Práctica 1

Whisper

# Introducción

El proyecto consistió en el desarrollo de un *Jupyter Notebook* para la construcción de un sistema que reconociera secuencias de dígitos a partir de audio utilizando un modelo transformer, y que además fuera capaz de calcular la suma de estos dígitos y presentarla en secuencia. La elección de un modelo transformer para esta tarea se basó en su eficacia demostrada en el procesamiento de secuencias, especialmente en tareas de procesamiento de lenguaje natural y reconocimiento de voz.

# Implementación

El primer paso en el cuaderno fue establecer un entorno controlado y reproducible. Para ello, se importaron las bibliotecas necesarias, como *torch*, *torchaudio*, *glob*, *random*, *scipy.signal*, y *numpy*. Se implementó la función *seed\_everything* para fijar la semilla de todos los generadores de números aleatorios, asegurando así la reproducibilidad de los experimentos.

Posteriormente, se procedió a definir la estructura del modelo transformer, comenzando con la creación de la clase *SelfAttention*, que es un componente fundamental de los transformers. Esta clase implementa la atención auto-dirigida que permite al modelo enfocarse en diferentes partes de la entrada para realizar predicciones más precisas. Se definieron las capas de normalización, las transformaciones lineales para las consultas, claves y valores, y la operación de atención propiamente dicha.

La clase *FeedForward* se diseñó para ser una capa adicional en el transformer, proporcionando una transformación no lineal que ayuda a aumentar la capacidad del modelo para aprender complejas representaciones de los datos.

La clase *Encoder*, que forma la mitad del modelo transformer, fue diseñada para procesar las secuencias de entrada. Se utilizó una combinación de atención auto-dirigida y transformaciones feed-forward para cada capa del *encoder*. La incorporación de la posición en el *encoder* fue crucial para mantener la información secuencial en los datos de entrada.

Para la parte de decodificación, se implementaron las clases *CausalSelfAttention* y *CrossAttention*. La atención causal asegura que las predicciones para un paso de tiempo dado solo dependan de los pasos anteriores, lo cual es esencial en la generación de secuencias. La atención cruzada, por otro lado, permite que el decodificador acceda a la salida del codificador, facilitando la alineación entre la entrada y la salida durante la generación de secuencias.

La clase *Decoder* integra estas dos formas de atención y añade una capa *feed-forward*, similar a la estructura del *encoder*, pero adaptada para la generación de secuencias.

La extracción de características del audio fue manejada por las clases *SpecAug* y *AudioFeatures*. *SpecAug* se usó para aumentar los datos de audio aplicando máscaras de tiempo y frecuencia, lo cual es una práctica común en el procesamiento de audio para mejorar la robustez del modelo. *AudioFeatures* transformó la señal de audio en un espectrograma de Mel, que luego fue procesado por una capa lineal para adaptarse a la dimensión del modelo.

Finalmente, la clase *AudioTransformer* integró todos estos componentes. Esta clase manejó el procesamiento completo, desde la entrada de audio hasta la generación de la secuencia de dígitos y su suma. La función *generate* de esta clase es particularmente importante, ya que define cómo el modelo produce la secuencia de salida a partir de una entrada de audio.

En cuanto a los datos, se crearon las clases *TrainDataset* y *TestDataset* para gestionar los conjuntos de entrenamiento y prueba, respectivamente. Estas clases se encargaron de cargar los archivos de audio, aplicar transformaciones de aumento de ruido y reverberación (mediante *NoiseAug* y *RIRAug*), y preparar las etiquetas de salida en el formato deseado.

El proceso de entrenamiento fue supervisado mediante una función de pérdida de entropía cruzada, y se utilizó el optimizador Adam para la actualización de los parámetros del modelo. El entrenamiento se llevó a cabo durante 20 épocas, mostrando una disminución consistente en la pérdida, lo cual indicaba que el modelo estaba aprendiendo efectivamente la tarea.

# Resultados

Ante unos primeros resultados de entorno al 68% de error, se hizo un ajuste de los hiperparámetros del modelo y se decidió escoger los valores de *d\_model*=1024, *nb\_layers*=8, *d\_ff*=1024, *n\_heads*=8, *d\_head*=32 y dropout=0.3. Cabe destacar que, debido a limitaciones de mi *hardware*, el entrenamiento se hizo con un tamaño de *batch* de 16 que me permitía realizar entrenamientos con menos de 6GB de ocupación de gráfica (entorno a uno 5,2GB).

Sin embargo, y pese a todos estos cambios, el modelo siguió mostrando una tasa de error relativamente alta, en concreto nos encontramos un error del 56.08%. Esto sugiere que, aunque el modelo puede aprender la tarea hasta cierto punto, todavía hay margen de mejora en su capacidad para generalizar a datos no vistos.

# Oportunidades de mejora

Una mejora clave sería la optimización de los hiperparámetros del modelo. Esto involucraría ajustar parámetros como el número de capas en los *encoders* y *decoders*, la dimensión de los modelos (*d\_model*), el número de cabezas en la atención multi-cabeza (*n\_heads*), y la dimensión de las capas *feed-forward* (*d\_ff*). Estos ajustes requieren un proceso de prueba y error o la implementación de una búsqueda automática de hiperparámetros, como una búsqueda en rejilla o una búsqueda bayesiana. En el código, esto implicaría cambiar los valores pasados a la clase *AudioTransformer* y posiblemente realizar ajustes en las clases *Encoder* y *Decoder* para manejar diferentes configuraciones de capas y dimensiones pero debido a limitaciones debido al *hardware*, esto implicaría demasiado tiempo.

La calidad y cantidad de datos de entrenamiento son fundamentales para el rendimiento del modelo. Una forma de mejorar esto sería incorporar un conjunto de datos más amplio y variado. Esto ayudaría al modelo a generalizar mejor a nuevos ejemplos. En términos de implementación, se podrían añadir más archivos de audio al dataset y potencialmente expandir las clases *TrainDataset* y *TestDataset* para manejar un mayor volumen de datos y una mayor diversidad en los tipos de audio.

El aumento de datos mediante técnicas como la distorsión del tiempo, la alteración del tono, o la adición de ruido de fondo podría mejorar la robustez del modelo. En el código actual, ya se utilizan técnicas de aumento como *SpecAug*, *NoiseAug*, y *RIRAug*. Sin embargo, estas podrían ampliarse o afinarse. Por ejemplo, se podrían experimentar con diferentes rangos para la máscara de frecuencia y tiempo en *SpecAug* o con diferentes niveles de SNR (relación señal-ruido) en *NoiseAug*.

Una consideración importante es la arquitectura del modelo. Experimentar con variantes de la arquitectura Transformer, como Transformer-XL o *Convolutions* Transformers, podría ofrecer mejoras en la capacidad del modelo para procesar secuencias largas y capturar dependencias a largo plazo. Esto implicaría reemplazar o modificar las clases existentes del modelo, como *Encoder*, *Decoder*, y *SelfAttention*, para incorporar estas nuevas arquitecturas.

Finalmente, el proceso de entrenamiento en sí mismo ofrece oportunidades de mejora. Una estrategia sería implementar técnicas de entrenamiento más sofisticadas, como el entrenamiento por currículum, donde el modelo primero aprende de ejemplos más simples y gradualmente se expone a ejemplos más complejos. Otra posibilidad sería utilizar técnicas de regularización más avanzadas para evitar el sobreajuste, como el *dropout* variacional o la normalización por lotes.

Cada una de estas mejoras requeriría modificaciones específicas en el código. Estas modificaciones podrían variar desde ajustes en los parámetros hasta la reestructuración completa de ciertas clases y métodos. La clave sería implementar estos cambios de manera iterativa, evaluando constantemente el impacto de cada ajuste en el rendimiento del modelo para guiar el proceso de optimización.

# Conclusiones

La realización de este proyecto ha sido un ejercicio profundo y enriquecedor en el ámbito del procesamiento de señales de audio y el aprendizaje automático, específicamente en la aplicación de modelos transformer para el reconocimiento de secuencias numéricas a partir de audio. La implementación de este sistema complejo, que integra la extracción de características del audio, la atención auto-dirigida y la generación de secuencias, ha demostrado no solo la flexibilidad de los modelos transformer sino también los desafíos inherentes a la tarea.

La creación de clases especializadas para manejar diferentes aspectos del modelo, como la atención, el codificador y el decodificador, y la extracción de características, refleja una profunda comprensión de la arquitectura transformer y sus aplicaciones en el procesamiento de secuencias. La inclusión de técnicas de aumento de datos y la consideración de la reproducibilidad son ejemplos de buenas prácticas en el diseño de experimentos de aprendizaje automático.

A pesar de los logros, el modelo presenta áreas de mejora, especialmente en su capacidad para generalizar a nuevos datos, como lo demuestra la tasa de error en la fase de prueba. Esto destaca la importancia de un enfoque iterativo y reflexivo en el desarrollo de modelos de aprendizaje automático, donde la evaluación continua y el ajuste fino juegan un papel crucial en la mejora del rendimiento.

El proyecto también subraya la relevancia de una comprensión por partes en el diseño de sistemas de IA, donde la calidad de los datos, la arquitectura del modelo, y los métodos de entrenamiento deben considerarse conjuntamente. Las posibles mejoras, como la optimización de hiperparámetros, el enriquecimiento de los datos de entrenamiento y la experimentación con diferentes arquitecturas, abren caminos hacia un modelo más robusto y preciso.

En resumen, este proyecto representa un valioso esfuerzo en la aplicación de modelos transformer en un contexto desafiante, ofreciendo *insights* significativos tanto en sus capacidades como en sus limitaciones.