CPU(s):                               12
  Lista de la(s) CPU(s) en línea:     0-11
ID de fabricante:                     GenuineIntel
  Nombre del modelo:                  Intel(R) Core(TM) i7-10750H
                                       CPU @ 2.60GHz
    Familia de CPU:                   6
    Modelo:                           165
    Hilo(s) de procesamiento por núcleo: 2
    Núcleo(s) por «socket»:           6
    «Socket(s)»                       1
    Revisión:                         2
    CPU(s) scaling MHz:               16%
    CPU MHz máx.:                     5000,0000
    CPU MHz mín.:                     800,0000
```

Figura 38. Características técnicas de la CPU del sistema

6.2.2 Python

Python es el lenguaje de programación utilizado en este TFG. Ha sido elegido debido a varias razones fundamentales. Por un lado, versatilidad y facilidad de uso, ya que es conocido por su sintaxis clara y legible lo que supone una herramienta valiosa para trabajos teóricos experimentales como el presente. Es el lenguaje preferido en la comunidad de la inteligencia artificial y el machine learning debido a que las bibliotecas y

frameworks más populares como TensorFlow y PyTorch están desarrollados en Python. En este TFG, se utiliza de manera integral de principio a fin para la adaptación del modelo de DEMUCS, la captura y reproducción del audio en tiempo real con bibliotecas como 'pyaudio' y 'sounddevice', y el desarrollo de la GUI utilizando la biblioteca de 'Tkinter', la cual permite el desarrollo de una interfaz intuitiva.

Seguidamente se ha procedido con la instalación de Anaconda como sistema de distribución de Python, garantizando la capacidad para gestionar paquetes y entornos virtuales de manera eficiente. Su uso garantiza que las dependencias y configuraciones necesarias para ejecutar los algoritmos de procesamiento de audio estén correctamente gestionadas. A través de Anaconda se ha creado un entorno virtual, el cual es un espacio con su propio intérprete de Python, destinado a gestionar las dependencias de este TFG. Esta estrategia se ha adoptado para evitar que las dependencias del proyecto interfieran con otros posibles proyectos en el mismo equipo de trabajo. De este modo, se asegura que las versiones de los paquetes necesarios para este TFG no afecten a otros proyectos y viceversa, garantizando el correcto funcionamiento y la estabilidad del entorno de desarrollo.

6.2.3 Instalación de Anaconda y configuración del entorno virtual.

Para el desarrollo de este TFG, se ha instalado Anaconda con el objetivo de crear un entorno virtual que incluya todas las dependencias necesarias. A continuación, se describen los pasos realizados para la instalación de Anaconda:

1. Dirigirse al sitio oficial de Anaconda y descargar la última versión disponible. En sistemas Linux, el archivo descargado será un script de Shell con extensión '.sh'.

2. Con el archivo con extensión '.sh' descargado, utilizar el comando en la terminal de linux 'chmod +x' para permitir que el archivo descargado se pueda ejecutar como un programa. Por ejemplo: **"chmod+x Anaconda3-2024.02-1-Linux-x86_64.sh"**

3. En el directorio dónde se encuentra el archivo se ha de abrir una terminal de Linux y ejecutar el siguiente comando para ejecutar el instalador de Anaconda: **"./Anaconda3-2024.02-1-Linux-x86_64.sh"**. Después de realizar la instalación, se recomienda para poder hacer uso de Anaconda en todo el sistema operativo, agregar la ruta de Anaconda al PATH para que los comandos

de Anaconda estén disponibles en la terminal. La ejecución en la terminal de Linux fue: **“echo ‘export PATH=” ~/anaconda3/bin: $PATH” ‘>> ~/. bashrc”.**

4. Seguidamente, se puede inicializar conda y preparar el entorno para el caso de la creación de un entorno virtual utilizando el siguiente comando para inicializar conda: **“conda init”.**

5. Finalmente, para poder asegurarse de que Anaconda se ha instalado correctamente, y que Conda está disponible, se puede verificar la versión a través del siguiente comando: **“conda –version”**. En este caso la versión empleada de Conda es conda 24.1.2.

Al haber realizado la instalación de Anaconda, el siguiente proceso es la creación del entorno virtual para este TFG. Un entorno virtual es una herramienta que se encarga de crear espacios aislados en los que se pueden instalar diferentes versiones de paquetes y dependencias sin que puedan interferir entre sí. En otras palabras, es una “mini-distribución” de Python que contiene su propio intérprete de Python y sus propios paquetes. Esto es especialmente útil en el presente trabajo para gestionar las diferentes dependencias y versiones de los paquetes. En este trabajo, es esencial que el entorno virtual incluya todos los paquetes y dependencias necesarias para el correcto funcionamiento tanto del modelo modificado de DEMUCS como de la aplicación de escritorio.

Se puede crear un entorno virtual de conda de varias maneras diferentes como puede ser: especificando los paquetes y dependencias a través de la terminal de Linux, o utilizando un archivo YAML (YAML Ain’t Markup Language). Un archivo YAML es un formato de serialización de datos diseñado para ser fácil de leer y escribir por las personas. Se utiliza ampliamente en la definición de configuraciones debido a su capacidad para representar datos estructurados de manera intuitiva y comprensible. En este TFG, se siguieron los siguientes pasos detallados para la creación del entorno virtual:

1. Para garantizar el correcto funcionamiento del algoritmo de red neuronal de DEMUCS se han de revisar los requisitos, versiones de paquetes y dependencias con el acceso a la documentación oficial. Es crucial configurar el entorno para utilizar GPU, ya que en este TFG se busca acelerar los procesos de separación de fuentes en el caso de la voz y el fondo en tiempo real. En la documentación proporcionada en DEMUCS, existe la posibilidad de crear un entorno con GPU ‘*environment-cuda.yml*’ y mediante CPU ‘*environment-*

cpu.yml', en este caso se ha optado por utilizar la primera opción debido a los motivos mencionados anteriormente.

2. Dentro del archivo '*environment-cuda.yml*' se indica el nombre del entorno virtual el cual va a ser creado, se especifican los paquetes que se deben instalar en el entorno y las diferentes versiones, las fuentes o repositorios los cuales indican los canales desde donde deben descargarse los paquetes, es decir desde donde conda se encarga de obtener los paquetes. La forma de trabajar con GPU se indica en apartados posteriores para que el modelo se ejecute mediante el cómputo paralelo de CUDA.

```
name: demucs

channels:
  - pytorch
  - conda-forge

dependencies:
  - python>=3.8,<3.10
  - ffmpeg>=4.2
  - pytorch>=1.8.1
  - torchaudio>=0.8
  - cudatoolkit>=10
  - tqdm>=4.36
  - numpy
  - pip
  - pip:
    - diffq>=0.2
    - dora-search
    - einops
    - hydra-colorlog>=1.1
    - hydra-core>=1.1
    - julius>=0.2.3
    - lameenc>=1.2
    - openunmix
    - musdb>=0.4.0
    - museval>=0.4.0
    - soundfile
    - submitit
    - treetable>=0.2.3
    - sounddevice
    - librosa
    - mir_eval
```

3. Al descargar el repositorio de DEMUCS con todos los scripts y archivos necesarios, se debe navegar hasta el directorio donde se encuentre el archivo '*environment-cuda.yml*'. Al estar en el directorio, se ha de abrir una terminal donde aparecerá a la izquierda del nombre del terminal en el entorno base 'bash' pero, este mismo es el que se desea modificar para poder crear el entorno

virtual. Se ha de ejecutar el siguiente comando en la terminal: **“conda env create -f environment-cuda.yml”.**

4. Al haber ejecutado el comando del paso anterior, se empieza a crear el entorno llamado ‘Demucs’ con todos los paquetes y dependencias necesarias para su correcto funcionamiento. Al haber creado el entorno, se podrá visualizar a la izquierda el mismo como se aprecia en la siguiente captura.

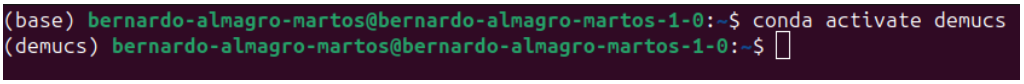


Figura 39. Activación del entorno ‘demucs’.

5. Para asegurarse de que todos los paquetes se han instalado de forma correcta, se pueden listar los paquetes con el siguiente comando para verificarlo.

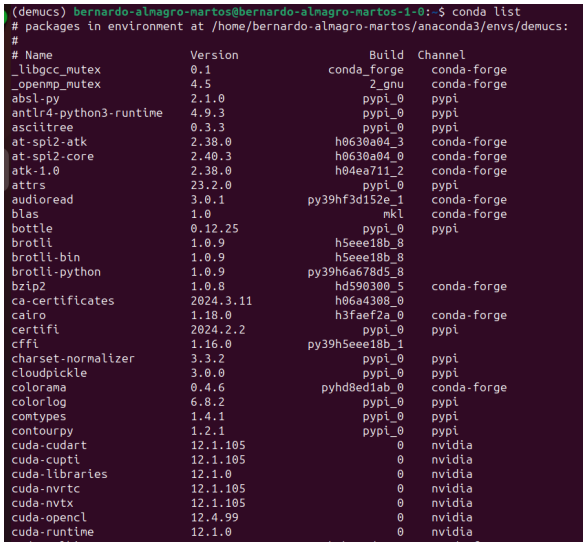


Figura 40. Listado de paquetes instalados en el entorno.

6. Para finalizar este subapartado, se ha de instalar un entorno de desarrollo integrado (IDE) como podría ser PyCharm utilizado en este TFG, ha sido elegido por ser un entorno popular y completo para el desarrollo de Python. PyCharm proporciona un entorno robusto y eficiente para el estudio y desarrollo de trabajos complejos como el de este TFG. La implementación y configuración de PyCharm fueron sencillas y directas, permitiendo un flujo de trabajo ágil y bien organizado.

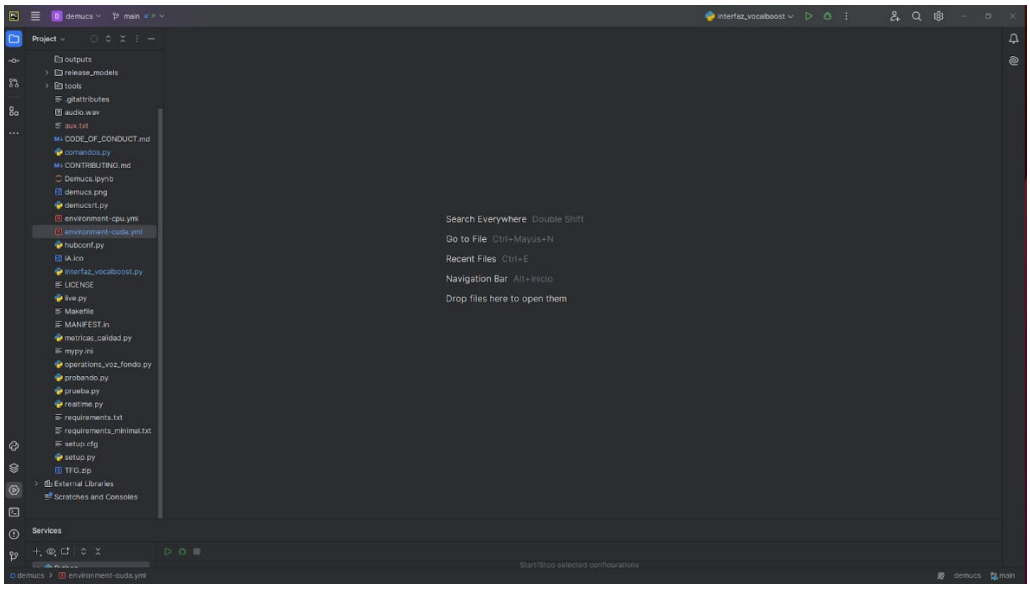


Figura 41. Entorno de Pycharm

Una vez completados estos pasos se habrá configurado el entorno virtual con todos los paquetes necesarios. Esto permitirá trabajar con el algoritmo de DNN 'Demucs' y las distintas modificaciones que se han de realizar para la separación de la voz y el fondo en tiempo real.

6.2.4 DEMUCS: Adaptación para baja latencia

Una de las partes más importantes de este TFG es el uso del algoritmo de DNN 'DEMUCS en su versión híbrida con la realización de ciertas modificaciones para la separación de voz y resto de sonidos en tiempo real con baja latencia. DEMUCS ha sido elegido debido a su alta eficiencia y precisión en separación de fuentes de audio. Utiliza modelos de redes neuronales profundas específicamente diseñados para tareas de separación de fuentes, lo cual permite descomponer una pista de audio en sus componentes individuales o 'stems', como la voz y el resto de sonidos en este caso. Además, es altamente optimizable para poder trabajar con GPUs, lo cual permite aprovechar la potencia de procesamiento de manera paralela para mejorar el rendimiento y la velocidad de las operaciones de este algoritmo, algo esencial para los objetivos de este proyecto. A continuación, se muestran una serie de pasos detallados que se han de seguir para obtener la separación en tiempo real:

1. Dentro del entorno virtual 'Demucs' se ha de descargar el repositorio de la versión 3 híbrida de DEMUCS para poder adaptarlo a su versión en tiempo real, mediante el siguiente comando: **"git clone https://github.com/facebookresearch/demucs.git".** Tras clonar el

repositorio, se habrá creado una carpeta con todos los ficheros necesarios para la ejecución.

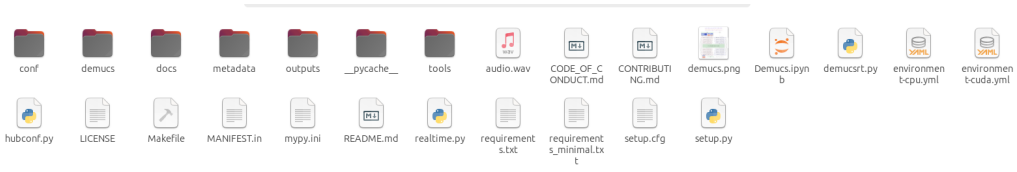


Figura 42. Directorio de trabajo DEMUCS

2. Para adaptar DEMUCS a las necesidades específicas de este TFG se han de realizar varias modificaciones. El script original que proporciona DEMUCS para la separación de los instrumentos musicales, *'separate.py'*, no permitía elegir saltos pequeños. La elección de poder seleccionar saltos pequeños supone tener bajo retardo, lo que permite preservar la sincronización del audio y vídeo, un problema crítico para que el usuario no note cierto retardo al escuchar la voz de los hablantes.

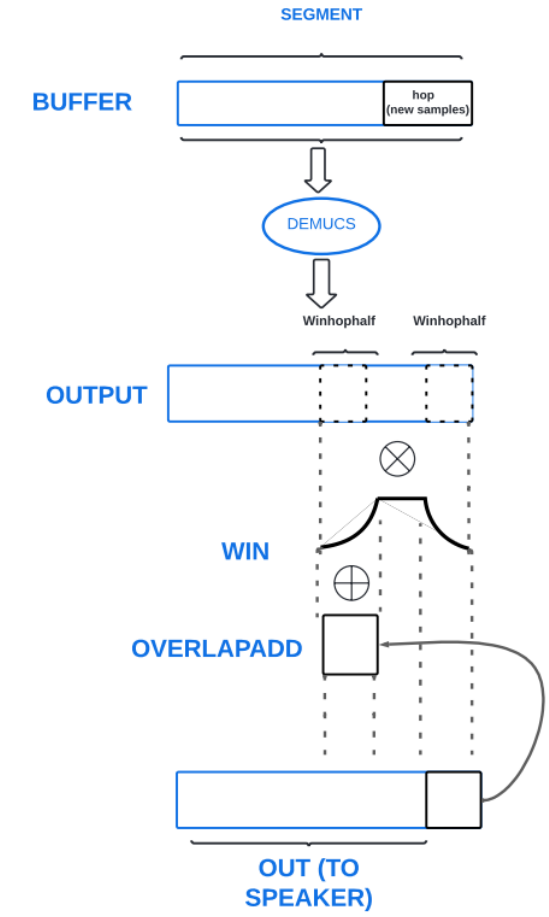


Figura 43. Proceso de adaptación del modelo para baja latencia

Primero de todo, en nuestro fichero *'realtime.py'* se crea una clase llamada RealTimeModel para gestionar la separación de audio en tiempo real. Esta clase ha sido creada para alimentar el modelo con pequeños bloques (o saltos) de muestras de audio y devolver las fuentes separadas para esos bloques. Esta clase permite indicar el tamaño del salto en muestras, que inicialmente se configura en 4410 muestras, es decir, 100 milisegundos (44.1 kHz). También permite indicar el tamaño del segmento de procesado en segundos, que es la cantidad de audio usada por DEMUCS (contexto temporal) para efectuar la separación. Se ha elegido 1 (s) debido a que representa un compromiso equilibrado entre contexto temporal y complejidad computacional. Además, como parámetro clave se puede elegir el dispositivo de procesamiento (GPU o CPU) según las capacidades del equipo del usuario. Al término de lo anteriormente enunciado, se van a explicar una serie de pasos que se han de seguir para la creación de esta clase de procesado en tiempo real:

1. Método del constructor y configuración del modelo: Para manejar y configurar el procesamiento en tiempo real de pequeños segmentos de audio, se ha de inicializar el modelo con los siguientes parámetros:

    - name: Nombre del modelo (archivo de pesos) a utilizar.
    - repo: Carpeta dónde se almacena el archivo de pesos.
    - segment: Longitud del segmento en segundos.
    - hop: Tamaño del salto en muestras.
    - device: Dispositivo a utilizar (por defecto, GPU si está disponible).

    El constructor 'def__init__' se ha de encargar de inicializar el modelo y establecerlo en modo de evaluación y almacena diversas propiedades del modelo como las que se enuncian a continuación:

    - fs: Frecuencia de muestreo del modelo.
    - channels: Número de canales de audio.
    - nsources: Número de fuentes que el modelo puede separar.
    - device: Dispositivo a utilizar.
    - hop: Tamaño del salto en muestras.
    - segmentsamp: Tamaño del segmento en muestras.
    - winhop: Tamaño de la ventana de Hanning.
    - winhophalf: Mitad del tamaño de la ventana de Hanning.

```
class RealTimeModel:
   def __init__(self, name: str,
repo: tp.Optional[Path] = None,
```

```
segment: float = 1,
hop: int = 4410,
device: str = "cuda" if th.cuda.is_available() else "cpu"):
        self.model = get_model(name=name, repo=repo)
        self.model.cpu()
        self.model.eval()

        fs = self.model.samplerate
        channels = self.model.audio_channels
        nsources = len(self.model.sources)

        self.device = device
        self.hop = hop
        segmentsamp = round(fs * segment)
        self.winhop = round(self.model.nfft / 4)
        winhophalf = round(self.winhop / 2)
```

2. Creación de la ventana de Hanning: Se crean ventanas de Hanning utilizando la biblioteca de tensores 'torch' para suavizar los bordes de los segmentos de audio. La ventana, se modifica añadiendo una sección plana en el medio con 'torch.cat', quedando tal y como se muestra en la Figura 43 (con longitud hop + *winhophalf*). Finalmente, se ajustan sus dimensiones con las funciones *'unsqueeze'* y *'expand'* para poder aplicarla a la vez a todas las fuentes y canales de salida.

```
# Hanning window
hann = th.hann_window(self.winhop, periodic=True, device=self.device)
hann = th.cat([hann[:winhophalf], th.ones(hop-winhophalf,
device=self.device), hann[-winhophalf:]])
hann = hann.unsqueeze(0).unsqueeze(1).unsqueeze(2)
self.win = hann.expand(1, nsources, channels, -1)
```

3. Inicialización de buffers de entrada:

   - 'self.overlapadd': Buffer para almacenar la parte solapada de los segmentos procesados.
   - 'self.buffer': Buffer para almacenar el segmento de audio actual, de tamaño igual a *segment* (1 segundo por defecto)

```
# Input buffers
self.overlapadd = th.zeros(1, channels, winhophalf, device=self.device)
self.buffer = th.zeros(1, channels, segmentsamp, device=self.device)
```

4. La principal funcionalidad de la clase *RealTimeModel* es el método *separate*, Este método se encarga de procesar cada segmento de audio y aplicar el

modelo de separación. En los siguientes puntos se muestra como se realiza el proceso de separación:

- Verificación del tamaño del bloque de entrada, es decir, se comprueba que el tamaño del nuevo fragmento de audio tenga el número de muestras y canales correcto.

```
def separate(self, frame: th.Tensor):
        if frame.shape != (1, self.model.audio_channels, self.hop):
raise ValueError(f"Frame must have size
1x{self.model.audio_channels}x{self.hop}")
```

- Actualización del buffer: El buffer anteriormente mencionado, que almacena todo el segmento de audio que entra en cada llamada al separador, se actualiza con las nuevas muestras, es decir, se desplazan las muestras hacia la izquierda dejando espacio para el nuevo bloque 'frame' al final del buffer.

```
# Desplaza el buffer y añade el nuevo frame
self.buffer[:, :, :-self.hop] = self.buffer[:, :, self.hop:]
self.buffer[:, :, -self.hop:] = frame
```

- Aplicación del modelo: Se aplica el modelo de separación propio de DEMUCS al contenido completo del buffer, dónde el resultado de salida 'output' es un tensor que contiene las fuentes separadas para el segmento de audio actual.

```
# Separate segment
output = apply_model(self.model, self.buffer, shifts=0, split=False,
device=self.device)
```

- Overlapp-Add: Se aplica la ventana de Hanning creada anteriormente a la parte final del segmento procesado para suavizar las transiciones entre segmentos. La parte solapada del segmento anterior, almacenada en 'self.overlapadd', se añade a la parte correspondiente del segmento actual (ver Figura 43) y se actualiza 'self.overlapadd' con la nueva parte final del segmento actual para su uso en la siguiente iteración. Finalmente, se entregan como salida las *hop* muestras situadas justo antes de las últimas *winhophalf* muestras.

```
# Overlap-add
winhophalf = round(self.winhop / 2)
output[:, :, :, -self.hop-winhophalf:] *= self.win
output[:, :, :, -self.hop-winhophalf:-self.hop] += self.overlapadd
self.overlapadd = output[:, :, :, -winhophalf:]
return output[:, :, :, -self.hop-winhophalf:-winhophalf]
```

A continuación, como ejemplo de uso, se explica de manera detallada cómo utilizar la clase *RealTimeModel* para separar audio en tiempo real:

1. Configuración del modelo en tiempo real. Se llama al constructor de la clase *RealTimeModel* indicando el valor de salto deseado, así como la ubicación del fichero de pesos pre-entrenados que se quiere usar . Desde fuera de la clase, se puede acceder al número de fuentes que el modelo puede separar, el número de canales de audio y la frecuencia de muestreo del modelo, donde en este caso se trabaja con audio estéreo a 44.100 Hz (estas propiedades dependen del fichero de pesos empleado). Se puede observar el proceso descrito anteriormente en el siguiente código:

```
# Overlap-add
winhophalf = round(self.winhop / 2)
output[:, :, :, -self.hop-winhophalf:] *= self.win
output[:, :, :, -self.hop-winhophalf:-self.hop] += self.overlapadd
self.overlapadd = output[:, :, :, -winhophalf:]
return output[:, :, :, -self.hop-winhophalf:-winhophalf]
```

2. Lectura del archivo de audio y preparación del buffer de salida: Se carga el archivo de audio de ejemplo, con un número de canales y frecuencia de muestreo que deben coincidir con las soportadas por el modelo, o y se ajusta la forma del tensor para que la primera dimensión sea de tamaño 1 (1 batch). Se prepara el buffer de salida con un tensor de ceros (‘waveout’), para poder almacenar los resultados de la separación de pistas.

```
# Audio de entrada y buffer de salida
wav = load_track(track, model.model.audio_channels,
model.model.samplerate)[None]
nsamples = wav.shape[2]
wavout = th.zeros(1, nsources, nchannels, nsamples)

# Tiempo de inicio
start_time = time.time()
```

3. Procesado de audio bloque a bloque: Se inicializa una variable ‘index’ para determinar la posición actual de las muestras de audio. El bucle while se encarga de procesar el audio en bloques hasta que se hayan procesado todas

las muestras del audio. En cada iteración del mismo se procesa un bloque de audio utilizando el método 'separate' del modelo y se actualiza el índice 'index' para avanzar al siguiente bloque, almacenando el resultado de la separación de fuentes en el tensor de salida 'waveout' en la posición correspondiente. Después de procesar el bloque actual, se incrementa 'index' en el tamaño del salto para avanzar al siguiente bloque de audio. Tal y como se muestra en el ejemplo siguiente, finalmente se extrae el índice asociado a la fuente "vocals" en el modelo, buscando en la lista de las fuentes que el modelo puede separar ('model.model.sources'), En el ejemplo, la voz separada se acaba reproduciendo en los altavoces.

```
# --PROCESADO DE AUDIO BLOQUE A BLOQUE--
hop = model.hop
index = 0

while nsamples > index + hop:
    wavout[:, :, :, index:index + hop] = model.separate(wav[:, :,
index:index + hop])
    index += hop
    print(f"Samples processed: {index} ({index / samplerate} seg)")

end_time = time.time()
elapsed_time = end_time - start_time

print("Elapsed time: ", elapsed_time)
print("Done, playing vocals ....")

idx_vocals = model.model.sources.index('vocals')
play_audio(wavout[0, idx_vocals, :, :][None], samplerate)
```

Para comprobar que la ejecución se produce correctamente en tiempo real, el tiempo de ejecución del programa ha de ser menor o igual que la duración del archivo de audio. Se han realizado diversas pruebas para obtener el retardo a partir del cual se produciría la ejecución en tiempo real en nuestro equipo de pruebas (ver la sección de *Resultados y Discusión*). Aumentar y reducir el retardo 'delay' supone dos inconvenientes:

1. Impacto del aumento del delay. Aumentar el retardo mejora la calidad de la separación y reduce la complejidad computacional. Sin embargo, esto puede causar una desincronización significativa entre el audio y el video. Este desajuste temporal resulta en una experiencia de usuario insatisfactoria, especialmente en aplicaciones como películas o programas de televisión, donde la sincronización entre audio y video es crucial.

2. Impacto de la reducción del delay. Reducir el retardo o tamaño del salto ('hop') disminuye la latencia, manteniendo una sincronización más estrecha entre el audio y el video. No obstante, este enfoque aumenta significativamente la carga computacional, ya que se realizan más llamadas al sistema de separación. Además, este incremento de la demanda de recursos puede introducir artefactos de separación por cortes en la señal de salida, afectado a la calidad del audio que se procesa.

Lo mencionado anteriormente dependerá de la capacidad del sistema del usuario y su tarjeta gráfica, que es la herramienta principal para poder realizar la separación en tiempo real. Aunque esta diferencia se ha ajustado en la aplicación de escritorio, que posteriormente será explicada, para que el usuario en función de sus capacidades pueda seleccionar el delay óptimo.

6.2.5 Captura y Reproducción de audio en tiempo real con Sounddevice

En este TFG, uno de los objetivos es poder capturar y reproducir la separación del diálogo en tiempo real mediante la captura del audio procedente de una aplicación con la finalidad de reproducir el resultado de separación a través de los altavoces del sistema. La biblioteca utilizada en este trabajo en relación con la reproducción y grabación de audio es Sounddevice. Es una biblioteca de Python que ofrece una interfaz intuitiva y eficaz para la reproducción y grabación de audio, utilizando la tecnología de PortAudio. Fue seleccionada debido a su uso sencillo, su compatibilidad con múltiples plataformas y su capacidad para integrarse con otras bibliotecas de procesamiento de datos en Python, como NumPy y PyTorch. Sounddevice permite gestionar audio en tiempo real, lo cual es crucial para aplicaciones que necesitan retroalimentación inmediata, como la separación de fuentes de audio. En este TFG, Sounddevice se utiliza para reproducir los resultados de la separación de audio en tiempo real.

```
# Audio estéreo
import sounddevice as sd
sd.play(muestras_audio, fs)
```

Se ha creado un fichero llamado *live.py* para implementar la separación en tiempo real con audio leído/reproducido con *sounddevice*. Uno de los aspectos más esenciales de este nuevo script es el método de captura de audio en el cual el programa, al inspeccionar el flujo de audio a separar, realiza todo el procesado de separación de las fuentes. Además, los resultados obtenidos de la separación se van reproduciendo a través de los altavoces:

1. Configuración de audio de entrada. El dispositivo de entrada 'device_in' se especifica a través de la línea de comandos. Seguidamente, se asegura que el dispositivo de entrada sea válido y esté disponible mediante la función 'query_devices'. Se determina el número de canales de entrada asegurando que 'channels_in' sea 1 (mono) o 2 (estéreo). Para la configuración del flujo o stream de entrada, se crea un objeto de tipo 'InputStream' de la biblioteca anteriormente mencionada *Sounddevice*, configurado con el dispositivo de entrada, la tasa de muestreo y el número de canales.

```
# CONFIGURACIÓN DE AUDIO DE ENTRADA
device_in = parse_audio_device(args.in_)
caps = query_devices(device_in, "input")
channels_in = min(caps['max_input_channels'], 2)

stream_in = sd.InputStream(
    device=device_in,
    samplerate=samplerate,
    channels=channels_in
)
```

2. Configuración de audio de salida. Similar a la configuración del audio de salida, se especifica el dispositivo de salida a través de la línea de comandos. La función 'query_devices' de nuevo sirve para asegurar la validez y disponibilidad del dispositivo de salida. Se determina de nuevo el número de canales de salida y asegurando de nuevo que 'channels_out' sea 1 (mono) o 2 (estéreo). Por último, se crea un objeto de tipo 'OutputStream', configurado con el dispositivo de salida seleccionado, la tasa de muestreo del modelo y el número de canales. El tamaño del bloque 'blocksize', es decir, el buffer de audio de salida interno de la librería *sounddevice*, se hace algo más grande que el tamaño del salto *hop* para asegurar que no haya pérdida de datos. En la siguiente captura se muestra la configuración realizada.

```
# CONFIGURACIÓN DE AUDIO DE SALIDA
device_out = parse_audio_device(args.out)
caps = query_devices(device_out, "output")
channels_out = min(caps['max_output_channels'], 2)

stream_out = sd.OutputStream(
    device=device_out,
    samplerate=samplerate,
    channels=channels_out,
    blocksize=model.hop + int(model.hop / 2)
)
```

6.2.5.1 Procesamiento de audio en tiempo real bloque a bloque

Una parte fundamental del método 'main' en el script *live.py* es el procesamiento del audio en tiempo real, es decir, cómo el script captura bloques de audio, los separa utilizando la clase *RealTimeModel* del script *realtime.py* (en concreto, la función de separación de segmentos de audio 'separate'), y cómo se realiza la mezcla y se envía el resultado a la salida. Antes de entrar en el bucle principal, se han de inicializar los stream de entrada y de salida para capturar el audio de la siguiente manera: **stream_in.start()** y **stream_out.start()**.

En el bucle principal de procesamiento 'while working', el script captura bloques de audio del dispositivo de entrada mediante la función 'read', indicando el número de muestras que se desean leer, en este caso el tamaño del salto. Al leer el flujo de audio de entrada, las muestras se encuentran en un array de numpy por lo que, al estar trabajando DEMUCS con tensores de torch, hay que convertirlo a un tensor.

```
while working:
    try:
        print("Leyendo audio frame...")
        frameorig, overflow = stream_in.read(model.hop)

        if overflow:
            print("overflow")

        frame = torch.from_numpy(frameorig).to(args.device)
```

A partir de aquí, el stream de audio de entrada se separa con el objetivo de conseguir la voz y resto de sonidos de forma separada. Además, se realiza una permutación debido a que el bloque de audio leído tiene dimensiones *nº muestras x nº canales*, mientras que el método 'separate' de *realtime.py* requiere un tensor de dimensiones *1 x nº canales x nº muestras*. El propósito de la línea de código *if not out.numel()* es evitar procesar y escribir datos de salida si no se ha producido ningún separación de fuente, por lo que si la salida 'out' está vacía, el bucle continúa con la siguiente interacción sin procesar el fragmento de audio actual. A continuación, se muestran las modificaciones realizadas.

```
print("Separando audio...")
with torch.no_grad():
    frame = th.permute(frame[None, :, :], (0, 2, 1))
    out = model.separate(frame)
    out = out[0, :, :, :]

if not out.numel():
    print("No se ha podido separar ningún audio.")
    continue
```

6.2.5.2 *Extracción de índices y aplicación de coeficientes de mezcla*

En este punto, tras obtener las fuentes separadas, hay que buscar en el modelo el listado de fuentes, en el cual cada fuente tiene asociado un índice. En este caso, se extraen los índices de las voces 'vocals' y el residuo 'residue' de la siguiente manera:

```
out_vocals = out[model.model.sources.index('vocals'), :, :]
out_residue = out[model.model.sources.index('residue'), :, :]
```

Una vez que el audio ha sido separado y se han determinado los índices de voz y residuo, se aplican los coeficientes de mezcla para ajustar los niveles de volumen de las diferentes fuentes obteniendo la suma de ambas fuentes ponderadas, 'mixed_signal'. En la aplicación de escritorio, estos dos parámetros han sido diseñados para que el usuario pueda elegir el volumen de voz y fondo según sus necesidades antes y durante el procesamiento del programa.

```
mix_vocals = volumen_voz * out_vocals
mix_residue = volumen_fondo * out_residue

mixed_signal = mix_vocals + mix_residue
```

6.2.5.3 *Conversión y envío a la salida*

Para que los altavoces o salida del sistema puedan reproducir el resultado final de la separación, hay que convertir de nuevo el tensor a un array de numpy, como se tenía inicialmente en el stream de entrada. Esto se debe a que la librería *sounddevice* sólo admite este último formato. Por lo tanto, se transpone a las dimensiones requeridas, se manda a la CPU y se convierte en un array de Numpy. Finalmente, se establece un indicador de 'underflow' que avisa cuando el buffer se queda sin muestras que reproducir, cosa que ocurre si el sistema es incapaz de suministrar muestras separadas en tiempo real, generando así huecos en el audio reproducido.

```
print("Escribiendo audio a la salida...")
underflow = stream_out.write(mixed_signal)

sample_count += model.hop
duration = sample_count / samplerate
print(f"Samples processed: {sample_count} ({duration} seg)")

if underflow:
    print("underflow")
```

6.2.5.4 *Configuración de dispositivos de audio virtual de forma manual*

Además de las modificaciones ilustradas anteriormente, se han de crear dispositivos de audio virtual que permitan el enrutamiento del audio a través de las

interfaces de entrada y salida del programa. Estos dispositivos de audio virtual son herramientas esenciales en el procesamiento de audio digital, especialmente cuando se necesita capturar, progresar y redirigir el audio de una aplicación a otra dentro de un sistema operativo. Este concepto, ha sido aplicado al presente TFG con una serie detallada de pasos. En este subapartado, se explica cómo realizar el enrutamiento de audio manualmente con las herramientas de PulseAudio, mientras en secciones posteriores se explica cómo se ha implementado de forma automática desde la aplicación.

1. Instalar el servidor de sonido **PulseAudio**, cuya función es proporcionar una interfaz común para manejar el audio en el sistema, es decir, es capaz de realizar tareas como mezclar múltiples flujos de audio, redirigir audio entre diferentes dispositivos y ajustar el volumen de forma independiente por aplicación. Esta herramienta permite enrutar las diferentes interfaces de audio virtual explicadas en los siguientes pasos. Para que este servidor de sonido funcione correctamente, se ha de instalar si no se encuentra en el sistema la herramienta de control de volumen **pavucontrol** que es la que permite interactuar con **Pulseaudio**.

```
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$ sudo apt install pulseaudio-utils
```

Figura 44. Instalación del servidor de sonido PulseAudio

```
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$ sudo apt -y install pavucontrol
```

Figura 45. Instalación del control de volumen Pavucontrol

2. Para crear las interfaces de audio virtual se ha de ejecutar en la terminal de Linux el siguiente comando: **sudo modprobe snd-aloop**. Este comando se utiliza para crear dispositivos de audio virtual. Es decir, el módulo snd-aloop se queda cargado en el kernel y se crea un dispositivo de audio virtual para permitir el enrutamiento de audio. En este caso, al ejecutarlo, se crean dos interfaces de audio virtual: Interfaz de salida "Loopback Analog Stereo" y interfaz de entrada “Monitor of loopback Analog Stereo”. La aplicación que genera audio, como por ejemplo un video de Youtube, envía el flujo de audio a la interfaz de salida "Loopback Analog Stereo", mientras que la interfaz de entrada “Monitor of loopback Analog Stereo” que es una interfaz de tipo “monitor” permite inspeccionar el flujo o audio que se envía a la interfaz de salida. Es decir, la aplicación como fuente de audio envía todo su audio a la interfaz de salida virtual, mientras que el programa desarrollado en este TFG, con las anteriores configuraciones realizadas de captura y reproducción de audio, captura el stream de audio escuchando la interfaz de entrada “Monitor of loopback Analog

Stereo" para realizar la separación de la voz y el fondo. Posteriormente, hay que dirigirse a la herramienta de **pavucontrol** para realizar de forma manual la configuración y enrutamiento de los dispositivos de audio virtual y ejecutar en orden los siguientes comandos:

pavucontrol	Arrancar el control de volumen
pulseaudio –start	Activar el servidor de sonido
sudo modprobe snd-aloop	Activar las interfaces de audio virtual

Con los dispositivos de audio virtual creados, el servidor **PulseAudio** en funcionamiento y nuestro programa ejecutándose, el primer paso consiste en configurar adecuadamente el enrutamiento de la captura y reproducción de audio. Al abrir el control de volumen (**pavucontrol**), en la pestaña "Grabación", se puede verificar que el programa **ALSA plug-in [python3.7]** en ejecución está preparado para capturar el flujo de audio, realizar el procesamiento y separación de las fuentes correspondientes desde el script *live.py*, lo que significa que se ha configurado correctamente. En esta configuración, es necesario seleccionar la interfaz de entrada "Monitor of Loopback Analog Stereo" en el desplegable de la derecha de la Figura 46. Esta interfaz se encargará de inspeccionar y leer el flujo de audio procedente de la interfaz de salida "Loopback Analog Stereo" para realizar la separación de fuentes. A continuación, se muestra la selección de la interfaz adecuada:

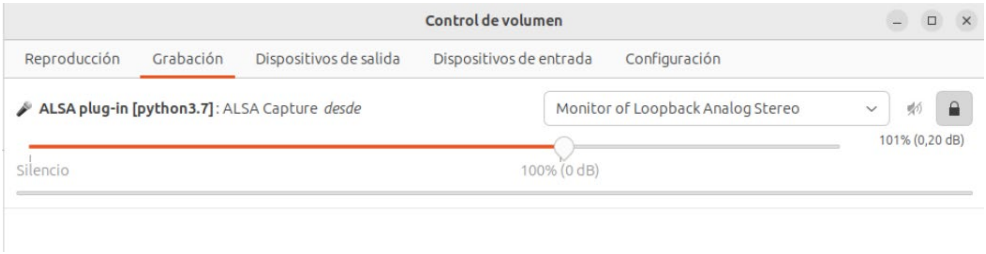


Figura 46. Asignación del dispositivo de grabación en el control de volumen.

En la pestaña de "Reproducción" de Pavucontrol, es necesario indicar a la aplicación que genera audio, en este caso YouTube, que envíe todo el flujo de audio a través de la interfaz de salida "Loopback Analog Stereo". Como se menciona en el paso anterior, la interfaz de tipo "Monitor" inspeccionará este flujo de audio. Finalmente, con la configuración del audio de salida realizada en el programa *live.py*, se puede observar nuevamente el programa **ALSA plug-in [python3.7]** en el que es necesario seleccionar en el display que el resultado de la separación de la voz y el resto de acompañamiento se reproduzca por los altavoces del sistema utilizado. De esta manera se completa el enrutamiento de audio para la separación de la voz y el fondo.

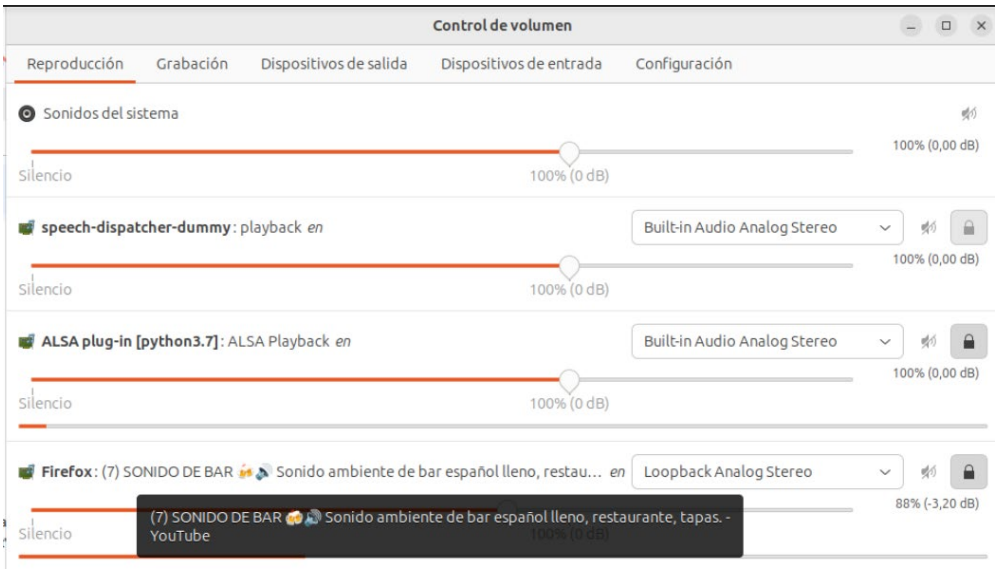


Figura 47. Asignación del dispositivo de reproducción en el control de volumen

6.2.6 Aplicación de Escritorio: Vocal Boost

Hasta ahora, se ha conseguido implementar el concepto de separación de fuentes (voz y acompañamiento) en tiempo real de forma manual, realizando las configuraciones necesarias mediante el control de volumen del sistema operativo Linux, tal y como se detalló en los pasos anteriores. A partir de aquí se presenta la implementación de esta funcionalidad en una aplicación de escritorio, diseñada para que el usuario pueda interactuar de manera intuitiva con diversos parámetros, como el volumen de voz, el volumen de fondo y el retardo en tiempo real. Esta sección describe cómo se ha desarrollado la interfaz gráfica para facilitar la utilización del programa, proporcionando una experiencia de usuario eficiente y accesible. La aplicación desarrollada en este TFG ha sido realizada en Python de manera que la parte visual de la interfaz gráfica, es decir, el Front-End sea amigable e intuitiva para el usuario sin que tenga que aplicar ningún tipo de conocimiento técnico previo. La misma ha sido diseñada con el objetivo de mejorar la experiencia de personas con problemas de audición y de usuarios que se encuentran en entornos ruidosos. Esta aplicación permite al usuario mediante un display, seleccionar los playbacks o streams de audio que desea procesar, ya sea un video con el propio reproductor del sistema operativo o Youtube por ejemplo. Al haber seleccionado el playback de audio activo, se ha de elegir el delay óptimo que permite mejorar el diálogo este parámetro solamente se podrá seleccionar antes de activar el programa debido a que cada vez que se seleccione se debería cargar el modelo de nuevo. El retardo (delay) dependerá de las capacidades computacionales del equipo de procesamiento utilizado, tal como se explicó anteriormente. Al seleccionar el retardo, el usuario puede ajustar la cantidad de volumen de voz y volumen de fondo según sus necesidades, y ambos parámetros pueden modificarse durante la separación de fuentes al activar el programa.

Con todas las modificaciones realizadas, el usuario podrá activar o desactivar el programa mediante un botón de 'Activar/Desactivar' según su conveniencia para ajustar el volumen de los diálogos en tiempo real.

En resumen, la aplicación permite capturar audio procedente de una fuente de audio mediante la selección del playback y la configuración de los parámetros mencionados anteriormente. Al activar el programa, se realiza la separación de fuentes y el resultado se reproduce por los altavoces. Todo este enrutamiento de audio se realiza de manera interna y automática. No obstante, todas estas configuraciones realizadas se explican de aquí en adelante para comprender el desarrollo de la aplicación. En el apartado *Evaluación objetiva de los modelos* dentro del capítulo *Resultados y Discusión*, se ha implementado un proceso de evaluación del programa desarrollado para la separación de la voz y el fondo en tiempo real y el propio modelo de DEMUCS para obtener los rendimientos en cuanto a métricas de calidad objetivas. En la Figura 48, se muestra la interfaz principal que permite al usuario interactuar.

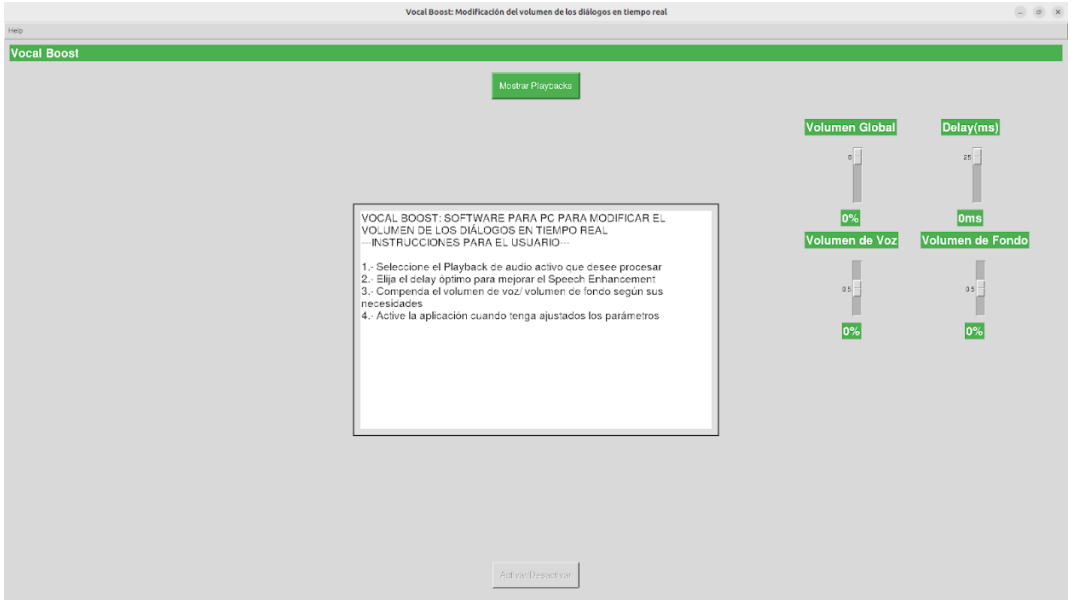


Figura 48. Interfaz principal de la aplicación de escritorio 'Vocal Boost'.

En la Figura 49, se muestra al lector la arquitectura de la aplicación 'Vocal Boost' de la forma en la que las bibliotecas o librerías interactúan con la propia interfaz gráfica, la aplicación la cual se desea capturar el flujo de audio para la separación de los diálogos, el subproceso de separación y los dispositivos de audio virtual.

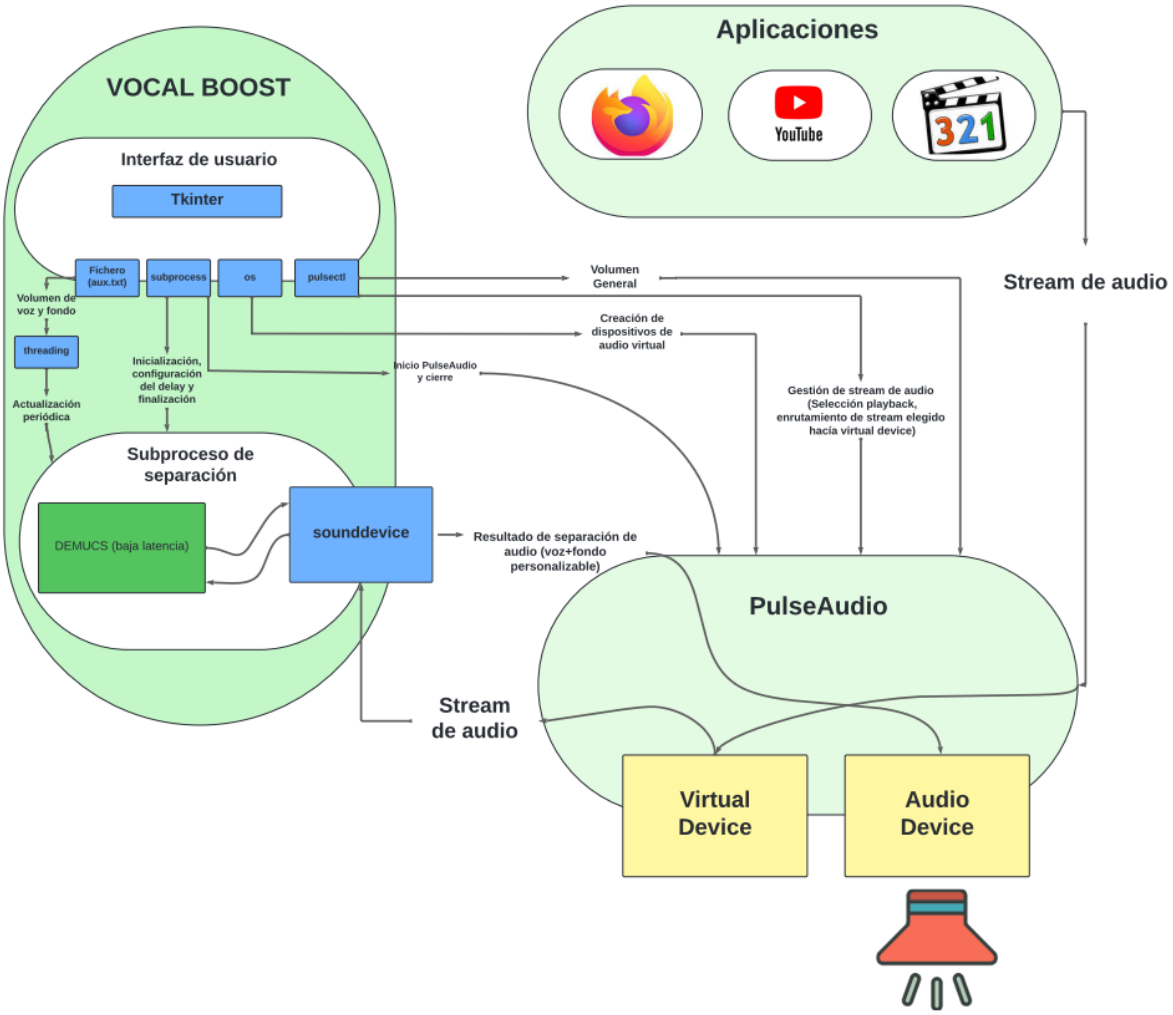


Figura 49. Diagrama de bloques de la arquitectura 'Vocal Boost'.

6.2.6.1 Funcionalidades de los scripts principales en Vocal Boost

La aplicación de escritorio 'Vocal Boost' se compone de tres scripts principales que trabajan en conjunto para ofrecer una experiencia de usuario eficiente y fluida en el procesamiento de audio en tiempo real como aspecto crucial en este TFG. Cada uno de los scripts tiene una función específica y desempeña un papel crucial en el correcto funcionamiento de la aplicación. A continuación, se describen las funcionalidades principales de los tres scripts que han sido clave.

comandos.py

El script comandos.py es el responsable del control y la gestión de procesos de audio en la aplicación. Emplea la biblioteca 'subprocess' para iniciar y detener el procesamiento de audio ejecutando el script *live.py* en un proceso separado, asegurando así que la interfaz gráfica no se bloquee y los parámetros de volumen y retardo se pasen correctamente. Además, incluye funciones para iniciar y detener el servidor de audio '**PulseAudio**', y para guardar y restaurar la configuración inicial de audio. Finalmente, se

facilita la búsqueda de las interfaces de audio disponibles y el enrutamiento del audio a través de interfaces de audio virtual para el monitoreo y para redireccionar el audio entre las mismas.

live.py

Este script se centra principalmente en el procesamiento de audio en tiempo real, utilizando, como se describió en apartados anteriores, el modelo de separación de fuentes DEMUCS con las modificaciones realizadas para su adaptación al procesamiento de audio en tiempo real, específicamente la clase *RealTimeModel* de *realtime.py*. Configura las interfaces de audio de entrada y salida, asegurando que los dispositivos correctos se utilicen para el procesamiento. A través de un bucle continuo 'while', carga y reproduce las pistas de audio, separando en tiempo real las fuentes de voz y fondo. Además, se incluyen funciones para leer y actualizar periódicamente los volúmenes de voz y fondo desde un archivo externo.

interfaz_vocalboost.py

El script *interfaz_vocalboost.py* define la GUI para esta aplicación de escritorio. Permite crear la ventana principal de la aplicación, incluye un menú de ayuda para proporcionar información sobre la aplicación. Permite al usuario seleccionar y mostrar los playbacks de audio activos a través de un desplegable y ajustar los valores de volumen tanto de voz como de fondo y retardo mediante barras deslizantes. Además, se proporcionan botones para activar y desactivar el procesamiento de audio, lanzar y detener el procesamiento de audio en tiempo real, y actualizar los valores de volumen de voz y fondo en un archivo de configuración externa para que el script live.py pueda leer los valores periódicamente.

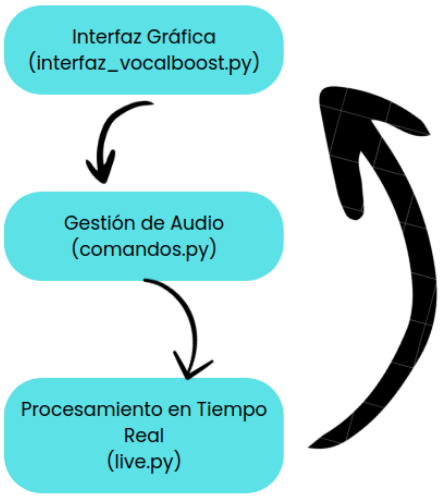


Figura 50. Funcionamiento básico de los scripts principales.

6.2.6.2 Diagrama de flujo de la aplicación

En aplicaciones como la desarrollada en este TFG, que requieren procesamiento intensivo de audio en tiempo real, la ausencia de hilos puede llevar a problemas significativos. Sin hilos, todas las operaciones se ejecutarían en el hilo principal del programa, lo que conlleva a un bloqueo de la interfaz de usuario, haciendo que la aplicación sea lenta y no responsiva durante tareas como la captura y el procesamiento de audio. Para evitar estos problemas y mejorar la eficiencia, se han utilizado hilos y subprocesos para manejar diferentes tareas de manera concurrente.

La interfaz de usuario **'interfaz_vocalboost.py'** que permite al usuario seleccionar los playbacks, ajustar volúmenes y demás funcionalidades explicadas en el apartado anterior. Cuando el usuario activa el procesamiento de audio, se lanza un subproceso que ejecuta al script **'live.py'**, permitiendo que el procesamiento de audio se realice sin bloquear la interfaz de usuario.

El script que permite realizar la captura, separación y reproducción de las fuentes de audio en tiempo real, **'live.py'**, tiene integrado un hilo que se ejecuta en segundo plano para actualizar los volúmenes de voz y fondo cada segundo, permitiendo modificar los valores de volumen de voz y fondo sin interrumpir el proceso de separación de fuentes.

Por otro lado, **'comandos.py'** que maneja el inicio y detención del servidor de PulseAudio, enrutamiento de audio y control del subproceso **'live.py'**. El lanzamiento del subproceso y detención del subproceso mencionado anteriormente se realiza desde la interfaz gráfica hasta **'comandos.py'**, en este se lanza el subproceso enviando los valores de volumen de voz, fondo y retardo por el usuario mediante argumentos de la línea de comandos. A continuación, se muestra el diagrama de flujo de la aplicación desarrollada.

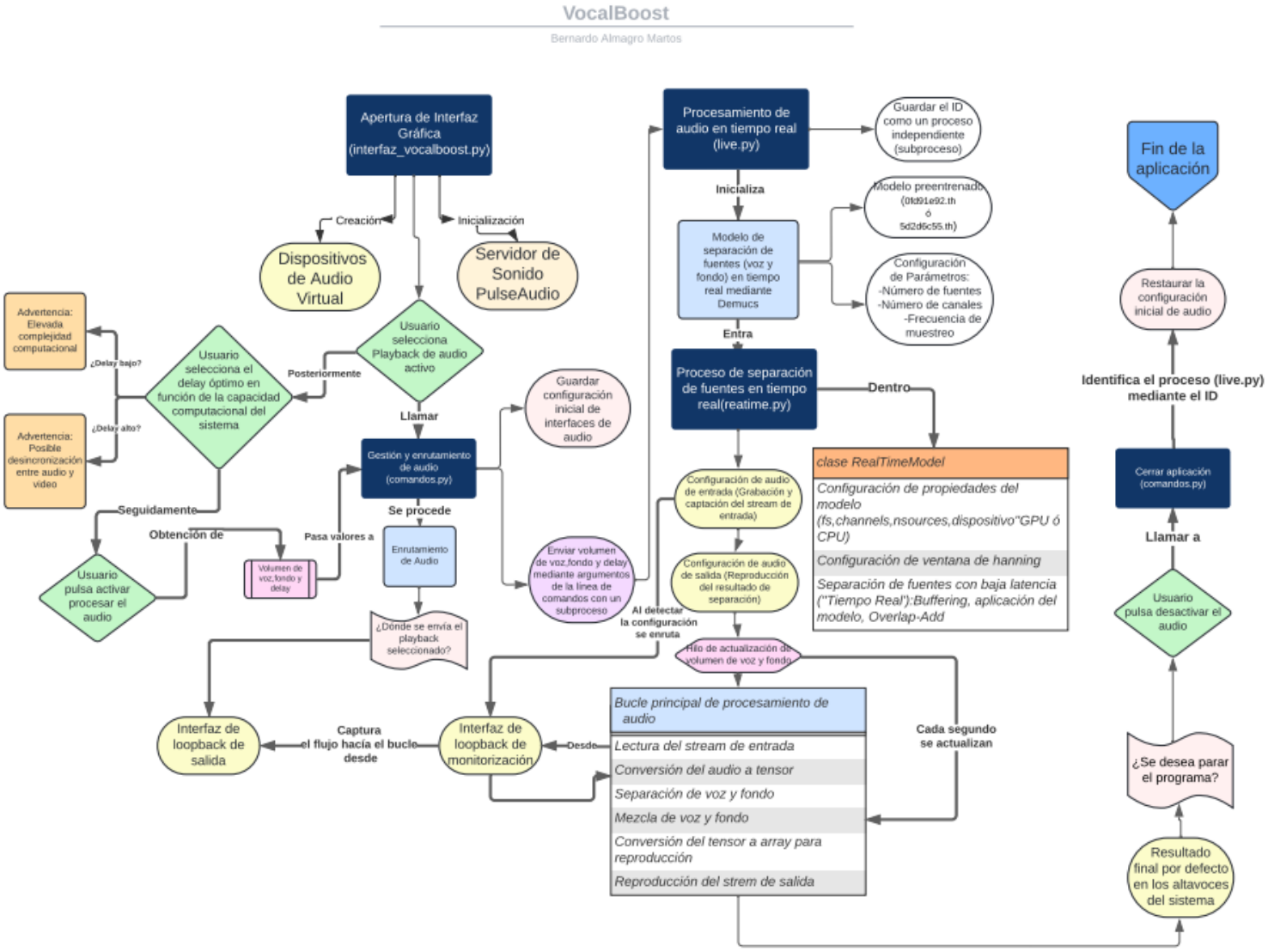


Figura 51. Diagrama de flujo 'Vocal Boost'.

Librería para el desarrollo de la interfaz gráfica

En el presente TFG se ha elegido la librería de Tkinter para el desarrollo de la GUI de 'Vocal Boost' debido a su simplicidad y eficacia. Tkinter es una biblioteca estándar de Python para la creación de interfaces gráficas que permite construir aplicaciones con ventanas, botones, deslizadores y otros componentes interactivos de manera rápida y sencilla. Además, es ampliamente compatible con diversas plataformas, lo que garantiza que la aplicación pueda ejecutarse en diferentes sistemas operativos sin necesidad de cambios significativos en el código.

En el trabajo realizado, Tkinter ha sido utilizado para crear una ventana principal que contiene los elementos necesarios para la configuración y control de la aplicación VocalBoost. Se han implementado botones personalizados para iniciar y detener el procesamiento de audio, deslizadores para ajustar el volumen de voz, volumen global, volumen de fondo y delay, además, se dispone de un display para seleccionar los playbacks de audio activos. En definitiva, la biblioteca de Tkinter ha facilitado la gestión de eventos y la actualización de la interfaz gráfica en tiempo real con la capacidad de reflejar de inmediato los cambios realizados por el usuario, mejorando la experiencia y la funcionalidad de la aplicación.

Figura 52. Librería utilizada para la realización de la GUI (Shekhar, 2023).

Librería para la gestión de audio

Se ha seleccionado la biblioteca **pulsectl** para el desarrollo y manejo del audio en la aplicación. **Pulsectl** es una biblioteca de Python que proporciona una interfaz para interactuar con **PulseAudio**, que como se explicó en apartados anteriores , es un servidor de sonido multiplataforma. La elección de **pulsectl** se debe a su capacidad para ofrecer un control detallado sobre los dispositivos de audio y flujos de sonido, lo cual es esencial para una aplicación como la presente que requiere la manipulación precisa del audio en tiempo real.

En este TFG, se han implementado funciones para iniciar y detener el servicio de PulseAudio. Asegurando que el servidor de sonido esté siempre activo cuando la aplicación 'Vocal Boost' esté en funcionamiento y viceversa. Además, se emplea en funciones como guardar y restaurar la configuración de audio inicial y el enrutamiento del audio.

```
from pulsectl import Pulse
def pulseaudio_start():
    # Inicia PulseAudio
    return subprocess.run(["pulseaudio", "--start"])
def pulseaudio_stop():
    # Detiene PulseAudio
    return subprocess.run(["pulseaudio", "--kill"])
```

Librería para el manejo de hilos

En la aplicación de escritorio desarrollada, se ha utilizado la biblioteca **threading** para gestionar tareas concurrentes. Threading es una técnica de programación que permite la ejecución simultánea de múltiples hilos (threads) dentro de un sólo proceso. Las ventajas que presenta haber utilizado esta biblioteca son: permitir que la aplicación realice varias tareas de manera más eficiente y mejorar la capacidad de respuesta de la interfaz gráfica al ejecutar ciertas tareas en segundo plano.

En el trabajo realizado, la librería **threading** se ha utilizado principalmente para la actualización continua de los volúmenes de la voz y el fondo de manera periódica. Esta implementación es crucial para mantener los niveles de volumen actualizados sin interferir con la experiencia del usuario.

```
import threading
import time
import operations_voz_fondo

# Declaramos como variable global working que servirá en el bucle de
procesamiento de audio
working = True
update_event = threading.Event()

# Función para actualizar los volúmenes
def actualizar_volumenes():
    global volumen_voz, volumen_fondo
    try:
        volumen_voz, volumen_fondo = operations_voz_fondo.read_volumes()
        print(f"Actualizados valores de volumen: Volumen Voz:
{volumen_voz}, Volumen Fondo: {volumen_fondo}")
    except Exception as e:
        print(f"Error al actualizar los volúmenes: {e}")

# Función para actualizar los volúmenes periódicamente
def actualizar_volumenes_periodicamente():
```

```
    while not update_event.is_set():  # Mientras que el evento no sea
set, se ejecutará este while
        actualizar_volumenes()
        time.sleep(1)

# Función para iniciar el hilo de actualización
def iniciar_hilo_actualizacion():
    hilo_actualizacion =
threading.Thread(target=actualizar_volumenes_periodicamente)
    hilo_actualizacion.daemon = True
    hilo_actualizacion.start()
```

Librería para el manejo de procesos

Una de las librerías de gran importancia empleadas en este TFG es **subprocess**, que permite ejecutar comandos del sistema operativo desde un script de Python para automatizar tareas como iniciar y detener el servidor de PulseAudio. Facilita la ejecución de scripts y programas externos, lo cual es esencial para lanzar el procesamiento de audio en tiempo real con *live.py*. En este TFG, una de las funciones principales de **subprocess** es lanzar el programa de separación de fuentes live.py con los parámetros de configuración necesarios (volumen de voz, volumen de fondo y retardo) en un proceso separado, permitiendo que el script *comandos.py* continúe ejecutándose sin bloquearse. En la siguiente captura de pantalla, se muestra cómo se lanza la aplicación mediante el uso de **subprocess.Popen**.

```
import subprocess

def launch_app(volumen_voz, volumen_fondo, delay):
    global process
    print("Volumen Voz:" + str(volumen_voz))
    print("Volumen Fondo:" + str(volumen_fondo))
    print("Delay:" + str(delay))

    if process is not None:
        print("El procesamiento ya está en ejecución")
        return

    save_initial_audio_routing()

    # Llama a live.py pasando los argumentos como argumentos de línea de
comandos
    process = subprocess.Popen(
        ['python', "/home/bernardo-almagro-martos/demucs/live.py", "--
volumen_voz", str(volumen_voz), "--volumen_fondo", str(volumen_fondo), "-
-delay", str(delay)]
    )

    # Lanzamos el proceso
    print(process.pid)
```

6.2.6.3 *Relación e interacción entre herramientas y bibliotecas*

6.2.6.4 *PulseAudio*

PulseAudio es el servidor de audio que gestiona la captura y reproducción del sonido. Captura el audio desde el micrófono y lo redirige hacia la aplicación de procesamiento Vocal Boost, y posteriormente hacia los dispositivos de salida como altavoces o auriculares. PulseAudio asegura que el audio capturado se envíe correctamente a los módulos de procesamiento y reproduce el audio mejorado. Configurado y controlado a través de **'pulsectl'** y 'os' para gestionar dispositivos de audio virtual, envío del volumen general del sistema, gestión de los streams de audio.

6.2.6.5 *Sounddevice*

Esta librería se encarga de capturar y reproducir el audio en tiempo real. Se conecta con 'PulseAudio' para capturar el audio procesado y reproducir las pistas de voz y fondo separadas.

6.2.6.6 *Pulsectl*

El propósito de esta biblioteca es controlar y configurar **'PulseAudio'** desde Python. Se utiliza junto con 'os' para crear dispositivos de audio virtual y gestionar las rutas de audio.

6.2.6.7 *Os*

El propósito de 'os' es manejar el sistema operativo para crear y configurar dispositivos de audio virtual de entrada y de salida. Trabaja junto con **'pulsectl'** y **'subprocess'** para gestionar la configuración de **'PulseAudio'** y el subproceso de separación dónde se encuentra DEMUCS.

6.2.6.8 *Demucs*

DEMUCS realiza la separación de fuentes de audio, permitiendo separar los diálogos respecto al resto de pistas de una película o vídeo. Se ejecuta dentro de un subproceso, utilizando **'sounddevice'** para manejar la entrada y salida de audio. La salida de Demucs incluye varias pistas individuales (por ejemplo, una pista con solo la voz y otra con el resto de pistas que conforman el fondo). Estas pistas separadas pueden ser procesadas de manera individual utilizando otros modelos o herramientas, como **Pytorch**, para aplicar mejoras adicionales a cada una.

6.2.6.9 Tkinter

Tkinter se utiliza para crear la GUI de 'Vocal Boost'. La GUI permite a los usuarios iniciar y detener la captura de audio, ajustar parámetros de procesamiento, inicia y controla el flujo de la aplicación, ejecución de subprocesos.

6.2.6.10 Subprocess

La biblioteca **'subprocess'** permite la ejecución de comandos del sistema desde el código Python. Esto es útil para configurar PulseAudio, iniciar y detener servicios y gestionar archivos de audio. **'subprocess'** se emplea para ejecutar scripts que configuran PulseAudio para capturar y reproducir audio correctamente además de iniciar DEMUCS.

6.2.6.11 Threading

Esta biblioteca para el manejo de hilos permite ejecutar tareas en paralelo, como la actualización periódica de volúmenes. Permite que la GUI de Tkinter siga respondiendo mientras se actualizan los valores de volumen de voz y fondo o se manejan otros procesos en segundo plano.

6.2.6.12 Código fuente relevante

En el desarrollo de la aplicación 'Vocal Boost', varias funciones y componentes clave permiten el procesamiento de audio en tiempo real y la interacción con el usuario. A continuación, se describen las partes más importantes del código fuente:

1. **Inicialización y Configuración de PulseAudio:** Para poder arrancar el control de volumen de 'pavucontrol' y el respectivo servidor de sonido de PulseAudio se han de definir los siguientes comandos en el script *comandos.py*, lanzados mediante subprocesos.

```
import os
import subprocess

def pulseaudio_start():
    # Inicia PulseAudio
    return subprocess.run(["pulseaudio", "--start"])

def pulseaudio_stop():
    # Detiene PulseAudio
    return subprocess.run(["pulseaudio", "--kill"])
```

2. **Enrutamiento de audio:** El enrutamiento de audio en la aplicación 'Vocal Boost' se maneja principalmente desde *interfaz_vocalboost.py* y se ejecuta a través de funciones definidas en *comandos.py*. A continuación, se explica cómo se realiza este enrutamiento y cómo se gestionan las demás operaciones clave.

Desde *interfaz_vocalboost.py*, se proporciona una interfaz para que el usuario seleccione los playbacks de audio activos que desea procesar para la mejora del diálogo. Estos playbacks se muestran en una lista desplegable. Al pulsar el botón llamado 'Mostrar Playbacks', el programa detecta las aplicaciones que están enviando audio al sistema y las muestra en la lista. Cuando el usuario selecciona una de estas aplicaciones, el playback correspondiente se envía automáticamente al script comandos.py para gestionar el enrutamiento del audio a través de las interfaces deseadas. La función **'on_playback_selected()'** llama al script *comandos.py* para que la función **'route_audio_to_loopback()'** , se encargue de enrutar el audio.

```
def mostrar_playbacks():
    try:
        pulse = Pulse('VocalBoost')
        playbacks = []
        sources = pulse.source_list()

        # Obtenemos la lista de streams de audio
        for stream in pulse.sink_input_list():
            playbacks.append(stream.proplist.get('application.name',
'Desconocido'))

        # Crear un desplegable para mostrar los playbacks
        if playbacks:
            playback_selected.set(value=playbacks[0])
            playbacks_dropdown = ttk.Combobox(frame_playbacks,
textvariable=playback_selected, values=playbacks, state="readonly",
width=40)
            playbacks_dropdown.pack(after=btn_abrir_playbacks, pady=(20,
0))  # Alinear con padding vertical

            # Asociamos el evento de Click al elemento de la lista
            playbacks_dropdown.bind("<<ComboboxSelected>>",
on_playback_selected)
        else:
            messagebox.showwarning(title="Advertencia", message="No se
encontraron playbacks disponibles.")

    except Exception as e:
        print(e)
        messagebox.showerror(title="Error", message=f"Error al obtener
los playbacks: {e}")
```

En el script *comandos.py* se define la función **'route_audio_to_loopback()'** que es la responsable de realizar el enrutamiento de audio. La finalidad de la función es realizar el enrutamiento de interfaces de audio virtual que se realizó en el subapartado *Configuración de dispositivos de audio virtual de forma manual*, de manera automática sin que el usuario tenga que realizar ningún ajuste en la herramienta de control de volumen de Pavucontrol. Primero, se inicializa la conexión con el servidor PulseAudio a través del objeto

‘Pulse('VocalBoost')’. A continuación, se buscan las interfaces de audio disponibles, específicamente la interfaz de entrada virtual ‘Monitor of Loopback Analog Stereo’ y la interfaz de salida virtual ‘Loopback Analog Stereo’. La creación de estas interfaces se realiza enviando el comando **sudo modprobe snd-aloop** mediante el script *comandos.py*, utilizando la biblioteca **os** para ejecutar comandos del sistema operativo.

Seguidamente, se recorre la lista de entradas de sink en PulseAudio y se busca un stream cuya propiedad **‘application.name’** coincida con el nombre del playback seleccionado. Si se encuentran tanto la interfaz de salida de loopback como el playback objetivo, se utiliza **‘pulse.sink_input_move’** para mover el playback a la interfaz de loopback ‘Loopback Analog Stereo’.

Además, si se detecta la interfaz de monitorización, se utiliza **‘pulse.source output move’** para redirigir el playback a dicha interfaz, permitiendo así la supervisión del audio enrutado. Esto implica que, cuando el flujo de audio seleccionado por el usuario se dirige a la interfaz de salida de loopback, la interfaz de entrada de loopback permite capturar o inspeccionar este flujo de audio. Esta interfaz de monitorización es utilizada por *live.py* para realizar la separación de la voz y el fondo en tiempo real. La configuración del audio de entrada para grabación y del audio de salida para reproducción de la separación realizada se detalló en el subapartado *Captura y Reproducción de audio en tiempo real con Sounddevice*. A continuación, se detalla del flujo completo del enrutamiento de audio realizado:

1. **Inicialización:**
   - Se establece una conexión con PulseAudio mediante el objeto **‘Pulse (‘VocalBoost’)’**.
   - Se carga el módulo **‘snd-aloop’** mediante el **comando ‘sudo modprobe snd-aloop’** para crear las interfaces de loopback de salida y entrada (monitorización) necesarias.

2. **Búsqueda de interfaces:**
   - Se realiza una búsqueda de las interfaces de audio **‘Loopback Analog Stereo’** y **‘Monitor of loopback Analog Stereo’** en PulseAudio.

3. **Identificación del Playback:**
   - Se busca en la lista de entradas de sink (sink inputs) un stream de audio cuyo nombre de aplicación **‘applicaction.name’** coincida con el playback seleccionado por el usuario.

4. **Enrutamiento:**
   - Si se encuentra tanto la interfaz de loopback de salida como el playback objetivo, se utiliza **'pulse.sink_input_move'** para mover el playback a la interfaz de loopback **'Loopback Analog Stereo'**.
   - Si se encuentra la interfaz de entrada (monitorización), se utiliza **'pulse.source_output_move'** para mover el playback a esa interfaz, permitiendo la captura del audio enrutado.

5. **Captura y procesamiento en tiempo real:**
   - El script *live.py* se configura como se explicó en apartados anteriores, para capturar el audio de entrada y en este caso para utilizar **'Monitor of loopback Analog Stereo'** como dispositivo de entrada.
   - *live.py* captura el audio desde esta interfaz de audio virtual de entrada y lo procesa en tiempo real para realizar la separación de la voz y el fondo. El resultado de la separación de fuentes se configura en el script *live.py*, en el que PulseAudio detecta el dispositivo de salida como una fuente de reproducción y muestra el resultado por los altavoces del equipo de usuario.

```
def route_audio_to_loopback(playback_name):
    try:
        with Pulse('VocalBoost') as pulse:
            loopback_sink = None
            null_output_monitor = None

            # Búsqueda de interfaces disponibles
            # Buscamos la loopback sink (Redirección)
            for sink in pulse.sink_list():
                if 'Loopback Analog Stereo' in sink.description:
                    loopback_sink = sink
                    break

            # Buscamos la loopback tipo monitor (Monitorización)
            for source in pulse.source_list():
                if 'Monitor of Loopback Analog Stereo' in
source.description:
                    null_output_monitor = source
                    break

            # Identificación del playback específico
            # Buscamos el playback específico
            target_playback = None
            for stream in pulse.sink_input_list():
                if stream.proplist.get('application.name') ==
playback_name:
                    target_playback = stream
                    break
```

```
            # Si se encuentran tanto la fuente de loopback como el stream
de audio
            if loopback_sink and target_playback:
                # Movemos el playback específico a la sink de loopback
                pulse.sink_input_move(target_playback.index,
loopback_sink.index)
                print(f"Redirigiendo {playback_name} a
{loopback_sink.description}")

                # Opcionalmente, mueve la source para monitorización
                if null_output_monitor:
                    pulse.source_output_move(target_playback.index,
null_output_monitor.index)
                    print(f"Monitorizando {playback_name} via
{null_output_monitor.description}")

    except Exception as e:
        print(e)
```

3. **Variación del volumen de voz y fondo en tiempo real:** Uno de los pasos más importantes para la realización de esta aplicación de escritorio es poder modificar el volumen de voz y fondo en tiempo real mientras el programa en sí está realizando la separación de fuentes. Por otro lado, la razón por la que no puede modificar el delay en la aplicación durante el procesamiento de separación de fuentes es debido a que este valor está directamente relacionado con el modelo de separación de fuentes, y cualquier cambio en el delay requeriría recargar y reconfigurar el modelo. En la aplicación, el retardo varía entre 25 a 250 (ms) el cual se convierte en un número de muestras que el modelo usa para poder procesar y separar el audio, lo que significa que debe ser configurado desde el inicio y no puede cambiarse dinámicamente sin reiniciar el procesamiento. El modelo **'RealTimeModel'** se inicializa con un hop o delay durante su creación, y cambiar este valor requeriría detener el procesamiento de audio, reconfigurar el modelo con el nuevo valor de delay y reiniciar el procesamiento de audio. Seguidamente, se van a explicar los pasos que se han seguido para modificar el volumen de la voz y el fondo mientras el usuario está reproduciendo una película o video y desea seleccionar la cantidad de voz y fondo según sus necesidades:
    1. La interfaz gráfica es el punto de interacción del usuario. Aquí, el usuario puede ajustar los valores de volumen de voz y fondo utilizando barras deslizantes (sliders) para ajustar la mezcla de audio (voz y fondo) según las preferencias del mismo.

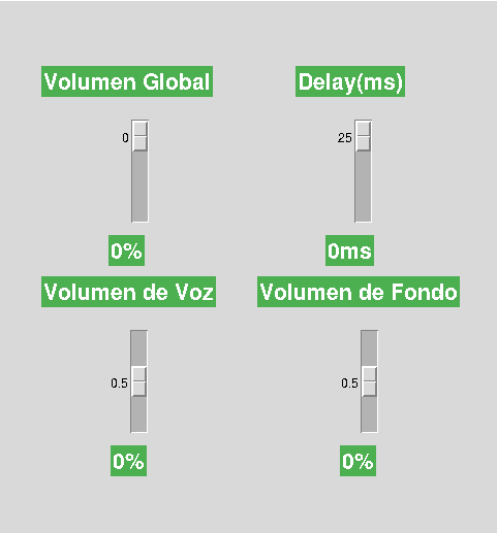


Figura 53. Barras deslizantes para volumen de voz y fondo.

2. Los sliders permiten al usuario ajustar los valores de voz y fondo de 0 a 1, donde 0 es silencio y 1 es el valor máximo de cada fuente. Dentro de la interfaz gráfica, se crea un función **'actualizar_volumenes()'** que se ejecuta cada vez que el usuario mueve una de las barras, permitiendo guardar respectivos valores. Ambos valores al haber sido guardados, se escriben en un archivo auxiliar ('aux.txt').

```
def actualizar_volumenes(*args):
    volumen_voz = barra_volumen_voz.get()
    volumen_fondo = barra_volumen_fondo.get()
    operations_voz_fondo.write_volumes(volumen_voz, volumen_fondo)

# Barra de volumen global
barra_volumen_global = crear_barra_volumen(label="Volumen Global",
from_=0, to=100, resolution=10, frame=volumen_superior,
update=update_volume, column=0, row=0, tipo="volumen", padx_right=20)

# Barra de retardo
barra_delay = crear_barra_volumen(label="Delay (ms)", from_=25, to=200,
resolution=25, frame=volumen_superior, empty=empty_volumen, column=1,
row=0, tipo="delay", padx_right=60)

# Barra de volumen de voz
barra_volumen_voz = crear_barra_volumen(label="Volumen de Voz", from_=0,
to=1, resolution=0.1, frame=volumen_inferior,
update=actualizar_volumenes, column=0, row=0, tipo="volumen",
padx_right=20)

# Barra de volumen de fondo
barra_volumen_fondo = crear_barra_volumen(label="Volumen de Fondo",
from_=0, to=1, resolution=0.1, frame=volumen_inferior,
update=actualizar_volumenes, column=1, row=0, tipo="volumen",
padx_right=20)
```

3. Se crea un script llamado *operations_voz_fondo.py* que se encarga de gestionar la escritura y lectura de los valores de volumen en el archivo auxiliar (‘aux.txt’). Esto permite que los cambios realizados en la interfaz gráfica se reflejen en el archivo. Dentro del mismo script, se definen dos funciones: **‘write_volumes()’**, escribe los valores de los volúmenes de la voz y el fondo en (‘aux.txt’) y **‘read_volumes()’** lee los valores de los volúmenes de la voz y el fondo desde (‘aux.txt’).

```
def write_volumes(voz, fondo):
    with open("aux.txt", "w") as file:
        file.write(f"{voz}\n")
        file.write(f"{fondo}\n")

def read_volumes():
    with open("aux.txt", "r") as file:
        lines = file.readlines()
        voz = float(lines[0].strip())
        fondo = float(lines[1].strip())
    return voz, fondo
```

4. El procesamiento en tiempo real se realiza en *live.py*, que lee continuamente los valores de los volúmenes desde el archivo (‘aux.txt’) y aplica estos valores al audio mientras se está reproduciendo. Para lograr esto, se ha implementado un hilo o hebra de actualización que permite gestionar los valores en segundo plano. Un hilo separado se encarga de leer los valores de los volúmenes desde (‘aux.txt’) periódicamente. Esto asegura que cualquier cambio realizado en los volúmenes en la interfaz gráfica se aplique casi de inmediato. Se crean 3 funciones que permiten solventar la actualización de los valores de volumen. Se define una función llamada **‘actualizar_volumenes()’** que lee los valores de volumen de voz y volumen de fondo desde el archivo (‘aux.txt’), esta función actualiza las variables globales **‘volumen_voz’** y **‘volumen_fondo’**. La siguiente función creada es **‘actualizar_volumenes_periodicamente()’** Proceque ejecuta **‘actualizar_volumenes()’** en un bucle, esperando un segundo entre cada actualización. Esto asegura que los valores de voz y fondo se actualicen regularmente. Por último, se define una función **‘iniciar_hilo_actualizacion()’** que crea un hilo o hebra para ejecutar **‘actualizar_volumenes_periodicamente()’**. El hilo ‘daemon’ se ejecuta en segundo plano y no bloquea la terminación del programa principal. Es decir, el hilo se cerrará automáticamente cuando el programa principal termine, incluso aunque el hilo no haya terminado su ejecución.

```
update_event = threading.Event()

sample_rate = 44100
volumen_voz = 0.5
volumen_fondo = 0.5

def actualizar_volumenes():
    global volumen_voz, volumen_fondo
    try:
        volumen_voz, volumen_fondo = operations_voz_fondo.read_volumes()
        print(f"Actualizados valores de volumen: Volumen Voz:
{volumen_voz}, Volumen Fondo: {volumen_fondo}")
    except Exception as e:
        print(f"Error al actualizar los volúmenes: {e}")

def actualizar_volumenes_periodicamente():
    while not update_event.is_set():  # Mientras que el evento no sea set
se ejecutará este while
        actualizar_volumenes()
        time.sleep(1)

def iniciar_hilo_actualizacion():
    hilo_actualizacion =
threading.Thread(target=actualizar_volumenes_periodicamente)
    hilo_actualizacion.daemon = True
    hilo_actualizacion.start()
```

5. La función principal **'main'** se encarga de la inicialización del procesamiento de audio, la configuración de los dispositivos de entrada y salida de audio (como se explica en el subapartado *Captura y Reproducción de audio en tiempo real con Sounddevice*), la gestión del hilo secundario para la actualización de los volúmenes, y el procesamiento de audio en tiempo real. En cuanto al hilo de actualización, este se llama al principio de la función '**main**' para comenzar a actualizar los volúmenes de voz y fondo que el usuario ha seleccionado, ejecutándose en segundo plano sin bloquear el hilo principal.

   Seguidamente, se entra en un bucle infinito **'while working'** dónde se captura y procesa el audio. Este bucle realiza las siguientes tareas, (como se explica en el subapartado *Procesamiento de audio en tiempo real bloque a bloque*):

   1. **Captura de audio:** Se lee el audio desde el dispositivo de entrada capturando el flujo desde la interfaz de loopback de monitorización.
   2. **Conversión a tensor:** El audio capturado se convierte a un tensor para su procesamiento.

3. **Separación de audio:** El modelo de separación de fuentes procesa el tensor para dividir el audio en las componentes de voz y fondo.

Antes de combinar la salida de ambas fuentes, dentro del hilo principal (bucle de procesamiento de audio), se utilizan los valores actualizados de **'volumen_voz'** y **'volumen_fondo'** para ajustar el volumen de las componentes de voz y fondo por separado. Finalmente, estas componentes extraídas mediante un índice de las respectivas fuentes se combinan y se convierten de un tensor a un array de numpy para reproducir por los altavoces el resultado de separación de fuentes mediante el stream de salida.

4. **Gestión de eventos y control de flujo:** Los eventos son acciones o sucesos que ocurren durante la ejecución de un programa y que pueden ser gestionados mediante funciones específicas llamadas 'handlers'. Tkinter proporciona un sistema robusto para gestionar eventos mediante el uso de 'bindings', que asocian eventos específicos a funciones de manejo.
    a. **Manejo de eventos en la interfaz gráfica:** En Tkinter, los eventos se asocian a widgets mediante el método *bind*. Uno de los eventos más comunes en la aplicación es la selección de un playback de audio desde una lista desplegable. Cuando el usuario selecciona un elemento de la lista, se llama a la función **'on_playback_selected()'** para enrutar el audio al playback correspondiente.

```
update_event = threading.Event()

sample_rate = 44100
volumen_voz = 0.5
volumen_fondo = 0.5

def actualizar_volumenes():
    global volumen_voz, volumen_fondo
    try:
        volumen_voz, volumen_fondo = operations_voz_fondo.read_volumes()
        print(f"Actualizados valores de volumen: Volumen Voz:
{volumen_voz}, Volumen Fondo: {volumen_fondo}")
    except Exception as e:
        print(f"Error al actualizar los volúmenes: {e}")

def actualizar_volumenes_periodicamente():
    while not update_event.is_set():  # Mientras que el evento no sea set
se ejecutará este while
        actualizar_volumenes()
        time.sleep(1)
```

```
def iniciar_hilo_actualizacion():
    hilo_actualizacion =
threading.Thread(target=actualizar_volumenes_periodicamente)
    hilo_actualizacion.daemon = True
    hilo_actualizacion.start()
```

b. **Control de flujo en la aplicación:** El control de flujo en 'Vocal Boost' se gestiona cuidadosamente para asegurar que todas las operaciones se realicen de manera ordenada y eficiente. Esto incluye la inicialización y configuración de PulseAudio, la interacción entre los módulos de GUI y procesamiento de audio, y la gestión de los cambios en los volúmenes de voz y fondo. La interfaz gráfica realizada interactúa con módulos de procesamiento de audio para realizar operaciones como iniciar o parar el audio y ajustar los volúmenes. Esta interacción se maneja mediante funciones de callback que se asocian a los eventos de la interfaz.

```
def procesar_audio():
    # Obtenemos los valores de las barras de volumen
    volumen_global, volumen_voz, volumen_fondo, delay =
obtener_valores_volumen()

    print("Volumen General: " + str(volumen_global))
    print("Volumen Voz: " + str(volumen_voz))
    print("Volumen Fondo: " + str(volumen_fondo))
    print("Delay: " + str(delay))

    print("Procesamiento de audio iniciado...")

    # Llamamos a la función launch_app en comandos.py con los valores de
las barras de volumen
    # Llamando a su vez a live.py pasándole los valores de voz, fondo y
delay como argumentos de la línea de comandos
    comandos.launch_app(volumen_voz, volumen_fondo, delay)
```

6.2.6.13 Lanzamiento del procesamiento de audio desde la interfaz gráfica

El procesamiento de audio en la aplicación 'Vocal Boost' es una característica central que permite ajustar los volúmenes de voz y fondo en tiempo real. Para garantizar que la experiencia del usuario sea óptima, es fundamental que este procesamiento se inicie y se gestione de manera eficiente desde la interfaz gráfica de la aplicación. Este proceso implica una serie de pasos cuidadosamente orquestados, desde la inicialización de componentes clave como PulseAudio hasta la ejecución de algoritmos avanzados de separación de fuentes de audio mediante el modelo de DEMUCS modificado para su adaptación en tiempo real. La interfaz gráfica, construida utilizando Tkinter, proporciona un punto de interacción intuitivo para los usuarios, permitiéndoles seleccionar la fuente de

audio, ajustar los volúmenes y el retardo, y activar o desactivar el procesamiento de audio. A través de una combinación de eventos de usuario, llamadas a funciones de configuración y ejecución de scripts de procesamiento, la aplicación garantiza que el audio se maneje de manera fluida y responsiva.

A continuación, se describe detalladamente cómo se realiza el lanzamiento del procesamiento de audio desde la interfaz gráfica, abarcando desde la configuración inicial hasta la ejecución del procesamiento en tiempo real. Este proceso incluye:

1. **Inicialización de la aplicación y configuración inicial.** Al iniciar la aplicación de escritorio 'Vocal Boost', se realiza la configuración inicial de PulseAudio y se crean los dispositivos de audio virtual. Esto asegura que todos los componentes necesarios estén listos para el procesamiento de audio. La configuración de PulseAudio y la creación de los dispositivos de audio virtual se realiza en el script *comandos.py*. En la siguiente captura se observa como se llama las funciones respectivas para asegurar que PulseAudio esté corriendo y que los dispositivos de audio estén configurados correctamente antes de que la interfaz gráfica sea presentada al usuario.

```
import comandos

comandos.pulseaudio_start()
comandos.init_snd_aloop()
```

2. **Configuración e inicialización de la interfaz gráfica.** La interfaz gráfica se construye como anteriormente se mencionó utilizando Tkinter y se configura para incluir varios widgets, como botones para activar o desactivar el procesamiento, lista desplegable para seleccionar los playbacks y las respectivas barras de volumen. Se muestra, a continuación, algunas de las configuraciones iniciales realizadas.

```
import tkinter as tk
from tkinter.font import Font
from tkinter import ttk

# Creamos la ventana principal y definimos el título en negrita
master = tk.Tk()
master.geometry("400x400")
master.title("Vocal Boost: Modificación del volumen de los diálogos en
tiempo real")

# Configuración de estilo para botones
style1 = ttk.Style()
style1.theme_use('alt')
```

```
style1.configure('TButton', background='#4CAF50', foreground='white',
font=('Arial', 12), padding=10)

# Título de la aplicación
font_title = Font(family="Helvetica", size=16, weight="bold")
title_label = ttk.Label(master, text="Vocal Boost", font=font_title,
background='#4CAF50', foreground='white')
title_label.pack(pady=10, padx=10, fill='x')
```

3. **Selección del playback de audio activo.** El usuario puede seleccionar un playback de audio activo desde una lista desplegable. Este evento desencadena la función explicada en el subapartado *Código fuente relevante en el punto 2, 'Enrutamiento de audio'* permitiendo enrutar el audio al playback correspondiente. En el siguiente fragmento de código, se define una función para mostrar los playbacks de audio disponibles y manejar la selección del usuario.

```
def mostrar_playbacks():
    try:
        pulse = Pulse('VocalBoost')
        playbacks = []
        sources = pulse.source_list()

        # Obtenemos la lista de streams de audio
        for stream in pulse.sink_input_list():
            playbacks.append(stream.proplist.get('application.name',
'Desconocido'))

        # Crear un desplegable para mostrar los playbacks
        if playbacks:
            playback_selected.set(value=playbacks[0])
            playbacks_dropdown = ttk.Combobox(frame_playbacks,
textvariable=playback_selected, values=playbacks, state="readonly",
width=40)
            playbacks_dropdown.pack(after=btn_abrir_playbacks, pady=(20,
8))  # Alinear con padding vertical

            # Asociamos el evento de click al elemento de la lista
            playbacks_dropdown.bind("<<ComboboxSelected>>",
on_playback_selected)
        else:
            messagebox.showwarning(title="Advertencia", message="No se
encontraron playbacks disponibles.")
    except Exception as e:
        print(e)
        messagebox.showerror(title="Error", message=f"Error al obtener
los playbacks: {e}")

def on_playback_selected(event):
    # Cuando se hace Click en un elemento de la lista, enrutamos el audio
al playback correspondiente
    playback_name = playback_selected.get()
    if playback_name:
```

```
        conecta_nuevo_audio_to_loopback(playback_name)
        boton_activar_desactivar.config(state=tk.NORMAL)  # Habilitar el
botón cuando se seleccione un playback
    else:
        messagebox.showerror(title="Error", message="Seleccione un
playback de la lista")
```

4. **Configuración de barras de volumen y delay.** El usuario puede ajustar los volúmenes de la voz y el fondo, así como el delay, utilizando barras de volumen. Estos valores se actualizan en tiempo real y se almacenan para ser utilizados durante el procesamiento de audio. En la interfaz gráfica, las barras de volumen de voz y fondo parten inicialmente del mismo valor (0.5) para que el usuario pueda empezar a mejorar el diálogo desde una perspectiva principal sin mejora. En los siguientes fragmentos de código, se muestra un ejemplo de cómo se han creado y configurado las barras de volumen y delay.

```
def crear_barra_volumen(label, from_, to, resolution, parent,
funcion_update, column, row, tipo, padx_right=0):
    frame_barra = ttk.Frame(parent)

    frame_barra.grid(column=column, row=row, padx=padx_right, pady=5)

    label_nombre = tk.Label(frame_barra, text=label, **label_style)  #
Aplicar el estilo al label
    label_nombre.grid(column=0, row=0, pady=(0, 10))  # Usamos grid para
una alineación precisa

    barra_volumen = tk.Scale(frame_barra, from_=from_, to=to,
resolution=resolution, orient='vertical',
                             command=funcion_update)
```

```
volumen_voz = tk.DoubleVar()
volumen_voz.set(0.5)

volumen_fondo = tk.DoubleVar()
volumen_fondo.set(0.5)

if label == "Volumen de Voz":
    barra_volumen.config(variable=volumen_voz)
elif label == "Volumen de Fondo":
    barra_volumen.config(variable=volumen_fondo)

# Configuración de la barra de volumen
barra_volumen.grid(column=0, row=1, pady=(10, 5))  # Alinear con grid
valor_inicial = "0%" if tipo == "volumen" else "0ms"
label_valor = tk.Label(frame_barra, text=valor_inicial, **label_style)  #
Aplicar el estilo al label
label_valor.grid(column=0, row=2, pady=(5, 0))  # Alinear con grid y
mover debajo de la barra

def actualizar_valor(event):
```

```
    valor = barra_volumen.get()
    if tipo == "volumen":
        label_valor.config(text=f"{valor}%")
    elif tipo == "delay":
        label_valor.config(text=f"{valor}ms")

barra_volumen.bind("<ButtonRelease-1>", actualizar_valor)
```

5. **Lanzamiento del procesamiento de audio.** El procesamiento de audio se lanza cuando el usuario activa la aplicación a través del botón 'Activar/Desactivar'. Esta acción inicialmente al seleccionar el playback de audio activo y el delay óptimo, llama a la función **'procesar_audio()'** que se encarga de obtener los valores de las barras de volumen y delay.

```
def obtener_valores_volumen():
    volumen_global = barra_volumen_global.get()
    volumen_voz = barra_volumen_voz.get()
    volumen_fondo = barra_volumen_fondo.get()
    delay = barra_delay.get() / 1000

    return volumen_global, volumen_voz, volumen_fondo, delay

def procesar_audio():
    # Obtenemos los valores de las barras de volumen
    volumen_global, volumen_voz, volumen_fondo, delay =
obtener_valores_volumen()

    print("Volumen General: " + str(volumen_global))
    print("Volumen Voz: " + str(volumen_voz))
    print("Volumen Fondo: " + str(volumen_fondo))
    print("Delay: " + str(delay))
    print("Procesamiento de audio iniciado.")

    comandos.launch_app(volumen_voz, volumen_fondo, delay)
```

Esta función **'procesar_audio()**' a su vez llama a una función llamada **'launch_app()'** en el script *comandos.py* que se encarga de lanzar el procesamiento de audio en tiempo real mediante el script *live.py*. La función **'launch_app()'** se encarga de lanzar el script live.py mediante un subproceso **'subprocess.Popen'** pasando los valores de voz, fondo y delay como argumentos de la línea de comandos al script *live.py*, lo que permite que el procesamiento de audio en tiempo real utilice estos valores.

6. **Procesamiento de audio en *live.py*.** El script *live.py* se encarga de leer los valores de volumen recibidos al lanzar la aplicación, aplicar el modelo DEMUCS para la separación de fuentes en el caso de la voz y el fondo en tiempo real y ajustar el volumen de voz y fondo para que se puedan modificar mientras se

procesa el audio. Una vez que *live.py* está en ejecución se emplea, como se explica en el subapartado *Código fuente relevante 'Variación del volumen de voz y fondo en tiempo real'*, un hilo secundario para actualizar periódicamente los valores de volumen leyendo desde el fichero **'operations_voz_fondo.py()'**, permitiendo que los valores se mantengan actualizados incluso si cambian después del lanzamiento inicial. En la función **main**, se configura el modelo DEMUCS y se prepara para la separación de fuentes en tiempo real con todas las configuraciones realizadas en el subapartado *Captura y Reproducción de audio en tiempo real con Sounddevice.*

En definitiva, el usuario hace click en el botón de activación/desactivación. La función de 'Activar/Desactivar' comprueba el estado de procesamiento de audio usando la variable **'estado_boton'**, si el procesamiento está inactivo (**'estado_boton'** es False), **'activar_desactivar()'** llama a **'procesar_audio()'** . Esta última función, obtiene los valores de las barras de volumen y delay llamando a **'obtener_valores_volumen()'**. Al obtener estos valores de las barras de volumen, **'procesar_audio()'** llama a **'launch_app()'** en el script *comandos.py*, pasando los valores obtenidos. **'launch_app()'** utiliza **'subprocess.Popen'** para lanzar el script *live.py* , pasando los valores de volumen delay como argumentos de la línea de comandos. En última instancia, *live.py* lee los argumentos de la línea de comandos y configura el procesamiento en tiempo real mientras que los valores de volumen y delay se actualizan periódicamente leyendo los valores desde el archivo ('aux.txt'). La función 'action_update()' se asegura que la barra de delay se deshabilite debido al estado del botón.

6.2.6.14 Desactivación del procesamiento de audio

La desactivación de la aplicación 'Vocal Boost' implica detener el procesamiento de audio en tiempo real y restaurar la configuración de audio original. Este subapartado incluye como se realiza la interacción del usuario desde que ejecuta el botón partiendo de que la aplicación previamente ha sido activada, la finalización del script *live.py* y la restauración de la configuración de audio.

El usuario al haber pulsado el botón de 'Activar/Desactivar' llama a una función internamente **'activar_desactivar()'** que gestiona el procesamiento de audio. Es decir, si el procesamiento de audio está activado, la función llama a **'stop_audio()'** para detenerlo. Además, se ha configurado una función para actualizar el estado del botón y ajustar la interfaz en consecuencia. Esta función **'action_update()'** desactiva la barra de delay

cuando el procesamiento de audio está activo y la habilita cuando el procesamiento de audio está inactivo. A continuación, se adjunta el código de ambas funciones.

```
def action_update():
    global estado_boton
    if estado_boton:
        barra_delay.config(state=tk.DISABLED)
    else:
        barra_delay.config(state=tk.ACTIVE)
    estado_boton = not estado_boton

def activar_desactivar():
    action_update()
    try:
        if estado_boton:
            stop_audio()
        else:
            procesar_audio()
    except Exception as e:
        messagebox.showerror(title="Error", message=f"Ocurrió un error al
activar/desactivar el procesamiento de audio: {str(e)}")
```

Cuando **'activar_desactivar()'** detecta que el procesamiento de audio está activo, llama a **'stop_audio()'**. Esta última función, se encarga de detener el script *live.py* y restaurar la configuración de audio original.

```
def stop_audio():
    print("Deteniendo procesamiento de audio...")
    comandos.kill_app()
    print("Procesamiento de audio detenido.")
```

La función **'kill_app()'** en *comandos.py* se encarga de finalizar el proceso *live.py* , se emplea el módulo **'os'** para enviar una señal de terminación al proceso y a esperar a que termine.

```
def kill_app():
    global process
    print('Terminar proceso')
    if process is not None:
        # Entra en el sistema operativo, reconoce el proceso mediante un
ID y envía una orden de parada.
        os.kill(process.pid, signal.SIGKILL)
        process.wait()
        process = None
    restore_initial_audio_routing()
```

Después de detener el proceso *live.py*, la función **'kill_app()'** llama a **'restore_initial_audio_routing()'** para restaurar la configuración de audio original. Esta función se asegura de que el enrutamiento de audio vuelva al estado anterior al inicio del procesamiento de audio para que el usuario vuelva a tener sus interfaces de audio de

entrada y de salida como estaban establecidas antes de ejecutar la aplicación. En el siguiente fragmento de código se desarrolla, tanto la función de guardar la configuración inicial de audio antes de comenzar el procesamiento de audio, como la función que se encarga de restaurar la configuración inicial al desactivar la aplicación.

```
def save_initial_audio_routing():
    global initial_sink
    with Pulse('VocalBoost') as pulse:
        # Guardar la configuración inicial del sink (salida de audio)
        default_sink = pulse.server_info().default_sink_name
        for sink in pulse.sink_list():
            if sink.name == default_sink:
                initial_sink = sink
                break

def restore_initial_audio_routing():
    global initial_sink
    with Pulse('VocalBoost') as pulse:
        if initial_sink:
            for stream in pulse.sink_input_list():
                pulse.sink_input_move(stream.index, initial_sink.index)
                print(f'Restaurar
{stream.proplist.get("application.name")} a {initial_sink.description}')
        else:
            print('Ninguna configuración inicial de salida encontrada.')
```

En definitiva, el usuario hace click en el botón de activación/desactivación, la función de 'Activar/Desactivar' comprueba el estado del procesamiento de audio usando la variable **'estado_boton'**. Si el procesamiento de audio está activo (**'estado_boton'** es True) la función **'activar_desactivar()'** llama a **'stop_audio()'**, por lo que, **'stop_audio()'** llama a **'kill_app()'** en el script *comandos.py* . Por otro lado, **'kill_app()'** envía una señal de terminación al proceso *live.py* y restaura la configuración de audio original llamando a **'restore_initial_audio_routing()'** . Por último, el procesamiento de audio se detiene y la configuración de audio se restaura.

6.2.6.15 Mensajes de alerta

En el desarrollo del software 'Vocal Boost', la gestión adecuada de mensajes de alerta y error es crucial para mejorar la experiencia del usuario y facilitar la resolución de problemas. En el caso de esta aplicación, se han implementado diversas alertas y mensajes de error que ayudan a los usuarios a entender y resolver situaciones inesperadas durante su uso. Este apartado describe algunos de los mensajes de alerta y error más relevantes y su importancia en la interacción con la aplicación.

1. **No se encontraron playbacks disponibles:** Cuando un usuario intenta mostrar los playbacks de audio activos, puede ocurrir que no se encuentren

dispositivos disponibles. En este caso, se muestra el mensaje de advertencia o alerta “No se encontraron playbacks disponibles”. Este mensaje informa al usuario de que en el sistema no hay streams de audio disponibles, lo que podría deberse a la falta de dispositivos de audio conectados o con problemas con el servicio Pulse Audio.

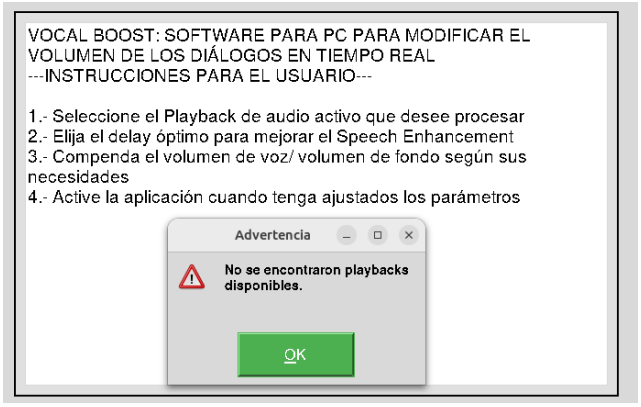


Figura 54. Advertencia / No se encontraron playbacks disponibles.

2. **Ocurrió un error al activar/ desactivar el procesamiento de audio:** El proceso de activar o desactivar el procesamiento de audio puede fallar a diversas razones, como problemas en la configuración del sistema o errores en el código de la aplicación. En tales casos, el mensaje informa al usuario de que ha ocurrido un problema durante la operación.

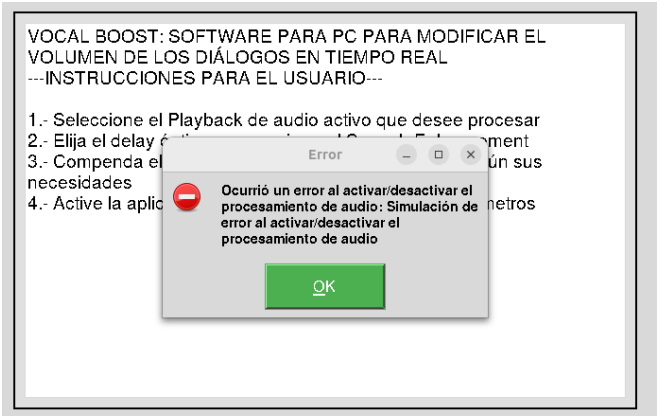


Figura 55. Error / Ocurrió un error al activar/desactivar el procesamiento de audio.

3. **No se pudo encontrar la interfaz de loopback:** Este mensaje se muestra cuando no se encuentra una interfaz de loopback disponible. La interfaz de loopback es uno de los pilares fundamentales para redirigir el audio, y si no está disponible, la operación de enrutamiento no puede completarse. Este mensaje alerta al usuario sobre este problema específico para que pueda verificar la configuración del sistema y la disponibilidad de la interfaz de loopback.

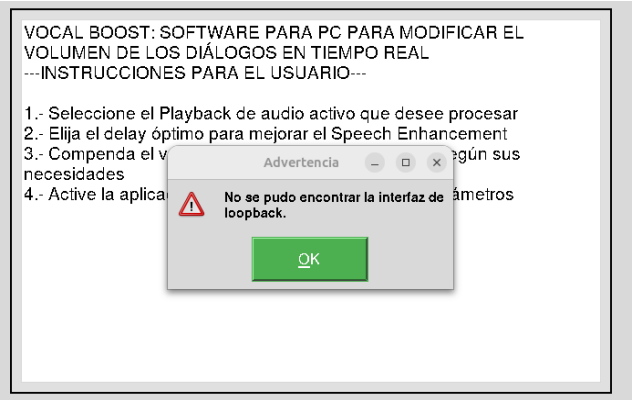


Figura 56. Advertencia / No se pudo encontrar la interfaz de loopback.

4. **No se pudo encontrar la reproducción especificada:** Este mensaje se muestra cuando la función no puede encontrar la reproducción de audio especificada (playback) entre las entradas de audio activas. Esto podría ocurrir si el nombre del playback proporcionado no coincide con ninguna de las reproducciones activas. Este mensaje informa al usuario sobre la necesidad de verificar que el playback que intenta enrutar existe y está activo.

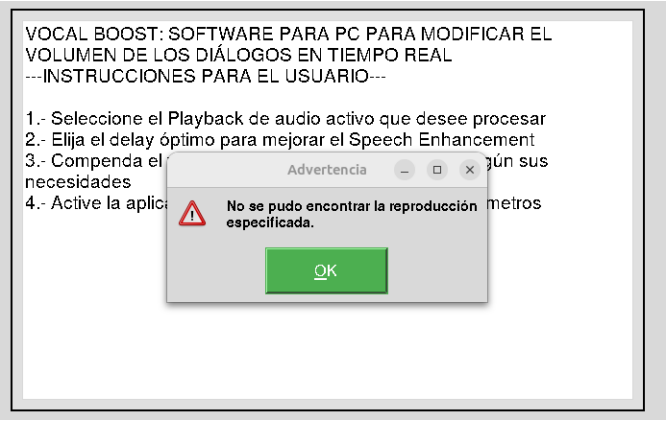


Figura 57. Advertencia / No se pudo encontrar la reproducción especificada.

5. **Aumento del delay:** Aumentar el delay en la separación de fuentes de audio, como la voz y el fondo, permite a la aplicación disponer de más tiempo para procesar el audio. Este incremento de tiempo puede mejorar la precisión del procesamiento en tiempo real, especialmente en condiciones de alta carga. Los efectos negativos que causa son la desincronización entre el audio y el vídeo, un delay mayor puede resultar en que el audio y el vídeo no se mantengan sincronizados. En contextos como películas o programas de televisión, esta falta de sincronización puede ser notable y molesta. Todo ello dependerá de las capacidades computacionales del equipo en términos de GPU.

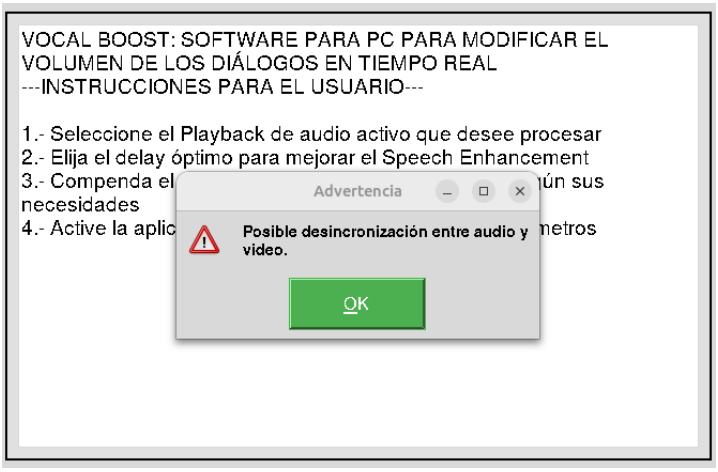


Figura 58. Advertencia / Posible desincronización entre audio y video.

6. **Reducción del delay:** Reducir el delay puede disminuir la latencia, manteniendo el audio y el video más sincronizados. Sin embargo, es crucial entender la diferencia entre el delay y el tamaño del salto (hop length) en un proceso de solapamiento y suma. El retardo depende de la ventana utilizada: si se usan ventanas solapadas en un 50%, el retardo será el doble del tamaño de salto, ya que las muestras anteriores al tamaño de salto aún están siendo procesadas desde la mitad de la ventana anterior. Reducir el tamaño del salto implica procesar segmentos de audio más pequeños y con mayor frecuencia, lo que aumenta la demanda de recursos computacionales en la GPU. Esto puede sobrecargar el sistema si no está adecuadamente equipado. Además, una mayor carga computacional puede resultar en artefactos de separación, afectando la claridad y fidelidad del audio. Estos artefactos pueden manifestarse como ruidos o distorsiones en la reproducción del audio procesado.

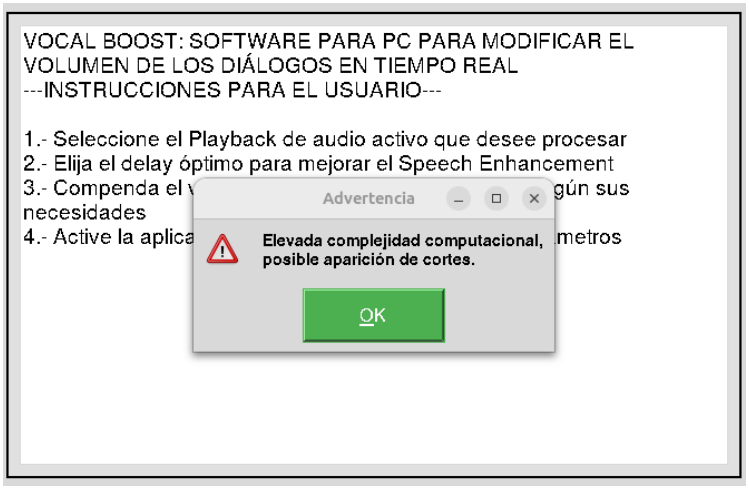


Figura 59. Advertencia / Elevada complejidad computacional posible aparición de cortes.

Los mensajes de advertencia y error enunciados arriba son solo una parte de los mecanismos implementados en la aplicación 'Vocal Boost' para garantizar una experiencia de usuario óptima. Además de estos mensajes, la aplicación maneja diversas excepciones y utiliza mensajes de depuración (**'print'**) que facilitan la identificación y resolución de problemas durante el desarrollo y la ejecución. Estos mecanismos adicionales han permitido una programación más robusta y eficiente, asegurando que la aplicación pueda manejar una amplia variedad de situaciones de manera adecuada.

7. RESULTADOS Y DISCUSIÓN

7.1 MÉTRICAS ESTÁNDAR PARA EVALUAR LA SEPARACIÓN DE FUENTES DE AUDIO

Evaluar la calidad de la separación es esencial para el desarrollo y la mejora de los algoritmos de separación de fuentes de audio. Las métricas de evaluación proporcionan una medida cuantitativa de la calidad de la separación, permitiendo comparar diferentes métodos y determinar cuáles ofrecen los mejores resultados en términos de claridad y fidelidad. Se calculan matemáticamente y producen un resultado en (dB) teniendo en cuenta que, cuanto mayor sea su valor, mejor será el resultado de separación de las fuentes. Además de las métricas objetivas, la evaluación subjetiva implica que evaluadores humanos escuchen las muestras y den una puntuación de la salida del sistema de separación de fuentes. Aunque son más costosas y lentas de obtener, pueden ser más fiables al reflejar la percepción humana real de la calidad del audio. En este TFG, se han empleado medidas objetivas pero las dos formas de medir la calidad de la separación proporcionan una evaluación integral de los algoritmos de separación de fuentes, permitiendo una mejora continua y una comparación objetiva entre diferentes enfoques (Manilow, y otros, 2020).

Componentes:

- S_{target}: la fuente original que se desea recuperar.
- e_{interf}: interferencia de otras fuentes en la fuente separada (por ejemplo, el sonido de la guitarra que se escucha en la pista de voz separada).
- e_{noise}: ruido no relacionado presente en la fuente separada (por ejemplo, ruido de fondo o artefactos introducidos durante la separación).
- e_{artif}: artefactos introducidos por el proceso de separación (por ejemplo, distorsiones que no estaban presentes en la grabación original).

SDR (Source-to-Distortion Ratio)

La SDR mide la relación entre la energía de la señal original y la energía de la distorsión presente en la señal separada. Evalúa cuánta distorsión se ha introducido durante el proceso de separación, debido a que estima la cantidad de señal original que se preserva en la señal separada.

$$SDR = 10\ log_{10}\left(\frac{\left||S_{target}|\right|^2}{\left||e_{interf} + e_{noise} + e_{artif}|\right|^2}\right) \quad (7)$$

Un SDR más alto indica una mejor calidad de la separación, ya que muestra que se ha conservado más de la señal original y se ha introducido menos distorsión.

SIR (Source-to-Interference Ratio)

La SIR mide la cantidad de interferencia de otras fuentes presentes en la señal separada. Esta métrica evalúa específicamente cuánta contaminación de otras fuentes está presente en la señal separada.

$$SIR = 10\ log_{10}\left(\frac{\left||S_{target}|\right|^2}{\left||e_{interf}|\right|^2}\right) \quad (8)$$

Un SIR más alto indica una menor interferencia de otras fuentes, lo que implica una mejor capacidad del modelo para aislar la fuente deseada. Es una medida crucial para entender la eficacia del algoritmo en la separación de la fuente objetivo de las demás fuentes presentes en la mezcla.

SAR (Source-to-Artifact Ratio)

La SAR mide la cantidad de artefactos no deseados introducidos en la señal separada. Los artefactos son ruidos adicionales o distorsiones que no estaban presentes en la grabación original y que se generan debido a las limitaciones del algoritmo de separación.

$$SAR = 10\ log_{10}\left(\frac{\left||S_{target} + e_{interf} + e_{noise}|\right|^2}{\left||e_{artif}|\right|^2}\right) \quad (9)$$

Un SAR más alto indica una menor cantidad de artefactos, lo que se traduce en una mayor fidelidad de la señal separada respecto a la original.

En este TFG se aborda la comparación y evaluación de las métricas anteriormente mencionadas de los dos modelos de separación de fuentes de audio: el modelo original de DEMUCS y el modelo de DEMUCS modificado de baja latencia para adaptación en tiempo real desarrollado en este TFG. El modelo DEMUCS es conocido por su capacidad de realizar una separación de alta calidad, mientras que la adaptación propuesta busca

optimizar el rendimiento en tiempo real, lo que es crucial para aplicaciones como la desarrollada donde la latencia baja es un requisito indispensable. Además, se analiza el impacto de la latencia en la calidad de separación y en el tiempo de procesado, probando el modelo adaptado con diferentes valores de latencia.

7.1.1 MUSDB18-HQ

Para llevar a cabo la evaluación, se utilizará el subconjunto de test de MUSDB18-HQ, una base de datos estándar en la investigación de separación de fuentes de audio. La base de datos contiene 150 mezclas completas (pistas de música) y sus respectivas fuentes individuales (vocals, bass, drums, other). MUSDB18-HQ se diferencia de MUSDB18 en la calidad de audio, ofreciendo archivos en formato de alta resolución (WAV a 44.1 KHz, 24 bits) (Rafii, y otros, 2019). En este TFG, MUSDB18-HQ se ha utilizado como base de datos principal para la evaluación de los modelos de separación de fuentes musicales. Cada pista de audio tiene su propia carpeta que contiene el archivo de mezcla y cada una de las fuentes individuales. Para el proceso de lectura de los archivos de audio, cada archivo de mezcla y sus fuentes correspondientes se leen utilizando la biblioteca de **'torchaudio'**. Las fuentes de bajo, batería y otros instrumentos se combinan en una pista de fondo. Además, para evitar problemas de memoria en términos de GPU, se ha limitado la longitud del audio a un valor máximo de 307200 muestras (~70 segundos a 44.1 kHz), tal y como se muestra en el *Anexo II: Problemas de GPU con archivos de audio grandes*. A continuación, se muestra una captura de pantalla para visualizar cómo se organiza la carpeta de test y los diferentes archivos de audio (Rafii, y otros, 2019).

Figura 60. Subconjunto de test de MUSDB18-HQ. (Rafii, y otros, 2019)

7.1.2 Librería 'museval'

La librería **'museval'** se utiliza en la comunidad de separación de fuentes musicales para evaluar la calidad de separación (SDR, SIR y SAR). En el presente TFG, se ha

implementado para evaluar de manera cuantitativa la calidad de los modelos de separación de fuentes, tanto el modelo original de DEMUCS como la versión modificada de DEMUCS para baja latencia.

```
# SDR, SIR, SAR
sdr, _, sir, sar = museval.evaluate(ref_sources, est_sources,
win=min_length)
```

7.1.3 Evaluación objetiva de los modelos

Como se explicó con anterioridad, la evaluación se realiza utilizando el conjunto de datos de 'test' de MUSDB18-HQ. Cada pista contiene los archivos de la mezcla completa y sus respectivas fuentes individuales (vocales, bajo, batería y otros). En primer lugar, para cada pista en el conjunto de datos se leen la mezcla completa y las fuentes individuales.

```
# Read audio files
xt, sr = ta.load(mix_path)
vt, _ = ta.load(voc_path)
bt, _ = ta.load(bass_path)
dt, _ = ta.load(drums_path)
ot, _ = ta.load(other_path)
```

Las fuentes individuales de bajo, batería y otros instrumentos se combinan en una única pista de fondo, lo que simplifica la evaluación de la separación de voz del resto. Este paso es importante para formar la referencia de comparación.

```
# Combine bass, drums, and other into one background track
rt = bt + dt + ot
```

Se ha implementado una función para seleccionar un segmento del archivo de audio que contenga las muestras más relevantes alrededor del máximo del valor absoluto de la señal vocal. Esto asegura que el segmento seleccionado contenga fuente de voz.

```
def select_indexes_around_max(tensor, N):
    if N > tensor.size(0):
        return 0, tensor.size(0)

    max_index = torch.argmax(torch.abs(tensor)).item()
    half_N = N // 2

    start_index = max(0, max_index - half_N)
    end_index = min(tensor.size(0), max_index + half_N + (N % 2))

    if end_index - start_index < N:
        if start_index == 0:
            end_index = N
        elif end_index == tensor.size(0):
            start_index = tensor.size(0) - N
```

```
    return start_index, end_index
```

Para poder estimar las fuentes de audio, los modelos de separación tanto el original como el modificado se aplican a las mezclas de audio. El modelo original procesa el segmento elegido de una vez, mientras que el modelo de DEMUCS modificado para baja latencia procesa la señal en trozos pequeños.

- **Modelo Original:**

```
def separate_orig_model(mixture):
    xt = mixture.unsqueeze(0).to(device)
    tstart = time.time()  # Inicio del tiempo de ejecución
    output = apply_model(model, xt, shifts=0, split=False,
device=device)[0, ...]
    tend = time.time()  # Fin del tiempo de ejecución
    output = output.to('cpu')
    vet = output[vocal_idx, ...]
    ret = torch.sum(output[backg_idx, ...], dim=0, keepdim=False)

    return vet, ret, np.array([tend - tstart])
```

- **Modelo de baja latencia:**

```
def separate_rt_model(mixture):
    xt = mixture.unsqueeze(0).to(device)
    nsources = len(model.model.sources)
    nchannels = mixture.shape[-2]
    nsamples = mixture.shape[-1]
    wavout = torch.zeros(1, nsources, nchannels, nsamples).to(device)
    hop = model.hop
    winhophalf = model.winhop // 2
    index = 0
    tstart = time.time()
    while index + hop < nsamples:
        wavout[:, :, :, index:index + hop] = model.separate(xt[:, :, :,
index:index + hop])
        index += hop
    tend = time.time()
    wavout = wavout[0, :, :, winhophalf:]  # align with original mixture
    wavout = wavout.to('cpu')
    vet = wavout[vocal_idx, ...]
    ret = torch.sum(wavout[backg_idx, ...], dim=0, keepdim=False)

    return vet, ret, np.array([tend - tstart])
```

Al realizar la separación de las fuentes (vocals y background), las señales que han sido separadas se evalúan empleando la librería descrita en el apartado anterior', **'museval',** la cual permite comparar las fuentes estimadas (vocals y background) con las fuentes originales de referencia.

```
# SDR, SIR, SAR
sdr, _, sir, sar = museval.evaluate(ref_sources, est_sources,
win=min_length)
```

Los resultados de la evaluación para cada fichero se almacenan y al final se obtiene la media de SDR, SIR y SAR para todos los ficheros. Se han incluido tanto las métricas objetivas como los tiempos de ejecución. En el caso de los tiempos de ejecución, se mide el tiempo que se tarda en procesar el fichero y se divide entre la duración del audio procesado, obteniendo así una medida de tiempo de ejecución por segundo de audio. Recordemos que las medidas han sido efectuadas en un equipo portátil con GPU GTX 1650.

```
# Save to file
results = {
    'sdr': sdr_list,
    'sir': sir_list,
    'sar': sar_list,
    'time': time_list,
    'dur': dur_list,
}
```

7.1.3.1 Resultados de separación de audio

Los modelos seleccionados en la comparación son los siguientes: DEMUCS original, DEMUCS en tiempo real con 100 ms de latencia y DEMUCS en tiempo real con 50 ms de latencia. En el caso de DEMUCS original, se procesa el audio completo de una vez. En el caso de DEMUCS en tiempo real, el audio se procesa en segmentos de 1 segundo, con la latencia (o salto) indicada.

En los tres casos, se han empleado dos conjuntos de pesos pre-entrenados. El primero es **'5d2d6c55.th'** (48 canales), que procede del modelo DEMUCS original, y el segundo es **'0fd91e92.th'** (32 canales), que ha sido entrenado para este TFG por el co-tutor Pablo Antonio Cabañas Molero, tal y como se explicó en el apartado *Enfoque General* dentro del capítulo 'Materiales y Métodos'. A continuación, se muestran cada una de las tablas con los resultados de ambos modelos y el análisis individual de cada tabla y también mutuo para denotar el mejor modelo preentrenado.

5d2d6c55.th

Métrica	Modelo Original	Modelo en tiempo real (100 ms)	Modelo en tiempo real (50 ms)
SDR Vocals (dB)	9,304463165	6,572340646	3,925311792
SDR Background (dB)	15,42863014	12,60875524	9,991395257
SIR Vocals (dB)	19,64951215	17,32464825	14,67413008

SIR Background(dB)	23,49685623	18,73827586	13,8950213
SAR Vocals(dB)	9,735356117	6,062507478	2,182490475
SAR Background(dB)	16,96500747	14,88304781	13,62527609
Tiempo de ejecución por segundo de audio (ms)	28.44590375	470.1605755	990.3365887

Tabla 1: Comparativa entre métricas de calidad objetiva usando el archivo de pesos '5d2d6c55.th'

En la tabla 7.1 se observa una disminución en las métricas SDR, SAR y SIR en los modelos en tiempo real en comparación con el modelo original. Para la métrica SDR, los valores para la voz y el fondo disminuyen progresivamente con la reducción del delay. Esto se debe a que los resultados de separación de DEMUCS son siempre peores en los extremos del segmento de audio procesado (Venkatesh, y otros, 2024). El motivo es que DEMUCS contiene algunas capas que tienen en cuenta todo el audio de entrada, lo que implica que el resultado para cada instante está afectado por todas las muestras futuras y pasadas. En nuestro caso, las muestras finales del segmento no cuentan con información del futuro para realizar la separación, lo que implica unos peores resultados (especialmente para valores muy bajos de latencia).

Por otro lado, el modelo en tiempo real con baja latencia (100 ms y 50 ms) muestra un incremento significativo en los tiempos de ejecución (470 ms y 990 ms, respectivamente). Esto se debe a la necesidad de aumentar el número de llamadas al modelo (por ejemplo, en el caso de 50 ms de latencia, la separación se realiza cada 50 ms con segmentos de 1 s, implicando el doble de complejidad computacional que en el caso de 100 ms). Los tiempos de DEMUCS original son especialmente bajos porque solo se realiza una llamada al modelo de separación, y además la paralelización realizada por la GPU es más efectiva al procesar el audio al completo.

0fd91e92.th

Métrica	Modelo Original	Modelo en tiempo real (100 ms)	Modelo en tiempo real (50 ms)
SDR Vocals(dB)	6,301888155	4,90097211	4,285265652
SDR Background(dB)	12,30300569	10,95065937	10,51724253
SIR Vocals(dB)	14,90100164	14,89812477	12,58683849
SIR Background(dB)	18,18510203	14,94416981	14,35540999
SAR Vocals(dB)	6,73986291	4,652214644	4,201831474
SAR Background(dB)	15,34372326	14,7587551	14,26095046
Tiempo de ejecución por segundo de audio (ms)	14.31021695	319.6759025	639.9395531

Tabla 2: Comparativa entre métricas de calidad objetiva usando el archivo de pesos '0fd91e92.th

Las métricas SDR, SIR y SAR son más bajas con el conjunto de pesos, **'0fd91e92.th'**, lo que indica una menor efectividad en la separación de voces y fondos, así como una mayor presencia de artefactos en las fuentes estimadas. Esto es debido a que este modelo posee un menor número de parámetros respecto al anterior (32 canales vs 48). Además, según la documentación de DEMUCS, el modelo **'5d2d6c55.th'** fue entrenado con 800 canciones adicionales no incluidas en MUSDBHQ, y que no están disponibles públicamente, mientras que el entrenamiento de **'0fd91e92.th'** se limitó a MUSDBHQ. Aunque no se alcanzan valores tan elevados en términos de calidad de separación de fuentes (SDR, SAR y SIR) en comparación con el modelo anterior, este modelo ha sido optimizado para funcionar de manera eficiente en tiempo real, adaptándose específicamente a la GPU disponible al aprovechar al máximo los recursos de hardware.

7.1.4 Medición del tiempo de ejecución

Para evaluar el rendimiento en términos de velocidad de los modelos de separación de fuentes (DEMUCS normal y DEMUCS modificado para baja latencia), se ha medido el tiempo de ejecución por segundo de audio procesado utilizando la GPU disponible (GTX 1650 en un equipo portátil). Esta métrica es fundamental para asegurar que la separación de audio se puede realizar en tiempo real. El tiempo de ejecución se mide antes y después del proceso de separación. Para comparar la eficiencia de ambos modelos, se calcula el tiempo promedio de ejecución por segundo de audio procesado. Es decir, esto se realiza dividiendo los tiempos totales de ejecución de cada modelo (**'time_list'**) por las respectivas duraciones de los audios procesados (**'dur_list'**).

```
time_mean = np.mean(time_list / dur_list, axis=0)
print(f'Exec time per audio second = {time_mean} sec')
```

7.1.4.1 Comparación de los tiempos de ejecución

	Modelo Original	Modelo en Tiempo Real (100 ms)	Modelo en Tiempo Real (50 ms)
5d2d6c55.th	28,45	470,16	990,34
0fd91e92.th	14,31	319,68	639,94

El modelo Original de DEMUCS con el entrenamiento **'0fd91e92.th'** es más eficiente, con un tiempo de ejecución casi la mitad del tiempo del modelo **'5d2d6c55.th'**. En el *Anexo II: Uso de GPU en Evaluación* se establece una comparación entre el modelo

original y el modelo de baja latencia con un delay de 100 ms. Es decir, la mayor complejidad computacional de los modelos de baja latencia se explica por la relación entre el tamaño del segmento de audio procesado en cada llamada y la latencia. Con un tamaño de segmento de 1 segundo, el modelo modificado para baja latencia (100 ms) implica que cada 100 ms de señal recibida desde el programa o aplicación que genera audio, éste es capturado por el propio programa y se llama al modelo con 100 ms nuevos y 900 ms que estaban almacenados en el buffer. Por tanto, procesar un segmento de 1 segundo en el modelo modificado para baja latencia (100 ms) requiere 10 llamadas al modelo, aumentando el tiempo de procesamiento mínimo a 10 veces el del modelo original sin restricciones de retardo. Esta necesidad de realizar cálculos en menos tiempo y la imposibilidad de paralelizar completamente todas las operaciones contribuyen al overhead adicional. En cambio, el modelo original presenta un menor tiempo de ejecución y un menor coste computacional debido a que se procesa 1 segundo de audio completo en una sola llamada al modelo. Esto significa que cada llamada al modelo de Demucs original toma 1 segundo de audio desde la entrada, lo procesa y devuelve las fuentes separadas. Esto supone que la latencia en el modelo original es muy elevada (1 segundo), ya que se espera a tener 1 segundo completo de audio antes de procesarlo. Aunque el coste computacional es menor porque se hace una única llamada al modelo para procesar 1 segundo de audio, lo que conlleva a menos operaciones en tareas de cómputo.

8. CONCLUSIONES Y LÍNEAS FUTURAS

8.1 CONCLUSIONES

En primera instancia, en este TFG titulado “Vocal Boost: software para PC para ajustar el volumen de los diálogos en tiempo real usando inteligencia artificial” se ha abordado con éxito el problema de la inteligibilidad de los diálogos en medios audiovisuales, desarrollando una herramienta que permite ajustar el volumen de los diálogos y el fondo (‘background’) en tiempo real. A continuación, se resumen las conclusiones más relevantes conseguidas en este TFG.

Para empezar, la adaptación del modelo DEMUCS para la separación de fuentes en tiempo real ha demostrado ser efectiva. La herramienta desarrollada permite una separación precisa de la voz del resto de acompañamiento, logrando una mejora significativa en la inteligibilidad del diálogo. El uso de técnicas avanzadas de procesamiento de señales, como la STFT y el método de solapamiento y suma, ha permitido mejorar la calidad de la separación y reducir la latencia. Aunque el entrenamiento proporcionado a la red neuronal ha sido limitado debido a los recursos de hardware en términos de GPU que dispone el ordenador utilizado, se realiza una separación de buena calidad entre ambas fuentes.

Cabe destacar que, siguiendo con lo mencionado con anterioridad, una de las mayores contribuciones de este TFG ha sido la optimización del modelo DEMUCS para operar con baja latencia en una aplicación en tiempo real. Lográndose mediante técnicas de buffering en tiempo real y la optimización del uso de recursos computacionales. El software desarrollado es capaz de procesar el audio en tiempo real, manteniendo una sincronización adecuada entre el audio y el video, lo que es esencial para el usuario que implemente la aplicación.

Por un lado, la GUI desarrollada en Python como aplicación de escritorio tiene limitaciones en cuanto al uso y la instalación debido a que el usuario debe tener el sistema operativo adecuado, hardware específico, instalar paquetes y dependencias que deriva en el uso de la misma para un usuario potencial. Sin embargo, en términos de front-end es intuitiva y fácil de usar incluyendo una guía para el usuario en la pantalla principal. Esta permite ajustar a los usuarios el retardo en función de las capacidades del hardware (GPU), el volumen de la voz y el fondo de una manera sencilla y eficiente. Aunque la interfaz de usuario está completa con los objetivos desarrollados en este TFG, se ha identificado la necesidad de mejorar funcionalidad y usabilidad. Además, sería beneficioso proporcionar

a un usuario potencial información sobre el estado del procesado en tiempo real para mejorar la experiencia del mismo.

Por otro lado, se han utilizado métricas de calidad objetiva como SDR, SAR y SIR para evaluar la efectividad del modelo en comparación con el modelo original de DEMUCS. En particular, se ha observado que el modelo original de DEMUCS ofrece una mejor calidad de separación en términos generales, mientras que la versión modificada presenta una ligera disminución en el rendimiento debido a la optimización para la baja latencia, lo que supone un empeoramiento en calidad de separación de fuentes, pero permitiendo un buen procesamiento en tiempo real. Aunque, el software desarrollado ha demostrado ser capaz de separar las fuentes de audio eficientemente, manteniendo a la salida una cierta calidad. Es importante conocer que el procesamiento en tiempo real de audio con alta calidad de separación requiere un uso intensivo de recursos computacionales, esto puede limitar la implementación del software en dispositivos con capacidades de hardware más limitadas.

Además, este trabajo realizado tiene un impacto significativo en la accesibilidad, especialmente para personas con dificultades auditivas en entornos ruidosos. Permite una mejor comprensión del contenido audiovisual. La capacidad de ajustar dinámicamente el volúmen del diálogo puede beneficiar a una amplia audiencia, incluyendo personas mayores, personas con pérdida auditiva y hablantes no nativos. El proyecto ha alcanzado sus objetivos principales, sin embargo, existen áreas que requieren mejoras adicionales. La interfaz de usuario, aunque funcional, necesita ser más optimizada en términos de presentación y facilidad de uso. La calidad de separación del modelo de DEMUCS es ligeramente inferior al original debido a las optimizaciones necesarias para reducir la latencia. Esto resalta la importancia de seguir investigando para mejorar la eficiencia sin comprometer la calidad. El procesamiento en tiempo real requiere un uso intensivo de recursos computacionales, lo que puede limitar la implementación en dispositivos con capacidades de hardware más limitadas como un teléfono móvil. Es esencial continuar trabajando y optimizando el uso de la GPU y CPU para asegurar un rendimiento adecuado en una amplia gama de dispositivos.

8.2 LÍNEAS FUTURAS

Para continuar con el desarrollo y mejora de este TFG, se proponen las siguientes líneas de investigación y desarrollo:

- **Optimización de la interfaz de usuario.** Mejorar la funcionalidad y usabilidad de la interfaz gráfica de usuario para hacerla más intuitiva y

atractiva. Incluir indicadores visuales y notificaciones que proporcionen información en tiempo real sobre el estado del procesamiento de audio.

- **Despliegue en contenedores Docker.** Desarrollar una versión del software que pueda ser ejecutada en contenedores Docker. Esto facilitará la implementación y escalabilidad del software, permitiendo su uso en diversos entornos y sistemas operativos sin necesidad de configuraciones complejas.

- **Despliegue en servidores web.** Adaptar el software para su ejecución en servidores web, permitiendo su acceso y uso a través de aplicaciones web. Esto ampliará significativamente el alcance y la accesibilidad del software, permitiendo su integración en plataformas de streaming y otras aplicaciones en línea.

- **Mejorar en la precisión del modelo.** Continuar con la investigación y el desarrollo de técnicas que mejoren la precisión y eficiencia del modelo de separación de fuentes. Explorar nuevas arquitecturas de redes neuronales y optimizar los parámetros del modelo para lograr una separación aún más precisa, por ejemplo.

- **Integración con plataformas de Streaming.** Explorar la posibilidad de integrar la herramienta con plataformas de streaming populares como Netflix y YouTube. Permitir que los usuarios ajusten la configuración de audio directamente en estas plataformas mejorará la accesibilidad y la experiencia de consumo de medios audiovisuales.

- **Optimización de recursos computacionales.** Continuar optimizando el uso de recursos computacionales para permitir que el software funcione eficientemente en una gama más amplia de dispositivos, incluyendo aquellos con capacidades de procesamiento más limitadas.

- **Ampliación de funcionalidades.** Implementar funciones adicionales como la reducción automática de ruido de fondo, aunque en el presente software hay una cierta reducción, mejoras en la interfaz de usuario y la capacidad de guardar configuraciones personalizadas de audio para diferentes tipos de contenido.

- **Optimizar el nivel de voz de salida de forma automática.** Usando herramientas de evaluación de la inteligibilidad de la voz en la mezcla, como STOI o PESQ, determinar de forma automática los valores de mezcla óptimos entre voz y fondo.

- **Optimizar el delay en función del hardware.** Minimizar el delay hasta que la carga computacional del hardware lo permita, monitorizando la carga del sistema y evitando su saturación.

- **Programación más elegante de algunas partes de la aplicación.** El paso de información entre hilos se hace a través de ficheros, lo que se puede mejorar con envío de paquetes de red a través de sockets. También se puede mejorar que el cambio de delay tenga que reiniciar la aplicación.

9. ANEXOS

9.1 GUÍA DE INSTALACIÓN DEL SOFTWARE 'VOCAL BOOST'

En este anexo se aportará la información necesaria para poder lanzar la aplicación por un usuario potencial, la creación del entorno virtual y todo lo que conlleva la ejecución de la aplicación de escritorio.

9.1.1 Requisitos previos

Para poder realizar el despliegue y correcto funcionamiento de la aplicación 'VocalBoost' es necesario cumplir con una serie de requisitos tanto de hardware como de software, los cuales se detallan a continuación.

9.1.1.1 Requisitos de Hardware

1. **Linux:** Para instalar este programa y la instalación de dependencias se recomienda hacerlo a través del sistema operativo de Linux ya que este programa ha sido diseñado para funcionar en distribuciones de Linux y la instalación en Windows puede derivar en problemas de versiones y dependencias. La versión recomendada es Ubuntu 18.04 o superior.
2. **Procesador (CPU):** Se recomienda procesador de 64 bits con al menos 4 núcleos.
3. **Memoria (RAM):** Mínimo 8 GB, pero se recomienda al menos 16 GB.
4. **Hardware:** GPU NVIDIA compatible con CUDA para poder realizar el procesamiento en tiempo real, aunque también se permite trabajar con la CPU. El software puede funcionar en modo CPU, pero no permitirá el procesamiento en tiempo real debido a la limitación en el rendimiento. Ejemplos de tarjetas gráficas: NVIDIA GeForce GTX 1060, RTX 2060, etc.
5. **Almacenamiento:** Para poder tener instalado este software, el espacio requerido en disco es de al menos 20 GB de espacio libre para la instalación de dependencias y procesamiento de datos.

9.1.1.2 Requisitos de Software

1. **Python:** Para trabajar con este programa se recomienda utilizar la versión **3.9** o superior de Python. Sin embargo, la versión mínima aceptable es la 3.7. Se puede descargar desde el sitio web https://www.python.org/downloads/ o mediante la terminal de Linux: ***sudo apt install -y python3.9*** y posteriormente verificar su instalación mediante: ***python –version***.

2. **Anaconda:** Instalar Anaconda como sistema de distribución de Python para gestión de entornos y paquetes. Se puede descargar desde https://www.anaconda.com/download.

3. **IDE:** Para poder trabajar de una forma más eficiente con este programa se debería instalar un entorno de desarrollo que permita al usuario visualizar la interacción de los componentes, el procesamiento de audio en tiempo real y el lanzamiento interno de comandos desde scripts. Se recomienda para trabajar con Python y el editor PyCharm. Se puede realizar la descarga de la versión: https://www.jetbrains.com/es-es/pycharm/download/other.html. Habría que elegir la edición de comunidad.

9.1.2 Descarga del software 'Vocal Boost'

Para poder tener acceso a todos los scripts y archivos necesarios para la ejecución del software se ha proporcionado un enlace a Google Drive para poder descargar el archivo comprimido. No es la mejor opción, pero tras varios problemas con la subida de la base de datos de MUSDB18-HQ y los modelos previamente entrenados, se ha seleccionado la opción de subir un enlace a drive para la descarga del software:

https://drive.google.com/file/d/1OmLn2bKaQ5ebKuEmI1nvlOhrArdbixeD/view?usp=sharing

9.1.3 Instalación de CUDA

Al instalar Anaconda, es recomendable, si no ha sido previamente instalado, instalar CUDA, debido a que se van a utilizar bibliotecas que aprovechan la aceleración por GPU. Por tanto, aunque posteriormente se pueda instalar con el fichero *environment-cuda.yaml,* se debe previamente tener instalada la última versión para evitar problemas de GPU. Se puede descargar la última versión con el siguiente enlace indicando el sistema operativo, en este caso **Linux,** la arquitectura (**'x86_64'**) y la distribución (**'Ubuntu'**),https://developer.nvidia.com/cuda-downloads. La versión utilizada en este sistema es la que se muestra en la Figura 61.

```
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$ conda list cudatoolkit
# packages in environment at /home/bernardo-almagro-martos/anaconda3/envs/demucs:
#
# Name                    Version                   Build  Channel
cudatoolkit               11.8.0              h4ba93d1_13    conda-forge
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$
```

Figura 61. Versión utilizada de CUDA.

9.1.4 Creación del entorno virtual e instalación de dependencias

Al haber descargado el archivo comprimido con el programa e instalado los requisitos previos, es recomendable utilizar un entorno virtual para evitar conflictos entre las dependencias de diferentes proyectos. El primer paso será instalar las dependencias necesarias a partir del archivo adjunto ***'environment-cuda.yml'***. Este contendrá todas las versiones de paquetes y dependencias instaladas con las correcciones correspondientes para adaptarlo a la ejecución del software. Además, para asegurarse que **'conda'** está disponible en todas las sesiones de terminal, agregar Anaconda al PATH modificando el archivo **'.bashrc' : echo 'export PATH="~/anaconda3/bin:$PATH"' >> ~/.bashrc source ~/.bashrc** . Hay que dirigirse al directorio o carpeta donde se encuentra el archivo llamado '*Vocalboost.zip'*, se ha de descomprimir en la carpeta especificada y se ha de crear el entorno abriendo una terminal de Linux en la misma carpeta y ejecutando el siguiente comando: **conda env create -f environment-cuda.yml**. Al haber ejecutado este comando se habrán instalado los paquetes y dependencias con las diferentes versiones para su correcta ejecución.

En el archivo '*yaml*' todas las dependencias contienen las versiones de instalación requeridas para el correcto funcionamiento. Es recomendable emplear las versiones señaladas en el fichero debido a que un cambio en la versión de cualquier paquete puede provocar un problema de compatibilidad conllevando a una ardua instalación. A continuación, se muestra una captura de pantalla de pantalla con las versiones de dependencias requeridas.

```
name: demucs

channels:
  - pytorch
  - conda-forge

dependencies:
  - python>=3.8,<3.10
  - ffmpeg>=4.2
  - pytorch>=1.8.1
  - torchaudio>=0.8
  - cudatoolkit>=10
  - tqdm>=4.36
  - numpy
  - pip
  - pip:
    - diffq>=0.2
    - dora-search
    - einops
    - hydra-colorlog>=1.1
    - hydra-core>=1.1
    - julius>=0.2.3
```

```
  - lameenc>=1.2
  - openunmix
  - musdb>=0.4.0
  - museval>=0.4.0
  - soundfile
  - submitit
  - treetable>=0.2.3
  - sounddevice
  - librosa
  - mir_eval
```

Cabe destacar que la instalación se ha de realizar con el archivo *'yml',* debido a que el software está previamente diseñado para trabajar con CUDA que es una librería que permite trabajar de forma paralela a la computación de la GPU. Al haber creado el entorno con éxito, y estando en la misma carpeta de instalación del entorno y del archivo ya descomprimido con los ficheros necesarios, se ha de ejecutar en la terminal el siguiente comando para activar el entorno **'*conda activate demucs*'**, ya que el nombre del entorno en el archivo *'yml'* se llama demucs. Si no se produjera la instalación de ciertas dependencias de forma automática, se podría hacer de forma manual a través de la terminal de comandos de Linux, especificando el paquete y la versión. El resultado de haber instalado el archivo *'yml'* y activado el entorno será el siguiente.

```
(base) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$ conda activate demucs
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$
```

Figura 62. Activación del entono de conda.

9.1.5 Pycharm

Para trabajar de una manera más cómoda con el programa y ver cuando entra un script y otro, cómo se modifican los parámetros de volumen de voz y fondo en tiempo real, es decir, líneas de depuración, se recomienda instalar el editor de Pycharm. Dentro de Pycharm se ha de configurar el entorno **'demucs'** para poder lanzar el programa desde la aplicación.

1. Dirigirse al menú principal y seleccionar **File** dentro de file buscamos la opción **Settings**, dentro de Settings se ha de buscar el desplegable **Project** donde se indicará el entorno. Para ello, se selecciona *Python interpreter.*

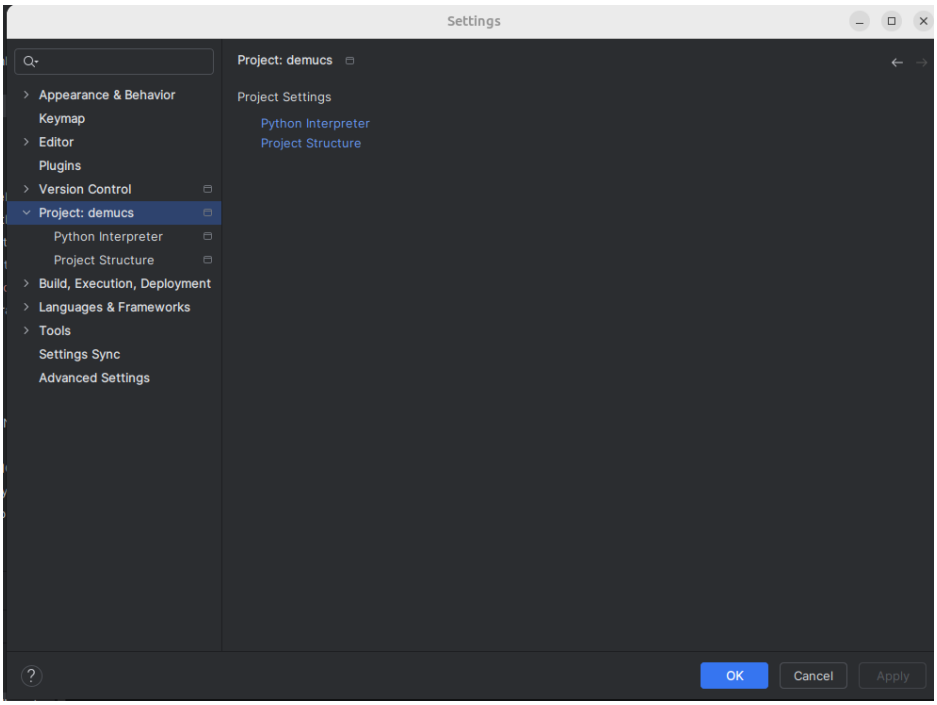


Figura 63. Menú de ajuste de proyectos de Pycharm.

2. Al seleccionar Python Interpreter, en la ventana en la parte derecha se ha de seleccionar *Add Local Interpreter.*

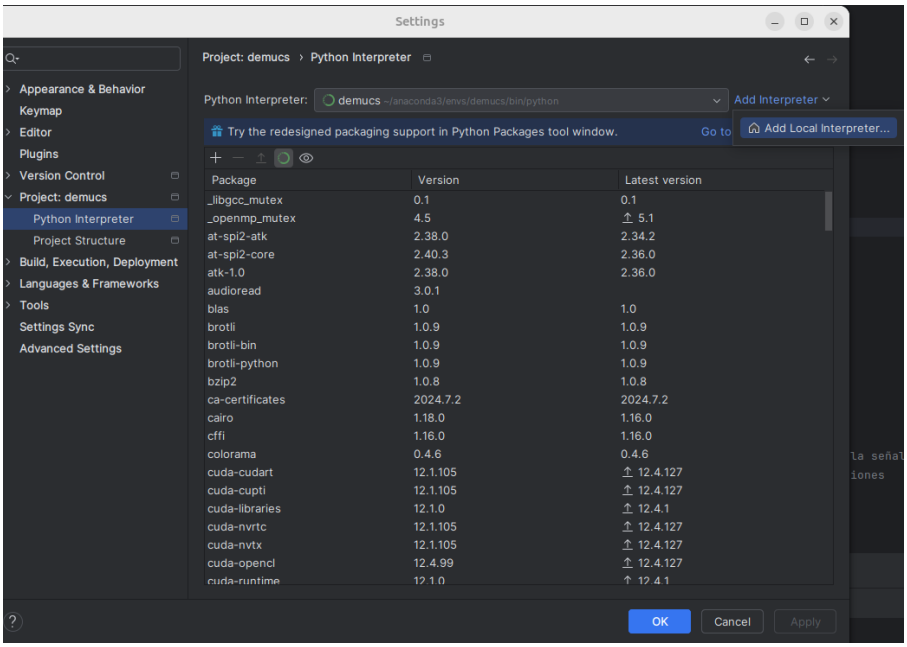


Figura 64. Selección para añadir el intérprete local en Pycharm.

3. Hay que dirigirse a *Conda Environment* donde se seleccionará *Use existing environment*, se ha de buscar en el desplegable hasta encontrar con el entorno creado. Esto permitirá tener todas las dependencias y bibliotecas adicionales a instalar en el entorno creado sin entrar en conflicto con otros proyectos

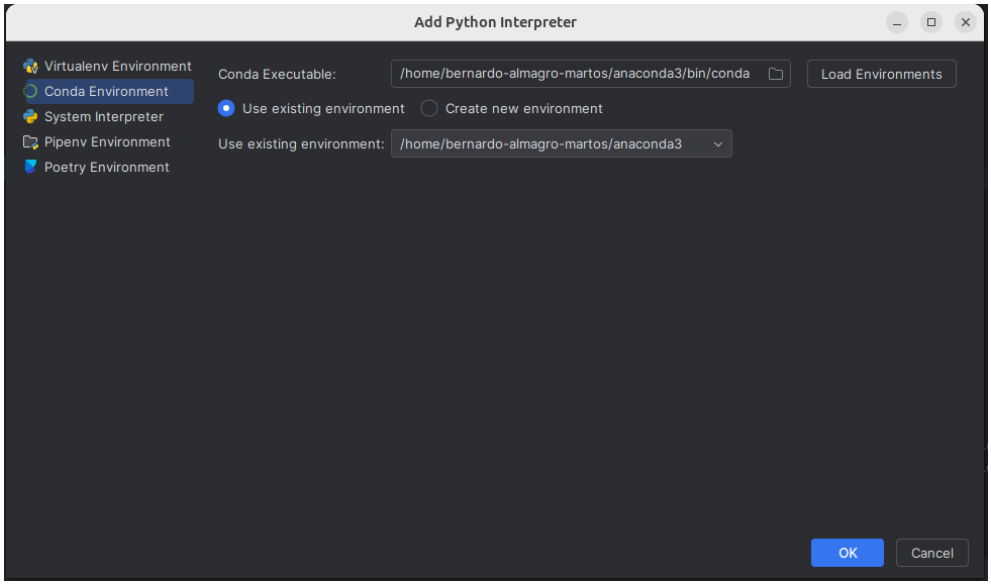


Figura 65. Elección del entorno existente 'demucs' en Pycharm.

4. En la pantalla principal de Pycharm, en la esquina inferior derecha, se puede observar si se ha elegido el entorno correcto para el proyecto.

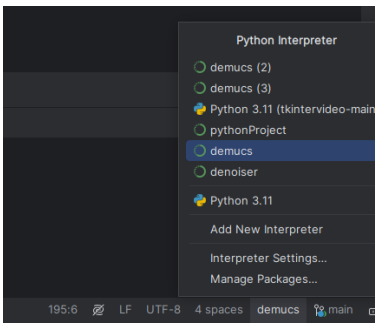


Figura 66. Intérpretes existentes en Pycharm.

9.1.6 Problemas de dependencias y librerías en el entorno

La mayoría de dependencias serán instaladas sin problema, aunque cabe la posibilidad de que haya cierta incompatibilidad de versiones y errores de verificación.

9.1.6.1 Error de encontrar 'conda'

Error al no encontrar el comando 'conda' después de la instalación, el PATH de Anaconda no se añadió correctamente al archivo de configuración de la terminal como ('**.bashrc**') .

conda: no se encontró la orden

Solución: Añadir Anaconda al PATH editando el archivo ('**. bashrc**')

9.1.6.2 Error de verificación de paquetes al intentar crear el entorno

La causa posible es que el Paquete de PyTorch esté corrupto o incompleto.

CondaVerificationError: The package for pytorch located at /home/bernardo-almagro-martos/anaconda3/pkgs/pytorch-2.2.1-py3.9_cuda12.1_cudnn8.9.2_0 appears to be corrupted.

Solución: Eliminar el paquete corrupto y volver a descargarlo

9.1.6.3 Error de clobber (conflicto de rutas compartidas) al instalar paquetes.

La causa se debe a los conflictos entre las rutas de archivos de los paquetes que se están intentando instalar.

ClobberError: This transaction has incompatible packages due to a shared path.

La solución que se puede llevar a cabo como se indica en el subapartado *Creación del entorno virtual e instalación de dependencias* ,es instalar los paquetes individualmente o actualizar conda. Uno de los paquetes que puede causar conflictos es Pytorch debido a que puede no ser compatible con la versión de CUDA y haya que actualizar ambas versiones para ser compatibles.

```
conda update conda
conda install pytorch
```

9.1.6.4 Error de compatibilidad de versiones al intentar instalar 'torchaudio'

Este error se debe a que la versión específica de **'torchaudio'** no es compatible o disponible para la versión de Python instalada en el entorno.

ERROR: Could not find a version that satisfies the requirement torchaudio==0.5.1

La solución puede tener dos alternativas: la primera, buscar una versión compatible de **'torchaudio'** con Python; la segunda es ajustar la versión de Python a esta librería hasta que sean compatibles.

```
conda create --name demucs python=3.9
conda activate demucs
pip install torchaudio==0.5.1
```

9.1.6.5 Error al encontrar la biblioteca de PortAudio al ejecutar 'sounddevice'

La causa que generaba este problema es que la biblioteca 'PortAudio' no esté instalada en el sistema.

OSError: PortAudio library not found

La solución a este problema generado es instalar PortAudio utilizando conda.

conda install -c conda-forge portaudio

9.1.6.6 Error al intentar instalar 'pyaudio' debido a la falta del compilador 'gcc'

El compilador GCC (GNU Compiler Collection) no estaba instalado el sistema. Si se produce el error se debe instalar ya que algunas bibliotecas de Python como '**pyaudio'**, requieren compilación desde el código fuente durante el proceso de instalación.

error: command 'gcc' failed: No such file or directory

GCC es el compilador que convierte ese código fuente en binarios ejecutables ('instrucciones en lenguaje máquina') que pueden ser utilizados por el sistema.

sudo apt-get install gcc

Los problemas anteriormente enunciados y brevemente explicados han surgido a la hora de instalar el entorno y algunas bibliotecas y librerías que sugería la IDE de Pycharm.

9.1.7 Configuración de la contraseña de Superusuario para interfaces de audio virtual

Para poder lanzar los dispositivos de audio virtual en 'Vocal Boost', es necesario modificar el archivo comandos.py y establecer la contraseña de superusuario adecuada. A continuación, se describen los pasos para realizar esta configuración:

1. **Localizar el archivo comandos.py:** El archivo comandos.py se encuentra en el directorio del software VocalBoost.
2. **Modificar la contraseña de Superusuario:** Abrir el archivo *comandos.py* con un editor por ejemplo el descrito con anterioridad, Pycharm. Buscar la siguiente función que se observa en la siguiente captura y reemplaza **'nueva_contraseña'** por la contraseña que corresponda al superusuario del sistema empleado.

```
def init_snd_aloop():
    sudoPassword = 'nueva_contraseña'
    command = 'modprobe snd-aloop'
    os.system('echo %s|sudo -S %s' % (sudoPassword, command))
```

9.1.8 Carga del modelo de separación de fuentes

Para no entrar en conflictos de ajuste de parámetros de script al cargar el modelo de separación de fuentes, simplemente hay que cambiar el repositorio en el script *live.py* por la ubicación de la carpeta donde se encuentre el archivo vocalboost.zip y lo demás se dejaría exactamente igual.

```
repo = "/directorio_carpeta_demucs/demucs/release_models/"

nombre_modelo1 = "0f91e92"  # Entrenado por Pablo Cabañas
nombre_modelo2 = "5d2d6c55"  # Repositorio de GitHub

device = "cuda"

# model = RealTimeModel(args.name, args.repo, device=args.device,
hop=conversor_delay())
model = RealTimeModel(nombre_modelo1, Path(repo), device=device,
hop=conversor_delay(delay))
```

Pudiendo seleccionar el modelo entrenado por Pablo Antonio Cabañas Molero o el modelo extraído del repositorio de DEMUCS.

```
def main():

    global volumen_voz, volumen_fondo
    volumen_voz = float(sys.argv[2])
    volumen_fondo = float(sys.argv[4])
    delay = float(sys.argv[6])

    print(f"Volumen Voz: {volumen_voz}, Volumen Fondo: {volumen_fondo},
Delay: {delay}")

    # Load separation model
    # repo =
"/media/jjean/82DCA32BDCA319D5/Trabajo/GitHubToys/_SOURCE_SEPARATION/demu
cs_live/release_models/"
    repo = "/home/bernardo-almagro-martos/demucs/release_models/"

    nombre_modelo1 = "0f91e92"  # Entrenado por Pablo Cabañas
    nombre_modelo2 = "5d2d6c55"  # Repositorio de GitHub
    device = "cuda"

    model = RealTimeModel(nombre_modelo1, Path(repo), device=device,
hop=conversor_delay(delay))
    nsources = len(model.model.sources)
    nchannels = model.model.audio_channels
    samplerate = model.model.samplerate
```

9.1.9 Instalación de Tkinter

Para que el usuario pueda visualizar la interfaz gráfica será necesario instalar tkinter de forma manual desde la línea de la terminal de Linux dentro del entorno de conda creado ***'sudo apt-get install python3-tk'***.

9.1.10 Instalación de Pavucontrol

Para asegurar que el usuario tiene **'pavucontrol'** instalado, el cuál es la interfaz gráfica para el control y configuración del servidor de sonido de PulseAudio, se ha de ejecutar el siguiente comando ***'sudo apt-get install pavucontrol'*.**

9.1.11 Despliegue de la aplicación 'Vocal Boost'

Al haber ya realizado todos los pasos anteriores en el entorno **'demucs'** y con éxito, se puede lanzar la aplicación de escritorio para que el usuario pueda modificar el volumen de los diálogos en tiempo real. Primero de todo, hay que asegurarse de estar correctamente en el entorno de *demucs* desde la terminal de linux y en el repositorio donde se alojan todos los archivos. Seguidamente, al activar el entorno podemos ejecutar el siguiente comando '***python interfaz_vocalboost.py'***, este permitirá ejecutar el script principal de interacción del usuario para abrir la GUI. Si no se ha producido ningún problema, el resultado de la ejecución de comandos e instalación de dependencias sería:

```
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$ conda deactivate
(base) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$ cd
(base) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$ conda activate demucs
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~$ cd demucs
(demucs) bernardo-almagro-martos@bernardo-almagro-martos-1-0:~/demucs$ python interfaz_vocalboost.py
```

Figura 67. Despliegue de la aplicación 'Vocal Boost'.

Dentro del script *live.py*, se puede modificar el modelo a utilizar para la separación de la voz y el fondo, por defecto, se ha utilizado **'0fd91e92.th'**, el cual funciona algo peor en cuanto a separación de fuentes, pero es más eficiente computacionalmente.

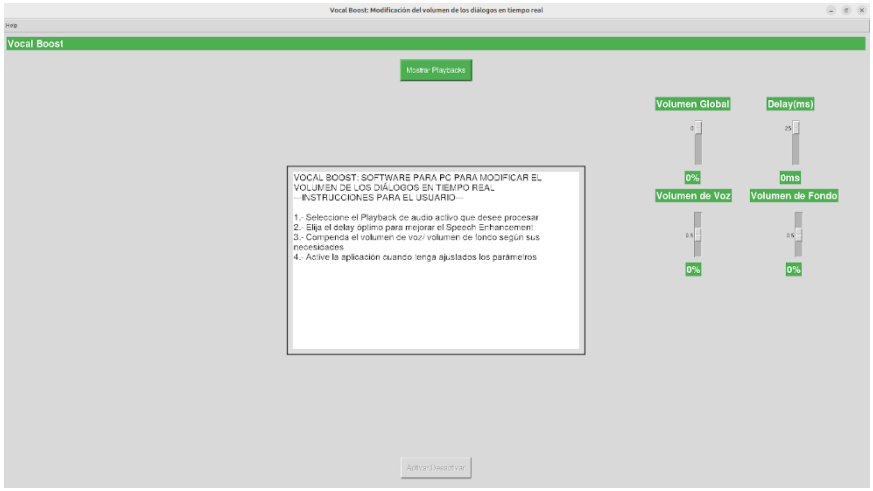


Figura 68. Aplicación de escritorio 'Vocal Boost'.

9.2 PROBLEMAS DE GPU CON ARCHIVOS DE AUDIO GRANDES

Durante el desarrollo y evaluación de los modelos de separación de fuentes musicales se presentaron varios desafíos relacionados con el uso de la GPU. Estos

problemas surgieron principalmente al procesar archivos de audio de gran tamaño. Este anexo detalla los problemas encontrados y las soluciones implementadas para mitigarlos.

1. **Error de memoria insuficiente (CUDA Out of Memory):** Al procesar archivos de gran tamaño inicialmente establecido la longitud máxima de los archivos de audio en 4096000 muestras, el modelo original de DEMUCS arrojaba un error de “CUDA Out of Memory”. Esto se debe a que la cantidad de memoria requerida para procesar el archivo completo excede la capacidad de la GPU disponible.

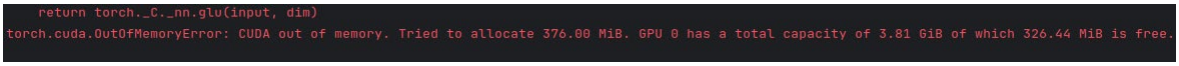


Figura 69. Error de memoria insuficiente en GPU.

Para que el modelo original de DEMUCS pudiera procesar archivos de audio sin error de memoria, se estableció de nuevo una longitud máxima de archivo ‘3072000’ para limitar el número de muestras procesadas. Este valor se determinó empíricamente, probando diferentes tamaños hasta encontrar el máximo que la GPU pudiera manejar sin errores.

9.2.1 Modelo de Demucs modificado

En el modelo de baja latencia, al procesar segmentos pequeños de audio, se han evitado los errores de memoria insuficiente y se han podido manejar archivos de audio grandes de manera eficiente.

9.2.2 Modelo de Demucs Original

Limitar el tamaño del archivo a ‘3072000’ muestras ha permitido que los archivos de audio se procesen sin problemas de memoria en la GPU. Sin embargo, esto puede llevar a una ligera pérdida de información, ya que los archivos de audio no se evalúan en su totalidad, sino sólo una parte de ellos ya que sólo es un proceso de evaluación. Esto puede resultar en la pérdida de efectos sonoros, menor precisión en la separación de fuentes y pérdida de contexto.

9.3 USO DE GPU EN EVALUACIÓN

En el presente anexo se documenta y analiza la utilización de la GPU durante la ejecución de dos modelos de separación de audio utilizando el previo entrenamiento realizado por el Profesor-Tutor Pablo Antonio Cabañas Molero, **’0fd91e92.th**’ el modelo original y el modelo en tiempo real con baja latencia con un retardo de 100 ms. La medición de la utilización de la GPU es crucial para comprender la eficiencia y la carga computacional

impuesta por cada modelo, especialmente en la aplicación 'Vocal Boost' implementada en este trabajo para realizar el procesamiento en tiempo real.

Para llevar a cabo este análisis, se utilizó la herramienta **'nvidia-smi'** junto con un script de Python basado en la biblioteca **'py3nvml'**. Estas herramientas permitieron registrar la utilización de la GPU, la memoria utilizada y el consumo de energía en intervalos de un segundo mientras se ejecutaban ambos modelos. Los datos recopilados se analizaron para obtener valores promedio que representan la carga impuesta por cada modelo en la GPU.

Resultados del modelo original de Demucs:

- **Utilización promedio de la GPU: 30%**
- **Utilización promedio de la memoria de la GPU: 6%**
- **Consumo promedio de energía: 7.5W**

Los valores recogidos del modelo original indican que no impone una carga significativa en la GPU del ordenador empleado, permitiendo un procesamiento más relajado y menos intensivo en términos de recursos.

Resultados del Modelo para baja latencia en tiempo real (100 ms):

- **Utilización promedio de la GPU: 75-81%**
- **Utilización promedio de la memoria de la GPU: 17%**
- **Consumo promedio de energía: 42W**

Estos resultados reflejan la mayor complejidad computacional y la necesidad de realizar operaciones rápidas y continuas para mantener la baja latencia requerida en la aplicación que ha sido desarrollada.

10. REFERENCIAS BIBLIOGRÁFICAS

Alulema, Alexis. 2022. The Code-It List. *Funciones de Activación.* [En línea] 23 de Septiembre de 2022. [Citado el: 29 de Julio de 2024.] https://www.alexisalulema.com/es/2022/09/23/funciones-de-activacion-en-tensorflow/.

Bahoura, M. 2019. *Efficient FPGA-Based Architecture of the Overlap-Add Method for Short-Time Fourier Analysis/Synthesis.* s.l. : Electronics, 2019. 10.3390/electronics8121533.

Barry, D. 2019. *Real-time Sound Source Separation For Music Applications.* Technological University Dublin. Dublín, Irlanda : s.n., 2019.

Calvo, Diego. 2017. Red Neuronal Convolucional. *Diego Calvo.* [En línea] 20 de Julio de 2017. [Citado el: 23 de Junio de 2024.] https://www.diegocalvo.es/red-neuronal-convolucional/.

Cano, Estefania, y otros. 2019. *Musical Source Separation: An Introduction.* 2019. págs. 31-40. 10.1109/MSP.2018.2874719.

Chevalier, Guillaume. 2018. *LARNN: Linear Attention Recurrent Neural Network.* Computer Science and Software Engineering, Laval University. Quebec, Canada : s.n., 2018. Informe técnico.

Dataquest. 2023. Tutorial: Introduction to Deep Learning. [En línea] 31 de Marzo de 2023. [Citado el: 28 de Julio de 2024.] https://www.dataquest.io/blog/tutorial-introduction-to-deep-learning/.

DataScientest. 2024a. Deep Learning o Aprendizaje profundo: ¿qué es? [En línea] 19 de Abril de 2024a. [Citado el: 2024 de Julio de 25.] https://datascientest.com/es/deep-learning-definicion.

—. 2024b. Inteligencia artificial : definición, historia, usos, peligros. [En línea] 2024b. [Citado el: 24 de Julio de 2024.] https://datascientest.com/es/inteligencia-artificial-definicion.

—. 2024c. Machine Learning: definición, funcionamiento, usos. [En línea] 26 de Julio de 2024c. https://datascientest.com/es/machine-learning-definicion-funcionamiento-usos.

DataScientist. 2024d. Perceptrón: ¿qué es y para que sirve? [En línea] 2024d. [Citado el: 2024 de Julio de 27.] https://datascientest.com/es/perceptron-que-es-y-para-que-sirve.

Déffossez, Alexandre. 2021. *Hybrid Spectrogram and Waveform Source Separation.* Proceedings of the MDX Workshop, arXiv. 2021. págs. 1-13, Artículo de congreso.

Défossez, A, y otros. 2021. *Music Source Separation in the Waveform Domain.* 2021. 1911.13254, 2019..

Défossez, A., y otros. 2021. Code for the paper Hybrid Spectrogram and Waveform Source Separation. *GitHub.* [En línea] 2021. [Citado el: 24 de Febrero de 2024.] https://github.com/facebookresearch/demucs.

Donahue, C, y otros. 2023. *SingSong: Generating musical accompaniments from singing.* Google Research. s.l. : arXiv, 2023. 2301.12662v1.

Drgas, Szymon. 2023. *A Survey on Low-Latency DNN-Based Speech Enhancement.* s.l. : Sensors, 2023. 1380.

Fuchs, Harald, Tuff, Simon y Bustad, Christofer. 2012. *Dialogue Enhancements: Technology and Experiments.* Ginebra : European Broadcasting Union (EBU), 2012. 1609-1469.

Gear4Music. 2024. Audiodinamix. [En línea] 2024. [Citado el: 29 de Julio de 2024.] https://www.gear4music.es/es/Grabacion-y-Ordenadores/Audionamix-IDC-Real-Time/5B47.

Heinzel, G, Rüdiguer, A y Schilling, R. 2002. *Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows.* Hannover : Max-Planck-Institut für Gravitationsphysik (Albert-Einstein-Institut), Teilinstitut Hannover, 2002.

Hennequin, R., y otros. 2020. *Spleeter: a fast and efficient music source separation tool with pre-trained models.* s.l. : Journal of Open Source Software, 2020. 5(50), p. 2154.

Hochreiter, Sepp y Schmidhuber, Jürgen. 1997. *LONG SHORT-TERM MEMORY.* Fakultat für Informatik, Technische Universität München. München, Germany : Neuronal Computation, 1997. págs. 1735-1780.

Honglie Chen, Rodrigo Mira, Stavros Petridis,Maja Pantic. 2024. *RT-LA-VocE: Real-Time Low-SNR Audio-Visual Speech Enhancement.* Meta AI. London : arXiv, 2024. 2407.07825v1.

IBM. 2024a. ¿Qué son las redes neuronales convolucionales? [En línea] 2024a. [Citado el: 28 de Julio de 2024.] https://www.ibm.com/es-es/topics/convolutional-neural-networks.

—. 2024b. ¿Qué son las redes neuronales recurrentes? [En línea] 24 de Junio de 2024b. [Citado el: 27 de Julio de 2024.] https://www.ibm.com/es-es/topics/recurrent-neural-networks?mhsrc=ibmsearch_a&mhq=redes%20neuronales%20recurrentes.

Lee, D.D y Seung, H.S. 1999. *Learning the parts of objects by non-negative matrix factorization, Nature.* 1999. págs. 788-791.

Liu, Y, y otros. 2020. *Voice and accompaniment separation in music using self-attention convolutional neural networks.* Ohio State University. Bangalore : Cornell University Library, 2020. 2003.08954v1.

Manilow, E, Seetharaman, P y Salamon, J. 2020. Open-Source Tools & Data for Music Separation. [En línea] 2020. [Citado el: 05 de Julio de 2024.] https://source-separation.github.io/tutorial/landing.html.

Rafii, Z., y otros. 2019. Zenodo. *MUSDB18-HQ - an uncompressed version of MUSDB18.* [En línea] 2019. [Citado el: 31 de Julio de 2024.] https://zenodo.org/record/3338373.

Robert, K. 2019. All About Circuits. *How to Train a Basic Perceptron Neural Network.* [En línea] 24 de Noviembre de 2019. [Citado el: 28 de Julio de 2024.] https://www.allaboutcircuits.com/technical-articles/how-to-train-a-basic-perceptron-neural-network/.

Rodríguez Serrano, Francisco José. 2014. *Separación de fuentes sonoras en señales musicales.* Ingeniería de Telecomunicación, Universidad de Jaén. Jaén : s.n., 2014. Tesis Doctoral. 978-84-8439-878-3.

Shekhar, R. 2023. LinkedIn. [En línea] 31 de Diciembre de 2023. [Citado el: 29 de Julio de 2024.] https://www.linkedin.com/pulse/painting-interactive-worlds-pythons-tkinter-unleash-your-shekhar-8txtc/.

Smaragdis, P y Brown, J.C. 2003. *Non-negative matrix factorization for polyphonic music transcription.* s.l. : IEEE, 2003.

Stack Sánchez, P. 2021. *Métodos de Aprendizaje Supervisado y no Supervisado para la Estimación de Microestructura Cerebral en Datos de DWMR.* Centro de Investigación en Matemáticas A.C. Guanajuato, Gto : s.n., 2021.

Stoller, D, Ewert, S y Dixon, S. 2018. *Wave-U-Net: A Multi-Scale Neural Network for End-to-End Audio Source Separation.* Paris : 19th International Society for Music Information Retrieval Conference (ISMIR), 2018.

Stöter, F.-R., y otros. 2019. *OpenUnmix - A Reference Implementation for Music Source Separation.* s.l. : Journal of Open Source Software, 2019. 4(41), p. 1667.

Strauss, M., y otros. 2021. *A Hands-on Comparison of DNNs for Dialog Separation Using Transfer Learning from Music Source Separation.* International Audio Laboratories Erlangen , Fraunhofer Institute for Integrated Circuits IIS. Erlangen, Germany : s.n., 2021. págs. 1-5, Informe técnico.

Suárez, S. 2023. Xataka. [En línea] 14 de Julio de 2023. [Citado el: 31 de Julio de 2024.] https://www.xataka.com/magnet/no-eres-tu-ellos-dialogos-peliculas-cada-vez-se-escuchan-peor-hay-explicacion.

The Neural Perspective. 2018. Recurrent Neural Network (RNN) Part 5: Custom Cells. [En línea] 2018 de Abril de 2018. [Citado el: 26 de Junio de 2024.] https://web.archive.org/web/20180416083110/https://theneuralperspective.com/2016/11/17/recurrent-neural-network-rnn-part-4-custom-cells/.

Torcoli, M., y otros. 2021. *DIALOG+ in Broadcasting: First Field Tests Using Deep-Learning-Based Dialogue Enhancement.* International Audio Laboratories Erlangen, Fraunhofer Institute for Integrated Circuits IIS. Alemania : s.n., 2021. Informe técnico.

University of Hamburg, Department of Informatics. 2023. Signal Processing, Traditional Speech Enhancement. [En línea] 08 de Diciembre de 2023. [Citado el: 05 de Julio de 2024.] https://www.inf.uni-hamburg.de/en/inst/ab/sp/research/intro.html.

Venkatesh, S, y otros. 2024. *Real-Time Low-Latency Music Source Separation Using Hybrid Spectrogram-TasNet.* s.l. : arXiv, 2024. arXiv:2402.17701v1.

Vocal Technologies Ltd. n.d. Vocal. [En línea] n.d. [Citado el: 01 de Julio de 2024.] https://vocal.com/noise-reduction/deep-neural-network-speech-enhancement/.

Wikipedia. n.d. Wikipedia, la enciclopedia libre. [En línea] n.d. [Citado el: 2024 de Julio de 01.] https://es.m.wikipedia.org/wiki/Espectro_de_frecuencias.