TP3: Clasificación de Audio

Tomás Gallo, Jazmín Sneider, Guadalupe Molina

November 2024

1 Introducción:

En este informe detallaremos los pasos realizados para mejorar nuestro modelo de aprendizaje supervisado, diseñado específicamente para el reconocimiento del género de audios musicales de 5 segundos. Empezaremos describiendo el proceso de preprocesamiento de datos, explicando cómo preparamos los audios para optimizar el desempeño del modelo. Posteriormente, abordaremos las distintas mejoras implementadas, incluyendo las transformaciones aplicadas a los datos y los ajustes en los hiperparámetros, con el objetivo de aumentar la precisión y la eficiencia del modelo.

2 Ejercicio 1: Configuración

Al realizar la separación de los datos en **training**, **test** y **validation**, identificamos un posible problema: si aplicábamos la división directamente sobre el conjunto completo, podía suceder que, por ejemplo, el conjunto de **testing** quedara compuesto únicamente de canciones de rock. Para evitar este desequilibrio, aprovechamos que los datos estaban organizados en carpetas por género y realizamos la división dentro de cada carpeta. Así, de cada género seleccionamos el 80% para **training**, el 10% para **validation** y el 10% restante para **testing**. De esta manera, logramos una distribución balanceada en cada subconjunto, preservando la diversidad de géneros en todas las etapas del modelo. Para nuestras experimentaciones iniciales, la learning rate inicial era de 0.0004, como optimizador utilizamos ADAM, no usamos ningún scheduler, realizamos 5 epochs. Utilizamos como input las waveforms de los audio. El modelo inicial tiene 3 capas densas y una hidden que tiene 64 nodos de entrada y de salida. La primera capa densa tiene una función de activación RELU. Ejecutando el modelo, sin más cambios, obtuvimos la siguientes losses en training y en validation.

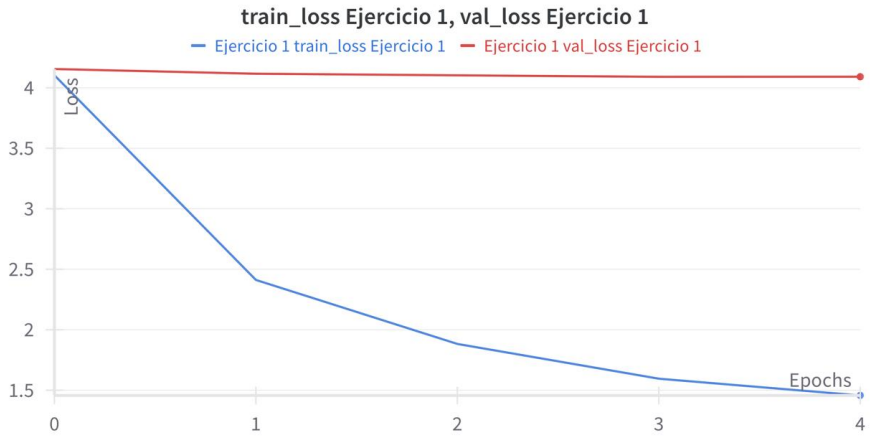

Figure 1: Comparación Train loss y Validation loss

Inicialmente el modelo es bastante malo prediciendo, con una función de pérdida bastante grande tanto en training como en validation, y con una accuracy entre 0,08 y 0,17. Esto nos indica que habrá que hacerle varias modificaciones al modelo si nuestro objetivo es optimizarlo.

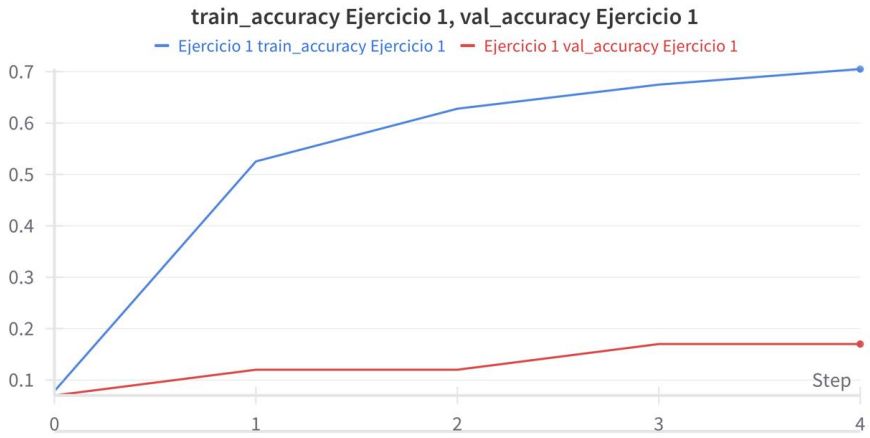

Figure 2: Accuracies de train y validation

Pudimos notar que la accuracy del test subía y su loss bajaba, lo que nos llevó a concluir que hubo overfitting a los datos de training ya que ni la loss del validation bajaba mucho ni su accuracy subía demasiado mientras que las métricas de training mejoraban.

3 Ejercicio 2: Arquitectura

Para explorar distintas opciones, decidimos probar diez modelos con configuraciones diferentes y seleccionar el mejor. Para definir las configuraciones de cada modelo, generamos de manera aleatoria la cantidad de hidden layers, eligiendo un número entre 2 y 15. Además, para cada modelo, definimos también de manera aleatoria la cantidad de **nodos** en cada una de sus **hidden layers**, seleccionando un valor entre 10 y 128. Guardamos todas las configuraciones utilizadas para poder analizarlas después de ejecutar los cinco modelos y determinar cuál de ellos ofrecía un mejor desempeño. Sin embargo, en términos generales, las mejoras con respecto al modelo original fueron pequeñas, y la precisión (**accuracy**) se mantuvo todavía en un nivel bastante bajo. Nuestro mejor modelo para este ejercicio tuvo un accuracy en validation de 0.2 y tenía 14 capas ocultas.

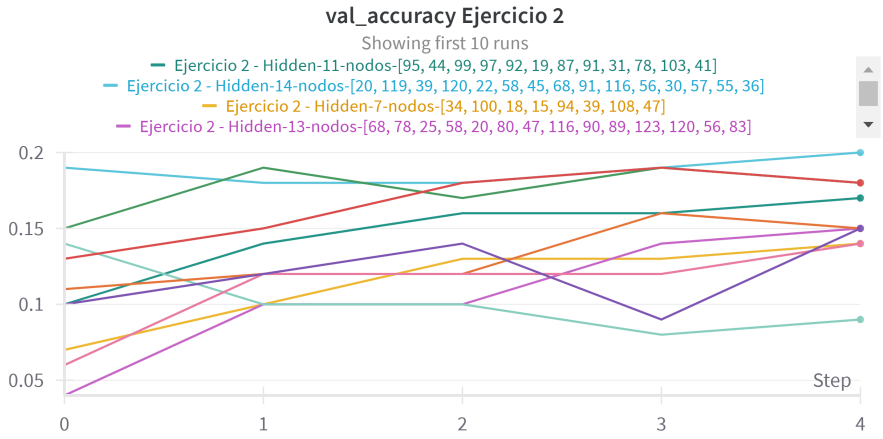

Figure 3: Accuracies de validation para los modelos

El peor modelo fue de 10 hidden layers con una accuracy de 0.09.

A continuación relizamos un grafico para poder comparar de manera mas detallada la performance del mejor y peor experimento.

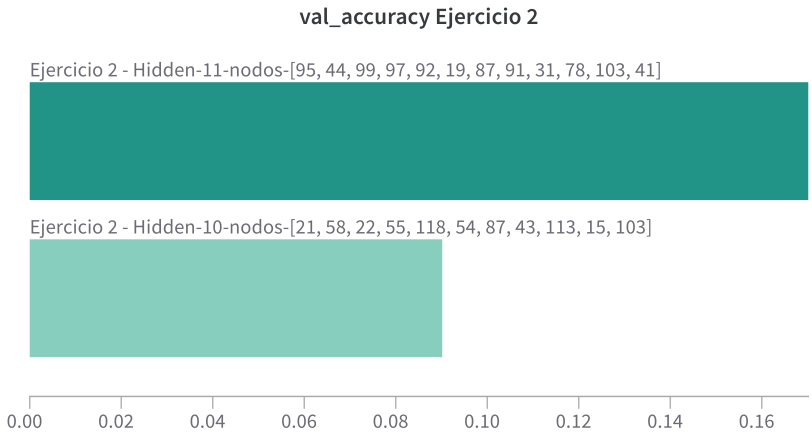

Figure 4: Comparacion del mejor y peor experimento

4 Ejercicio 3: Arquitectura CNN

Las capas convolucionales son excelentes para identificar patrones espaciales y características específicas en imágenes o datos visuales. Entonces en lugar de usar directamente los waveforms o formas de onda como datos de entrada, aplicamos una transformación para convertirlos en espectrogramas, una representación gráfica que muestra cómo las frecuencias de audio cambian con el tiempo. Para ello, definimos funciones auxiliares que transforman los datos de audio en espectrogramas y adaptamos las funciones de entrenamiento y validación para poder entrenar el modelo utilizando esta nueva forma de entrada. Desarrollamos dos arquitecturas de redes CNN para probarlas.

- En la primera, configuramos dos capas convolucionales:

    1. La primera capa convolucional tiene un canal de entrada, 32 canales de salida, y un kernel de 9x9, que define el tamaño del filtro que se aplica a la imagen. Para conservar el tamaño original del espectrograma, añadimos un padding de 4.
    2. La segunda capa convolucional toma como entrada los 32 canales generados por la primera capa y los expande a 64 canales de salida, con un kernel de 3x3 y un padding de 1, de modo que el tamaño de la imagen también se conserve.

    Cada capa convolucional incorpora un maxpooling de 2x2, que reduce el tamaño de la imagen a la mitad, simplificando los datos y resaltando las características más importantes. Dado que los espectrogramas tienen un tamaño inicial de 201x552, el average pooling total reduce el tamaño

cuatro veces. Tras pasar por ambas capas, la salida final tiene un tamaño calculado de 64x50x138. Antes de enviar estos datos a la capa totalmente conectada (fully connected), aplicamos una función de torch.flatten. La función de activación ReLU se utiliza en cada capa convolucional para introducir no linealidades. Finalmente, la primera capa fully connected en el modelo original utiliza una activación ReLU normal.

- La segunda configuración posee 3 capas: en las dos primeras capas convolucionales maxpool y la última averagepool.

  Creamos dos modelos que si bien poseían una cantidad de capas convolucionales distintas ambas tenían una única hiddden layer con una entrada de 128 nodos y una salida del mismo tamaño. Con estas arquitecturas CNN la performance del modelo mejoró muchísimo, alcanzando una accuracy de 0.47 en CNN1 y CNN2 de 0.42.

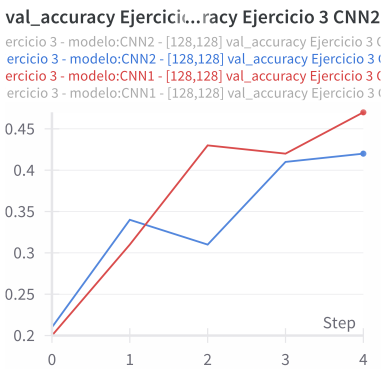

Figure 5: Comparación Train loss y Validation loss

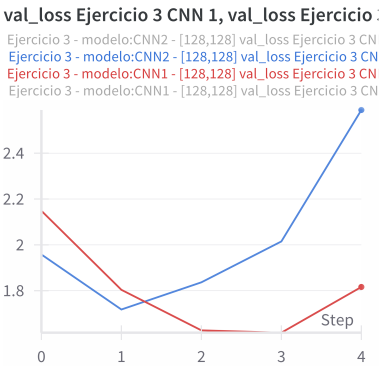

Figure 6: Comparación Train accuracy y Validation accuracy

Finalmente decidimos quedarnos con el CNN1 como mejor modelo ya que tuvimos mejores resultados y era un poco más rápido de ejecutarse.

5 Ejercicio 4: Funciones de Activación

Con el mejor modelo que habíamos obtenido hasta el momento, la arquitectura **CNN2**, realizamos experimentos para probar distintas **funciones de activación** en cada capa y observar su efecto en el rendimiento del modelo. Las funciones de activación introducen no linealidad en el modelo, permitiéndole aprender patrones más complejos a partir de los datos. Para poder variar las funciones de activación, editamos la clase de la CNN original, añadiendo una configuración que permitía elegir entre varias opciones: **ReLU**, **Leaky ReLU**,

Sigmoid, o una combinación extra que mezclaba las tres funciones en diferentes capas. Tras ejecutar varias pruebas, encontramos que la configuración que mejor desempeño alcanzó fue **Leaky Relu**, con una precisión (**accuracy**) final de 0.41. Sin embargo, decidimos mantener a Relu como la función que vamos a usar durante los incisos siguientes. Esto se debe a que en ejecuciones anteriores del código, Relu siempre le ganó a las otras dos, aunque en este caso perdió. También nos guiamos por el gráfico, donde Relu en todas las epochs le gana a las otras dos, salvo en la última y este comportamiento se repitió siempre en todas las ejecuciones. En este caso, aunque probamos distintas funciones de activación, la precisión del modelo no mostró una mejora significativa. Esto resulta inusual, ya que en todas las veces anteriores que corrimos este mismo inciso, logramos observar un aumento considerable en la precisión a partir del inciso 3. Aunque la causa exacta de esta falta de mejora sigue sin estar clara, creemos que el uso de una función de activación generalmente contribuye a mejorar el desempeño. Por esta razón, hemos decidido conservar la función de activación que mostró el mejor rendimiento. Además, consideramos que como aún quedan aspectos por optimizar, como el tamaño del batch, el tipo de optimizador y otros parámetros; ajustes en estos factores podrían influir positivamente en la precisión general del modelo sumándose a la función de activación.

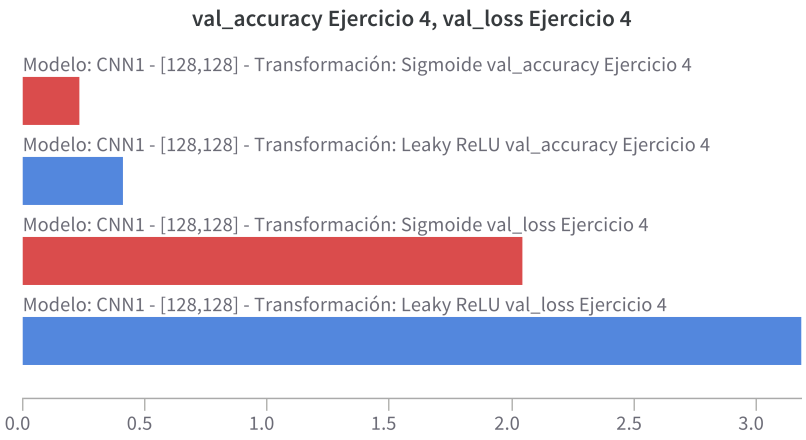

Figure 7: Comparacion peor y mejor experimento de función de activación

En este gráfico podemos observar y comparar la loss y accuracy entre el peor y mejor experimento que realizamos, los cuales en este caso son la función sigmoidea y la función Leaky Relu.

6 Ejercicio 5: Optimizadores

En este inciso, probamos distintos optimizadores para nuestro modelo, utilizando los datos de espectrogramas para el entrenamiento y validación. Elegimos dos optimizadores: **Adam** y **SGD**. Para cada uno, aplicamos dos tipos de ajustes en la **learning rate**: *exponential decay* y *reduce on plateau*. Estos ajustes permiten que el modelo ajuste su aprendizaje a medida que avanza.

1. **Adam con exponential decay**: En esta configuración, utilizamos el optimizador Adam con una disminución exponencial en la tasa de aprendizaje. Esto significa que, con cada epoch, la tasa de aprendizaje se reduce multiplicándola por un valor constante, en este caso, el valor **gamma** (0.95). Esto permite que el modelo disminuya la velocidad de aprendizaje progresivamente a medida que se acerca a un mínimo, evitando oscilaciones bruscas.

2. **Adam con reduce on plateau**: Aquí utilizamos el optimizador Adam con un ajuste de tasa de aprendizaje que se reduce cuando el modelo se estanca, es decir, cuando el desempeño en el conjunto de validación deja de mejorar. El parámetro **factor** (0.5) indica cuánto se reducirá la tasa de aprendizaje cada vez que se detecte estancamiento, mientras que **patience** (2) especifica la cantidad de épocas que el modelo esperará antes de ajustar la tasa.

3. **SGD con exponential decay**: Esta configuración emplea el optimizador SGD (gradiente estocástico descendente) con la misma estrategia de disminución exponencial de la tasa de aprendizaje usando el valor de gamma de 0.95. Al igual que en Adam, esto ayuda al modelo a acercarse a un mínimo de forma controlada.

4. **SGD con reduce on plateau**: Finalmente, probamos el optimizador SGD con reducción de la tasa de aprendizaje en plateau. Esto significa que reducimos la tasa en un 50% (según el **factor**) si el modelo no mejora tras cinco epochs (**patience**).

Al final de estas pruebas, comparamos los valores de precisión de validación de cada configuración para analizar cuál obtiene mejores resultados para nuestros datos.

7 Ejercicio 6: Entrenamiento con distintos batch-sizes y epochs

En este inciso, buscamos analizar cómo el tamaño del batch y la cantidad de epochs afectan el rendimiento del modelo. Para ello, se modifican dos hiperparámetros clave en el proceso de entrenamiento: el **batch size** y el **número de epochs**. El batch size es el número de datos de entrenamiento que se procesan juntos en una sola iteración durante el entrenamiento del modelo. En lugar de alimentar al modelo con todo el conjunto de datos de una sola vez, se divide en pequeños lotes (o batches). Este parámetro tiene un impacto directo en la eficiencia del entrenamiento y en la capacidad del modelo para generalizar a nuevos datos:

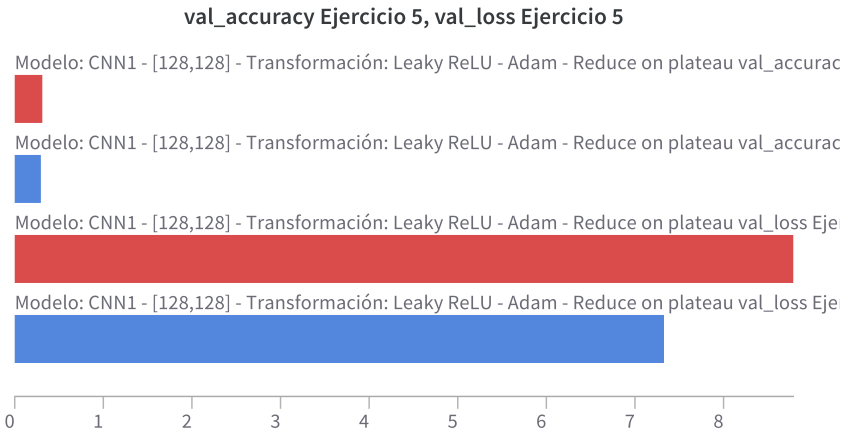

Figure 8: Comparación del peor y mejor experimento

- **Batch pequeño:** Un tamaño de batch pequeño significa que el modelo se actualiza más frecuentemente con cada lote, lo que puede resultar en una convergencia más rápida, aunque más ruidosa. También puede ayudar al modelo a generalizar mejor, pero a costa de un mayor tiempo de entrenamiento debido a la mayor cantidad de actualizaciones.
- **Batch grande:** Un tamaño de batch grande reduce la frecuencia de actualización de los parámetros del modelo, lo que puede llevar a una convergencia más estable, pero a veces más lenta.

El número de epochs es el número de veces que el conjunto completo de datos de entrenamiento pasa a través del modelo. Con más epochs, el modelo tiene más oportunidades de aprender y ajustar sus parámetros, pero también existe el riesgo de sobreajuste si el número de épocas es excesivo. Para llevar a cabo el experimento, usamos un código que combina distintas configuraciones de batch size y epochs para evaluar cómo cada uno afecta la precisión de validación del modelo. El proceso se realizó en dos pasos principales: Modificación del tamaño del batch: Se definió una función llamada batch_size_modifier, que toma como entrada el conjunto de entrenamiento (train_ds) y un tamaño de batch específico. Esta función crea un DataLoader con el tamaño de batch indicado, permitiendo que el modelo procese los datos en lotes del tamaño especificado y los utilice para el entrenamiento. Entrenamiento y validación: Se probaron diferentes combinaciones de batch size y epochs. Los valores de batch size elegidos fueron: 16, 20, 32, 64, 128 y 256, mientras que las epochs a probar fueron 3, 5, 7 y 10. Para cada combinación, el modelo se entrenó durante el número de epochs especificado y se evaluó la precisión de validación al final de cada ciclo de entrenamiento. Los resultados de la precisión de validación fueron almacenados en un diccionario resultados, donde las claves fueron las tuplas

(batch_size, epoch) y los valores fueron las precisiones obtenidas en el conjunto de validación.

8 Ejercicio 7: Regularización

En este experimento, se implementaron dos técnicas de **regularización** con el fin de mejorar la generalización del modelo y evitar el sobreajuste durante el entrenamiento: **L2** y **Dropout**.

8.1 Regularización L2

La regularización **L2**, también conocida como *ridge regularization*, es una técnica que penaliza grandes valores en los parámetros del modelo. Se añade un término al cálculo de la función de pérdida que es proporcional al cuadrado de los valores de los pesos del modelo. Este término es el siguiente:

$$L2 = \lambda \sum_{i=1}^{n} w_i^2$$

Donde w_i son los pesos del modelo y λ es un hiperparámetro que controla la magnitud de la penalización. La regularización L2 fomenta la reducción de los valores de los pesos, lo que puede ayudar a evitar que el modelo se ajuste demasiado a las particularidades del conjunto de entrenamiento, lo cual podría causar sobreajuste.
Al reducir los pesos de manera proporcional, L2 ayuda a que el modelo sea más robusto a pequeñas fluctuaciones en los datos, promoviendo una mejor capacidad de generalización. En resumen, L2 penaliza la complejidad del modelo y lo empuja a aprender representaciones más simples y generalizables.

8.2 Dropout

Por otro lado, **Dropout** es una técnica que consiste en "apagar" aleatoriamente un porcentaje de neuronas durante el entrenamiento en cada iteración. Específicamente, durante una pasada hacia adelante en la red, se seleccionan aleatoriamente neuronas y se les asigna un valor de cero, lo que equivale a que esas neuronas no participen en la activación de esa iteración en particular. Este proceso se repite para cada *batch* de datos en cada época.
El objetivo principal de Dropout es evitar que el modelo dependa demasiado de ciertas neuronas o de ciertas conexiones entre ellas. Al forzar a la red a que aprenda representaciones más dispersas y menos dependientes de patrones específicos de las neuronas, Dropout actúa como una forma de regularización que ayuda a prevenir el sobreajuste. Es particularmente útil en redes neuronales profundas, donde existe un mayor riesgo de que el modelo memorice los datos de entrenamiento en lugar de generalizar a datos nuevos.
La tasa de **dropout** (porcentaje de neuronas desactivadas) es un hiperparámetro que se puede ajustar. Un valor típico de dropout está en el rango de 0.2 a 0.5,

lo que significa que entre el 20% y el 50% de las neuronas se apagan durante cada iteración.

8.3 Uso de L2 y Dropout en el Experimento

En este experimento, se utilizaron ambas técnicas para regularizar el modelo y mejorar su capacidad de generalización. Mientras que L2 se aplica a todos los pesos del modelo, asegurando que no se vuelvan demasiado grandes, **Dropout** ayuda a que el modelo sea menos dependiente de las características específicas de las neuronas durante el entrenamiento. La combinación de estas dos técnicas proporciona una forma robusta de evitar el sobreajuste y mejorar el rendimiento del modelo en datos no vistos.

9 Mejor modelo y conclusión

Al aplicar nuestro mejor modelo a los datos de prueba, obtuvimos una precisión de 0.36, lo cual resulta decepcionante considerando los buenos resultados que logramos construir a lo largo del trabajo práctico. En varias ocasiones, los ajustes en las funciones de activación y los cambios en los optimizadores nos permitieron obtener precisiones de validación superiores a 0.50, lo que sugería un gran potencial de mejora. Sin embargo, trabajar en Google Colab presentó algunas dificultades que limitaron nuestros avances. A pesar de haber establecido una semilla, no logramos replicar los resultados de manera consistente, ya que cada nueva ejecución generaba resultados diferentes, afectando la constancia. Además, los servidores de Colab se desconectaban con frecuencia o nos restringían el acceso a las GPU, lo que nos obligaba a reiniciar las ejecuciones desde cero, perdiendo así la base necesaria para optimizar de manera efectiva. Estos obstáculos dificultaron significativamente el desarrollo de una estrategia sólida para mejorar el modelo.