

Tecnología Digital VI: Inteligencia Artificial

Olivier Saint-Nom, Valentina de Diego, Lucas Piñera

Trabajo Práctico 4

Encodeador de Música

Introducción	1
1) Encodear la canción en un vector latente	2
2) Análisis exploratorio de vectores latentes	8
3) Encodear música nueva	16
4) Generación de Música	17
Conclusiones finales del trabajo	23

Introducción

El propósito de este trabajo práctico es explorar la aplicación de técnicas de encodeo de canciones en vectores latentes para su posterior reconstrucción. Utilizaremos la base de datos GTZAN, que contiene 990 canciones de 30 segundos cada una, clasificadas en diez géneros musicales. La tarea principal consiste en diseñar y entrenar una red neuronal capaz de transformar estas canciones en vectores latentes más compactos, con el objetivo de evaluar la calidad de la reconstrucción. Además, se realizarán análisis exploratorios de los vectores latentes para entender las relaciones entre las canciones en este nuevo espacio.

En los primeros pasos, primero establecimos las bases para el trabajo práctico bajando los archivos de las canciones a tratar y estableciendo la tasa de muestreo. Luego, introducimos la clase MusicDataset, diseñada para manejar la carga y estructuración de los datos de la base de datos GTZAN. En esta clase, se recorren los directorios de géneros, se identifican las clases (géneros) presentes y se aplica una transformación de espectrograma a los datos. También

dividimos el conjunto de datos en conjuntos de entrenamiento, validación y prueba. Esto lo hicimos con una semilla aleatoria para garantizar la reproducibilidad y determinando tamaños para los conjuntos de validación y prueba. Creamos dataloaders para los conjuntos de entrenamiento, validación y prueba, estableciendo un tamaño de lote y configurando opciones para la carga eficiente de datos.

1) Encodear la canción en un vector latente

En esta primera instancia, se aborda la construcción de una red neuronal dedicada al encodeo de canciones en vectores latentes. La red que buscamos crear es capaz de reconstruir las canciones a partir de estos vectores, manteniendo su estructura original. Para esto, llevamos a cabo experimentos con diferentes tamaños de vectores latentes, buscando así el tamaño más pequeño posible.

Desarrollamos una estructura de autoencoder diseñada para la tarea de procesamiento de datos musicales. La arquitectura se basa en un codificador (`Encoder`) y un decodificador (`Decoder`). Estas clases incorporan capas de convolución y deconvolución respectivamente, configuradas con hiperparámetros como el número de canales, tamaños de kernel, pasos de stride y de padding. La clase final, llamada `Autoencoder`, combina estas dos partes para formar el modelo completo. En esta inicialización del modelo, incluimos una ejecución de prueba `dummy_input` utilizando un tensor aleatorio para comprender las dimensiones del bottleneck y el tamaño de la salida, y facilitarnos el proceso de experimentación.

En la fase de entrenamiento, utilizamos datos de entrenamiento y validación usando la función de pérdida Mean Squared Error (`MSELoss`) y el optimizador Adam para impulsar el proceso de aprendizaje. El modelo luego se entrena a través de múltiples épocas, con el modelo más eficaz siendo guardado basándose en la pérdida en el conjunto de validación.

La evaluación del modelo se lleva a cabo en el conjunto de prueba, calculando la precisión alcanzada. Se carga el modelo óptimo y se realiza una demostración visual del proceso de reconstrucción utilizando un ejemplo de audio y su correspondiente spectrograma.

Una vez que la estructura base de nuestro modelo estaba armada, pasamos a la parte de experimentación de hiper parámetros para analizar cómo estos afectaban el espacio latente del autoencoder y, por ende, la calidad de la reconstrucción musical. El objetivo principal fue encontrar el conjunto óptimo que resultara en una representación compacta, y en la reconstrucción de los audios más precisa.

Ajustamos variables cruciales como el número de canales (c_1 , c_2 , c_3 , c_4), los tamaños de kernel (k_1 , k_2 , k_3 , k_4), los pasos de stride (s_1 , s_2 , s_3 , s_4), y los valores de padding (p_1 , p_2 , p_3 , p_4). Cada uno de los parámetros indicando a qué capa de convolución pertenecía. Además, modificamos la tasa de aprendizaje (lr), el número de épocas (num_epochs), y el tamaño del lote (batch_size).

Por otro lado, probamos ambas estructuras simétricas y asimétricas con respecto a los hiper parámetros utilizados en las convoluciones. Cuando probamos la estructura asimétrica, aunque facilitaba mucho los cálculos manuales de las dimensiones (descrito posteriormente), por lo general daba peores resultados. Por esto, decidimos inclinarnos hacia una estructura simétrica, en el sentido de que los hiper parámetros usados por el encoder son los mismos que en el decoder, espejados.

Este proceso de ajuste se realizó repetidamente, evaluando cómo cada cambio impacta en la calidad de la representación latente y en la reconstrucción del audio. Buscábamos particularmente encontrar el conjunto de hiperparámetros que permitiera lograr la representación más compacta, minimizando el tamaño del vector latente, como se indica en la consigna. Además de eso, también buscábamos garantizar que la reconstrucción resultante sea auditivamente coherente y respete la estructura original de la canción.

Para ir guardando las diferentes combinaciones y sus respectivos resultados con la red neuronal, creamos un Weights and Biases para visualizar las pérdidas de cada una de las pruebas realizadas y a su vez los hiperparámetros correspondientes.

Hiperparámetros

Durante la prueba de combinaciones de hiper parámetros, definitivamente encontramos un obstáculo ya que estos valores afectan el tamaño del output cada vez que atravesaba una capa. Por esta razón, seguimos las siguientes fórmulas para verificar que las modificaciones en las dimensiones que se hacían en cada capa convolucional y deconvolucional sean las correctas:

$$\text{output común} = (\text{input_size} + 2\text{padding} - \text{kernel})/\text{stride} + 1$$

$$\text{output traspuesta} = (\text{input_size} - 1) * \text{stride} - 2\text{padding} + \text{kernel}$$

De esta manera, pudimos encontrar combinaciones de los hiper parámetros stride, padding, channels y kernel que no generaban problemas dimensionales en la red neuronal.

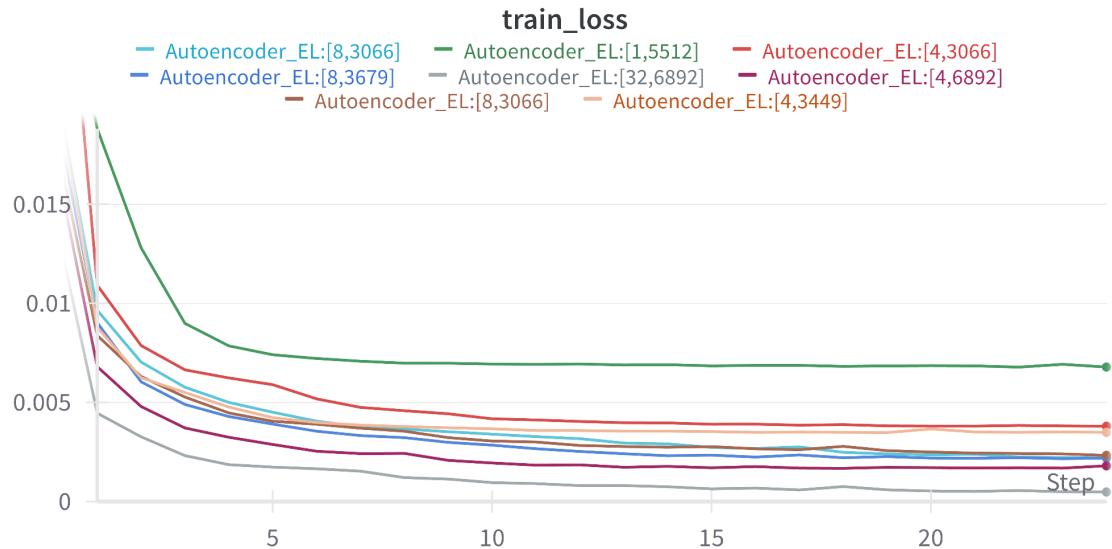


Figura 1 - Media de pérdida según épocas de las arquitecturas con los datos de entrenamiento. El nombre se compone del tipo de arquitectura + EL(espacio latente)

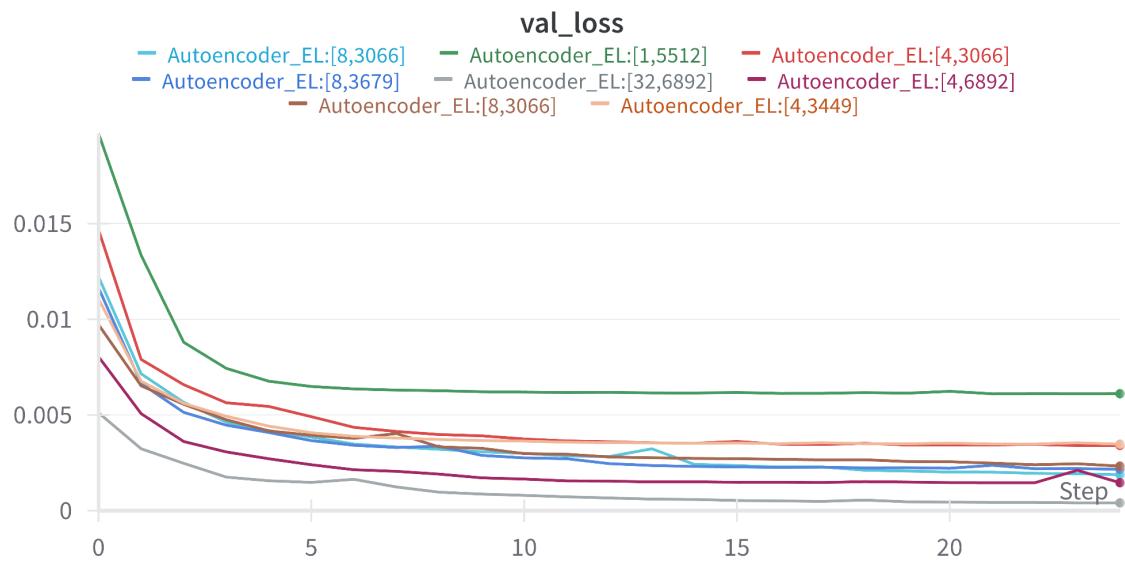


Figura 2 - Media de pérdida según épocas de las arquitecturas con los datos de validación.

Viendo los resultados en Weights and Biases mostrados por la *Figura 1* y *Figura 2*, pudimos ver, como esperábamos, que aquellas que tienen mayor espacio latente, tienen menos pérdida, ambos en entrenamiento y en evaluación. Esto se da porque un espacio latente más grande tiene más dimensiones para codificar información y puede preservar mejor la información esencial durante el proceso de codificación y decodificación. Por otro lado, aquellos intentos de espacios latentes muy chicos, muestran una pérdida muy alta y una reconstrucción del audio muy robusta y poco precisa. Es por esto que de nuestros intentos, buscábamos el tamaño más chico posible que pueda igual ofrecer una reconstrucción aceptable de los audios.

Arquitectura elegida: Autoencoder_EL:[8, 3066]

Name	c_1	c_2	c_3	c_4	k_1	k_2	k_3	k_4	p_1	p_2	p_3	p_4	s_1	s_2	s_3	s_4	val_loss	train_loss
Autoencoder_EL:[1,5512]	64	32	8	4	8	6	5	2	4	4	4	4	6	3	2	3	0,006110	0,006785
Autoencoder_EL:[32,6892]	64	32	32	4	6	4	4	2	4	3	1	2	4	2	2	3	0,000404	0,000481
Autoencoder_EL:[4,3066]	64	32	4	4	8	6	5	2	4	4	4	4	6	3	2	3	0,003405	0,003798
Autoencoder_EL:[4,3449]	64	32	4	8	6	6	5	2	4	5	4	4	4	4	2	2	0,003472	0,003491
Autoencoder_EL:[4,6892]	64	32	4	4	6	4	4	2	4	3	1	2	4	2	2	3	0,001458	0,001802
Autoencoder_EL:[8,3066]	64	32	8	4	8	6	5	2	4	4	4	4	6	3	2	2	0,002325	0,002329
Autoencoder_EL:[8,3066]	64	32	8	4	8	6	5	2	4	4	4	4	6	3	2	3	0,001877	0,002199
Autoencoder_EL:[8,3679]	64	32	8	4	8	6	4	2	4	4	4	4	5	3	2	2	0,002161	0,002188

Figura 3 - Tabla de parámetros en las arquitecturas con sus valores de pérdida en validación y entrenamiento

El modelo seleccionado es el resaltado en la *Figura 3*. Este modelo de 8 canales con longitud de 3066, indica que el espacio latente más pequeño posible logrado para que no pierda su estructura, es de 24528, reduciendo a un 22% del tamaño de la canción original. Lo elegimos porque era el más chico posible que sea capaz de reconstruir el audio de manera precisa y de buena calidad, llegando a una pérdida en el conjunto de evaluación con 25 epochs de 0.001877.

Funciones de activación

Para entrenar nuestra red neuronal, probamos diferentes enfoques en cuanto a las funciones de activación. Para empezar, la función de activación relu directamente fue descartada por su desventaja al no tratar adecuadamente los valores negativos, ya que nuestro conjunto de datos contiene valores, como imagen [-1,1]. Luego, probamos con la función de activación **Silu**, **Tanh** y **Leaky Relu**, dandonos los siguientes rendimientos:

Se puede ver que en validación, la función de activación Leaky Relu dio un resultado de menor pérdida luego de 25 epochs. Además, como se muestra en la *Figura 4*, se notaba una mejor calidad en los audios con Leaky Relu, en comparación con las otras funciones de activación, por lo que elegimos esta para ser la función de activación usada por nuestra red neuronal.

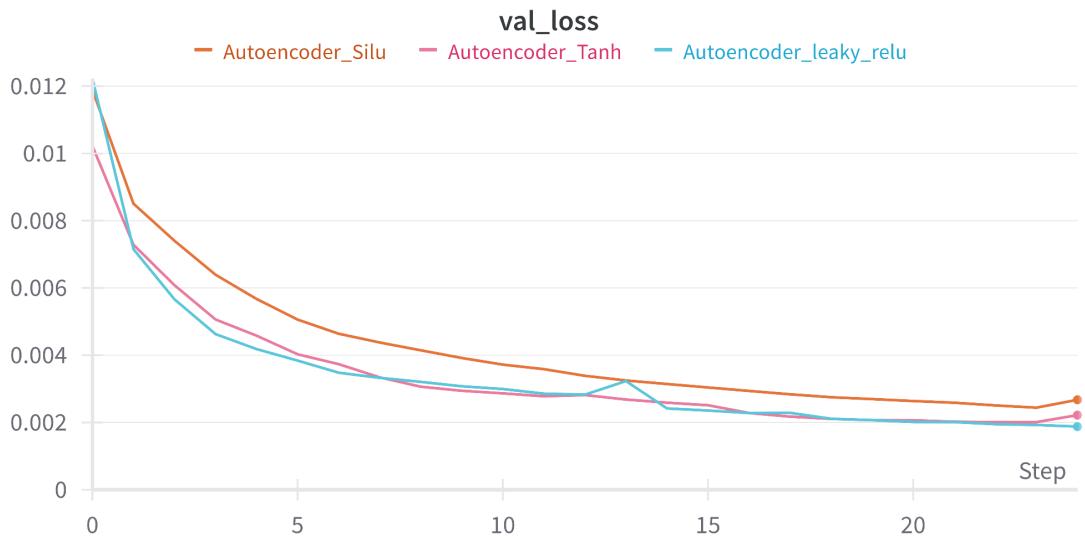


Figura 4 - Comparación de pérdidas de validación experimentando con 3 funciones de activación que permiten el manejo de valores negativos.

Por último, habiendo encontrado la mejor combinación de hiperparametros, con el espacio latente más chico posible y la mejor función de activación en cuanto a rendimiento, el audio reconstruido resultó tener la siguiente forma (*Figura 5*).

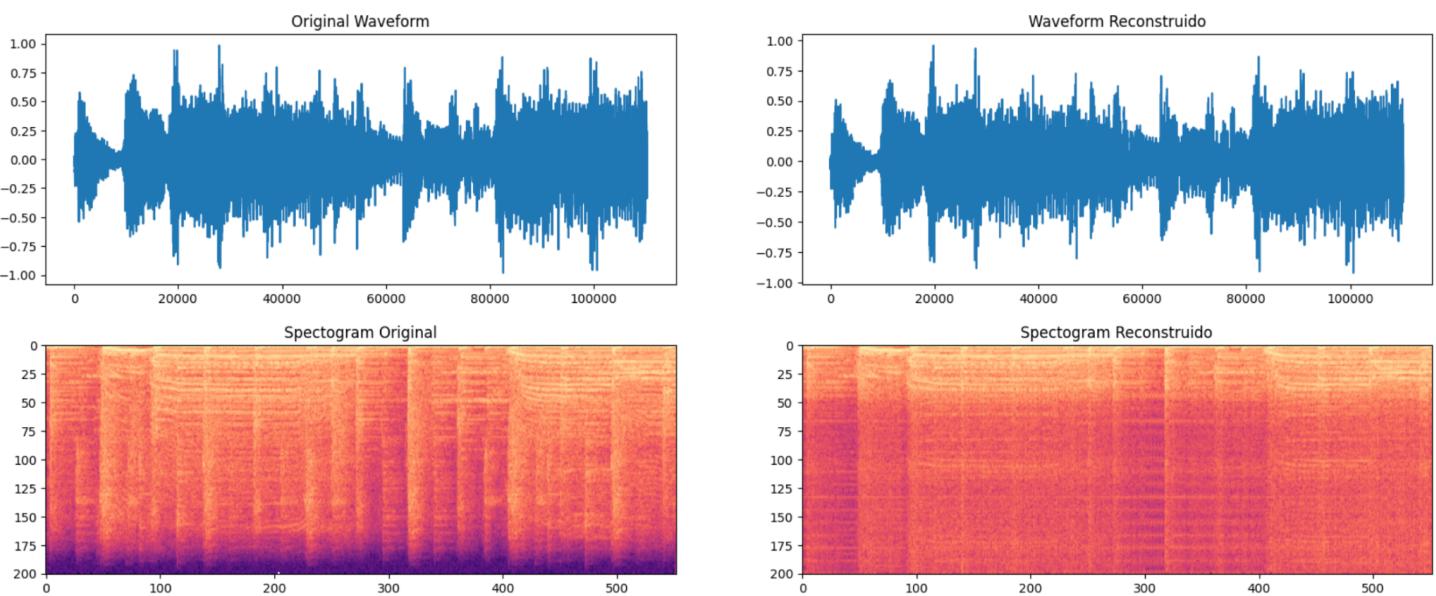


Figura 5 - Representación visual de la reconstrucción de una canción con la arquitectura ganadora con un espacio latente de 24528.

2) Análisis exploratorio de vectores latentes

En este inciso, buscamos explorar las relaciones entre las canciones únicamente en el espacio latente. En este sentido, realizamos análisis para cuatro tamaños de vectores: el más pequeño propuesto encontrado en el primer inciso, uno pequeño (la mitad), uno extra pequeño (un octavo) y uno más grande (el doble). Haciendo esto, pudimos encontrar relaciones entre los vectores y ver como la resolución afecta estas relaciones. En esta etapa hicimos 2 experimentaciones, clasificación y clustering, en las cuales vimos su rendimiento y su potencial según el tamaño del espacio latente. Para lograr esto seguimos los siguientes pasos de implementación:

Importación de Modelos y Entrenamiento como Autoencoders:

En esta fase, importamos cuatro modelos diferentes, cada uno hecho para aprender representaciones latentes de diferentes tamaños: `extra_chico` ($X1/4$), `chico` ($X1/2$), `normal` ($X1$), y `grande` ($X4$). Estas medidas fueron creadas multiplicando el espacio latente mínimo obtenido en el ejercicio 1, por $\frac{1}{4}$, $\frac{1}{2}$, 1 y 4 respectivamente. Es decir:

- `extra_chico` ($X1/8$) → Tamaño = $\frac{1}{4} \cdot 24528 = 6132$ → 5% del tamaño de la canción.
- `chico` ($X1/2$) → Tamaño = $\frac{1}{2} \cdot 24528 = 12264$ → 11% del tamaño de la canción.
- `normal` ($X1$) → Tamaño = $1 \cdot 24528 = 24528$ → 22% del tamaño de la canción.
- `grande` ($X4$) → Tamaño = $4 \cdot 24528 = 98112$ → 88% del tamaño de la canción.

Para estos modelos, entrenamos el auto encoder y nos quedamos con el espacio latente. Esto fue hecho una única vez y los guardamos con el nombre de `modelo_{tamaño}_2.ckpt` para agilizar el proceso. Una vez guardados los modelos, pudimos empezar a experimentar con ellos y ver su rendimiento.

- **Clasificación:**

En esta etapa de análisis, nos focalizamos en crear modelos de clasificación de los espacios latentes, aprendiendo según su género a través de los siguientes pasos:

1. *Creación de DataLoaders:*

En este paso, cargamos los modelos preentrenados y los aplicamos a conjuntos de datos de entrenamiento, validación y prueba. La salida del encoder (el espacio latente codificado), se utiliza para crear nuevos DataLoaders, de manera que tengan una forma adecuada para la entrada del clasificador en el siguiente paso.

2. *Clasificación:*

Aca definimos un modelo de clasificación (M5 con 4 capas convolucionales) y se entrena para cada tamaño de modelo (chico, normal, grande). El objetivo es evaluar si los espacios latentes aprendidos son informativos y discriminativos para clasificar géneros musicales. Después analizamos la precisión de esta clasificación en el conjunto de prueba, para evaluar la calidad de las representaciones latentes en términos de su utilidad para la tarea de clasificación.

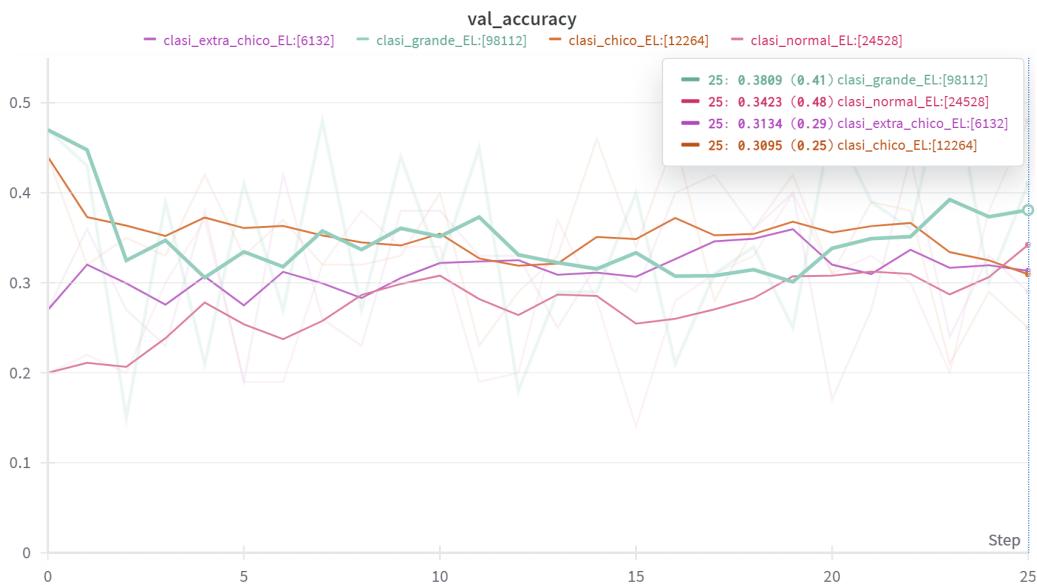


Figura 6: Accuracy de los cuatro modelos a lo largo de las epochs realizadas en el conjunto de datos de validación.

3. Evaluación:

Para ver el rendimiento de la clasificación, vimos cómo se desempeñaba el modelo en el conjunto de datos de test. Pudimos notar que la clasificación de los espacios latentes de todos los tamaños tuvieron una precisión por debajo del 50%, lo que significa que los espacios latentes de los audios no son lo suficientemente representativos para su género.

Tamaño	Precisión
Extra Chico	38.00%
Chico	46.00%
Normal	40.00%
Grande	47.00%

Figura 7: Tabla de modelos según el tamaño del espacio latente y las precisiones alcanzadas

Como se muestran en los resultados de las *Figura 6 y 7*, está claro que el tamaño del espacio latente tiene un efecto notable en la precisión de clasificación del modelo. La precisión del modelo grande es ampliamente más precisa que el modelo extra chico. Sin embargo, nos pareció interesante ver que el chico obtuvo una precisión mayor al normal, lo cual nos indica que nuestro modelo inicial se podría haber achicado aún más.

• Clustering:

En esta fase, realizamos clustering en el espacio latente utilizando el algoritmo K-Means en conjunto de datos de train, el cual tenía 790 canciones de las 990 del dataset total. El objetivo fue agrupar automáticamente las muestras de audio en categorías similares según sus representaciones latentes. Utilizamos PCA (*Figuras 8.1 a 8.4*) para poder reducir las dimensiones en 2D y 3D para luego realizar clustering y calcular métricas como homogeneidad y completitud para evaluar la calidad del agrupamiento.

Realizamos dos representaciones distintas para visualizar su rendimiento, la primera en forma de 3D y otra en 2D. Tuvimos en cuenta que es muy difícil para vectores con muchos parámetros como estos, poder ser representados en un espacio tan compacto, por lo que no esperábamos

grandes desempeños. Algo a destacar tras hacer esta visualización, es que a mayor tamaño del espacio latente, más separadas estaban las canciones, pudiendo ver a simple vista que tener más espacio latente mejora la habilidad para separar. Aun así, quisimos explorar la posibilidad y esencialmente conocer las diferencias entre el agrupamiento a través de K-Means y el agrupamiento por género.

Usamos distintos valores de cantidad de cluster para dividir y comparar su rendimiento. Ya que las canciones en nuestro Dataset se dividían en 10 géneros destinos, decidimos partir de ese número para empezar a analizar las distintas formas de clusterizar. La cantidad de clusters se definió a partir de `n_clusters_values = [10, 12, 15, 20, 25, 30]`.

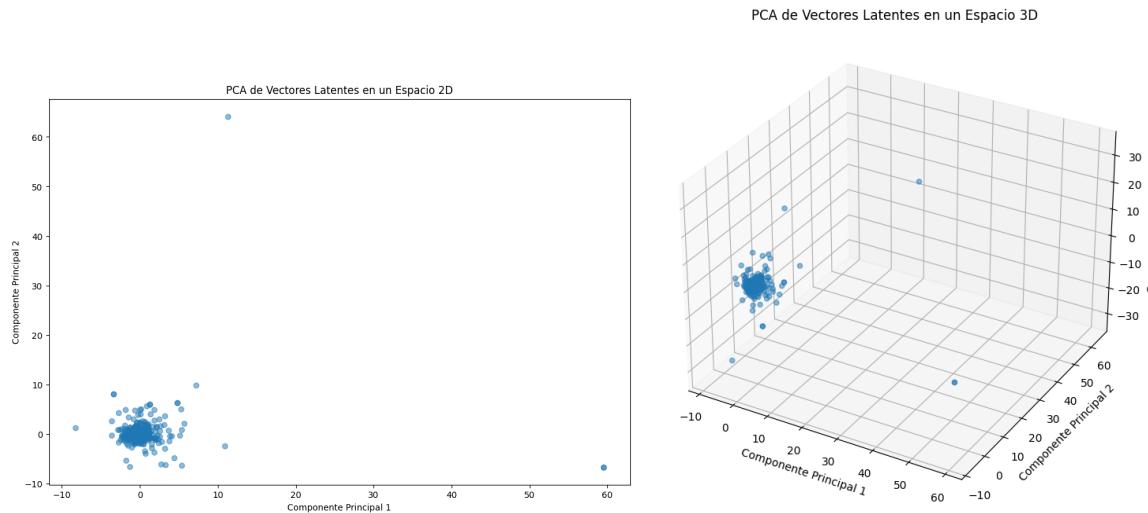


Figura 8.1: PCA 2D y 3D del modelo extra chico

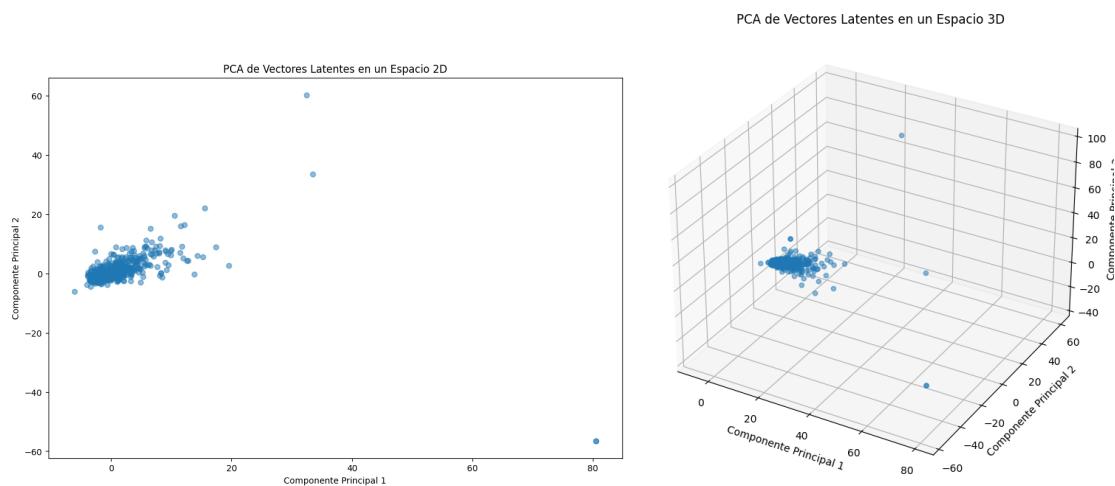


Figura 8.2: PCA 2D y 3D del modelo chico

PCA de Vectores Latentes en un Espacio 3D

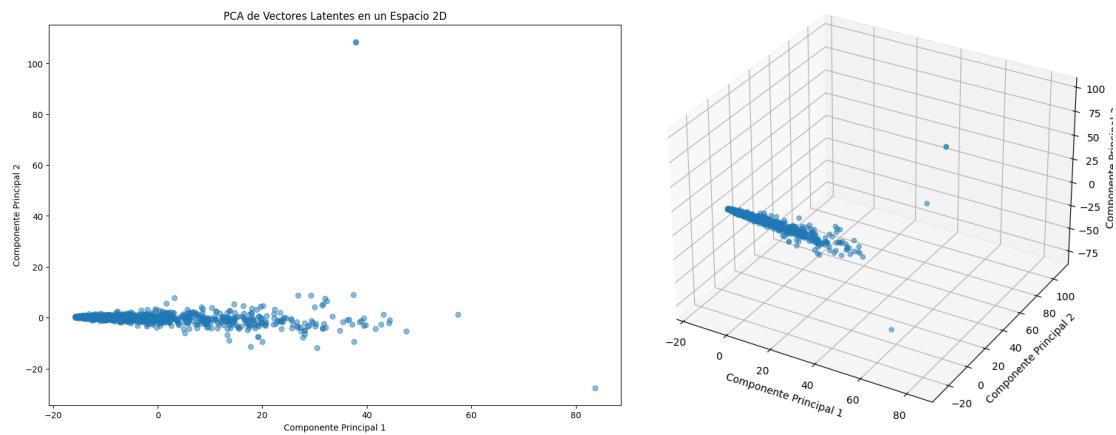


Figura 8.3: PCA 2D y 3D del modelo normal

PCA de Vectores Latentes en un Espacio 3D

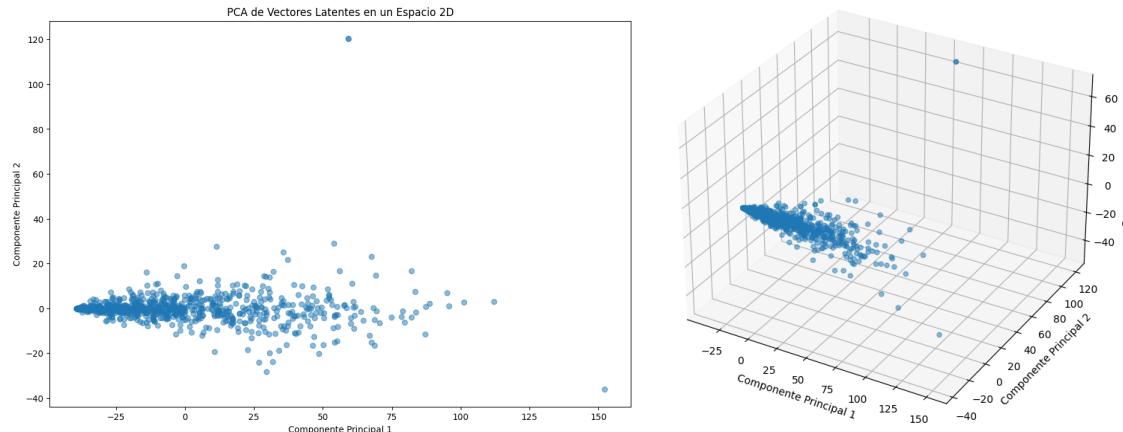


Figura 8.4: PCA 2D y 3D del modelo grande

Luego de realizar el análisis mediante técnicas de clustering, pudimos ver que las representaciones en 2D y en 3D no presentaron mucha diferencia. Por otro lado, vimos una mejora leve cuando se trataba de espacios latentes mayores, aunque no es la diferencia que esperábamos dada la brecha grande de tamaño de los espacios latentes.

extra_chico			
	Clusters	Homogeneidad	Compleitud
0	10	0.068248	0.136322
1	12	0.089295	0.137480
2	15	0.096019	0.141353
3	20	0.114296	0.137983
4	25	0.128286	0.133962
5	30	0.150260	0.141061
chico			
	Clusters	Homogeneidad	Compleitud
0	10	0.071155	0.120756
1	12	0.097297	0.131249
2	15	0.115083	0.137068
3	20	0.128585	0.125701
4	25	0.145224	0.136851
5	30	0.164846	0.142820
normal			
	Clusters	Homogeneidad	Compleitud
0	10	0.102199	0.128795
1	12	0.113034	0.135700
2	15	0.128026	0.132996
3	20	0.141039	0.130855
4	25	0.155336	0.127687
5	30	0.164713	0.131733
grande			
	Clusters	Homogeneidad	Compleitud
0	10	0.094459	0.115641
1	12	0.098588	0.110401
2	15	0.115303	0.113926
3	20	0.128585	0.117677
4	25	0.144820	0.120917
5	30	0.159581	0.122248

Figura 9.1: Métricas de homogeneidad y completitud para los clusters realizados en 2D

extra_chico			
	Clusters	Homogeneidad	Compleitud
0	10	0.058213	0.138551
1	12	0.067618	0.141240
2	15	0.080001	0.137310
3	20	0.099151	0.128628
4	25	0.119021	0.140166
5	30	0.140617	0.146497
chico			
	Clusters	Homogeneidad	Compleitud
0	10	0.072329	0.122916
1	12	0.076839	0.118436
2	15	0.103626	0.131935
3	20	0.112462	0.127981
4	25	0.133043	0.134613
5	30	0.149501	0.137004
normal			
	Clusters	Homogeneidad	Compleitud
0	10	0.085908	0.120907
1	12	0.095406	0.126429
2	15	0.119773	0.133350
3	20	0.127363	0.122555
4	25	0.143468	0.130877
5	30	0.159962	0.131659
grande			
	Clusters	Homogeneidad	Compleitud
0	10	0.082552	0.101963
1	12	0.087719	0.102688
2	15	0.107897	0.114004
3	20	0.121216	0.112075
4	25	0.140502	0.117524
5	30	0.156268	0.120984

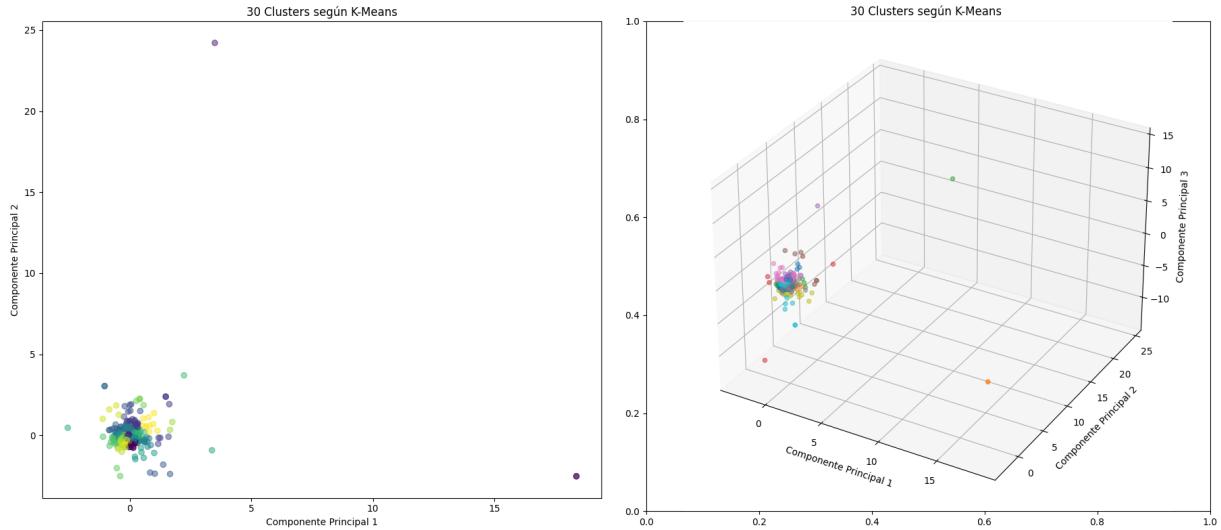
Figura 9.2: Métricas de homogeneidad y completitud para los clusters realizados en 3D

En general, pudimos ver que la agrupación en 10 clusters, a pesar de ser la misma cantidad que los géneros en nuestros datos, es la agrupación con peor rendimiento, posiblemente debido a la falta de relación en los espacios latentes con el género del mismo. Además, notamos que al aumentar la cantidad de cluster, la homogeneidad muestra un aumento sutil, pero esto no sucede con la completitud. Esta situación puede explicarse por el hecho de que, al aumentar el número de clusters, estos tienden a agruparse en conjuntos más específicos, logrando así una mayor homogeneidad dentro de cada grupo, aunque no sean del mismo género musical.

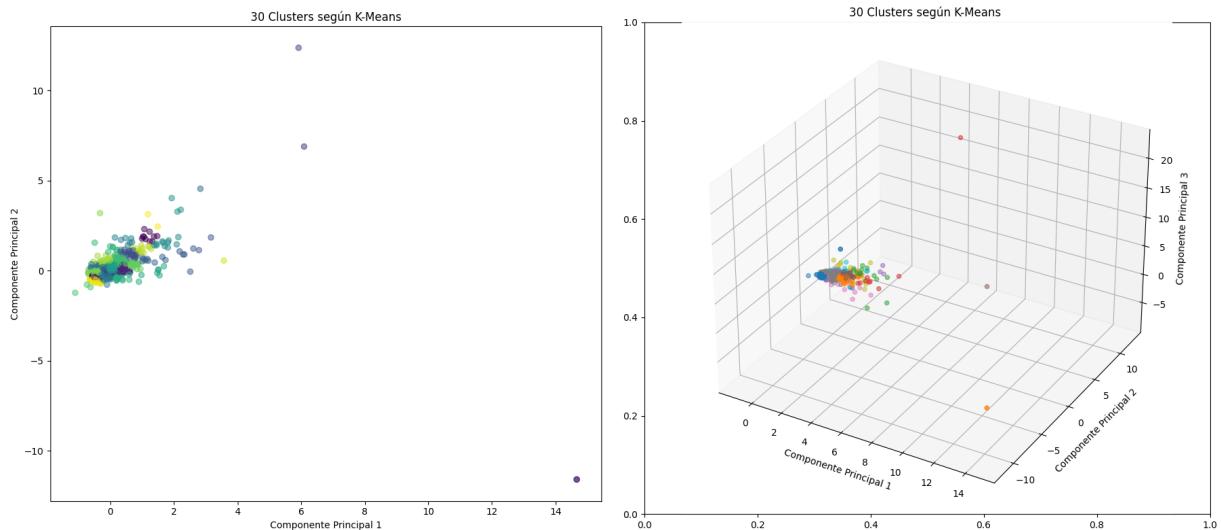
Aun así, se puede observar que en general, el modelo de tamaño normal y en especial el grande muestran una leve mejora en los resultados en cada instancia. Esto sugiere que contar con un espacio latente más amplio puede ofrecer una caracterización más efectiva de los géneros, y por ende una mejor formación de clusters.

A continuación mostramos las agrupaciones que dieron mejores resultados para cada tamaño realizado:

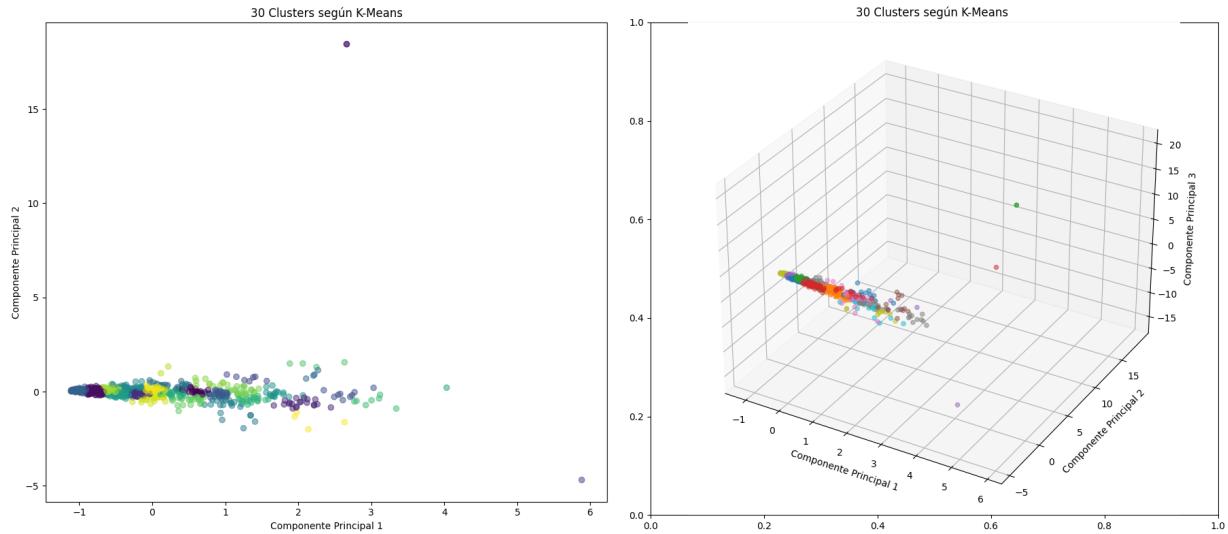
`extra_chico` con agrupación de 30 clusters:



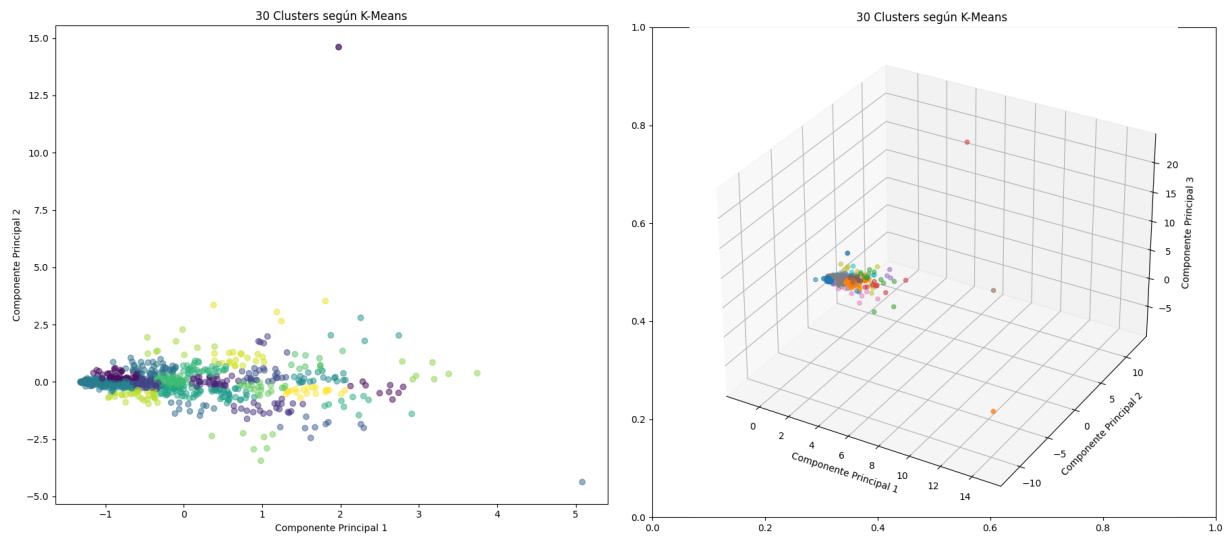
`chico` con agrupación de 30 clusters:



`normal` con agrupación de 30 clusters:



grande con agrupaci\'on de 30 clusters:



3) Encodear m\'usica nueva

En este ejercicio, trabajamos en la tarea de encodear m\'usica nueva, es decir, en la capacidad del modelo de autoencoder para procesar y representar canciones nuevas que no forman parte del conjunto de datos original. Para realizar esto, decidimos probar con cuatro canciones pertenecientes a cuatro g\'eneros distintos (hip hop, disco, jazz y pop) para obtener una representaci\'on amplia de la m\'usica.

Primero de todo, nos encontramos con el obstáculo de ver que cuando analizamos canciones externas a nuestro conjunto de datos, el sample rate era diferente y por ende la reproducción final aparecía fallida. Por eso creamos una función llamada `sample_rate` que ajusta el sample rate de una canción nueva al valor deseado (22050 Hz). Esto fue esencial para que la canción tenga la misma tasa de muestreo que las canciones del conjunto de datos original. Aparte del sample rate, también tuvimos que ajustar la cantidad de canales de las canciones nuevas, ya que el formato de entrada que espera nuestro modelo es de 1 canal, y las canciones que descargamos tenían 2 canales. Para esto creamos una función `channel` que se encarga de cargar una canción nueva utilizando `torchaudio` y, si tiene más de un canal, reduce la cantidad de canales a 1 tomando el promedio. Ambas funciones fueron necesarias para que los audios externos del dataset dado coincidan con el formato de entrada que espera el modelo entrenado.

Luego, solo importamos el modelo previamente entrenado y cargamos la canción nueva seleccionada que fue ajustada en términos de sample rate y channel, y evaluamos el rendimiento.

La performance del modelo en datos no conocidos es la misma que con los datos del dataset inicial, siempre y cuando la arquitectura del autoencoder sea compatible y respete sus restricciones de entrada, como los canales y el sample rate. Al escuchar los audios de estas nuevas canciones luego de ser comprimidos y descomprimidos, notamos una buena calidad de reconstrucción, demostrando buenos resultados de nuestra arquitectura.

4) Generación de Música

En esta parte de la exploración, nos centramos en la generación de música utilizando el modelo de autoencoder entrenado en vectores latentes de canciones. Iniciamos cargando el mejor modelo obtenido durante el entrenamiento y definimos una función para obtener los vectores latentes y etiquetas de género de los conjuntos de datos de entrenamiento, validación y prueba.

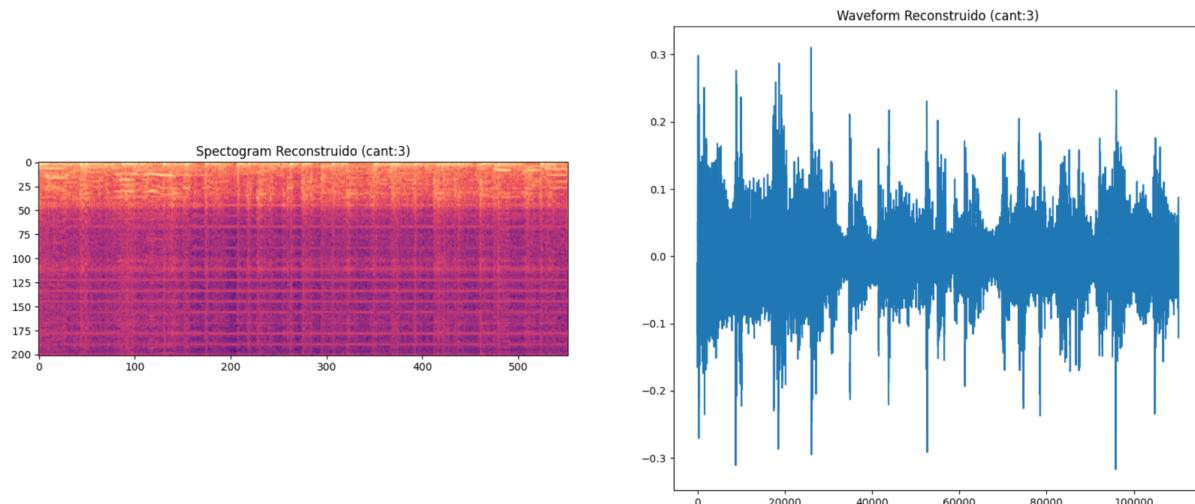
1. Promedio de canciones

Para hacer esta primera experimentación, separamos los vectores latentes por género, específicamente para el género de reggae, que son los géneros que queríamos probar combinar. Acá, realizamos un chequeo de la calidad de la reconstrucción decodificando y reproduciendo dos ejemplos de audios a partir de los vectores latentes.

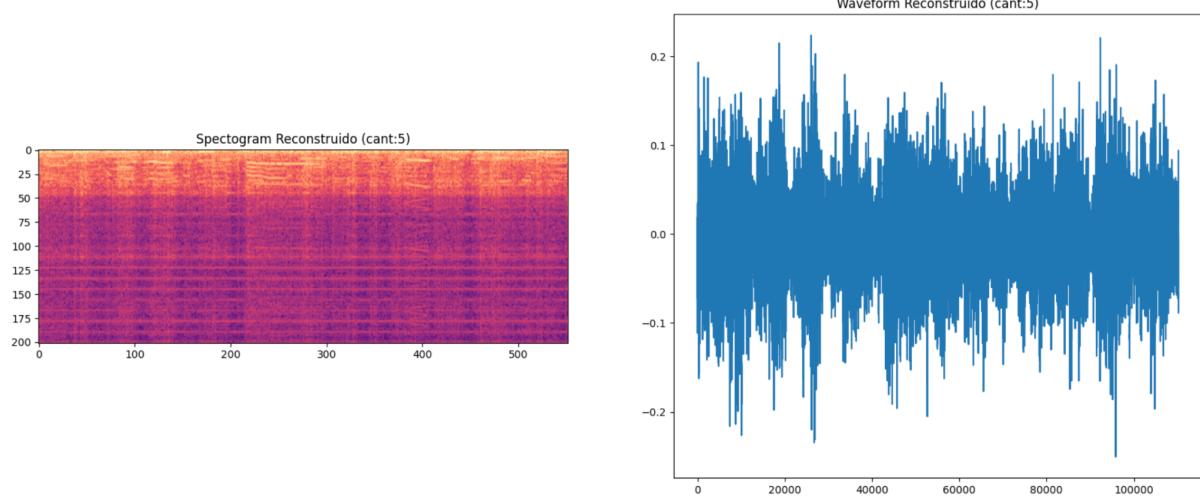
Luego definimos una función para realizar operaciones con vectores latentes, que en este caso era promediar combinaciones de vectores. Esta función es `promedio_combinaciones_latentes(lista, x)` que toma una lista de vectores latentes y realiza el promedio de los primeros x vectores de la lista. A partir de los vectores latentes del género reggae, generamos nuevos vectores promediando diferentes cantidades de ellos. Específicamente, tratamos de combinar 3 canciones, 5 canciones, 10 canciones, 50 canciones y 70 canciones.

Finalmente, cargamos el modelo nuevamente y decodificamos los vectores latentes promediados de todas las combinaciones mencionadas, reproduciendo los audios generados.

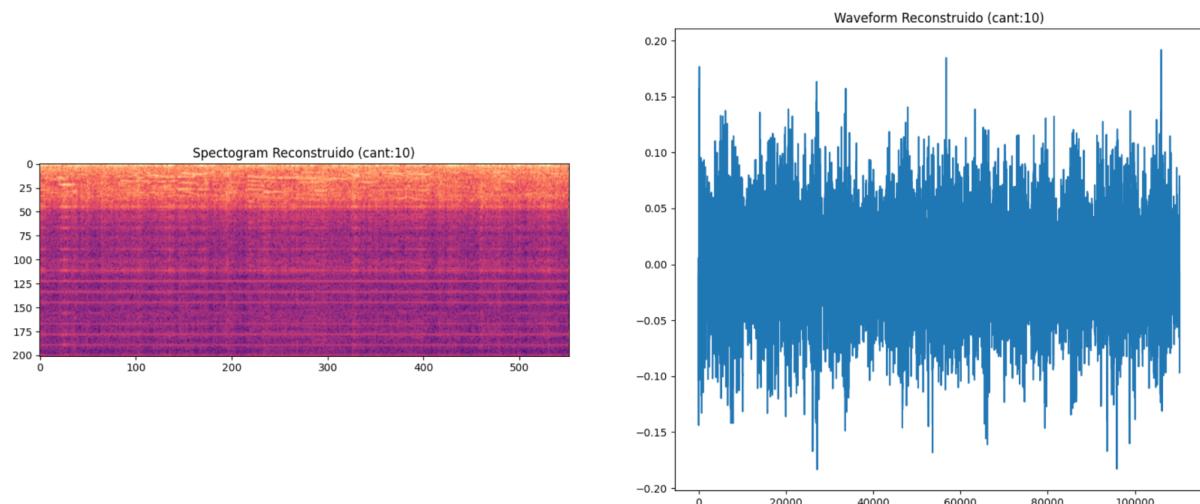
Combinación de 3 canciones:



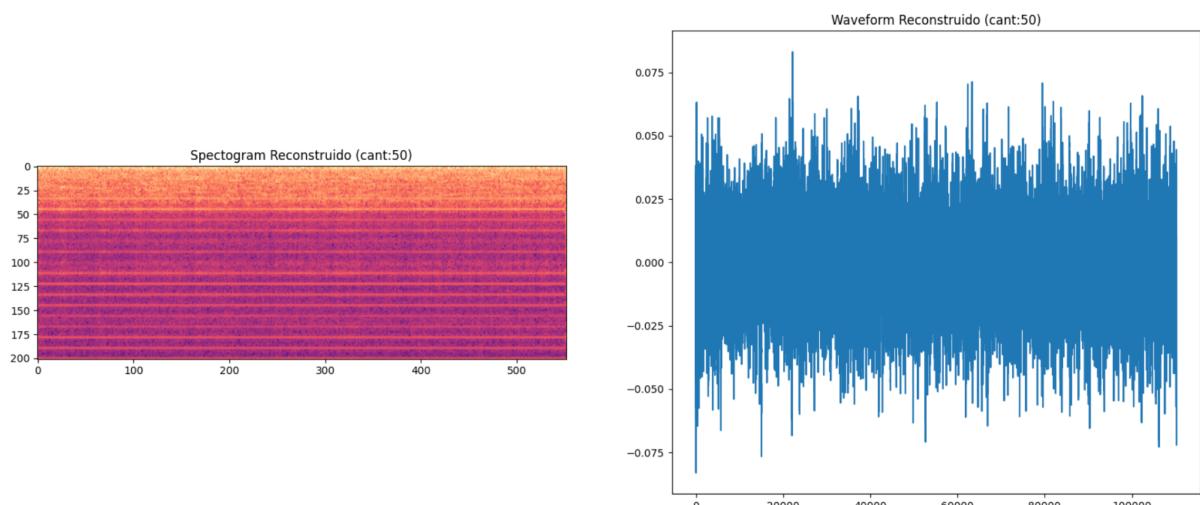
Combinación de 5 canciones:



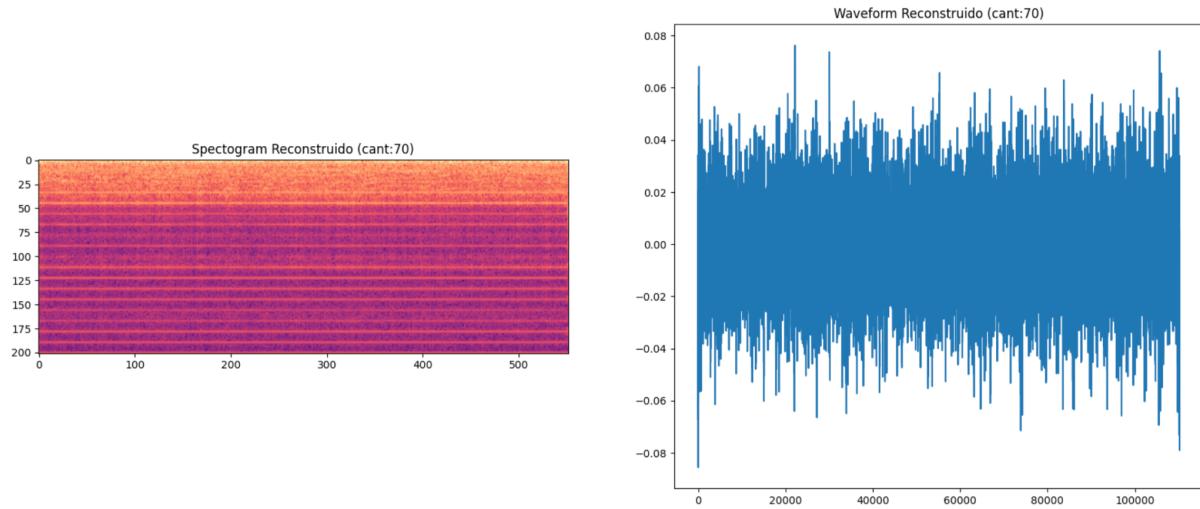
Combinación de 10 canciones:



Combinación de 50 canciones:



Combinación de 70 canciones:



Pudimos notar que a medida que aumentamos la cantidad de canciones promediadas, el resultado final decrece en coherencia musical. Esto se ve a través de los espectrogramas generados, notando que a medida que más canciones se combinan, el audio generado presenta menos características marcadas. Cuando se promedian los vectores latentes de varias canciones, se está realizando una especie de fusión de información de diferentes fuentes musicales, lo cual puede no ser coherente al oído. Al agregar más canciones, es posible que se introduzcan elementos incompatibles que, al promediarse, resulten en un sonido difuso que no representa claramente ninguna de las canciones originales. Esto puede deberse a que las características específicas de cada canción, que podrían ser distintivas y significativas por separado, se diluyen o cancelan entre sí al promediarse.

Es por esto que los resultados más coherentes que recibimos fue el de combinar 3 canciones de reggae donde se podían apreciar los rasgos característicos de las 3 canciones, seguido por el resultado de combinar 10 canciones, siendo menos coherente pero igual preservando algunas características musicales.

2. Concatenación de canciones

Como segundo intento de generación de música, decidimos probar una manera menos invasiva al manipular canciones, y combinarlas separadamente. Para hacer esto, agregamos los vectores latentes del género country para combinar con los que ya teníamos de reggae.

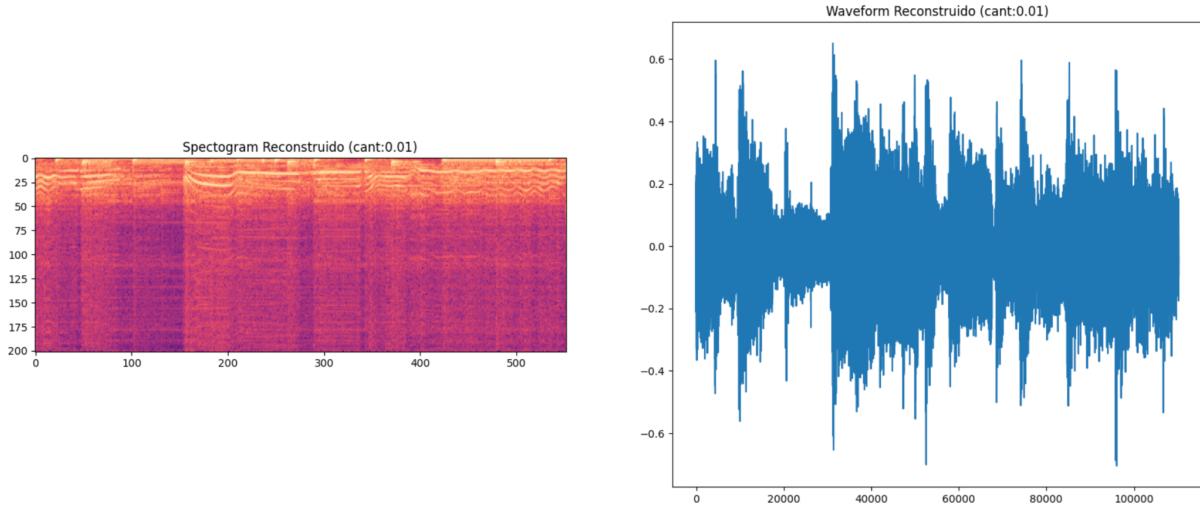
Tomamos la mitad de cada vector latente generado a partir de las formas de onda codificadas de country y reggae (`half_encoded_country` y `half_encoded_reggae`, respectivamente). Luego concatenamos estos vectores latentes y formamos un nuevo vector `concatenated`, que luego se decodifica para formar el resultado final. De esta forma, quisimos imitar una especie de ‘remix’ entre dos canciones de diferentes géneros, cuyos resultados mantuvieron la esencia musical de ambas canciones al preservar sus características.

3. Suma de ruido a las canciones

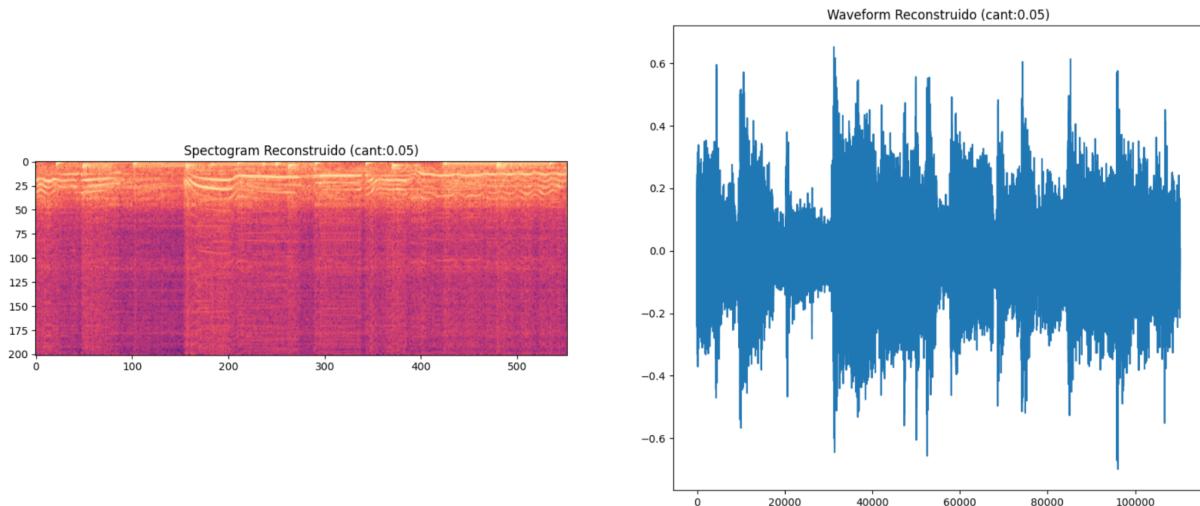
Por último, decidimos probar como se podía generar nueva música a partir de agregar ruido a canciones. Para hacer esto, generamos una función que tomaba un vector latente y un valor numérico y los sumaba. Luego, este mismo espacio latente nuevo, al ser decodeado, resultaba en una nueva canción con un ruido agregado. Esto lo probamos con cuatro diferentes valores de ruido: 0.01, 0.05, 0.1 y 0.5.

Tras escuchar los resultados a partir de esta experimentación, vimos que a medida que aumentaba el valor del ruido, más disruptivo se escuchaba el audio resultante, lo cual era lo esperable. Esto sucede debido a que con mayor cantidad de ruido, los elementos especiales que componen las canciones cada vez se ven más opacadas por ruido sin sentido, generando así música sin sentido y con poca coherencia musical. Estos efectos se pueden ver claramente en los espectrogramas generados a continuación.

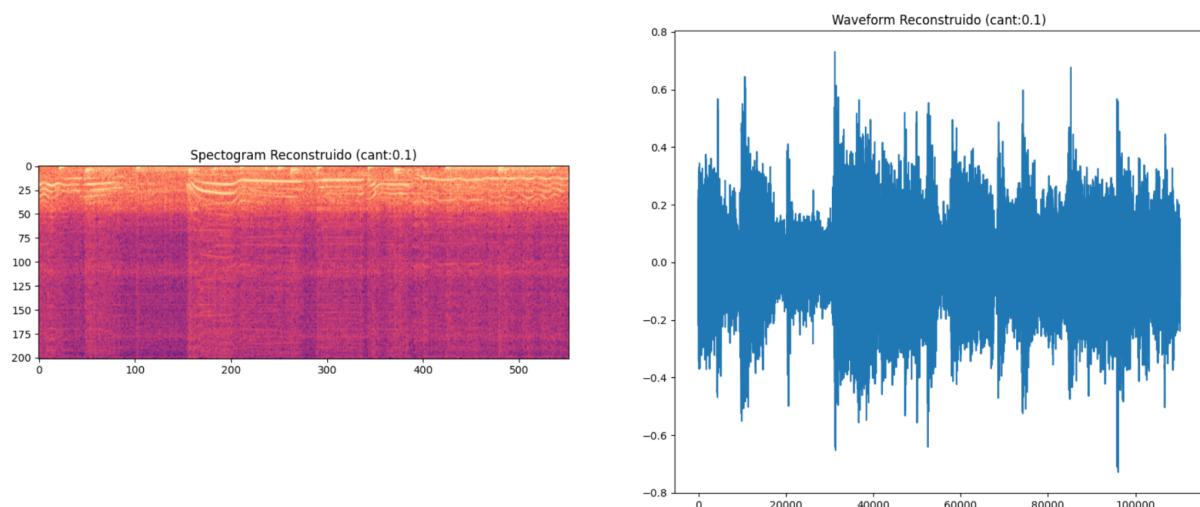
Ruido de 0.01:



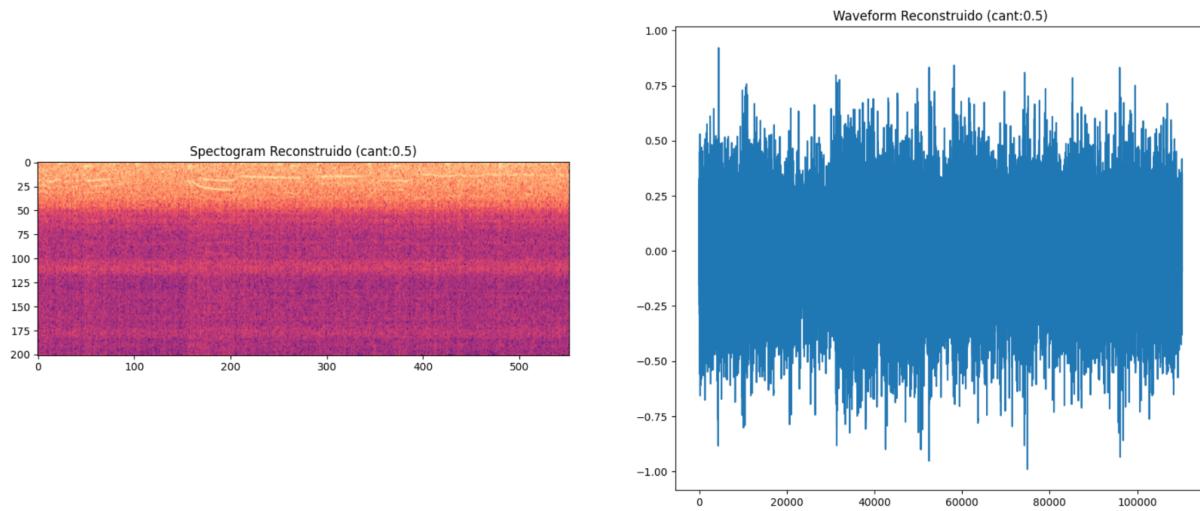
Ruido de 0.05:



Ruido de 0.1:



Ruido de 0.5:



Conclusiones finales del trabajo

En este trabajo, exploramos técnicas de encodeo musical mediante autoencoders. Diseñamos y entrenamos un modelo para transformar canciones en vectores latentes, buscando el equilibrio entre representación compacta y reconstrucción precisa. Experimentamos con hiperparámetros, destacando el modelo "Autoencoder_EL: [8, 3066]" como el óptimo con un espacio latente mínimo (24528).

Realizamos análisis exploratorios en los espacios latentes para ver su comportamiento dependiendo de su tamaño. Probamos 3 variantes a nuestra propuesta original y concluimos que al menor espacio latente, mayor era la pérdida de definición, ya que tienen menor capacidad de representación y falta de características que las relacionan. Comprobamos estos hallazgos utilizando dos experimentos distintos, uno a través de clasificación con una red neuronal convolucional y otro con el método de K-means de clustering.

Además, abordamos la posibilidad de generar música nueva mediante la modificación y combinación de distintas canciones. Superamos desafíos de formato y ajustamos el sample rate y canales para integrar música externa. En la generación, combinamos vectores latentes de reggae,

observando que menos canciones resultaron en mejor coherencia musical. Exploramos un enfoque de "remix" concatenando géneros diferentes y continuamos indignados con maneras de agregar ruido para modificar canciones existentes. La generación de música a partir de estos enfoques nos ofreció una visión de las posibilidades creativas que existen, aunque también reveló la importancia de la coherencia musical al combinar información de múltiples fuentes.

Este trabajo no solo amplió nuestro conocimiento en el campo de procesamiento musical y autoencoders, sino que destacó la importancia de ajustar cuidadosamente los hiperparámetros y otros aspectos de la arquitectura para obtener representaciones eficientes. Aprendimos en profundidad sobre la creación y manejo de los hiperparámetros de las redes convolucionales en mayor profundidad, poniéndolos en práctica. Específicamente, indagamos sobre los filtros, sus tamaños y cómo deben ser operados en situaciones reales para encodear información en un espacio latente lo suficientemente representativo. En este trabajo no solo profundizamos nuestro entendimiento en la tarea de encodear audios, ya que estos conocimientos se pueden ampliar para otras tareas como encodear imágenes. Ahora contamos con una herramienta que nos permite comprender mejor lo que sucede en el fondo de esta práctica.