Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Monitorización a través de la voz y mediante redes neuronales de pacientes de la enfermedad de Parkinson

Autor: Juan Pablo Gallego Van Megroot
Tutor: Agustín Álvarez Marquina

Madrid, 14 de Enero de 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: Monitorización a través de la voz y mediante redes neuronales de pacientes de la enfermedad de Parkinson

Madrid, 14 de Enero de 2025

Autor: Juan Pablo Gallego Van Megroot

Tutor: Agustín Álvarez Marquina
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

La enfermedad de Parkinson es un trastorno neurodegenerativo crónico que afecta progresivamente las capacidades motoras y, en etapas tempranas, puede manifestarse mediante alteraciones en la voz y el habla.

Este Trabajo de Fin de Grado desarrolla y compara dos enfoques basados en redes neuronales para la detección de la enfermedad de Parkinson a partir de grabaciones de voz en español.

Se implementó una red neuronal convolucional (CNN) construida desde cero con *TensorFlow/Keras*, además de un ajuste fino (*fine-tuning*) de un modelo preentrenado denominado *Audio Spectrogram Transformer (AST)*. En ambos casos se utilizó la librería Optuna para optimizar los hiperparámetros de los modelos. Se utilizaron los conjuntos de datos de PC-GITA y NeuroVoz, además de una combinación de los dos.

Los resultados muestran que la CNN alcanza una exactitud al rededor del 70 % en los tres conjuntos de datos, mientras que el AST logra valores alrededor del 80 %.

Esto corrobora el potencial de los AST y el transfer learning, superando incluso a modelos con arquitecturas *ad hoc* en entornos con datos moderados. También se abordan consideraciones sobre impacto ético, privacidad de datos y limitaciones del estudio. En conclusión, este trabajo evidencia la viabilidad de la voz como biomarcador y la utilidad de modelos de *Deep Learning* para la identificación no invasiva de la enfermedad, facilitando su posible implementación clínica.

Abstract

Parkinson's disease is a chronic neurodegenerative disorder that progressively affects motor abilities and, in its early stages, can manifest through voice and speech alterations.

This Final Thesis Project develops and compares two neural network-based approaches for detecting Parkinson's disease using Spanish voice recordings.

A convolutional neural network (CNN) was implemented from scratch with *TensorFlow/Keras*, along with fine-tuning of a pre-trained model called the *Audio Spectrogram Transformer (AST)*. In both cases, the Optuna library was used to optimize the models' hyperparameters. The PC-GITA and NeuroVoz datasets were utilized, as well as a combination of the two.

The results show that the CNN achieves an accuracy around 70 % across all three datasets, while the AST achieves values around 80 %.

This corroborates the potential of ASTs, even beating CNN models with *ad hoc* architectures in moderate data environments. Ethical impact considerations, data privacy, and study limitations are also discussed. In conclusion, this project demonstrates the feasibility of voice as a biomarker and the utility of *Deep Learning* models for non-invasive disease identification, facilitating potential clinical implementation.

Agradecimientos

Al culminar este Trabajo de Fin de Grado, deseo expresar mi más sincero agradecimiento a todas aquellas personas que, de una forma u otra, han contribuido a su realización y me han acompañado en este importante camino.

En primer lugar, quiero manifestar mi gratitud a todos los profesores que han formado parte de mi educación durante estos años en el Grado en Ingeniería Informática. Sus enseñanzas, dedicación y valiosos consejos han sido fundamentales para mi crecimiento académico y personal. Cada uno de ellos, con su particular estilo y conocimiento, ha dejado una huella imborrable en mi formación, y les estaré siempre agradecido por haberme guiado en el apasionante mundo de la ingeniería.

De manera especial, quiero destacar la inestimable labor de mi tutor, Agustín Álvarez Marquina. Su orientación, paciencia y apoyo constante han sido pilares fundamentales para el desarrollo de este proyecto. Le agradezco profundamente su motivación, sus acertadas sugerencias y el haberme brindado la oportunidad de trabajar en un tema tan fascinante y con tanto potencial como la detección de la enfermedad de Parkinson a través de la voz. Su confianza en mí ha sido un motor invaluable para llevar a buen puerto este trabajo.

Este trabajo no habría sido posible sin la inspiración que me brindó mi querido abuelo. Él padeció la enfermedad de Parkinson y su lucha diaria, su entereza ante la adversidad, fueron un ejemplo de vida que me marcó profundamente. A su memoria dedico este proyecto, con la esperanza de que los avances en este campo puedan contribuir a mejorar la calidad de vida de quienes sufren esta enfermedad y la de sus seres queridos.

Por último, pero no por ello menos importante, quiero expresar mi más profundo agradecimiento a mi familia. Su amor incondicional, su apoyo constante y sus palabras de aliento han sido mi refugio y mi fortaleza en los momentos de dificultad. A ellos les debo no solo la oportunidad de haber llegado hasta aquí, sino también la motivación para seguir adelante y superarme cada día. Gracias por creer en mí, incluso cuando yo mismo dudaba, y por ser mi pilar fundamental en cada etapa de mi vida.

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación

La enfermedad de Parkinson es un trastorno neurodegenerativo crónico que afecta al sistema nervioso central, principalmente a las áreas encargadas del control y coordinación del movimiento. Según la Organización Mundial de la Salud (OMS), en 2019, más de 8,5 millones de personas en todo el mundo padecían esta enfermedad, y su prevalencia sigue en aumento debido al envejecimiento de la población. [1]

Uno de los desafíos más significativos en el manejo del Parkinson es su detección temprana. Los síntomas motores, como temblores y rigidez muscular, suelen manifestarse cuando ya ha ocurrido una pérdida sustancial de neuronas dopaminérgicas. Sin embargo, existen síntomas no motores, como alteraciones en la voz y el habla, que pueden presentarse en etapas iniciales y servir como indicadores precoces.[2]

El análisis de las señales de voz ofrece una vía prometedora para la detección temprana y no invasiva del Parkinson. Las tecnologías actuales permiten capturar y procesar grabaciones de voz con relativa facilidad, y mediante técnicas avanzadas de procesamiento de señales y aprendizaje automático, es posible extraer patrones característicos asociados con la enfermedad.[3]

En este contexto, las redes neuronales y el aprendizaje profundo han demostrado un gran potencial en tareas de clasificación y reconocimiento de patrones complejos. Su capacidad para aprender representaciones de alto nivel a partir de datos brutos las convierte en herramientas ideales para abordar el desafío de detectar el Parkinson mediante el análisis de la voz.[4]

Además de su relevancia científica y médica, este trabajo tiene una motivación personal profunda para mí. Mi abuelo padeció la enfermedad de Parkinson, y durante sus últimos años de vida fui testigo de los efectos devastadores que esta enfermedad puede tener sobre una persona y su entorno. Ver cómo sus habilidades motoras se deterioraban y cómo, poco a poco, tareas cotidianas se volvían imposibles de realizar fue una experiencia profundamente impactante y dolorosa. La enfermedad no solo afecta al paciente, sino también a su familia y seres queridos, que enfrentan el

sufrimiento y las limitaciones impuestas por el Parkinson.

Esta experiencia personal despertó en mí el interés por investigar más a fondo esta enfermedad y explorar posibles caminos para mejorar la vida de quienes la padecen. Elegí este trabajo con la esperanza de contribuir, aunque sea en una pequeña medida, al desarrollo de herramientas y tecnologías que puedan facilitar una detección más temprana del Parkinson y, en última instancia, ofrecer opciones que ayuden a retrasar su avance.

1.2. Objetivos generales y específicos

El objetivo general de este trabajo es desarrollar un sistema basado en redes neuronales que permita detectar la enfermedad de Parkinson a partir de grabaciones de voz, empleando características acústicas extraídas de dichas grabaciones. Para alcanzar este objetivo, se plantean los siguientes objetivos específicos:

- **Revisión bibliográfica**: Realizar una búsqueda exhaustiva sobre el estado del arte en la detección de Parkinson mediante análisis de voz, con especial énfasis en trabajos que empleen aprendizaje profundo y redes neuronales. Identificar las mejores prácticas y metodologías utilizadas en estudios similares.
- **Preparación del conjunto de datos**: Conseguir un conjunto de datos adecuado de grabaciones de voz de pacientes con Parkinson y sujetos de control. Limpiar y preprocesar los datos para asegurar su calidad, eliminando ruido y normalizando las señales para su análisis.
- **Diseño e implementación de la arquitectura neuronal**: Desarrollar una red neuronal capaz de clasificar las muestras de voz de pacientes con Parkinson y sujetos de control. Explorar y definir la arquitectura óptima de la red, tomando en cuenta los resultados de la revisión bibliográfica y asegurando que el modelo alcance una exactitud mínima del 70 %, manteniendo un balance adecuado entre la exactitud alcanzada y la pérdida de valor (*value_loss)*. Este valor (70 %) fue elegido teniendo en cuenta la complejidad del problema.
- **Entrenamiento y validación del modelo**: Entrenar y validar el modelo en conjuntos de datos separados, asegurando que no exista solapamiento de pacientes entre los conjuntos de entrenamiento, validación y prueba. Ajustar los hiperparámetros y utilizar técnicas de regularización para optimizar el desempeño del modelo y mejorar su capacidad de generalización.
- **Análisis de resultados**: Evaluar el rendimiento del modelo y comparar estos resultados con los reportados en la literatura y realizar un análisis detallado de los errores, identificando áreas de mejora para futuras líneas de investigación.
- **Documentación y preparación para la defensa**: Documentar cada etapa del proyecto, incluyendo la metodología, los resultados obtenidos y el análisis realizado. Preparar una memoria final que incluya todos los detalles del

proyecto, así como una presentación visual para la defensa del trabajo, asegurando claridad en la comunicación de la metodología y los resultados del sistema desarrollado.

1.3. Metodología empleada

Para alcanzar los objetivos planteados, se desarrolló un sistema basado en el análisis de grabaciones de voz utilizando dos enfoques de aprendizaje profundo: una Red Neuronal Convolucional (CNN) y un Audio Spectrogram Transformer (AST) pre-entrenado. Se emplearon dos conjuntos de datos en español, PC-GITA y NeuroVoz, que contienen grabaciones de pacientes con Parkinson y sujetos de control. Las grabaciones fueron preprocesadas, segmentadas y se les aplicó técnicas de aumentación de datos.

En el primer enfoque, se generaron espectrogramas Mel a partir de los audios, los cuales sirvieron como entrada para la CNN. La arquitectura de la CNN y sus hiperparámetros fueron optimizados utilizando la librería Optuna. En el segundo enfoque, se realizó un ajuste fino (fine-tuning) del modelo AST pre-entrenado, utilizando también Optuna para la optimización de hiperparámetros.

El rendimiento de ambos modelos se evaluó mediante métricas como la exactitud y la matriz de confusión. Se realizó una comparativa entre ambos enfoques para determinar cuál ofrece mejores resultados en la detección de la enfermedad de Parkinson a partir de la voz.

1.4. Estructura del documento

El presente documento se organiza en los siguientes capítulos:

- **Capítulo 1: Resumen**
  Se proporciona una síntesis del trabajo, incluyendo los objetivos, la metodología empleada, los resultados obtenidos y las conclusiones principales.
- **Capítulo 2: Abstract**
  Se proporciona una síntesis del trabajo, incluyendo los objetivos, la metodología empleada, los resultados obtenidos y las conclusiones principales en inglés.
- **Capítulo 3: Introducción**
  Se contextualiza la importancia del problema abordado, se expone la motivación personal y científica detrás del trabajo, y se definen los objetivos generales y específicos. Además, se describe la metodología general empleada y se presenta la estructura del documento.
- **Capítulo 4: Estado del Arte**
  Se realiza una revisión exhaustiva de las investigaciones y desarrollos previos

relacionados con la detección de Parkinson mediante análisis de voz, incluyendo las técnicas y modelos más recientes y relevantes.

- **Capítulo 5: Metodología**
  Se detallan los procedimientos y técnicas empleadas en el desarrollo del trabajo, incluyendo la descripción del conjunto de datos, los métodos de extracción y selección de características, el diseño de la arquitectura neuronal y los procesos de entrenamiento y validación del modelo.

- **Capítulo 6: Resultados**
  Se presentan los resultados obtenidos tras el entrenamiento y evaluación del modelo, acompañados de gráficos, tablas y análisis que faciliten su interpretación.

- **Capítulo 7: Conclusiones**
  Se discuten los logros y aportaciones del trabajo, las limitaciones encontradas y se proponen líneas futuras de investigación que podrían continuar y mejorar el trabajo realizado.

- **Capítulo 8: Análisis de Impacto**
  Se reflexiona sobre el impacto social y ético del trabajo, las consideraciones sobre privacidad y seguridad de datos, y cómo el proyecto contribuye a los Objetivos de Desarrollo Sostenible.

- **Capítulo 9: Bibliografía**
  Se listan todas las fuentes bibliográficas consultadas y citadas a lo largo del documento, siguiendo las normas de citación académica establecidas.

- **Capítulo 10: Anexos**
  Se incluyen materiales complementarios como el código fuente completo, configuraciones adicionales, pruebas complementarias y manuales técnicos, que respaldan y amplían la información presentada en los capítulos principales.

Capítulo 2

Estado del Arte

2.1. Detección de la Enfermedad de Parkinson mediante Análisis de Voz

La enfermedad de Parkinson no cuenta con una prueba diagnóstica definitiva, lo que ha impulsado la investigación en herramientas no invasivas que puedan apoyar el diagnóstico clínico. El uso de la voz como biomarcador ha ganado considerable atención debido a que las alteraciones en el habla son síntomas tempranos y comunes en pacientes con esta enfermedad. En este contexto, diversos modelos de aprendizaje profundo han sido desarrollados para detectar patrones característicos en señales de voz.

Los enfoques existentes para la detección automática de la enfermedad de Parkinson (EP) a partir del habla se dividen principalmente en dos categorías:

1. **Ingeniería de características**: Se extraen manualmente características relevantes de las señales de voz, como coeficientes cepstrales, medidas de perturbación vocal y estadísticas del espectro de frecuencia. Luego, estas características se utilizan como entrada en modelos de clasificación como Perceptrones Multicapa (MLP), Redes Neuronales Convolucionales (CNN) o Redes Neuronales de Memoria a Largo Plazo (LSTM).

2. **Enfoques de extremo a extremo (end-to-end)**: Estos modelos aprenden representaciones directamente de los datos brutos, sin una etapa explícita de extracción de características. Se entrenan para identificar patrones en espectrogramas o incluso en las formas de onda originales de las señales de voz.

La efectividad de estos modelos varía según el tipo de datos utilizados (vocales sostenidas, palabras o frases) y las arquitecturas implementadas. En la Tabla 4.1 se presenta un resumen de estudios representativos en la detección de la enfermedad de Parkinson mediante análisis de voz, incluyendo los conjuntos de datos, tipos de entrada, modelos utilizados y precisiones alcanzadas.

Cuadro 2.1: Comparativa de enfoques en la detección de la EP a partir del habla.

Ref.	**Año**	**Conjunto de Datos**	**Tipo de Datos**	**Modelos DL**	**Exactitud**
[7]	2019	UCI ML Repository (188 EP, 64 Control)	Características vocales	CNN de 9 capas	87 %
[8]	2019	PC-GITA (50 EP, 50 Control)	Espectrogramas	ResNet con Transfer Learning	92 %
[9]	2021	PC-GITA (50 EP, 50 Control)	Señal de voz cruda; Señal de fuente de voz	CNN + MLP	67 %; 69 %
[10]	2021	GYENNO SCIENCE PD Research Center (30 EP, 15 Control)	Características dinámicas del habla	LSTM Bidireccional	77 %
[11]	2021	PC-GITA (50 EP, 50 Control)	Características espectrales	MLP	91 %
[12]	2022	PC-GITA; GYENNO SCIENCE PD Research Center	Mel-espectrogramas	CNN 2D distribuida en el tiempo + CNN 1D	82 %; 85 %
[13]	2022	PC-GITA (50 EP, 50 Control)	Mel-espectrogramas	ResNet-101 + LSTM	98 %
[14]	2023	SAKAR database (18 EP, 20 Control)	BSCC, MSCC, BFCC, MFCC	CNN	90 %
[15]	2024	Base de datos italiana (28 EP, 22 Control)	Espectrogramas	CNN + LSTM	96 %

La mayoría de los estudios emplean transformaciones de las señales de voz al dominio tiempo-frecuencia, como espectrogramas o espectrogramas MEL, y extraen características como los Coeficientes Cepstrales de Frecuencia Mel (MFCC) para alimentar los modelos de Deep Learning (DL). Sin embargo, la efectividad de modelos que utilizan directamente las señales de audio crudas sin preprocesamiento ha sido poco explorada.

2.2. Aplicaciones de Redes Neuronales en Diagnósticos Médicos

El aprendizaje profundo ha revolucionado el campo del diagnóstico médico, permitiendo el análisis y clasificación de datos complejos como imágenes médicas y

señales fisiológicas. Las CNN, por su capacidad para extraer características espaciales y locales, han sido ampliamente utilizadas en el reconocimiento de patrones en imágenes y señales. Las LSTM, por otro lado, son efectivas en el procesamiento de secuencias temporales, como series de tiempo y lenguaje natural.

En el caso de la detección de EP, las CNN han sido aplicadas para analizar espectrogramas de voz, capturando patrones espectrales asociados con la enfermedad. Estudios como el de [8] y el de [13] utilizaron arquitecturas pre-entrenadas mediante transferencia de aprendizaje, alcanzando altas precisiones en la clasificación.

Los modelos híbridos que combinan CNN y LSTM también han mostrado resultados prometedores, aprovechando la capacidad de las CNN para extraer características locales y de las LSTM para modelar dependencias temporales en las señales de voz.

Además, se ha explorado el uso de enfoques multimodales que integran diferentes tipos de datos (por ejemplo, características acústicas y visuales) para mejorar la precisión diagnóstica. Esta integración permite capturar una gama más amplia de patrones y reduce la dependencia de un solo tipo de característica.

2.3. Comparación de Enfoques y Modelos Existentes

La elección del modelo y las técnicas empleadas en la detección de EP mediante voz depende de varios factores:

- **Disponibilidad y calidad de datos**: Modelos de DL profundos requieren grandes cantidades de datos para evitar el sobreajuste. La disponibilidad de bases de datos balanceadas y representativas es crucial.
- **Complejidad y recursos computacionales**: Modelos más complejos pueden capturar patrones más sutiles, pero requieren mayores recursos para su entrenamiento y despliegue.

En general, los modelos de aprendizaje profundo han demostrado un mejor rendimiento en la detección de EP a partir de la voz, especialmente cuando se emplean arquitecturas avanzadas y datos preprocesados adecuadamente.

En este trabajo, se propone un enfoque que combina la extracción de características relevantes mediante técnicas de procesamiento de señales y el uso de redes neuronales para la clasificación. Este enfoque busca aprovechar la capacidad de los modelos de DL para aprender representaciones complejas, al mismo tiempo que se mantiene una estructura sencilla y manejable.

De acuerdo, reescribiré el capítulo de Metodología para que incluya toda la información relevante del primer texto, conservando el contexto y la estructura del TFG completo que me has proporcionado. Aquí está la versión mejorada:

Capítulo 3

Metodología

En este capítulo se describe detalladamente el procedimiento seguido para la obtención y tratamiento de los datos, así como para la construcción y entrenamiento de dos enfoques diferentes en la detección de la enfermedad de Parkinson mediante voz. El primero de ellos consiste en una red neuronal convolucional (CNN) desarrollada en *TensorFlow/Keras* y optimizada mediante la librería Optuna [16]. El segundo enfoque emplea el modelo *Audio Spectrogram Transformer (AST)* [17] preentrenado sobre *AudioSet*, ajustado mediante *fine-tuning* con la biblioteca *Hugging Face Transformers* y optimizado con *Optuna*. Se presenta la organización de los datos, los procesos de preprocesamiento, las arquitecturas y herramientas utilizadas, y se explican los pasos de entrenamiento y validación realizados en cada caso.

3.1. Revisión bibliográfica y estado del arte

- **Investigación exhaustiva:** Se realizó una búsqueda detallada de literatura científica, incluyendo artículos, tesis y trabajos relacionados con la detección de la enfermedad de Parkinson (EP) mediante análisis de voz, el uso de redes neuronales en aplicaciones médicas, y específicamente el uso de redes neuronales convolucionales (CNNs) y Audio Spectrogram Transformers (ASTs) en el análisis de audio.
- **Identificación de mejores prácticas:** Se analizaron las metodologías y técnicas más efectivas empleadas en estudios previos, con especial atención a los métodos de preprocesamiento de datos, extracción de características, arquitecturas de redes neuronales, estrategias de entrenamiento y métricas de evaluación. El objetivo fue adoptar o adaptar las estrategias más prometedoras al presente trabajo, considerando las limitaciones y ventajas de cada enfoque.

3.2. Obtención y preprocesamiento de los datos

3.2.1. Datasets utilizados

Para la realización de este trabajo se emplearon dos conjuntos de datos principales: **PC-GITA** [6] y **NeuroVoz** [5]. El primero fue proporcionado por el tutor del

trabajo, mientras que el segundo fue descargado desde el repositorio de Zenodo[1].

Los dos conjunto de datos empleados en el proyecto incluyen grabaciones de voz de pacientes con Parkinson (*PD*) e individuos sanos (*Control*), abarcando diversas tareas que van desde la pronunciación de vocales sostenidas y repeticiones de sílabas, hasta lecturas y monólogos. Para el desarrollo de este trabajo, se optó por emplear exclusivamente los fragmentos de lectura (*reading / utterances*) y monólogo (*monologue*), ya que muestran mayor diversidad y riqueza lingüística, aportando más información relevante para la clasificación. Se seleccionaron las grabaciones de lectura de texto y habla espontánea por su mayor riqueza fonética y prosódica, que se consideraron más informativas para la detección de la EP que las tareas más simples.

3.2.2. Selección y organización de las grabaciones

Tras un análisis preliminar, se descartaron las secciones correspondientes a vocales sostenidas y repeticiones de sílabas, de modo que se consolidó una estructura de archivos que agrupa, para cada conjunto de datos, los audios de cada participante en dos grandes categorías: `hc` para los individuos sanos y `pd` para los pacientes con Parkinson. A continuación, se organizó la información en carpetas cuyo nombre corresponde al *speaker ID*, facilitando así la separación posterior en entrenamiento y prueba.

En el caso concreto de **PC-GITA**, la distribución final incluyó 57 carpetas en la categoría `hc` y 60 carpetas en la categoría `pd`, una por cada hablante disponible. Para **NeuroVoz**, se aplicó una estructura similar, obteniendo 58 carpetas en la categoría `hc` y 54 carpetas en la categoría `pd`. Cabe destacar que no fue necesario eliminar audios de ninguna clase porque ya se encontraban lo suficiente balanceadas para la tarea en cuestión.

3.2.3. Eliminación de silencios y segmentación

Para homogenizar y limpiar los audios, se procedió a:

- **Eliminar silencios superiores a 0.5s**: Se aplicó un algoritmo de detección de silencio para identificar y eliminar segmentos de silencio superiores a 0.5 segundos al inicio, final y dentro de cada grabación. Esto se hizo para reducir la influencia de pausas no informativas y mejorar la precisión del modelo.
- **Segmentar los audios en fragmentos de 2segundos**: Con el propósito de estandarizar la longitud de las muestras, se fraccionaron las grabaciones de modo que cada archivo resultante tuviese exactamente 2 segundos. En caso de que una grabación tuviera una duración inferior a 2 segundos, se aplicó padding con ceros al final para completar la duración. Si la grabación tenía una duración mayor a 2 segundos se fue dividiendo en ventanas de 2 segundos consecutivas. Se eligió esta duración para proporcionar suficiente información

[1] `https://zenodo.org/records/10777657`

fonética y prosódica sin ser excesivamente larga, lo que podría dificultar el entrenamiento del modelo.

- **Normalización:** Las señales de voz se normalizaron para tener una amplitud máxima de 1, lo que ayuda a mejorar el rendimiento del modelo al evitar problemas de escala en los datos.

3.2.4. Aumentación de datos (Data Augmentation)

Adicionalmente, se incluyeron técnicas de *data augmentation* con el objetivo de incrementar la robustez de los modelos ante variaciones en la señal de voz. Entre las transformaciones más destacadas se encuentran la adición de ruido blanco, desplazamientos temporales de la onda (*time-shifting*), cambios de velocidad, modificaciones del tono (*pitch shifting*) y efectos de reverberación. Estas operaciones se aplicaron de manera aleatoria a una fracción de los fragmentos de audio durante la etapa de entrenamiento, tanto en el primer enfoque (CNN), como en el segundo (AST). En concreto se aplicaron las siguientes técnicas:

- **Adición de ruido blanco:** Se añadió ruido blanco gaussiano a las señales de voz con diferentes relaciones señal-ruido (SNR) para simular diferentes condiciones de grabación.
- **Desplazamiento temporal (Time Shifting):** Se desplazó aleatoriamente la señal de voz en el tiempo hacia adelante o hacia atrás, dentro de un rango predefinido, para simular variaciones en la velocidad del habla.
- **Cambio de velocidad (Time Stretching):** Se modificó la velocidad de reproducción de las señales de voz en un factor aleatorio, dentro de un rango predefinido, para simular diferentes ritmos de habla.
- **Cambio de tono (Pitch Shifting):** Se modificó el tono de las señales de voz en un factor aleatorio, dentro de un rango predefinido, para simular variaciones en la entonación.
- **Reverberación:** Se aplicó un efecto de reverberación a las señales de voz para simular diferentes ambientes acústicos.

3.3. Generación y tratamiento de representaciones espectrales

En la primera aproximación, basada en redes convolucionales, se hizo uso explícito de espectrogramas para alimentar al modelo. Concretamente, se generaron **espectrogramas Mel** con 64 componentes en el eje de frecuencia ($n\_mels = 64$), empleando una ventana de análisis de corta duración (ventana de Hanning de 2048 muestras con un solapamiento del 50 %). Estos espectrogramas se convirtieron luego a escala logarítmica, obteniendo así **log-Mel spectrograms**, para obtener una representación más compacta y perceptualmente relevante de la señal de voz.

El proceso de generación se realizó mediante librerías como *Librosa*, normalizando cada espectrograma de modo que presente media cero y desviación estándar unitaria. Posteriormente, cada espectrograma se almacenó en una matriz de dimensiones (64,T) para su posterior procesamiento por la CNN.

Por otra parte, en la segunda aproximación (AST), no fue necesaria la producción manual de espectrogramas, ya que el *feature extractor* incluido en el modelo, proporcionado por la biblioteca Hugging Face Transformers, se encarga de convertir las formas de onda de entrada en la representación espectral específica para la arquitectura *Transformer*, que incluye el cálculo de espectrogramas y su normalización. En este caso, el preprocesamiento se redujo a la normalización del audio a 16kHz, algo necesario para garantizar la compatibilidad con el modelo preentrenado, y a la misma segmentación en fragmentos de 2segundos que en el otro proceso. Las señales de voz se remuestrearon a 16 kHz antes de ser procesadas por el Feature Extractor del AST, ya que esta es la frecuencia de muestreo requerida por el modelo preentrenado.

3.4. Extracción de características e ingeniería de atributos

En las fases iniciales de experimentación, se consideró la posibilidad de entrenar modelos directamente sobre atributos estadísticos y temporales del audio (por ejemplo, medidas de energía, entropía, skewness, entre otras). No obstante, los resultados obtenidos se situaron alrededor del 70 % de precisión, lo que resultaba menos prometedor en comparación con la clasificación basada en **log-Mel spectrograms**, que superó el 75 % en los primeros ensayos. Además de esto, la comparación de resultados vista en 2.1 evidencia este hecho.

Por ende, se decidió centrar los esfuerzos en la representación espectral y prescindir de características manuales, con miras a aprovechar la capacidad de aprendizaje automático de las redes neuronales profundas.

3.5. Diseño de los modelos

3.5.1. Primer modelo: CNN

Arquitectura y librerías utilizadas

El primer modelo se desarrolló en *Python* mediante la librería *TensorFlow/Keras* para el diseño de la CNN, encargada de procesar los **log-Mel spectrograms** descritos. La arquitectura se definió de forma paramétrica para permitir la optimización de hiperparámetros con Optuna. Los hiperparámetros (número de capas, filtros, funciones de activación, tasas de aprendizaje, etc.) se sometieron a un proceso de optimización mediante la biblioteca de Optuna con el objetivo de maximizar la precisión con una pérdida de validación *(validation loss)* controlada (<0.8). La red consta

de varias capas convolucionales seguidas de capas de pooling, capas de dropout y una capa densa final con una función de activación sigmoide para la clasificación binaria (Parkinson vs. Control).

Procesamiento de espectrogramas y normalización

Cada espectrograma, de dimensión (64,T), se expandió en un canal adicional, resultando en un tensor (64,T,1). Para el entrenamiento, se agruparon los ejemplos en lotes (*batches*), y a cada lote se le aplicaron técnicas de aumento de datos, si así se requería, antes de introducirlos a la red.

Optimización de hiperparámetros

El proceso de optimización de hiperparámetros tuvo una primera fase manual, con la que pude experimentar con distintos valores para tener una idea inicial de cómo reaccionaba el modelo con los cambios realizados. Tras esto, implementé la biblioteca Optuna para automatizar este proceso.

Para cada ensayo (*trial*), se entrenó la CNN un número acotado de épocas y se registró la exactitud en un conjunto de validación. El mejor conjunto de parámetros se seleccionó tras múltiples iteraciones, obteniendo precisiones cercanas al 50 % en los primeros prototipos y mejorando progresivamente hasta alcanzar un 82.73 % en experimentos individuales. Se utilizó Optuna para optimizar los hiperparámetros de la CNN, incluyendo el número de capas convolucionales, el número de filtros por capa, el tamaño del kernel, la función de activación, la tasa de dropout, el optimizador (Adam, RMSProp, SGD), la tasa de aprendizaje y el tamaño del lote (batch size).

Ambiente de ejecución y hardware

La ejecución de los modelos CNN con *Optuna* se llevó a cabo en un entorno *Docker* con *TensorFlow* habilitado para *GPU*. Se utilizaron recursos de cómputo avanzados para acelerar la búsqueda de hiperparámetros; sin embargo, la exigencia de memoria de este enfoque resultó moderada en comparación con el método basado en *AST*.

3.5.2. Segundo modelo: Ajuste fino (*fine-tuning*) de Audio Spectrogram Transformer (AST)

Arquitectura y modelo base

El segundo enfoque parte del modelo **Audio Spectrogram Transformer (AST)**, pre-entrenado en el conjunto *AudioSet*. Este modelo, disponible a través de la librería *Hugging Face Transformers*, utiliza una estructura similar a un *Vision Transformer (ViT)* [18] pero aplicada a espectrogramas de audio. Para adaptar el modelo a la clasificación de *Parkinson vs. Control*, se remplazó la capa de salida original y se

ajustaron los pesos internos mediante *fine-tuning*. Se reemplazó la capa de clasificación original del modelo por una nueva capa densa con una función de activación sigmoide para la clasificación binaria.

Procesamiento de la onda sonora

En vez de proporcionar manualmente los espectrogramas, se empleó *AutoFeatureExtractor*, que se encarga de convertir las formas de onda (remuestreadas a 16 kHz) en la representación de entrada que el AST necesita. De este modo, la arquitectura aprende a extraer y refinar atributos útiles para la tarea de detección del Parkinson.

Entrenamiento y validación cruzada (k-fold)

Para evaluar la robustez del modelo, se aplicó validación cruzada con k=5, dividiendo los participantes (*speakers*) en cinco pliegues, de manera que cada pliegue sirvió secuencialmente como conjunto de prueba, mientras que los restantes se destinaron al entrenamiento. Los hablantes se dividieron en 5 grupos, y en cada iteración, un grupo se utilizó como conjunto de prueba y los 4 grupos restantes como conjunto de entrenamiento. Esto permitió obtener una estimación más robusta del rendimiento del modelo. En cada pliegue:

- Se entrenó el AST durante varias épocas (hasta un máximo de 50).
- Se monitoreó la exactitud en validación y se realizaron guardados (*checkpoints*) del mejor modelo.
- Se calculó la exactitud, pérdida de valor y matriz de confusión para estudiar los posibles errores de clasificación.

Coste computacional y entorno de ejecución

En este segundo método, el costo computacional fue considerablemente más elevado. Se requirió de una **GPU RTX 4090 con 24 GB de VRAM** para entrenar eficientemente el AST, aprovechando su elevada capacidad de procesamiento y memoria para manejar el gran número de parámetros del modelo. Además, se empleó también un contenedor *Docker* específico para *PyTorch* y librerías de *Hugging Face*, facilitando la instalación de dependencias y asegurando la reproducibilidad del entorno.

Con dicha infraestructura, se llevaron a cabo sesiones de entrenamiento prolongadas (aproximadamente 24 horas continuas) en las que se exploraron distintas configuraciones y técnicas de regularización. Durante estas ejecuciones, se lograron precisiones individuales de hasta 0.89, tanto para la clase *Healthy Control* como para *Parkinson's Disease*, lo que representa un avance significativo frente a la aproximación inicial basada en la CNN.

3.6. Entrenamiento y validación del modelo

- **División de datos:** Los conjuntos de datos se dividieron en dos subconjuntos: entrenamiento y prueba a un ratio de 5 a 1 (80 % de los datos en el subconjunto de entrenamiento y 20 % de los datos en el conjunto de prueba), y seleccionados aleatoriamente. La división se realizó por hablante (speaker-independent), es decir, los hablantes en el conjunto de entrenamiento eran diferentes de los hablantes en el conjunto de prueba. Esto se hizo para evaluar la capacidad de generalización del modelo a nuevos hablantes no vistos durante el entrenamiento.
  - **Conjunto de entrenamiento:** Se utilizó para entrenar los modelos CNN y AST.
  - **Conjunto de prueba:** Se utilizó para evaluar el rendimiento final de los modelos entrenados.
- **Entrenamiento del modelo:**
  - **CNN:** El modelo CNN se entrenó utilizando el conjunto de entrenamiento y los log-Mel spectrograms como entrada. Se utilizó el optimizador Adam y la función de pérdida de entropía cruzada binaria. El rendimiento del modelo se monitorizó en el conjunto de prueba durante el entrenamiento para evitar el sobreajuste.
  - **AST:** El modelo AST se entrenó mediante fine-tuning utilizando el conjunto de entrenamiento y las señales de voz remuestreadas a 16 kHz como entrada. Se utilizó el optimizador AdamW y la función de pérdida de entropía cruzada binaria. El rendimiento del modelo se monitorizó en el conjunto de prueba durante el entrenamiento para evitar el sobreajuste. Se aplicó early stopping para detener el entrenamiento cuando el rendimiento en el conjunto de prueba dejaba de mejorar.
- **Evaluación del modelo:** El rendimiento de los modelos se evaluó utilizando las siguientes métricas:
  - **Exactitud (Accuracy):** Porcentaje de clasificaciones correctas.
  - **Pérdida de valor (Value Loss):** Promedio de la diferencia absoluta entre las predicciones del modelo y los valores reales.
  - **Matriz de confusión:** Tabla que muestra el número de verdaderos positivos (TP), verdaderos negativos (TN), falsos positivos (FP) y falsos negativos (FN) para cada clase.

3.7. Optuna: Optimización de Hiperparámetros

Optuna es una biblioteca de optimización de hiperparámetros de código abierto diseñada para automatizar y optimizar el proceso de búsqueda de los mejores parámetros para modelos de aprendizaje automático. Su principal objetivo es facilitar la experimentación eficiente mediante técnicas avanzadas como la optimización

bayesiana, lo que permite explorar de manera inteligente el espacio de hiperparámetros y encontrar configuraciones que maximicen el rendimiento del modelo con un número reducido de ensayos.

En el presente trabajo, Optuna se empleó para optimizar los hiperparámetros de ambos modelos. Los hiperparámetros sujetos a optimización en el primer modelo incluyeron:

- **Número de capas convolucionales**: Se determinan entre 1 y 5 capas para capturar diferentes niveles de abstracción en las características del espectrograma.
- **Número de filtros por capa**: Se crean entre 16 y 256 filtros por capa, con incrementos de 16, para ajustar la capacidad de aprendizaje del modelo.
- **Tamaño del kernel**: Se elige entre tamaños de *kernel* de (3,3), (5,5) y (7,7) para cada capa convolucional.
- **Función de activación**: Se seleccionan la funciones de activación entre `relu`, `elu` o `tanh` que introducen no linealidad en el modelo.
- **Tasa de *dropout***: Se establecen tasas de *dropout* entre 0.0 y 0.7 para regularizar el modelo y prevenir el sobreajuste.
- **Optimización**: Se determina el optimizador entre *Adam*, *RMSProp* y *SGD*, además de ajustar la tasa de aprendizaje *(learning rate)* correspondiente, con valores entre 1e-5 y 1e-2.
- **Tamaño de lote**: Se elige un tamaño de lote *(batch size)* entre los valores 8, 16 y 32. Estos valores fueron seleccionados tras un previo proceso de optimización manual.

Además de optimizar los hiperparámetros de la CNN, Optuna se utilizó para ajustar parámetros clave del *Audio Spectrogram Transformer (AST)* durante su proceso de *fine-tuning*. Se utilizó Optuna para optimizar los hiperparámetros del proceso de fine-tuning, incluyendo la tasa de aprendizaje, el tamaño del lote, el número de épocas de entrenamiento, el warmup ratio, el clipping del gradiente, la tasa de dropout, el weight decay y la paciencia para el early stopping. La búsqueda incluyó tanto hiperparámetros relacionados con el optimizador como configuraciones específicas del entrenamiento y del modelo. Los parámetros optimizados fueron los siguientes:

- **Tasa de aprendizaje (*learning_rate*)**: Se exploraron valores en un rango logarítmico entre 10e-6 y 10e-3 para determinar la mejor tasa de aprendizaje para el optimizador *AdamW*.
- **Batch Size (*batch_size*)**: Se probaron tamaños de lote entre 8, 16, 24 y 32, buscando un balance entre la estabilidad del entrenamiento y el uso eficiente de la memoria de GPU.

- **Número de Épocas (*num_epochs*)**: Se varió entre 10 y 64 épocas, en pasos de 10, para encontrar el punto de convergencia óptimo del modelo sin sobreentrenamiento.
- **Warmup Ratio (*warmup_ratio*)**: Se exploraron valores entre 0.0 y 0.3 (en pasos de 0.05), ajustando el número de pasos iniciales del entrenamiento en los que la tasa de aprendizaje crece linealmente.
- **Clipping del Gradiente (*clip_value*)**: Se optimizó entre 0.1 y 2.0 (en pasos de 0.1) para prevenir explosiones de gradientes durante el entrenamiento.
- **Dropout Rate (*dropout_rate*)**: Se varió entre 0.0 y 0.8 (en pasos de 0.1), regulando la tasa de desactivación aleatoria de neuronas para reducir el sobreajuste.
- **Weight Decay (*weight_decay*)**: Se exploraron valores logarítmicos entre 10e-6 y 10e-1 para controlar la magnitud de los pesos durante la regularización del optimizador *AdamW*.
- **Paciencia para el Early Stopping (*patience*)**: Se optimizó entre 3 y 10 épocas consecutivas sin mejora en la precisión de validación, ajustando la sensibilidad del criterio de parada temprana.

El proceso de optimización con Optuna siguió el esquema de validación cruzada (*k-fold*) con k=5. En cada pliegue, se evaluaron múltiples configuraciones de hiperparámetros, seleccionando finalmente aquella que maximizó la precisión global promedio en validación. Este enfoque permitió identificar combinaciones efectivas que llevaron el modelo AST a alcanzar precisiones cercanas al 0.79 en validación cruzada y 0.89 en métricas individuales de *precision*, *recall* y f1-score.

El uso de Optuna para este modelo, aunque computacionalmente costoso, resultó en una configuración ajustada de hiperparámetros que aprovechó al máximo las capacidades del modelo AST, demostrando la utilidad de esta herramienta en la optimización de modelos complejos.

Optuna implementa un *study* que guía la exploración del espacio de hiperparámetros utilizando estrategias eficientes. En este trabajo, se configuró el estudio para maximizar la precisión en el conjunto de validación mediante la evaluación de múltiples configuraciones de hiperparámetros en cada ensayo (*trial*). Además, se empleó almacenamiento persistente con SQLite para permitir la continuidad de la optimización en caso de interrupciones.

La integración de Optuna permitió:

- **Automatización**: Reducir la necesidad de intervención manual en la selección de hiperparámetros, agilizando el proceso de experimentación.
- **Eficiencia**: Utilizar métodos de optimización bayesiana para explorar de manera inteligente el espacio de hiperparámetros, evitando evaluaciones redundantes y enfocándose en regiones prometedoras.

- **Reproducibilidad**: Documentar y registrar automáticamente los resultados de cada ensayo, facilitando el análisis posterior y la comparación de distintas configuraciones. También se utilizaron semillas *(seeds)* a la hora de hacer estos estudios, un parámetro que modifica las operaciones aleatorias y las convierte en operaciones reproducibles en el futuro, siempre y cuando se utilice la misma semilla.

3.8. Análisis de resultados

- **Comparación de modelos:** Se comparó el rendimiento de los modelos CNN y AST en los tres conjuntos de datos (PC-GITA, NeuroVoz y la combinación de ambos) utilizando las métricas de evaluación mencionadas anteriormente.
- **Análisis de errores:** Se analizaron las matrices de confusión para identificar los tipos de errores cometidos por los modelos (falsos positivos y falsos negativos) y para comprender mejor sus fortalezas y debilidades.
- **Interpretación de resultados:** Se interpretaron los resultados en el contexto de la literatura existente sobre la detección de la EP mediante análisis de voz y se discutieron las implicaciones de los hallazgos para el diagnóstico temprano de la enfermedad.
- **Identificación de limitaciones:** Se identificaron las limitaciones del estudio, como el tamaño relativamente pequeño de los conjuntos de datos y la heterogeneidad en las condiciones de grabación.

3.9. Resumen de la metodología

A modo de síntesis, el proceso metodológico se estructuró en:

1. **Obtención y preparación de datos**: Recolección, limpieza de silencios, segmentación a 2 s y aumentación de audios.
2. **Generación de representaciones**: Uso de espectrogramas Mel en la CNN y de un extractor de características interno en el AST.
3. **Construcción de modelos**:
    - *CNN + Optuna*: Diseño de arquitectura convolucional, exploración intensiva de hiperparámetros y entrenamientos iterativos con validación cruzada.
    - *AST fine-tuning + Optuna*: Adaptación de un modelo pre-entrenado con exploración de hiperparámetros, validación cruzada y uso de recursos de cómputo avanzados.
4. **Entrenamiento, validación y comparación**: Medición de métricas de clasificación, matrices de confusión y análisis de resultados. El segundo método

obtuvo globalmente un mejor desempeño, al costo de mayores requerimientos computacionales y tiempo de entrenamiento.

Esta metodología permitió comparar dos aproximaciones claramente diferenciadas: la construcción de un modelo personalizado desde cero y la adaptación de un modelo de *deep learning* con conocimiento previo del dominio del audio. Las observaciones revelan que la robustez de un modelo pre-entrenado, como *AST*, resulta ventajosa frente a arquitecturas diseñadas *ad hoc*, especialmente al abordar una tarea compleja con un volumen de datos moderado.

3.10. Documentación y presentación

- **Registro detallado:** Se documentó de forma minuciosa cada etapa del proceso metodológico, incluyendo las decisiones tomadas, los desafíos enfrentados, las soluciones implementadas, los resultados obtenidos y el análisis realizado.
- **Preparación de la memoria final:** Se redactó una memoria final que describe de forma clara y concisa el trabajo realizado, siguiendo la estructura estándar de un Trabajo de Fin de Grado (TFG). La memoria incluye una introducción, una revisión del estado del arte, una descripción detallada de la metodología, la presentación de los resultados, una discusión de los mismos, las conclusiones y las líneas futuras de investigación.
- **Defensa del trabajo:** Se preparó una presentación oral para la defensa del TFG, que resume los aspectos más relevantes del trabajo y destaca los principales hallazgos y contribuciones. La presentación se apoyó en material visual, como gráficos y tablas, para facilitar la comprensión de los resultados.

3.11. Logros y aportaciones del trabajo

Uno de los logros más significativos de este trabajo ha sido la implementación de dos enfoques distintos para la detección de la enfermedad de Parkinson a partir de señales de voz: un modelo de redes neuronales convolucionales (CNN) desarrollado desde cero y un proceso de ajuste fino de un modelo preentrenado, el *Audio Spectrogram Transformer (AST)*. A continuación se destacan los principales aportes y hallazgos derivados de esta investigación:

- **Diseño y validación de un pipeline reproducible**: Se creó un flujo de trabajo completo que abarca la *adquisición* y *preprocesamiento* de las grabaciones de voz, la *generación de representaciones* espectrales o el uso de extractores internos de características, el *entrenamiento* de modelos de *Deep Learning* y la *evaluación* y *comparación* mediante métricas de clasificación. Este pipeline resulta reproducible y escalable a otros dominios o datasets.
- **Optimización automatizada de hiperparámetros con Optuna**: Tanto para la arquitectura de CNN diseñada *ad hoc* como para el modelo AST preentrenado, se integraron rutinas de optimización con la biblioteca *Optuna*, lo que

permitió *explorar* y *afinar* de forma sistemática parámetros como el número de capas, filtros, tasa de aprendizaje, regularización, y otros. Esta estrategia ha contribuido a maximizar los resultados de cada aproximación.

- **Comparación objetiva de enfoques clásicos vs. preentrenados**: El estudio reveló que, ante datos con menor volumen o mayor complejidad, la *transferencia de conocimiento* desde modelos preentrenados como AST otorga ventajas considerables en términos de exactitud y generalización. Esto se evidenció en un incremento considerable de la precisión a la hora de clasificar sujetos sanos y pacientes de Parkinson, frente a la CNN construida desde cero.
- **Contribución potencial al diagnóstico temprano del Parkinson**: Si bien el presente trabajo no reemplaza la evaluación médica tradicional, demuestra el *potencial* de la voz como biomarcador y respalda la idea de que las redes neuronales profundas pueden extraer patrones relevantes de la señal. Una detección temprana podría ayudar a *optimizar* tratamientos y a *monitorizar* la evolución de la enfermedad.
- **Mejora de la robustez mediante aumentación de datos**: La aplicación de técnicas de *data augmentation* (añadido de ruido, variaciones de tono, cambios de velocidad, etc.) ayudó a *aumentar la generalización* de los modelos y a *reducir* el sobreajuste, lo que sienta un precedente para seguir explorando metodologías de augmentación adaptadas a la voz.

En conjunto, estos resultados refuerzan la idea de que *las señales de voz son una fuente de información valiosa* en el ámbito clínico y que, con técnicas de aprendizaje profundo, es posible acercarse a **clasificaciones de alta exactitud** en la detección de la enfermedad de Parkinson.

3.12. Limitaciones encontradas

Durante el desarrollo del presente trabajo, se identificaron distintas limitaciones que pueden servir de punto de partida para mejoras futuras:

- **Volumen de datos limitado**: Las bases de datos utilizadas (PC-GITA y NeuroVoz) si bien aportan diversidad de hablantes y tareas de habla, aún no alcanzan un *tamaño masivo* que permita entrenar *desde cero* arquitecturas extremadamente profundas sin caer en problemas de sobreajuste.
- **Heterogeneidad en las condiciones de grabación**: Algunas grabaciones presentan diferencias en el entorno acústico, tipo de micrófono y distancia de grabación, lo cual introduce *ruido* o variaciones que pueden afectar la consistencia de los datos. Aunque la aumentación de datos mitiga parcialmente este problema, no lo elimina por completo.
- **Coste computacional elevado para modelos preentrenados**: El modelo AST requirió de una *GPU de alto rendimiento* (RTX 4090) y entrenamiento prolongado para lograr convergencia en validación cruzada. Esto limita la *accesibilidad* a usuarios con infraestructura computacional más modesta.

- **Interpretabilidad limitada**: Aunque se logró una precisión elevada, las redes neuronales profundas pueden comportarse como "cajas negras". No siempre es sencillo interpretar *qué características de la señal de voz* son más relevantes para la clasificación, lo que supone un reto en aplicaciones médicas con alta necesidad de *transparencia* y explicación de resultados.
- **Representatividad de la muestra**: Existen otros factores clínicos y lingüísticos (edad, nivel educativo, dialectos, etc.) que podrían influir en la calidad de las grabaciones y en los patrones vocales. Una **muestra más amplia** y estratificada podría ofrecer conclusiones más sólidas sobre la generalización del modelo.

Pese a estas limitaciones, los resultados obtenidos son prometedores, sirviendo de base para esfuerzos de investigación posteriores.

3.13. Líneas futuras de investigación

Con base en la experiencia adquirida y en los hallazgos de esta investigación, se proponen las siguientes líneas de trabajo que podrían complementar y *extender* la labor realizada:

- **Ampliación de bases de datos**: Recopilar y unificar bases de datos de mayor tamaño y diversidad lingüística (distintos idiomas, variaciones dialectales, rangos de edad, etc.). De esta forma, se podría *entrenar modelos de gran escala* que capturen mejor la variabilidad inter e intra-sujeto.
- **Enfoques multimodales**: Integrar, además de la voz, *otros biomarcadores* como imágenes cerebrales (RM, PET), señales de movimiento o datos de *smartphones* (acelerómetros, giroscopios), de modo que el *diagnóstico* se base en información proveniente de múltiples canales.
- **Interpretabilidad y explicabilidad**: Profundizar en técnicas de *explainable AI* (por ejemplo, Grad-CAM aplicado a espectrogramas [19]) para **identificar** las regiones de la señal de voz que tienen mayor relevancia en la decisión de un modelo. Esto podría incrementar la *aceptación* de estos sistemas en entornos clínicos.
- **Optimizaciones orientadas a entornos con recursos limitados**: Investigar arquitecturas ligeras (por ejemplo, modelos *TinyML [20]* o *Distil Transformers[21]*) que puedan ejecutarse en dispositivos móviles o sistemas embebidos, **facilitando** su aplicación en entornos clínicos con poca disponibilidad de equipos de alta gama.
- **Validación clínica y ensayos reales**: Colaborar con centros médicos para llevar a cabo *evaluaciones* y ensayos clínicos que validen la eficacia de estos modelos en **situaciones reales**, integrándolos como parte de un protocolo de diagnóstico o seguimiento de pacientes con Parkinson.

Capítulo 4

Resultados

En este capítulo se describen y analizan los resultados de la experimentación con los dos enfoques principales: el modelo basado en redes neuronales convolucionales (*CNN*) optimizado con *Optuna*, y el modelo *Audio Spectrogram Transformer (AST)* ajustado (*fine-tuning*), también optimizado con *Optuna*. Ambos modelos han sido probados con el dataset de PC-GITA, el de NeuroVoz, y la combinación de ambos (en adelante, *Combinado*). Se incluyen tablas de métricas y representaciones gráficas (matrices de confusión) que permiten visualizar su desempeño en cada caso.

4.1. Métricas de evaluación

Para cuantificar la efectividad de los modelos en la detección de la enfermedad de Parkinson a partir de grabaciones de voz, se emplearon las siguientes métricas:

- **Exactitud (Accuracy)**: Proporción de predicciones correctas respecto al total de muestras evaluadas.
- **Pérdida de Valor (Value Loss)**: Mide la diferencia entre las predicciones del modelo y los valores reales. Esta métrica evalúa la magnitud del error de las predicciones del modelo, sin hacer distinción entre falsos positivos y falsos negativos. Un valor alto de *Value Loss*, incluso con una buena exactitud, indica que el modelo está generando errores de gran magnitud en sus predicciones y que su rendimiento no es óptimo. Una baja pérdida de valor indica que el modelo es capaz de hacer predicciones cercanas a los valores reales, lo que sugiere un buen ajuste a los datos de entrenamiento y una buena capacidad de generalización.
- **Matriz de Confusión**: Representación tabular que muestra el número de aciertos y errores de clasificación, diferenciando entre *Parkinson* (PD) y *Control* (HC).

4.2. Resultados del modelo basado en Redes Neuronales Convolucionales (CNN) con Optuna

El primer conjunto de experimentos corresponde al modelo **CNN**, cuyos hiperparámetros fueron optimizados mediante la librería *Optuna* a fin de mejorar su desempeño en cada uno de los datasets. En la Tabla 4.1 se indican los valores más relevantes hallados por el proceso de optimización para cada caso.

Cuadro 4.1: Principales hiperparámetros de la CNN encontrados por Optuna en cada dataset

Hiperparámetro	**PC-GITA / NeuroVoz**	**Combinado**
learning_rate	0.00123	0.00490
batch_size	8	16
optimizer	rmsprop	adam
n_conv_layers	3	4
filters_layer_0	224	208
kernel_size_layer_0	[5, 5]	[3, 3]
activation_layer_0	relu	elu
pooling_layer_0	max	max
dropout_rate_conv_0	0.4150	0.5746

En general, *Optuna* recomienda arquitecturas con varias capas convolucionales, combinando funciones de activación `relu` o `elu`, y con uso de técnicas de *dropout* tanto en las capas convolucionales como en las densas para reducir el sobreajuste.

4.2.1. Resultados del modelo con PC-GITA

Tras entrenar y evaluar la CNN en el dataset PC-GITA, se alcanzó una exactitud máxima en un experimento individual del 77.8 % y una pérdida de valor (val_loss) de 0.68. La Figura 4.1 muestra la matriz de confusión de este experimento, con 219 aciertos en la clase *HC* y 167 aciertos en la clase *PD*. En un experimento de validación cruzada con k-fold (5), se alcanzó una exactitud media del 70.7 %, con lo cual se confirma la estabilidad del modelo.

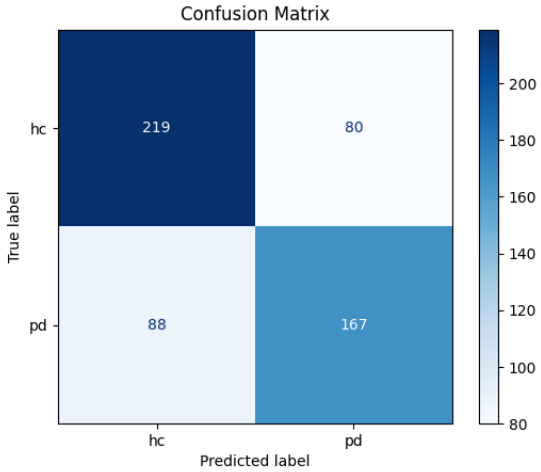

Figura 4.1: Matriz de confusión del mejor modelo CNN en el conjunto de prueba de PC-GITA.

Es importante destacar que la matriz de confusión permite identificar el tipo de errores que el modelo comete. En el contexto de la detección de Parkinson, los falsos negativos podrían tener implicaciones más graves al no identificar a pacientes enfermos, mientras que los falsos positivos podrían requerir pruebas adicionales.

4.2.2. Resultados del modelo con NeuroVoz

El entrenamiento y evaluación en NeuroVoz da lugar a una exactitud del 82.73 % en el mejor experimento individual, con una pérdida de valor de 0.80. Como se puede observar en la Figura 4.2, la CNN confunde a 43 sujetos *HC* y 36 sujetos *PD* de un total de 404. En un experimento de validación cruzada con k-fold (5), se alcanzó una exactitud media de aproximadamente un 78 %.

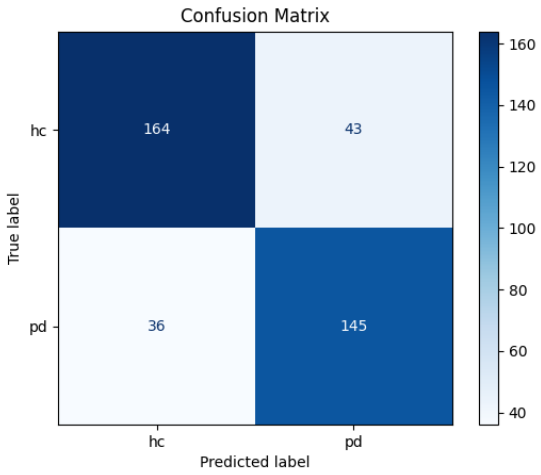

Figura 4.2: Matriz de confusión del mejor modelo CNN en el conjunto de prueba de NeuroVoz.

4.2.3. Resultados del modelo con la combinación de PC-GITA y NeuroVoz

Con el conjunto de datos **combinado**, la CNN conserva un rendimiento relativamente elevado, teniendo un 77.35 % de exactitud y una pérdida de valor de 0.63 en su mejor experimento individual. La matriz de confusión de la Figura 4.3 muestra cómo se obtiene un total de 398 aciertos en clase *HC* y 320 en *PD*, sobre 923 instancias en total. En un experimento de validación cruzada con k-fold (5), la exactitud media fue del 69 %, resultado se queda por debajo de nuestro objectivo de 70 % propuesto inicialmente.

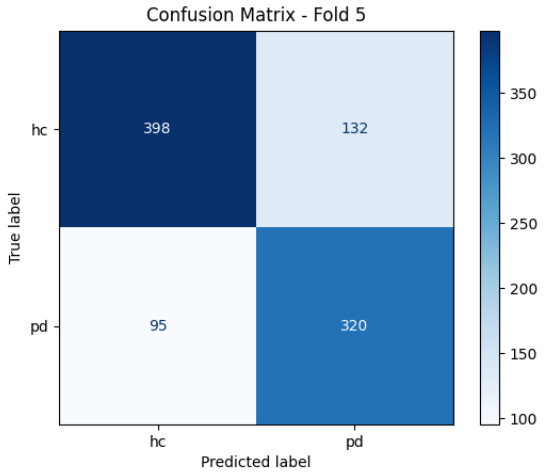

Figura 4.3: Matriz de confusión del mejor modelo CNN en el conjunto de prueba combinado.

4.3. Resultados del modelo Audio Spectrogram Transformer (AST) con ajuste fino y Optuna

El segundo grupo de experimentos se llevó a cabo con un **Audio Spectrogram Transformer (AST)** preentrenado en *AudioSet*, al que se aplicó *fine-tuning* en cada uno de los datasets disponibles. Al igual que en la CNN, se empleó *Optuna* para el ajuste de hiperparámetros, ajustando *learning rates*, *batch sizes*, regularización, etc. En estos experimentos, todos los resultados son de kfold (5).

4.3.1. Resultados del modelo con PC-GITA

La Figura 4.4 presenta la matriz de confusión de uno de los mejores modelos AST. Se observa un total de 215 aciertos en *HC* y 221 en *PD*, con 44 y 38 errores respectivamente. Esto implica que el modelo obtuvo 436 aciertos de 518 muestras, rondando el 84 % de exactitud.

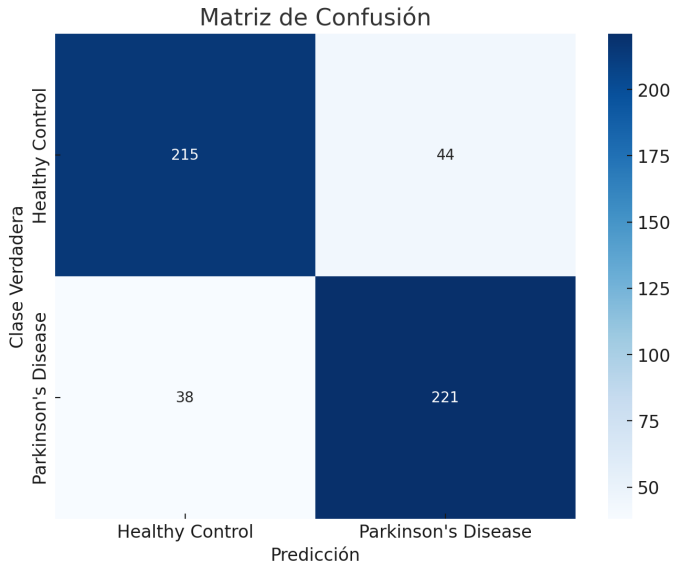

Figura 4.4: Matriz de confusión del modelo AST en el conjunto de prueba de PC-GITA.

Podemos apreciar como el modelo AST es capaz de obtener mejores resultados que el modelo CNN en este experimento, teniendo casi 6 puntos porcentuales más en exactitud.

4.3.2. Resultados del modelo con NeuroVoz

La Figura 4.5 ilustra la matriz de confusión obtenida por el AST en NeuroVoz. Se logran 243 aciertos en la clase *HC* y 243 en la clase *PD*, con 50 y 61 errores, respectivamente, sobre un total de 597 muestras. Esto corresponde a un **81 %** de exactitud, inferior a la marca del 82.73 % conseguida con la CNN.

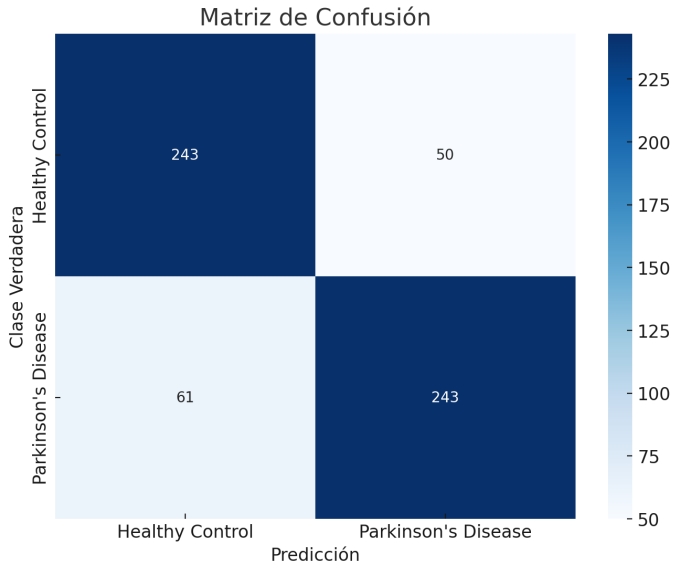

Figura 4.5: Matriz de confusión del modelo AST en el conjunto de prueba de NeuroVoz.

4.3.3. Resultados del modelo con la combinación de PC-GITA y NeuroVoz

Finalmente, en el conjunto **combinado**, el AST alcanza cerca de un **81 %** de exactitud, como se ve en la Figura 4.6.

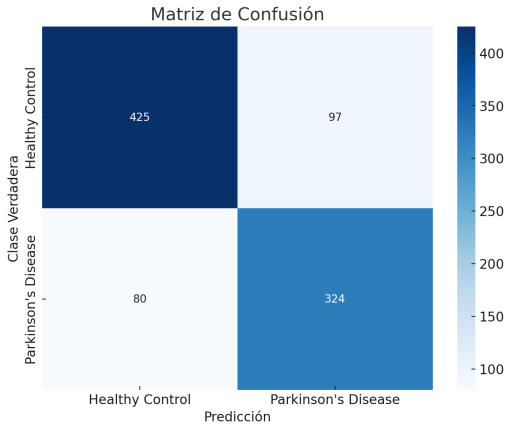

Figura 4.6: Matriz de confusión del modelo AST en el conjunto de prueba combinado.

El rendimiento del AST en el dataset combinado confirma que este modelo tiene una mayor capacidad de generalización que el modelo de CNN, obteniendo significados significativamente superiores, aunque con un coste computacional bastante mayor.

4.4. Comparativa de resultados

La Tabla 4.2 resume la exactitud de cada modelo en los tres conjuntos de evaluación.

Cuadro 4.2: Comparativa general entre la CNN y el AST en los conjuntos PC-GITA, NeuroVoz y Combinado

Dataset	**CNN (Accuracy)**	**AST (Accuracy)**
PC-GITA	70.7 %	84 %
NeuroVoz	78 %	81 %
Combinado	69 %	81 %

Como puede observarse, el **AST** supera a la **CNN** en todos los experimentos realizados, con diferencias de entre 3 y 13.3 puntos porcentuales en exactitud. La diferencia es sustancial, evidenciando la superioridad del modelo AST contra el modelo CNN.

4.5. Discusión de los resultados

Los hallazgos descritos sugieren que la detección de la enfermedad de Parkinson a través de parámetros acústicos de la voz puede realizarse con relativamente alta fiabilidad usando redes profundas de tipo *AST*. Si bien el *CNN* resulta atractivo por sus menores requerimientos computacionales, en estos experimentos no ha logrado superar la eficacia del *AST*.

Entre los factores que pueden estar influyendo en este rendimiento se destacan:

- **Volumen y diversidad de datos**: Aunque los datasets PC-GITA y NeuroVoz abarcan distintas poblaciones, el volumen global sigue siendo moderado para un *Transformer*, que suele requerir grandes cantidades de datos. Esta teoría se ve reforzada al ver que el resultado del experimento con el conjunto de datos combinado tuvo mejor resultado que los experimentos con los conjuntos de datos por separado.
- **Parámetros y regularización**: La *CNN* optimizada con *Optuna* (véanse parámetros clave en Tabla 4.1) ha mostrado su eficacia al emplear funciones de activación como `relu`/`elu`, *dropout* en capas convolucionales y densas, y *optimizers* bien ajustados (`rmsprop`, `adam`) con *learning rates* relativamente bajos.
- **Coste computacional**: El *AST* requiere mayor capacidad de procesamiento y memoria, lo cual puede dificultar encontrar la configuración óptima de hiperparámetros si no se dispone del hardware adecuado.

En conclusión, el **AST** es lo suficientemente superior como para dejar de lado los beneficios de los menores costes computacionales del **CNN** sobre todo teniendo en

cuenta que en un contexto de tanta sensibilidad como este, los costes computacionales no suelen ser un factor de tanta importancia. Además, el **AST** podría mejorar sus métricas aún más con más datos y un ajuste fino más exhaustivo, siendo una línea de investigación abierta para futuros trabajos.

Un aspecto importante a destacar es que en 4 de los 6 experimentos, la mayoría de los errores de clasificación corresponden a casos en los que el modelo predice que la muestra pertenece a la clase *Parkinson* (PD) cuando en realidad corresponde a un sujeto *Control* (HC). Este tipo de error, aunque influye en la exactitud del modelo, es preferible en este contexto clínico, ya que podría llevar a una evaluación más exhaustiva de aquellos individuos con un mayor riesgo aparente, minimizando la posibilidad de pasar por alto a un paciente con la enfermedad.

Capítulo 5

Análisis de Impacto

5.1. Impacto social y ético

La detección temprana de la enfermedad de Parkinson a través del análisis de la voz tiene un potencial transformador en la calidad de vida de los pacientes y en la eficacia de los tratamientos. Al permitir un seguimiento más continuo y asequible, se favorece la personalización de la terapia y se mejora el pronóstico. Sin embargo, esta tecnología conlleva consideraciones éticas: existe el riesgo de generar ansiedad si se comete un fallo de falso positivo y potencialmente retrasar un diagnóstico en caso de falso negativo. Además, la automatización de estas pruebas requiere precaución para evitar sesgos o juicios erróneos que puedan derivar en diagnósticos equivocados. Desde esta perspectiva, es fundamental establecer protocolos claros de responsabilidad y transparencia, particularmente en el uso de algoritmos de aprendizaje profundo, que, en ocasiones, se perciben como "cajas negras".

5.2. Consideraciones sobre privacidad y seguridad de datos

Las grabaciones de voz y los metadatos de pacientes constituyen un tipo de información especialmente sensible, pues permiten la identificación personal y contienen rasgos biométricos únicos. Para asegurar el cumplimiento de la normativa de protección de datos (p. ej., el Reglamento General de Protección de Datos —RGPD— en Europa), resulta indispensable:

- **Anonimizar o seudonimizar** las grabaciones antes de su procesamiento y almacenamiento.
- **Solicitar consentimiento informado** para la captura y el uso de la voz con fines de investigación o diagnóstico.
- **Aplicar protocolos de seguridad** que garanticen la confidencialidad e integridad de la información.
- **Establecer medidas de retención y supresión de datos** claras para prevenir el uso indebido o el almacenamiento innecesariamente prolongado.

Asimismo, resulta esencial la correcta formación del personal sanitario o investigador en ciberseguridad y en el manejo responsable de datos clínicos, evitando vulneraciones que afecten la privacidad de los participantes.

5.3. Contribución a los Objetivos de Desarrollo Sostenible

Este proyecto se alinea con varios de los Objetivos de Desarrollo Sostenible (ODS) establecidos por las Naciones Unidas, entre los cuales destacan:

- **ODS 3: Salud y Bienestar**. El desarrollo de métodos de diagnóstico temprano y no invasivo apoya la meta de garantizar una vida sana y promover el bienestar de todas las personas, contribuyendo a mejorar la atención a pacientes con Parkinson.
- **ODS 9: Industria, Innovación e Infraestructura**. El uso de técnicas de inteligencia artificial e infraestructuras tecnológicas para el procesamiento masivo de datos promueve la innovación médica y científica.
- **ODS 10: Reducción de las Desigualdades**. La implementación de sistemas de bajo costo y accesibles, basados en la voz, puede facilitar la detección y el seguimiento de la enfermedad en regiones con recursos limitados, contribuyendo a reducir la brecha en el acceso a una atención de calidad.

En conjunto, el trabajo realizado demuestra el potencial de la tecnología *deep learning* para repercutir de manera positiva en la sociedad, promoviendo diagnósticos más precisos y accesibles, y fomentando la colaboración entre los sectores tecnológico, sanitario y académico para avanzar hacia sistemas de salud más justos y sostenibles.

Bibliografía

[1] Organización Mundial de la Salud. (2019). *Enfermedad de Parkinson*. World Health Organization.

[2] Picó Berenguer, M., & Yévenes Briones, H. A. (2019). Trastornos del habla en la enfermedad de Parkinson. Revisión. *Revista Científica Ciencia Médica*, 22(1), 36-42. SciELO Bolivia.

[3] Agencia SINC. (2018). *Cómo diagnosticar el párkinson a través del habla*. Agencia Sinc.

[4] Rey Paredes, Marta, Universidad Politécnica de Madrid. (2024). *Voice-based Parkinson's disease detection through neural networks and data augmentation*. Open Access UPM.

[5] Janína Mendes-Laureano, Jorge A. Gómez-García *NeuroVoz: a Castillian Spanish corpus of parkinsonian speech* (2024) [2403.02371].

[6] Orozco, Juan Rafael, Arias-Londoño, Julian D., Vargas-Bonilla, J., González-Rátiva, María, & Noeth, Elmar. (2014). *New Spanish speech corpus database for the analysis of people suffering from Parkinson's disease*.

[7] Hakan Gunduz. *Deep learning-based Parkinson's disease classification using vocal feature sets*. IEEE Access, 7, 115540–115551, 2019.

[8] Marek Wodzinski, Andrzej Skalski, Daria Hemmerling, Juan Rafael Orozco Arroyave, & Elmar Noth. *Deep learning approach to Parkinson's disease detection using voice recordings and convolutional neural network dedicated to image classification*. In 2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), pp. 717–720, 2019.

[9] Nonavinakere Prabhakera Narendra, Björn Schuller, & Paavo Alku. *The detection of Parkinson's disease from speech using voice source information*. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29, 1925–1936, 2021.

[10] Changqin Quan, Kang Ren, & Zhiwei Luo. *A deep learning based method for Parkinson's disease detection using dynamic features of speech*. IEEE Access, 9, 10239–10252, 2021.

[11] Biswajit Karan & Sitanshu Sekhar Sahu. *An improved framework for Parkinson's disease prediction using Variational Mode Decomposition-Hilbert spectrum of speech signal*. Biocybernetics and Biomedical Engineering, 41(2), 717–732, 2021.

[12] Changqin Quan, Kang Ren, Zhiwei Luo, Zhonglue Chen, & Yun Ling. *End to-end deep learning approach for Parkinson's disease detection from speech signals*. Biocybernetics and Biomedical Engineering, 42(2), 556–574, 2022.

[13] Mehmet Bilal Er, Esme Isik, & Ibrahim Isik. *Parkinson's detection based on combined CNN and LSTM using enhanced speech signals with Variational mode decomposition*. Biomedical Signal Processing and Control, 70, 103006, 2021.

[14] Miyara Mounia, Boualoulou Nouhaila, Nsiri Benayad, & Belhoussine Drissi Taoufiq. *Use of ANN, LSTM and CNN classifiers for the new MSCC and BSCC methods in the detection of Parkinson's disease by voice analysis*. International Journal of Advanced Computer Science and Applications, 14(12), 2023.

[15] Vivek Kumar Pandey, Sitanshu Sekhar Sahu, Biswajit Karan, & Sudhan shu Kumar Mishra. *Parkinson disease prediction using CNN-LSTM model from voice signal*. SN Computer Science, 5(4), 381, 2024.

[16] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, & Masanori Koyama. *Optuna: A next-generation hyperparameter optimization framework*. Accepted at KDD 2019 Applied Data Science track, arXiv:1907.10902 [cs.LG], 2019.

[17] Yuan Gong, Yu-An Chung, & James Glass. *AST: Audio Spectrogram Transformer*. Accepted at Interspeech 2021, arXiv:2104.01778 [cs.SD], 2021. https://doi.org/10.48550/arXiv.2104.01778

[18] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, & Neil Houlsby. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. ICLR 2021, arXiv:2010.11929 [cs.CV], 2021. https://doi.org/10.48550/arXiv.2010.11929

[19] Kudryavtsev, Vasiliy, Mkrtchian, Grach, & Gorodnichev, Mikhail. *Capsule-based and TCN-based Approaches for Spoofing Detection in Voice Biometry*. Engineering, Technology and Applied Science Research, 14, 18409-18414, 2024.

[20] Rakhee Kallimani, Krishna Pai, Prasoon Raghuwanshi, Sridhar Iyer, & Onel L. A. López. *TinyML: Challenges and Opportunities for Embedded Machine Learning*. Multimedia Tools and Applications, 2023, arXiv:2303.13569 [cs.LG]. https://doi.org/10.48550/arXiv.2303.13569

[21] Yaning Zhang, Qiufu Li, Zitong Yu, & Linlin Shen. *Distilled Transformers with Locally Enhanced Global Representations for Face Forgery Detection*. Accepted by Pattern Recognition, arXiv:2412.20156 [cs.CV], 2024. https://doi.org/10.48550/arXiv.2412.20156

Anexos

5.4. Código fuente completo

5.4.1. Código de CNN con kfold (5)

```
import os
import gc
import numpy as np
import tensorflow as tf
import librosa
import random
import matplotlib.pyplot as plt
import json
from datetime import datetime
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.utils.class_weight import compute_class_weight
from sklearn.model_selection import StratifiedGroupKFold
import warnings

warnings.filterwarnings('ignore')
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

def configure_gpu():
    physical_devices = tf.config.list_physical_devices('GPU')
    if physical_devices:
        print(physical_devices)
        try:
            for gpu in physical_devices:
                tf.config.experimental.set_memory_growth(gpu, True)
                tf.config.experimental.set_virtual_device_configuration(
                    gpu,
                    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=4096)]
                )
        except RuntimeError as e:
            print(e)

def get_speaker_ids(dataset_path):
    speaker_ids = {'hc': [], 'pd': []}
    for label in ['hc', 'pd']:
        label_dir = os.path.join(dataset_path, label)
        if not os.path.isdir(label_dir):
            continue
        for speaker_id in os.listdir(label_dir):
            speaker_path = os.path.join(label_dir, speaker_id)
            if os.path.isdir(speaker_path):
                speaker_ids[label].append(speaker_id)
    return speaker_ids

def generate_log_mel_spectrogram(audio, sr, n_mels=64):
    spectrogram = librosa.feature.melspectrogram(y=audio, sr=sr, n_mels=n_mels)
    log_mel_spectrogram = librosa.power_to_db(spectrogram, ref=np.max)
    return log_mel_spectrogram

def create_folds(speaker_ids, n_splits=5, random_state=42):
    spk_list = []
    spk_labels = []
    for label in ['hc', 'pd']:
        spk_list.extend(speaker_ids[label])
        spk_labels.extend([label]*len(speaker_ids[label]))
    le = LabelEncoder()
    y_encoded = le.fit_transform(spk_labels)
    spk_array = np.array(spk_list)
    gkf = StratifiedGroupKFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    for train_idx, test_idx in gkf.split(spk_array, y_encoded, groups=spk_array):
        train_speakers = {'hc': [], 'pd': []}
        test_speakers = {'hc': [], 'pd': []}
        for idx in train_idx:
            l = le.inverse_transform([y_encoded[idx]])[0]
            train_speakers[l].append(spk_array[idx])
        for idx in test_idx:
            l = le.inverse_transform([y_encoded[idx]])[0]
            test_speakers[l].append(spk_array[idx])
```

```
        yield train_speakers, test_speakers

def load_speaker_data(dataset_path, speaker_id, label, sr=22050, duration=2, n_mels=64):
    X_audio = []
    speaker_path = os.path.join(dataset_path, label, speaker_id)
    if not os.path.exists(speaker_path):
        return None
    wav_files = [f for f in os.listdir(speaker_path) if f.endswith('.wav')]
    if not wav_files:
        return None
    for file_name in wav_files:
        file_path = os.path.join(speaker_path, file_name)
        try:
            audio, _ = librosa.load(file_path, sr=sr)
            desired_length = int(duration * sr)
            if len(audio) < desired_length:
                audio = np.pad(audio, (0, desired_length - len(audio)), mode='constant')
            else:
                audio = audio[:desired_length]
            log_mel_spec = generate_log_mel_spectrogram(audio, sr, n_mels=n_mels)
            log_mel_spec = (log_mel_spec - np.mean(log_mel_spec)) / np.std(log_mel_spec)
            log_mel_spec = np.expand_dims(log_mel_spec, axis=-1)
            X_audio.append(log_mel_spec)
        except:
            continue
    return np.array(X_audio) if X_audio else None

def load_split_data(dataset_path, train_speakers, test_speakers):
    X_train, y_train, X_test, y_test = [], [], [], []
    for label in ['hc', 'pd']:
        for speaker in train_speakers[label]:
            speaker_data = load_speaker_data(dataset_path, speaker, label)
            if speaker_data is not None and len(speaker_data) > 0:
                X_train.extend(speaker_data)
                y_train.extend([label] * len(speaker_data))
    for label in ['hc', 'pd']:
        for speaker in test_speakers[label]:
            speaker_data = load_speaker_data(dataset_path, speaker, label)
            if speaker_data is not None and len(speaker_data) > 0:
                X_test.extend(speaker_data)
                y_test.extend([label] * len(speaker_data))
    if not X_train or not X_test:
        raise ValueError("No data loaded for training or testing set.")
    X_train = np.array(X_train)
    y_train = np.array(y_train)
    X_test = np.array(X_test)
    y_test = np.array(y_test)
    return (X_train, y_train), (X_test, y_test)

def build_audio_model(input_shape_audio, params):
    from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout
    from tensorflow.keras.layers import BatchNormalization, Activation, AveragePooling2D
    input_audio = Input(shape=input_shape_audio)
    x = input_audio
    for i in range(params['n_conv_layers']):
        num_filters = params[f'filters_layer_{i}']
        kernel_size = params[f'kernel_size_layer_{i}']
        activation = params[f'activation_layer_{i}']
        x = Conv2D(num_filters, kernel_size, padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        if params[f'pooling_layer_{i}'] == 'max':
            x = MaxPooling2D(pool_size=(2,2))(x)
        else:
            x = AveragePooling2D(pool_size=(2,2))(x)
        dropout_rate = params[f'dropout_rate_conv_{i}']
        if dropout_rate > 0.0:
            x = Dropout(dropout_rate)(x)
    x = Flatten()(x)
    for i in range(params['n_dense_layers']):
        units = params[f'units_dense_{i}']
        activation = params[f'activation_dense_{i}']
        x = Dense(units)(x)
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        dropout_rate = params[f'dropout_rate_dense_{i}']
        if dropout_rate > 0.0:
            x = Dropout(dropout_rate)(x)
    output = Dense(1, activation='sigmoid')(x)
    model = tf.keras.models.Model(inputs=input_audio, outputs=output)
    return model

def convert_to_native(obj):
    if isinstance(obj, dict):
        return {k: convert_to_native(v) for k, v in obj.items()}
    elif isinstance(obj, list):
        return [convert_to_native(elem) for elem in obj]
    elif isinstance(obj, np.ndarray):
        return obj.tolist()
    elif isinstance(obj, (np.float32, np.float64, np.float_)):
        return float(obj)
    elif isinstance(obj, (np.int32, np.int64, np.int_)):
        return int(obj)
    else:
        return obj
```

```
def main():
    dataset_path = 'Dataset2s'
    outputs_dir = "images"
    os.makedirs(outputs_dir, exist_ok=True)
    log_file = os.path.join(outputs_dir, 'training_log.jsonl')
    seed = 4
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)
    speaker_ids = get_speaker_ids(dataset_path)
    best_params = {
        'learning_rate': 0.0015514911774146848,
        'batch_size': 8,
        'optimizer': 'rmsprop',
        'n_conv_layers': 5,
        'filters_layer_0': 192,
        'kernel_size_layer_0': (7, 7),
        'activation_layer_0': 'elu',
        'pooling_layer_0': 'max',
        'dropout_rate_conv_0': 0.039263549683244386,
        'filters_layer_1': 224,
        'kernel_size_layer_1': (5, 5),
        'activation_layer_1': 'tanh',
        'pooling_layer_1': 'avg',
        'dropout_rate_conv_1': 0.5676059006777794,
        'filters_layer_2': 128,
        'kernel_size_layer_2': (5, 5),
        'activation_layer_2': 'relu',
        'pooling_layer_2': 'avg',
        'dropout_rate_conv_2': 0.07470380113457614,
        'filters_layer_3': 16,
        'kernel_size_layer_3': (7, 7),
        'activation_layer_3': 'relu',
        'pooling_layer_3': 'max',
        'dropout_rate_conv_3': 0.44485527356852145,
        'filters_layer_4': 208,
        'kernel_size_layer_4': (5, 5),
        'activation_layer_4': 'relu',
        'pooling_layer_4': 'avg',
        'dropout_rate_conv_4': 0.45644680622770967,
        'n_dense_layers': 2,
        'units_dense_0': 128,
        'activation_dense_0': 'elu',
        'dropout_rate_dense_0': 0.12744450091311246,
        'units_dense_1': 480,
        'activation_dense_1': 'elu',
        'dropout_rate_dense_1': 0.04480013509625047
    }
    fold_results = []
    fold_index = 0
    for train_speakers, test_speakers in create_folds(speaker_ids, n_splits=5, random_state=seed):
        fold_index += 1
        (X_train, y_train), (X_test, y_test) = load_split_data(dataset_path, train_speakers, test_speakers)
        le = LabelEncoder()
        y_train_encoded = le.fit_transform(y_train)
        y_test_encoded = le.transform(y_test)
        model = build_audio_model(X_train.shape[1:], best_params)
        if best_params['optimizer'] == 'rmsprop':
            optimizer = tf.keras.optimizers.RMSprop(learning_rate=best_params['learning_rate'])
        elif best_params['optimizer'] == 'adam':
            optimizer = tf.keras.optimizers.Adam(learning_rate=best_params['learning_rate'])
        else:
            optimizer = tf.keras.optimizers.SGD(learning_rate=best_params['learning_rate'])
        model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
        class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(y_train_encoded), y=y_train_encoded)
        class_weight_dict = dict(enumerate(class_weights))
        reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=50, min_lr=1e-6, verbose=1)
        early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=50, restore_best_weights=True, verbose=1)
        checkpoint = tf.keras.callbacks.ModelCheckpoint(os.path.join(outputs_dir, f'best_model_fold{fold_index}.h5'), monitor='val_loss', save_best_only=True, ve
        epochs = 200
        history = model.fit(
            X_train, y_train_encoded,
            validation_data=(X_test, y_test_encoded),
            epochs=epochs,
            batch_size=best_params['batch_size'],
            class_weight=class_weight_dict,
            callbacks=[reduce_lr, early_stopping, checkpoint],
            verbose=1
        )
        test_loss, test_accuracy = model.evaluate(X_test, y_test_encoded, verbose=0)
        y_pred_probs = model.predict(X_test, verbose=0)
        y_pred_classes = (y_pred_probs > 0.5).astype(int).flatten()
        cm = confusion_matrix(y_test_encoded, y_pred_classes)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
        disp.plot(cmap=plt.cm.Blues, values_format='d')
        plt.title(f'Confusion Matrix - Fold {fold_index}')
        cm_path = os.path.join(outputs_dir, f'confusion_matrix_fold{fold_index}.png')
        plt.savefig(cm_path)
        plt.close()
        plt.figure(figsize=(10, 4))
        plt.subplot(1, 2, 1)
        plt.plot(history.history['loss'], label='Training Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss')
        plt.title('Model Loss')
```

```
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()
        plt.subplot(1, 2, 2)
        plt.plot(history.history['accuracy'], label='Training Accuracy')
        plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
        plt.title('Model Accuracy')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.legend()
        plt.tight_layout()
        history_path = os.path.join(outputs_dir, f'training_history_fold{fold_index}.png')
        plt.savefig(history_path)
        plt.close()
        run_timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_data = {
            'fold': fold_index,
            'timestamp': run_timestamp,
            'chosen_hyperparameters': best_params,
            'evaluation': {
                'test_loss': float(test_loss),
                'test_accuracy': float(test_accuracy)
            },
            'confusion_matrix': cm.tolist(),
            'train_speakers': {k: len(v) for k, v in train_speakers.items()},
            'test_speakers': {k: len(v) for k, v in test_speakers.items()},
            'training_history': {
                'loss': convert_to_native(history.history['loss']),
                'val_loss': convert_to_native(history.history['val_loss']),
                'accuracy': convert_to_native(history.history['accuracy']),
                'val_accuracy': convert_to_native(history.history['val_accuracy'])
            }
        }
        log_data_serializable = convert_to_native(log_data)
        with open(log_file, 'a') as f:
            json.dump(log_data_serializable, f)
            f.write('\n')
        fold_results.append((test_loss, test_accuracy))
        del model
        del history
        tf.keras.backend.clear_session()
        gc.collect()
    mean_test_accuracy = np.mean([fr[1] for fr in fold_results])
    print(f'Mean Test Accuracy over 5 folds: {mean_test_accuracy:.4f}')

if __name__ == '__main__':
    configure_gpu()
    main()
```

5.4.2. Código de CNN con Optuna

```
import os
import gc  # Import garbage collector
import numpy as np
import tensorflow as tf
import librosa
import random
import matplotlib.pyplot as plt
import json
from datetime import datetime
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.utils.class_weight import compute_class_weight
import optuna  # Import Optuna
from optuna.trial import FixedTrial
import warnings

# Suppress TensorFlow warnings for cleaner output
warnings.filterwarnings('ignore')
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppress TensorFlow logging

# Configure TensorFlow to use the GPU with dynamic memory growth and limit memory usage
def configure_gpu():
    physical_devices = tf.config.list_physical_devices('GPU')
    if physical_devices:
        print('Using GPU:', physical_devices)
        try:
            for gpu in physical_devices:
                tf.config.experimental.set_memory_growth(gpu, True)
                # Optionally limit GPU memory (e.g., to 4096 MB)
                tf.config.experimental.set_virtual_device_configuration(
                    gpu,
                    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=4096)]
                )
        except RuntimeError as e:
            print(e)
    else:
        print('No GPU available, using CPU.')

configure_gpu()

def get_speaker_ids(dataset_path):
```

```
    """
    Retrieves all speaker IDs categorized by label ('hc' and 'pd').
    """
    speaker_ids = {'hc': [], 'pd': []}
    for label in ['hc', 'pd']:
        label_dir = os.path.join(dataset_path, label)
        if not os.path.isdir(label_dir):
            print(f"Warning: Label directory '{label_dir}' does not exist.")
            continue
        for speaker_id in os.listdir(label_dir):
            speaker_path = os.path.join(label_dir, speaker_id)
            if os.path.isdir(speaker_path):
                speaker_ids[label].append(speaker_id)
    return speaker_ids

def generate_log_mel_spectrogram(audio, sr, n_mels=64):
    """
    Generates a log-mel spectrogram from an audio signal.
    """
    spectrogram = librosa.feature.melspectrogram(y=audio, sr=sr, n_mels=n_mels)
    log_mel_spectrogram = librosa.power_to_db(spectrogram, ref=np.max)
    return log_mel_spectrogram

def load_data(dataset_path, speaker_ids, augment=False, audio_augmenter=None):
    """
    Loads and processes audio data for specified speakers.
    """
    X_audio = []
    y = []

    # Parameters for audio processing
    sr = 22050  # Sample rate
    duration = 2  # Duration in seconds
    n_mels = 64  # Number of Mel bands

    for label in ['hc', 'pd']:
        label_dir = os.path.join(dataset_path, label)
        for speaker_id in speaker_ids[label]:
            speaker_path = os.path.join(label_dir, speaker_id)
            for file_name in os.listdir(speaker_path):
                if file_name.endswith('.wav'):
                    file_path = os.path.join(speaker_path, file_name)
                    try:
                        audio, _ = librosa.load(file_path, sr=sr)
                        # Ensure the audio is of the desired length
                        desired_length = int(duration * sr)
                        if len(audio) < desired_length:
                            audio = np.pad(audio, (0, desired_length - len(audio)), mode='constant')
                        else:
                            audio = audio[:desired_length]

                        # Apply audio augmentation if needed
                        if augment and audio_augmenter:
                            audio = audio_augmenter(audio)

                        # Generate log-mel spectrogram
                        log_mel_spec = generate_log_mel_spectrogram(audio, sr, n_mels=n_mels)
                        # Normalize spectrogram
                        log_mel_spec = (log_mel_spec - np.mean(log_mel_spec)) / np.std(log_mel_spec)
                        # Expand dimensions for channel
                        log_mel_spec = np.expand_dims(log_mel_spec, axis=-1)

                        X_audio.append(log_mel_spec)
                        y.append(label)
                    except Exception as e:
                        print(f"Error processing {file_path}: {e}")

    X_audio = np.array(X_audio).astype(np.float32)
    y = np.array(y)
    return X_audio, y

def build_audio_model(trial, input_shape_audio):
    """
    Builds a CNN model optimized by Optuna that processes audio data.
    """
    from tensorflow.keras.layers import (Input, Conv2D, MaxPooling2D, Flatten, Dense,
                                         Dropout, BatchNormalization, Activation, GlobalAveragePooling2D)

    input_audio = Input(shape=input_shape_audio, name='audio_input')

    x = input_audio

    # Number of convolutional layers (1 to 5)
    n_conv_layers = trial.suggest_int('n_conv_layers', 1, 5)

    for i in range(n_conv_layers):
        num_filters = trial.suggest_int(f'filters_layer_{i}', 16, 256, step=16)
        kernel_size = trial.suggest_categorical(f'kernel_size_layer_{i}', [(3,3), (5,5), (7,7)])
        activation = trial.suggest_categorical(f'activation_layer_{i}', ['relu', 'elu', 'tanh'])
        x = Conv2D(num_filters, kernel_size, padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        if trial.suggest_categorical(f'pooling_layer_{i}', ['max', 'avg']) == 'max':
            x = MaxPooling2D(pool_size=(2,2))(x)
        else:
```

```
            x = tf.keras.layers.AveragePooling2D(pool_size=(2,2))(x)
        dropout_rate = trial.suggest_float(f'dropout_rate_conv_{i}', 0.0, 0.7)
        if dropout_rate > 0.0:
            x = Dropout(dropout_rate)(x)

    x = Flatten()(x)

    # Number of dense layers (1 to 3)
    n_dense_layers = trial.suggest_int('n_dense_layers', 1, 3)
    for i in range(n_dense_layers):
        units = trial.suggest_int(f'units_dense_{i}', 32, 512, step=32)
        activation = trial.suggest_categorical(f'activation_dense_{i}', ['relu', 'elu', 'tanh'])
        x = Dense(units)(x)
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        dropout_rate = trial.suggest_float(f'dropout_rate_dense_{i}', 0.0, 0.7)
        if dropout_rate > 0.0:
            x = Dropout(dropout_rate)(x)

    output = Dense(1, activation='sigmoid', name='output_layer')(x)

    model = tf.keras.models.Model(inputs=input_audio, outputs=output)

    return model

def convert_to_native(obj):
    """
    Recursively converts NumPy data types in the input object to native Python types.
    """
    if isinstance(obj, dict):
        return {k: convert_to_native(v) for k, v in obj.items()}
    elif isinstance(obj, list):
        return [convert_to_native(elem) for elem in obj]
    elif isinstance(obj, np.ndarray):
        return obj.tolist()
    elif isinstance(obj, (np.float32, np.float64, np.float_)):
        return float(obj)
    elif isinstance(obj, (np.int32, np.int64, np.int_)):
        return int(obj)
    else:
        return obj

def get_train_test_speaker_ids(all_speaker_ids):
    """
    Splits the speaker IDs into fixed train and test sets.
    """
    train_speaker_ids = {'hc': [], 'pd': []}
    test_speaker_ids = {'hc': [], 'pd': []}

    for label in ['hc', 'pd']:
        speakers = all_speaker_ids[label].copy()
        random.shuffle(speakers)
        num_speakers = len(speakers)
        num_test = max(1, int(0.2 * num_speakers))  # 20% test speakers
        train_speaker_ids[label] = speakers[num_test:]
        test_speaker_ids[label] = speakers[:num_test]

    return train_speaker_ids, test_speaker_ids

def main():
    dataset_path = 'dataa'  # Update with your dataset path
    desktop_path = os.path.join(os.path.expanduser('~'), 'Desktop')
    outputs_dir = "images"
    os.makedirs(outputs_dir, exist_ok=True)  # Ensure Outputs directory exists
    log_file = os.path.join(outputs_dir, 'training_log.jsonl')

    # Set random seed for reproducibility
    seed = 42
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)

    # Get all speaker IDs
    all_speaker_ids = get_speaker_ids(dataset_path)

    # Fix train/test split
    train_speaker_ids, test_speaker_ids = get_train_test_speaker_ids(all_speaker_ids)

    # Load training data with augmentation
    print("Loading training data...")
    X_train_audio, y_train = load_data(
        dataset_path, train_speaker_ids, augment=False, audio_augmenter=None)
    # Load testing data without augmentation
    print("Loading testing data...")
    X_test_audio, y_test = load_data(
        dataset_path, test_speaker_ids, augment=False)

    # Encode labels
    le = LabelEncoder()
    y_train_encoded = le.fit_transform(y_train)  # 'hc' -> 0, 'pd' -> 1
    y_test_encoded = le.transform(y_test)

    # Define Optuna study with SQLite storage for persistence
    storage_name = 'sqlite:///study.db'  # SQLite database stored in current directory
    study_name = 'hyperparameter_optimization'
```

```
study = optuna.create_study(
    study_name=study_name,
    direction='maximize',
    storage=storage_name,
    load_if_exists=True
)

# Define the log file path
log_file = os.path.join(outputs_dir, 'training_log.jsonl')

# Initialize a mutable object to track the best validation accuracy
best_val_accuracy = {'val_acc': 0.0}

# Define the objective function with logging and memory management
def objective(trial):
    try:
        # Clear any previous TensorFlow graphs
        tf.keras.backend.clear_session()

        # Suggest hyperparameters
        learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-2, log=True)
        batch_size = trial.suggest_categorical('batch_size', [8, 16, 32])
        optimizer_name = trial.suggest_categorical('optimizer', ['adam', 'rmsprop', 'sgd'])

        # Build model with hyperparameters
        input_shape_audio = X_train_audio.shape[1:]
        model = build_audio_model(trial, input_shape_audio)

        # Choose optimizer
        if optimizer_name == 'adam':
            optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
        elif optimizer_name == 'rmsprop':
            optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
        elif optimizer_name == 'sgd':
            optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)

        model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

        # Compute class weights
        class_weights = compute_class_weight(
            class_weight='balanced', classes=np.unique(y_train_encoded), y=y_train_encoded)
        class_weight_dict = dict(enumerate(class_weights))

        # Callbacks
        early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, verbose=0)

        # Training
        epochs = 50  # You can adjust this as needed
        history = model.fit(
            X_train_audio, y_train_encoded,
            validation_data=(X_test_audio, y_test_encoded),
            epochs=epochs,
            batch_size=batch_size,
            class_weight=class_weight_dict,
            callbacks=[early_stopping],
            verbose=0  # Set verbose=0 to reduce output
        )

        # Extract the maximum validation accuracy and corresponding loss
        val_accuracy = history.history.get('val_accuracy', [])
        val_loss = history.history.get('val_loss', [])
        if val_accuracy:
            max_val_acc = max(val_accuracy)
            epoch_max_val_acc = val_accuracy.index(max_val_acc)
            corresponding_val_loss = val_loss[epoch_max_val_acc]
        else:
            max_val_acc = 0.0
            corresponding_val_loss = float('inf')

        # Evaluate the model on test data
        test_loss, test_accuracy = model.evaluate(
            X_test_audio, y_test_encoded, verbose=0)

        # Check if this trial has the best validation accuracy so far
        if max_val_acc > best_val_accuracy['val_acc']:
            best_val_accuracy['val_acc'] = max_val_acc
            print(f"New best validation accuracy: {max_val_acc:.4f} (Trial {trial.number})")

            # Make predictions to generate confusion matrix
            y_pred_probs = model.predict(X_test_audio, verbose=0)
            y_pred_classes = (y_pred_probs > 0.5).astype(int).flatten()
            y_true = y_test_encoded

            # Generate confusion matrix
            cm = confusion_matrix(y_true, y_pred_classes)
            disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
            disp.plot(cmap=plt.cm.Blues, values_format='d')
            plt.title('Confusion Matrix for Best Validation Accuracy')

            # Save confusion matrix to Outputs directory
            confusion_matrix_path = os.path.join(
                outputs_dir, 'confusion_matrix_best_hyperparameters.png')
            try:
                plt.savefig(confusion_matrix_path)
                print(f"Confusion matrix saved to {confusion_matrix_path}")
```

```
            except Exception as e:
                print(f"Failed to save confusion matrix to Outputs directory: {e}")

            # Ensure 'images' directory exists inside Outputs and save additional confusion matrix
            images_dir = os.path.join(outputs_dir, 'images')
            try:
                os.makedirs(images_dir, exist_ok=True)
                additional_confusion_matrix_path = os.path.join(images_dir, 'CNNNeuroVoz.png')
                plt.savefig(additional_confusion_matrix_path)
                print(f"Confusion matrix also saved to {additional_confusion_matrix_path}")
            except Exception as e:
                print(f"Failed to save confusion matrix to images directory: {e}")

            plt.close()

        # Log the trial results
        log_data = {
            'trial_number': trial.number,
            'params': trial.params,
            'max_val_accuracy': float(max_val_acc),
            'val_loss_at_max_accuracy': float(corresponding_val_loss),
            'test_accuracy': float(test_accuracy),
            'test_loss': float(test_loss),
            'timestamp': datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        }

        # Append the log data to the log file
        with open(log_file, 'a') as f:
            json.dump(log_data, f)
            f.write('\n')

        # Clean up to release memory
        del model
        del history
        tf.keras.backend.clear_session()
        gc.collect()

        return max_val_acc  # Since the study is maximizing val_accuracy

    except Exception as e:
        # Handle exceptions gracefully
        print(f"Trial {trial.number} failed with exception: {e}")

        # Clean up to release memory even if trial fails
        tf.keras.backend.clear_session()
        gc.collect()

        # Reraise the exception to let Optuna handle it
        raise e

# Optimize the study
study.optimize(objective, n_trials=2000, timeout=None, n_jobs=1)  # Set n_jobs=1 for sequential execution

# Print best hyperparameters
print('Best hyperparameters:', study.best_params)

# Optionally, retrain the model with the best hyperparameters
print("Retraining the model with the best hyperparameters...")
best_trial = study.best_trial
best_params = best_trial.params

# Build the model with the best hyperparameters using FixedTrial
fixed_trial = FixedTrial(best_params)
input_shape_audio = X_train_audio.shape[1:]
model = build_audio_model(fixed_trial, input_shape_audio)

# Choose optimizer
if best_params['optimizer'] == 'adam':
    optimizer = tf.keras.optimizers.Adam(learning_rate=best_params['learning_rate'])
elif best_params['optimizer'] == 'rmsprop':
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=best_params['learning_rate'])
elif best_params['optimizer'] == 'sgd':
    optimizer = tf.keras.optimizers.SGD(learning_rate=best_params['learning_rate'])

model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

# Compute class weights
class_weights = compute_class_weight(
    class_weight='balanced', classes=np.unique(y_train_encoded), y=y_train_encoded)
class_weight_dict = dict(enumerate(class_weights))

# Callbacks
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss', factor=0.5, patience=10, min_lr=1e-6, verbose=1)
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=15, restore_best_weights=True, verbose=1)
checkpoint = tf.keras.callbacks.ModelCheckpoint(
    os.path.join(outputs_dir, 'best_model.h5'), monitor='val_loss', save_best_only=True, verbose=1)

# Training with the best hyperparameters
epochs = 50
print("Starting model training with the best hyperparameters...")
history = model.fit(
    X_train_audio, y_train_encoded,
    validation_data=(X_test_audio, y_test_encoded),
```

```
        epochs=epochs,
        batch_size=best_params['batch_size'],
        class_weight=class_weight_dict,
        callbacks=[reduce_lr, early_stopping, checkpoint],
        verbose=1
    )

    # Evaluation
    print("Evaluating the model on test data...")
    test_loss, test_accuracy = model.evaluate(
        X_test_audio, y_test_encoded, verbose=0)
    print(f'Test loss: {test_loss:.4f}, Test accuracy: {test_accuracy:.4f}')

    # Predictions
    print("Making predictions on test data...")
    y_pred_probs = model.predict(X_test_audio, verbose=0)
    y_pred_classes = (y_pred_probs > 0.5).astype(int).flatten()
    y_true = y_test_encoded

    # Confusion Matrix
    print("Generating confusion matrix...")
    cm = confusion_matrix(y_true, y_pred_classes)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
    disp.plot(cmap=plt.cm.Blues, values_format='d')
    plt.title('Confusion Matrix with Best Hyperparameters')

    # Save confusion matrix to Outputs directory
    confusion_matrix_path = os.path.join(
        outputs_dir, 'confusion_matrix_best_hyperparameters.png')
    try:
        plt.savefig(confusion_matrix_path)
        print(f"Confusion matrix saved to {confusion_matrix_path}")
    except Exception as e:
        print(f"Failed to save confusion matrix to Outputs directory: {e}")

    # Ensure 'images' directory exists inside Outputs and save additional confusion matrix
    images_dir = os.path.join(outputs_dir, 'images')
    try:
        os.makedirs(images_dir, exist_ok=True)
        additional_confusion_matrix_path = os.path.join(images_dir, 'CNNNeuroVoz.png')
        plt.savefig(additional_confusion_matrix_path)
        print(f"Confusion matrix also saved to {additional_confusion_matrix_path}")
    except Exception as e:
        print(f"Failed to save confusion matrix to images directory: {e}")

    plt.close()

    # Prepare log data
    run_timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_data = {
        'timestamp': run_timestamp,
        'best_hyperparameters': study.best_params,
        'train_speakers': train_speaker_ids,
        'test_speakers': test_speaker_ids,
        'training_history': history.history,
        'final_evaluation': {
            'test_loss': float(test_loss),
            'test_accuracy': float(test_accuracy)
        },
        'confusion_matrix': cm.tolist()
    }

    # Convert log_data to native Python types
    log_data_serializable = convert_to_native(log_data)

    # Log the run details
    run_log_file = os.path.join(outputs_dir, 'final_run_log.json')
    print("Logging run details...")
    try:
        with open(run_log_file, 'w') as f:
            json.dump(log_data_serializable, f, indent=4)
        print(f"Run details logged to {run_log_file}")
    except Exception as e:
        print(f"Failed to log run details: {e}")

    # Clean up after final training
    del model
    del history
    tf.keras.backend.clear_session()
    gc.collect()

if __name__ == '__main__':
    main()
```

5.4.3. Código de AST con kfold (5)

```
import os
import random
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import torchaudio
import librosa
from torch.utils.data import Dataset, DataLoader, Subset, WeightedRandomSampler
from transformers import AutoFeatureExtractor, ASTForAudioClassification, get_linear_schedule_with_warmup
from sklearn.model_selection import GroupKFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay
from collections import Counter
import warnings
import gc
import matplotlib.pyplot as plt
import json
from datetime import datetime
from tqdm import tqdm

warnings.filterwarnings("ignore")
scaler = torch.cuda.amp.GradScaler()

def configure_gpu():
    if torch.cuda.is_available():
        device = torch.device('cuda')
        torch.cuda.empty_cache()
        torch.backends.cudnn.benchmark = True
        torch.backends.cudnn.deterministic = False
    else:
        device = torch.device('cpu')
    return device

device = configure_gpu()

def add_noise(data, noise_factor=0.005):
    noise = np.random.randn(len(data))
    augmented_data = data + noise_factor * noise
    return augmented_data

def shift_time(data, shift_max=0.2):
    shift = int(random.uniform(-shift_max, shift_max) * len(data))
    augmented_data = np.roll(data, shift)
    if shift > 0:
        augmented_data[:shift] = 0
    else:
        augmented_data[shift:] = 0
    return augmented_data

def pitch_shift(data, sr, n_steps=2):
    return librosa.effects.pitch_shift(data, sr=sr, n_steps=n_steps)

def time_stretch(data, rate=1.1):
    return librosa.effects.time_stretch(data, rate=rate)

def speed_tune(data, speed_factor=1.1):
    return librosa.effects.time_stretch(data, rate=speed_factor)

def add_reverb(data, room_size=0.9, wet_level=0.5, dry_level=0.5, damping=0.2, pre_delay=0.04, wet_only=False):
    return librosa.effects.preemphasis(data, coef=0.95)

class ParkinsonsDataset(Dataset):
    def __init__(self, data_path, augment=False):
        self.file_list = []
        self.labels = []
        self.speakers = []
        self.augment = augment
        for class_label, class_name in enumerate(['hc', 'pd']):
            class_dir = os.path.join(data_path, class_name)
            if not os.path.exists(class_dir):
                continue
            for speaker in os.listdir(class_dir):
                speaker_dir = os.path.join(class_dir, speaker)
                if os.path.isdir(speaker_dir):
                    for file in os.listdir(speaker_dir):
                        if file.endswith('.wav'):
                            file_path = os.path.join(speaker_dir, file)
                            self.file_list.append(file_path)
                            self.labels.append(class_label)
                            self.speakers.append(speaker)
    def __len__(self):
        return len(self.file_list)
    def __getitem__(self, idx):
        audio_path = self.file_list[idx]
        label = self.labels[idx]
        speech_waveform, sampling_rate = torchaudio.load(audio_path)
        if sampling_rate != 16000:
            resampler = torchaudio.transforms.Resample(orig_freq=sampling_rate, new_freq=16000)
            speech_waveform = resampler(speech_waveform)
        speech_array = speech_waveform.squeeze().numpy()
        if self.augment:
            if random.random() < 0.5:
                speech_array = add_noise(speech_array)
            if random.random() < 0.5:
                speech_array = shift_time(speech_array)
            if random.random() < 0.5:
                speech_array = pitch_shift(speech_array, sr=16000)
            if random.random() < 0.5:
                speech_array = time_stretch(speech_array)
            if random.random() < 0.5:
                speech_array = speed_tune(speech_array)
```

```
            if random.random() < 0.5:
                speech_array = add_reverb(speech_array)
        return speech_array, label

def collate_fn(batch):
    speech_list = [item[0] for item in batch]
    label_list = [item[1] for item in batch]
    inputs = feature_extractor(
        speech_list,
        sampling_rate=16000,
        return_tensors="pt",
        padding=True
    )
    labels = torch.tensor(label_list)
    return inputs, labels

def train_epoch(model, train_loader, optimizer, scheduler, device, scaler, grad_accumulation_steps, clip_value):
    model.train()
    total_loss = 0
    correct_predictions = 0
    for batch_idx, batch in enumerate(tqdm(train_loader, desc="Training")):
        inputs, labels = batch
        inputs = {key: val.to(device) for key, val in inputs.items()}
        labels = labels.to(device)
        with torch.cuda.amp.autocast():
            outputs = model(**inputs, labels=labels)
            loss = outputs.loss
        scaler.scale(loss / grad_accumulation_steps).backward()
        if (batch_idx + 1) % grad_accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), clip_value)
            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad()
            scheduler.step()
        total_loss += loss.item() * inputs['input_values'].size(0)
        _, preds = torch.max(outputs.logits, dim=1)
        correct_predictions += torch.sum(preds == labels)
    total_loss /= len(train_loader.dataset)
    epoch_accuracy = correct_predictions.double() / len(train_loader.dataset)
    return total_loss, epoch_accuracy.item()

def evaluate_model(model, test_loader, device):
    model.eval()
    total_loss = 0
    correct_predictions = 0
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            inputs, labels = batch
            inputs = {key: val.to(device) for key, val in inputs.items()}
            labels = labels.to(device)
            outputs = model(**inputs, labels=labels)
            loss = outputs.loss
            total_loss += loss.item() * inputs['input_values'].size(0)
            _, preds = torch.max(outputs.logits, dim=1)
            correct_predictions += torch.sum(preds == labels)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())
    total_loss /= len(test_loader.dataset)
    epoch_accuracy = correct_predictions.double() / len(test_loader.dataset)
    report = classification_report(all_labels, all_preds, target_names=['Healthy Control', 'Parkinson\'s Disease'])
    cm = confusion_matrix(all_labels, all_preds)
    return total_loss, epoch_accuracy.item(), report, cm

def main():
    global feature_extractor
    data_path = 'Dataset2sa'
    outputs_dir = 'images'
    os.makedirs(outputs_dir, exist_ok=True)
    log_file = os.path.join(outputs_dir, 'training_log.jsonl')
    # Random seed
    seed = 35
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    dataset = ParkinsonsDataset(data_path)
    label_counts = Counter(dataset.labels)
    labels_np = np.array(dataset.labels)
    speakers_np = np.array(dataset.speakers)
    groups = speakers_np
    kf = GroupKFold(n_splits=5)
    feature_extractor = AutoFeatureExtractor.from_pretrained("MIT/ast-finetuned-audioset-10-10-0.4593")
    model = ASTForAudioClassification.from_pretrained(
        "MIT/ast-finetuned-audioset-10-10-0.4593",
        num_labels=2,
        ignore_mismatched_sizes=True
    ).to(device)
    if hasattr(model.classifier, 'dense'):
        nn.init.xavier_uniform_(model.classifier.dense.weight)
        nn.init.zeros_(model.classifier.dense.bias)
    elif hasattr(model.classifier, 'proj'):
```

```
    nn.init.xavier_uniform_(model.classifier.proj.weight)
    nn.init.zeros_(model.classifier.proj.bias)
learning_rate = 0.0000146547083690754
batch_size = 8
grad_accumulation_steps = 1
num_epochs = 50
warmup_ratio = 0.25
clip_value = 1.0
dropout_rate = 0.5
model.classifier.dropout = dropout_rate
class_weights = torch.tensor([1.0, 1.0], device=device)
criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = optim.AdamW(model.parameters(), lr=learning_rate, weight_decay=0.022890575468600547)
total_steps = 0
for train_index, test_index in kf.split(dataset.file_list, labels_np, groups):
    total_steps += (len(train_index) // batch_size) * num_epochs
def get_scheduler(optimizer):
    return get_linear_schedule_with_warmup(
        optimizer=optimizer,
        num_warmup_steps=int(total_steps * warmup_ratio),
        num_training_steps=total_steps
    )
all_val_accuracies = []
fold = 0
for train_index, test_index in kf.split(dataset.file_list, labels_np, groups):
    fold += 1
    print(f"\n===== Starting Fold {fold} =====\n")
    train_subset = Subset(dataset, train_index)
    test_subset = Subset(dataset, test_index)
    train_subset.dataset.augment = True
    sampler = None
    if label_counts[0] != label_counts[1]:
        weights = [1/label_counts[label] for label in [dataset.labels[i] for i in train_index]]
        sampler = WeightedRandomSampler(weights, len(weights))
    train_loader = DataLoader(
        train_subset,
        batch_size=batch_size,
        shuffle=(sampler is None),
        sampler=sampler,
        collate_fn=collate_fn,
        num_workers=8,
        pin_memory=True
    )
    test_loader = DataLoader(
        test_subset,
        batch_size=batch_size,
        shuffle=False,
        collate_fn=collate_fn,
        num_workers=8,
        pin_memory=True
    )
    fold_model = ASTForAudioClassification.from_pretrained(
        "MIT/ast-finetuned-audioset-10-10-0.4593",
        num_labels=2,
        ignore_mismatched_sizes=True
    ).to(device)
    if hasattr(fold_model.classifier, 'dense'):
        nn.init.xavier_uniform_(fold_model.classifier.dense.weight)
        nn.init.zeros_(fold_model.classifier.dense.bias)
    elif hasattr(fold_model.classifier, 'proj'):
        nn.init.xavier_uniform_(fold_model.classifier.proj.weight)
        nn.init.zeros_(fold_model.classifier.proj.bias)
    fold_model.classifier.dropout = dropout_rate
    opt = optim.AdamW(fold_model.parameters(), lr=learning_rate, weight_decay=1e-2)
    sched = get_scheduler(opt)
    best_val_accuracy = 0.0
    epochs_no_improve = 0
    patience = 10
    for epoch in range(num_epochs):
        print(f"Fold {fold}, Epoch {epoch+1}/{num_epochs}")
        train_loss, train_accuracy = train_epoch(fold_model, train_loader, opt, sched, device, scaler, grad_accumulation_steps, clip_value)
        val_loss, val_accuracy, val_report, val_cm = evaluate_model(fold_model, test_loader, device)

        # Log the results
        print(f"Fold {fold}, Epoch {epoch+1}/{num_epochs} - Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, Val Loss: {val_loss:.4f}, Val
        print("Validation Classification Report:")
        print(val_report)

        # Save the best model
        if val_accuracy > best_val_accuracy:
            best_val_accuracy = val_accuracy
            epochs_no_improve = 0
            fold_model.save_pretrained(f'best_parkinsons_ast_model_fold_{fold}')
            feature_extractor.save_pretrained(f'best_parkinsons_ast_feature_extractor_fold_{fold}')
            print(f"=> New best model saved for fold {fold}.")
        else:
            epochs_no_improve += 1
            print(f"No improvement in validation accuracy for {epochs_no_improve} epoch(s).")
            if epochs_no_improve >= patience:
                print("Early stopping triggered!")
                break

    # Load the best model for evaluation
    best_model = ASTForAudioClassification.from_pretrained(f'best_parkinsons_ast_model_fold_{fold}').to(device)
    final_val_loss, final_val_accuracy, final_val_report, final_val_cm = evaluate_model(best_model, test_loader, device)
```

```
        all_val_accuracies.append(final_val_accuracy)

        # Plot and save the confusion matrix
        cm_display = ConfusionMatrixDisplay(confusion_matrix=final_val_cm, display_labels=['Healthy Control', 'Parkinson\'s Disease'])
        cm_display.plot(cmap=plt.cm.Blues)
        plt.title(f'Confusion Matrix for Fold {fold}')
        cm_path = os.path.join(outputs_dir, f'confusion_matrix_fold_{fold}.png')
        plt.savefig(cm_path)
        plt.close()
        print(f"Confusion matrix saved to {cm_path}")

        print(f"Fold {fold} - Final Validation Accuracy: {final_val_accuracy:.4f}")

        # Clean up
        del fold_model
        torch.cuda.empty_cache()
        gc.collect()

    overall_acc = np.mean(all_val_accuracies)
    print(f'\n===== Cross-validation completed =====')
    print(f'Cross-validation accuracy: {overall_acc:.4f}')
    del model
    torch.cuda.empty_cache()
    gc.collect()

if __name__ == '__main__':
    main()
```

5.4.4. Código de AST con Optuna

```
import os
import random
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchaudio
import librosa
from torch.utils.data import Dataset, DataLoader, Subset, WeightedRandomSampler
from transformers import AutoFeatureExtractor, ASTForAudioClassification, get_linear_schedule_with_warmup
from sklearn.model_selection import GroupKFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from collections import Counter
import warnings
import gc
import matplotlib.pyplot as plt
import json
from datetime import datetime
from tqdm import tqdm
import optuna

warnings.filterwarnings("ignore")
scaler = torch.cuda.amp.GradScaler()

def configure_gpu():
    if torch.cuda.is_available():
        device = torch.device('cuda')
        torch.cuda.empty_cache()
        torch.backends.cudnn.benchmark = True
        torch.backends.cudnn.deterministic = False
    else:
        device = torch.device('cpu')
    return device

device = configure_gpu()

def add_noise(data, noise_factor=0.005):
    noise = np.random.randn(len(data))
    augmented_data = data + noise_factor * noise
    return augmented_data

def shift_time(data, shift_max=0.2):
    shift = int(random.uniform(-shift_max, shift_max) * len(data))
    augmented_data = np.roll(data, shift)
    if shift > 0:
        augmented_data[:shift] = 0
    else:
        augmented_data[shift:] = 0
    return augmented_data

def pitch_shift(data, sr, n_steps=2):
    return librosa.effects.pitch_shift(data, sr=sr, n_steps=n_steps)

def time_stretch(data, rate=1.1):
    return librosa.effects.time_stretch(data, rate=rate)

def speed_tune(data, speed_factor=1.1):
    return librosa.effects.time_stretch(data, rate=speed_factor)

def add_reverb(data, room_size=0.9, wet_level=0.5, dry_level=0.5, damping=0.2, pre_delay=0.04, wet_only=False):
    return librosa.effects.preemphasis(data, coef=0.95)
```

```
class ParkinsonsDataset(Dataset):
    def __init__(self, data_path, augment=False):
        self.file_list = []
        self.labels = []
        self.speakers = []
        self.augment = augment
        for class_label, class_name in enumerate(['hc', 'pd']):
            class_dir = os.path.join(data_path, class_name)
            if not os.path.exists(class_dir):
                continue
            for speaker in os.listdir(class_dir):
                speaker_dir = os.path.join(class_dir, speaker)
                if os.path.isdir(speaker_dir):
                    for file in os.listdir(speaker_dir):
                        if file.endswith('.wav'):
                            file_path = os.path.join(speaker_dir, file)
                            self.file_list.append(file_path)
                            self.labels.append(class_label)
                            self.speakers.append(speaker)
    def __len__(self):
        return len(self.file_list)
    def __getitem__(self, idx):
        audio_path = self.file_list[idx]
        label = self.labels[idx]
        speech_waveform, sampling_rate = torchaudio.load(audio_path)
        if sampling_rate != 16000:
            resampler = torchaudio.transforms.Resample(orig_freq=sampling_rate, new_freq=16000)
            speech_waveform = resampler(speech_waveform)
        speech_array = speech_waveform.squeeze().numpy()
        if self.augment:
            if random.random() < 0.5:
                speech_array = add_noise(speech_array)
            if random.random() < 0.5:
                speech_array = shift_time(speech_array)
            if random.random() < 0.5:
                speech_array = pitch_shift(speech_array, sr=16000)
            if random.random() < 0.5:
                speech_array = time_stretch(speech_array)
            if random.random() < 0.5:
                speech_array = speed_tune(speech_array)
            if random.random() < 0.5:
                speech_array = add_reverb(speech_array)
        return speech_array, label

def collate_fn(batch):
    speech_list = [item[0] for item in batch]
    label_list = [item[1] for item in batch]
    inputs = feature_extractor(
        speech_list,
        sampling_rate=16000,
        return_tensors="pt",
        padding=True
    )
    labels = torch.tensor(label_list)
    return inputs, labels

def train_epoch(model, train_loader, optimizer, scheduler, device, scaler, grad_accumulation_steps, clip_value):
    model.train()
    total_loss = 0
    correct_predictions = 0
    for batch_idx, batch in enumerate(tqdm(train_loader, desc="Training")):
        inputs, labels = batch
        inputs = {key: val.to(device) for key, val in inputs.items()}
        labels = labels.to(device)
        with torch.cuda.amp.autocast():
            outputs = model(**inputs, labels=labels)
            loss = outputs.loss
        scaler.scale(loss / grad_accumulation_steps).backward()
        if (batch_idx + 1) % grad_accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), clip_value)
            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad()
            scheduler.step()
        total_loss += loss.item() * inputs['input_values'].size(0)
        _, preds = torch.max(outputs.logits, dim=1)
        correct_predictions += torch.sum(preds == labels)
    total_loss /= len(train_loader.dataset)
    epoch_accuracy = correct_predictions.double() / len(train_loader.dataset)
    print(f"Train Loss: {total_loss:.4f}, Train Accuracy: {epoch_accuracy:.4f}")
    return total_loss, epoch_accuracy.item()


def evaluate_model(model, test_loader, device):
    model.eval()
    total_loss = 0
    correct_predictions = 0
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            inputs, labels = batch
            inputs = {key: val.to(device) for key, val in inputs.items()}
            labels = labels.to(device)
```

```
            outputs = model(**inputs, labels=labels)
            loss = outputs.loss
            total_loss += loss.item() * inputs['input_values'].size(0)
            _, preds = torch.max(outputs.logits, dim=1)
            correct_predictions += torch.sum(preds == labels)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())
    total_loss /= len(test_loader.dataset)
    epoch_accuracy = correct_predictions.double() / len(test_loader.dataset)
    report = classification_report(all_labels, all_preds, target_names=['Healthy Control', 'Parkinson\'s Disease'])
    cm = confusion_matrix(all_labels, all_preds)
    print(f"Validation Loss: {total_loss:.4f}, Validation Accuracy: {epoch_accuracy:.4f}")
    return total_loss, epoch_accuracy.item(), report, cm


def objective(trial):
    global feature_extractor
    random.seed(35)
    np.random.seed(35)
    torch.manual_seed(35)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(35)
    data_path = 'Dataset2s'
    dataset = ParkinsonsDataset(data_path)
    label_counts = Counter(dataset.labels)
    labels_np = np.array(dataset.labels)
    speakers_np = np.array(dataset.speakers)
    groups = speakers_np
    kf = GroupKFold(n_splits=5)
    feature_extractor = AutoFeatureExtractor.from_pretrained("MIT/ast-finetuned-audioset-16-16-0.442")
    learning_rate = trial.suggest_loguniform('learning_rate', 1e-6, 1e-3)
    batch_size = trial.suggest_categorical('batch_size', [8, 16, 24, 32])
    num_epochs = trial.suggest_int('num_epochs', 10, 64, step=10)
    warmup_ratio = trial.suggest_float('warmup_ratio', 0.0, 0.3, step=0.05)
    clip_value = trial.suggest_float('clip_value', 0.1, 2.0, step=0.1)
    dropout_rate = trial.suggest_float('dropout_rate', 0.0, 0.8, step=0.1)
    weight_decay = trial.suggest_loguniform('weight_decay', 1e-6, 1e-1)
    grad_accumulation_steps = 1
    patience = trial.suggest_int('patience', 3, 10)
    total_steps = 0
    for train_index, test_index in kf.split(dataset.file_list, labels_np, groups):
        total_steps += (len(train_index) // batch_size) * num_epochs
    def get_scheduler(optimizer):
        return get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=int(total_steps * warmup_ratio),
            num_training_steps=total_steps
        )
    all_val_accuracies = []
    fold = 0
    for train_index, test_index in kf.split(dataset.file_list, labels_np, groups):
        fold += 1
        train_subset = Subset(dataset, train_index)
        test_subset = Subset(dataset, test_index)
        train_subset.dataset.augment = True
        sampler = None
        if label_counts[0] != label_counts[1]:
            weights = [1/label_counts[label] for label in [dataset.labels[i] for i in train_index]]
            sampler = WeightedRandomSampler(weights, len(weights))
        train_loader = DataLoader(
            train_subset,
            batch_size=batch_size,
            shuffle=(sampler is None),
            sampler=sampler,
            collate_fn=collate_fn,
            num_workers=4,
            pin_memory=True
        )
        test_loader = DataLoader(
            test_subset,
            batch_size=batch_size,
            shuffle=False,
            collate_fn=collate_fn,
            num_workers=4,
            pin_memory=True
        )
        fold_model = ASTForAudioClassification.from_pretrained(
            "MIT/ast-finetuned-audioset-16-16-0.442",
            num_labels=2,
            ignore_mismatched_sizes=True
        ).to(device)
        if hasattr(fold_model.classifier, 'dense'):
            nn.init.xavier_uniform_(fold_model.classifier.dense.weight)
            nn.init.zeros_(fold_model.classifier.dense.bias)
        elif hasattr(fold_model.classifier, 'proj'):
            nn.init.xavier_uniform_(fold_model.classifier.proj.weight)
            nn.init.zeros_(fold_model.classifier.proj.bias)
        fold_model.classifier.dropout = dropout_rate
        opt = optim.AdamW(fold_model.parameters(), lr=learning_rate, weight_decay=weight_decay)
        sched = get_scheduler(opt)
        best_val_accuracy = 0.0
        epochs_no_improve = 0
        for epoch in range(num_epochs):
            print(f"Fold {fold}, Epoch {epoch+1}/{num_epochs}")
            train_loss, train_accuracy = train_epoch(fold_model, train_loader, opt, sched, device, scaler, grad_accumulation_steps, clip_value)
```

```
            val_loss, val_accuracy, val_report, val_cm = evaluate_model(fold_model, test_loader, device)
            if val_accuracy > best_val_accuracy:
                best_val_accuracy = val_accuracy
                epochs_no_improve = 0
            else:
                epochs_no_improve += 1
                if epochs_no_improve >= patience:
                    print("Early stopping triggered!")
                    break
        all_val_accuracies.append(best_val_accuracy)
        del fold_model
        torch.cuda.empty_cache()
        gc.collect()
    overall_acc = np.mean(all_val_accuracies)
    print(f"Overall cross-validation accuracy: {overall_acc:.4f}")
    return overall_acc

if __name__ == '__main__':
    # Define Optuna study with SQLite storage for persistence
    storage_name = 'sqlite:///study.db'  # SQLite database stored in current directory

    # Create the study with storage
    study = optuna.create_study(
        study_name="audio_classification_optimization",
        storage=storage_name,
        direction="maximize",
        load_if_exists=True
    )

    # Optimize the study
    study.optimize(objective, n_trials=1000)

    # Print the best parameters
    print(f"Best parameters: {study.best_params}")
    print(f"Best value: {study.best_value}")

    # Save the study results for analysis
    study.trials_dataframe().to_csv("study_results.csv", index=False)
    print("Study results saved to study_results.csv")
```

5.4.5. Script para eliminar silencios mayores a 0.5s

```
import os
import sys
import shutil
import webrtcvad
from pydub import AudioSegment
from pydub.utils import make_chunks

def vad_split(audio, sample_rate=16000, frame_duration=30, padding_duration=300):
    """
    Split audio using VAD into non-silent segments.

    Parameters:
        audio (AudioSegment): The input audio.
        sample_rate (int): The sample rate for VAD.
        frame_duration (int): Duration of each frame in ms.
        padding_duration (int): Minimum duration to consider a segment as speech.

    Returns:
        List[AudioSegment]: List of non-silent audio segments.
    """
    vad = webrtcvad.Vad(2)  # 0: least aggressive, 3: most aggressive

    audio = audio.set_frame_rate(sample_rate).set_channels(1).set_sample_width(2)
    raw_audio = audio.raw_data
    frame_size = int(sample_rate * frame_duration / 1000) * 2  # 16-bit PCM

    segments = []
    voiced_frames = []
    num_padding_frames = padding_duration // frame_duration
    ring_buffer = []

    for i in range(0, len(raw_audio), frame_size):
        frame = raw_audio[i:i+frame_size]
        if len(frame) < frame_size:
            break
        is_speech = vad.is_speech(frame, sample_rate)
        ring_buffer.append((frame, is_speech))
        if len(ring_buffer) > num_padding_frames:
            ring_buffer.pop(0)

        if is_speech:
            voiced_frames.append(frame)
        elif voiced_frames:
            segments.append(AudioSegment(
                b''.join(voiced_frames),
                frame_rate=sample_rate,
                sample_width=2,
                channels=1
            ))
```

```
            voiced_frames = []

    if voiced_frames:
        segments.append(AudioSegment(
            b''.join(voiced_frames),
            frame_rate=sample_rate,
            sample_width=2,
            channels=1
        ))

    return segments

def process_wav_file_vad(file_path):
    try:
        audio = AudioSegment.from_wav(file_path)
    except Exception as e:
        print(f"[ERROR] Could not load '{file_path}': {e}")
        return

    original_duration = len(audio) / 1000.0

    segments = vad_split(audio)
    if not segments:
        print(f"[WARNING] No speech detected in '{file_path}'. Skipping.")
        return

    processed_audio = AudioSegment.empty()
    for segment in segments:
        processed_audio += segment

    processed_duration = len(processed_audio) / 1000.0

    try:
        processed_audio.export(file_path, format="wav")
        print(f"[SUCCESS] Processed '{file_path}': {original_duration:.2f}s -> {processed_duration:.2f}s")
    except Exception as e:
        print(f"[ERROR] Could not save '{file_path}': {e}")

def process_all_wav_files_vad(root_dir):
    for dirpath, dirnames, filenames in os.walk(root_dir):
        for filename in filenames:
            if filename.lower().endswith('.wav'):
                file_path = os.path.join(dirpath, filename)
                print(f"[PROCESSING] {file_path}")
                process_wav_file_vad(file_path)

def main_vad():
    # Verify that FFmpeg is installed
    if not shutil.which("ffmpeg"):
        print("[ERROR] FFmpeg is not installed or not found in PATH. Please install FFmpeg before running this script.")
        sys.exit(1)

    # Assume the script is run from inside RestructuredDataset/
    current_dir = os.getcwd()
    print(f"[INFO] Starting VAD processing in directory: {current_dir}")

    process_all_wav_files_vad(current_dir)

    print("[INFO] All files have been processed.")

if __name__ == "__main__":
    main_vad()
```

5.4.6. Script para segmentar los audios

```
import os
import shutil

def create_datasets(base_dir, times, dataset_names):
    """
    Creates new datasets based on specified time durations by copying .wav files.

    Parameters:
    - base_dir (str): Path to the 'RestructuredDataset - 3 Split/' directory.
    - times (list): List of time subfolder names (e.g., ['0.5s', '1s', '2s', '3s']).
    - dataset_names (dict): Mapping from time to new dataset folder name.
    """

    # Identify all top-level categories (e.g., 'hc', 'pd', etc.)
    categories = [d for d in os.listdir(base_dir)
                  if os.path.isdir(os.path.join(base_dir, d))
                  and d not in dataset_names.values()]  # Exclude already created datasets

    print(f"Found categories: {categories}")

    for time in times:
        dataset_dir = os.path.join(base_dir, dataset_names[time])
        os.makedirs(dataset_dir, exist_ok=True)
        print(f"\nCreating dataset for '{time}' at '{dataset_dir}'")

        for category in categories:
            source_category_dir = os.path.join(base_dir, category)
```

```
            target_category_dir = os.path.join(dataset_dir, category)
            os.makedirs(target_category_dir, exist_ok=True)
            print(f"  Processing category: {category}")

            # Iterate over main classes within the category
            main_classes = [d for d in os.listdir(source_category_dir)
                            if os.path.isdir(os.path.join(source_category_dir, d))]

            for main_class in main_classes:
                source_main_class_dir = os.path.join(source_category_dir, main_class)
                target_main_class_dir = os.path.join(target_category_dir, main_class)
                os.makedirs(target_main_class_dir, exist_ok=True)

                # Iterate over sub-classes within the main class
                sub_classes = [d for d in os.listdir(source_main_class_dir)
                               if os.path.isdir(os.path.join(source_main_class_dir, d))]

                for sub_class in sub_classes:
                    source_time_dir = os.path.join(source_main_class_dir, sub_class, time)
                    if os.path.isdir(source_time_dir):
                        # Iterate over all .wav files in the time directory
                        for file_name in os.listdir(source_time_dir):
                            if file_name.lower().endswith('.wav'):
                                source_file = os.path.join(source_time_dir, file_name)
                                target_file = os.path.join(target_main_class_dir, file_name)

                                # Check for file name conflicts
                                if os.path.exists(target_file):
                                    base, ext = os.path.splitext(file_name)
                                    new_file_name = f"{base}_{sub_class}{ext}"
                                    target_file = os.path.join(target_main_class_dir, new_file_name)
                                    print(f"    Renaming and copying '{file_name}' to '{new_file_name}' to avoid conflict.")

                                shutil.copy2(source_file, target_file)
                                print(f"    Copied '{file_name}' to '{target_main_class_dir}'")
                    else:
                        print(f"    Time folder '{time}' does not exist in sub-class '{sub_class}'. Skipping.")

    print("\nAll datasets have been created successfully.")

def main():
    # Define the times and corresponding dataset names
    times = ['0.5s', '1s', '2s', '3s']
    dataset_names = {
        '0.5s': 'Dataset0.5s',
        '1s': 'Dataset1s',
        '2s': 'Dataset2s',
        '3s': 'Dataset3s'
    }

    # Get the current directory (assumed to be 'RestructuredDataset - 3 Split/')
    base_dir = os.getcwd()
    print(f"Base directory: {base_dir}")

    create_datasets(base_dir, times, dataset_names)

if __name__ == "__main__":
    main()
```

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 14 23:12:15 CET 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)