Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Querétaro

TC3002B Desarrollo de aplicaciones avanzadas de ciencias computacionales

Módulo 2 Análisis y Reporte sobre el desempeño del modelo.

Profesores:

Benjamín Valdés Aguirre

Pedro Oscar Pérez Murueta

Manuel Iván Casillas del Llano

Presenta:

José Emiliano Riosmena Castañón – A01704245

Fecha:

Domingo, 13 de abril del 2025

Índice

Introducción

En este reporte, hablaremos sobre sobre el proceso seguido para construir un modelo de inteligencia artificial para generar música del género Metal. Para ello, en este análisis se presentará una discusión sobre el conjunto de datos utilizado, el uso de las técnicas y algoritmos utilizados para el desarrollo del modelo, y los desafíos presentados durante la implementación de este último.

Por último, veremos los resultados del desempeño de nuestro modelo, analizando su capacidad de generar nueva música, y qué tan coherente es la música generada en comparación con música real. De este modo podremos concluir si el modelo es efectivo para cumplir su objetivo.

Descripción del Dataset

Para entender cómo funcionara el modelo, vamos a comprender la estructura de nuestros datos. El dataset es un conjunto de archivos de audio los cuales contienen diferentes muestras de piezas musicales con una duración de aproximadamente 30 segundos por archivo. Estos últimos están clasificados en diferentes géneros representados por subcarpetas.

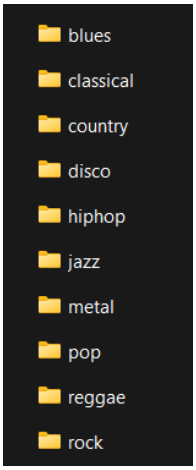

Figura 1.0. Muestra de los géneros del dataset.

Cada género tiene un total de 100 archivos de audio, y en el dataset tenemos un total de 10 géneros, los cuales se muestran en la figura 1.0. Esto nos da como resultado un total de 1000 archivos de audio en total. Esto significa que estamos trabajando con un conjunto muy grande, pues el dataset en su totalidad es de más de un 1 GB.

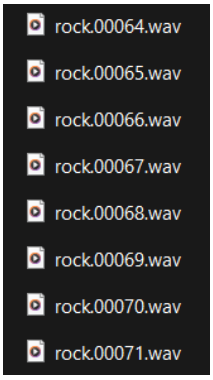

Figura 1.1. Muestra de los archivos de audio del conjunto rock.

Sin embargo, al principio de este documento, mencionamos que el objetivo del modelo es generar música para el género metal. Por lo cual, únicamente necesitamos el conjunto del género metal del dataset. Lo cual nos permitirá trabajar más rápido ya que solo estamos utilizando una pequeña parte de todo el conjunto de datos.

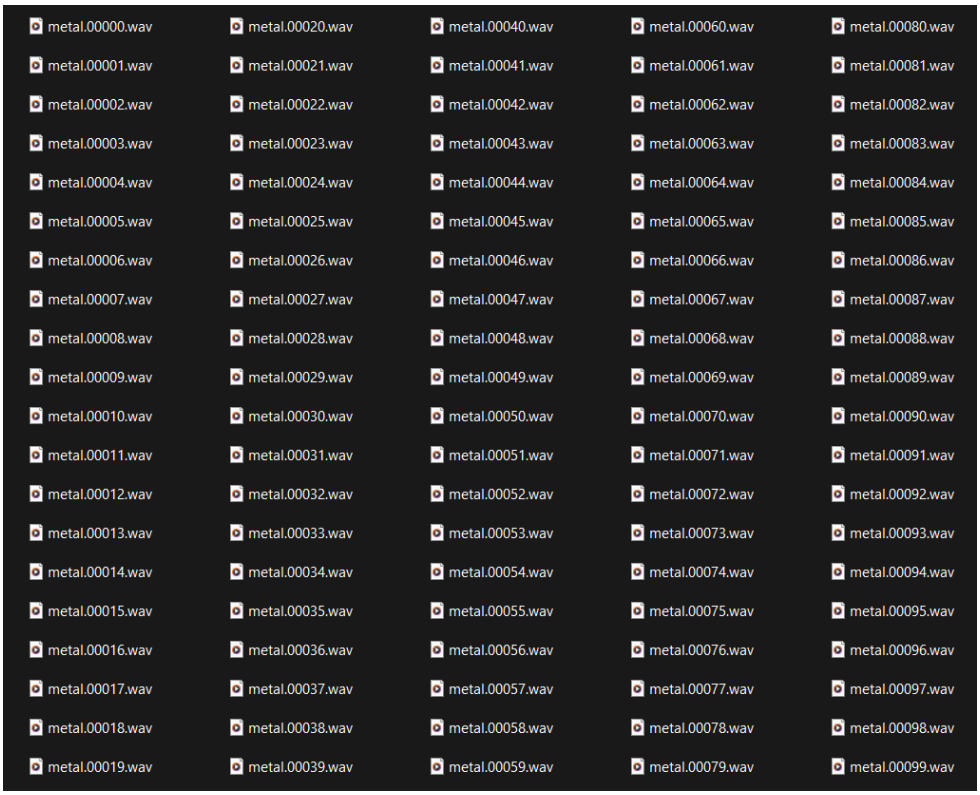

Figura 1.2. Archivos de audio del género metal.

Un detalle por aclarar es que como mencionamos al anteriormente, cada género contiene 100 archivos de audio, y cómo solo estamos trabajando con el género metal, solo tenemos a la mano 100 archivos, lo cual puede ser muy poco para lograr un alcance como el que esperamos para este modelo. Para ello, lo más adecuado sería buscar más archivos y juntarlos en la carpeta. Sin embargo, nos enfrentaríamos al riesgo de que los datos vengan de forma diferente, por ejemplo, que no tengan una duración cercana a 30 segundos, o la calidad de los audios, etc. Eso significa que necesitamos una alternativa para poder trabajar con más archivos.

Uso de ETL (Extract – Transform – Load)

Para la extracción de los datos, en este dataset particular, los dos datos ya venían en un formato decente para su utilización. Pues, los archivos de audio ya vienen clasificados en los diferentes géneros. Pero en este caso, solo queremos trabajar con los datos del género metal. Por lo cual, necesitamos manejar únicamente el género metal durante el preprocesamiento.

Para un modelo como el que vamos a trabajar, necesitamos convertir nuestros archivos de audio de forma que sea más fácil para el modelo aprender los patrones y poder generar música nueva. Para eso, hemos implementado una función para convertir los audios en un espectrograma. Un espectrograma es una imagen que muestra cómo cambian las frecuencias de audio en el tiempo. Y al ser una representación visual sobre cómo suena el género metal, le permitirá al modelo entender mejor la textura y complejidad de una canción. Anteriormente mencionamos que necesitamos una alternativa para tener más archivos de audio con los que pueda entrenar el modelo. Para ello, en esta misma función, cada archivo se divide en diferentes segmentos de 6 segundos, de modo que eso nos permita tener más espectrogramas para cada fragmento de las canciones, aumentando el número de muestras a 500, lo cual nos puede ayudar a mejorar el entrenamiento y que el modelo pueda aprender más detalles y comprender la música que recibe.

```python
def process_audio(file_path):
    signal, sr = torchaudio.load(file_path)
    signal = torch.mean(signal, dim=0, keepdim=True)

    total_samples = signal.shape[1]
    chunks = total_samples // SAMPLES
    processed = []

    for i in range(chunks):
        start = i * SAMPLES
        end = start + SAMPLES
        chunk = signal[:, start:end]

        mel = torchaudio.transforms.MelSpectrogram(
            sample_rate=SR,
            n_mels=N_MELS
        )(chunk)
        mel_db = torchaudio.transforms.AmplitudeToDB()(mel)
        processed.append(mel_db)

    return processed
```

Figura 2.0. Función para fragmentar y convertir los audios a espectrogramas.

Ya tenemos nuestra función definida, lo único que nos haría falta seria la conversión de nuestros archivos, lo cual es bastante sencillo, recorremos todos los archivos de audio del género metal del dataset, y para cada uno lo fragmentamos en muestras de 6 segundos, y guardamos su espectrograma como un archivo .pt, el cual es un tensor de PyTorch.

```python
def process_dataset():
    for file in tqdm(os.listdir(INPUT_DIR), desc="Processing files", colour="green"):
        if file.endswith(".wav"):
            in_path = os.path.join(INPUT_DIR, file)
            chunks = process_audio(in_path)
            base_name = os.path.splitext(file)[0]
            for idx, chunk in enumerate(chunks):
                out_path = os.path.join(OUTPUT_DIR, f"{base_name}_{idx:03d}.pt")
                torch.save(chunk, out_path)
```

Figura 2.1. Función para procesar cada archivo de audio y guardar sus espectrogramas.

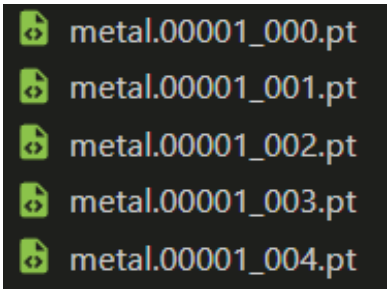

Figura 2.2. Muestra de los espectrogramas de un archivo de audio fragmentado.

De ese modo ya tenemos nuestros datos procesados y podemos empezar a entrenar un modelo para tratar de generar nueva música de metal. En el siguiente capitulo, veremos la implementación del modelo y cómo se realizará su entrenamiento.

Implementación de modelo

Llegamos a la implementación de nuestro modelo, hemos mencionado el objetivo principal, el cual es generar nueva música del género metal, para ello estaremos trabajando con un GAN. Un GAN es una arquitectura de Deep Learning que permite generar datos nuevos que imitan a datos reales. Está compuesto por dos redes neuronales que compiten entre sí:

- Generador: Este intenta crear datos falsos, en nuestro caso música, que parezca real. Toma ruido aleatorio como entrada y genera una salida que simula datos reales.
- Discriminador: Intenta distinguir entre datos reales y datos falsos, en este caso, los datos reales del dataset y datos falsos, los cuales son generados por el generador. Su objetivo es decir si una muestra es real o falsa.

En el entrenamiento, el generador intenta engañar al discriminador creando datos cada vez más realistas. El discriminador intenta mejorar su capacidad para detectar datos falsos. Este proceso se conoce como entrenamiento adversarial, ya que las dos redes se enfrentan en un juego de suma cero, y ambas mejoran en sus respectivas tareas. Vamos a ver la implementación de ambas redes. Hemos separado los modelos en un archivo diferente y el entrenamiento en otro archivo.

Generador

Definimos una clase Generator para la red generadora del GAN, como su nombre lo indica, su objetivo es generar imágenes, en este caso de espectrogramas ya que estamos trabajando con audio, por medio de un vector aleatorio, en este caso, el ruido. En la arquitectura, utilizamos una red neuronal convolucional transpuesta, que toma un vector de ruido “z” de 100 canales y lo va expandiendo hasta obtener una imagen de 1 canal. Utilizamos el método forward para definir cómo fluye la entrada z a través de la red.

```python
class Generator(nn.Module):
  def __init__(self):
    super().__init__()
    self.net = nn.Sequential(
      nn.ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), padding=0),
      nn.BatchNorm2d(512),
      nn.ReLU(True),

      nn.ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=1),
      nn.BatchNorm2d(256),
      nn.ReLU(True),

      nn.ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=1),
      nn.BatchNorm2d(128),
      nn.ReLU(True),

      nn.ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=1),
      nn.Tanh(),
    )

  def forward(self, z):
    return self.net(z)
```

Figura 3.0. Clase para el generador del GAN.

Discriminador

Del mismo modo, definimos una clase Discriminator para la red discriminadora del GAN. Como mencionábamos, su propósito es clasificar si una imagen, en este caso el espectrograma es real o falso. En este caso se trata de una red convolucional tradicional, la cual reduce progresivamente la resolución de la entrada mientras aumenta el número de canales para extraer características y tomar una decisión. Su salida es un escalar por imagen la cual se puede interpretar como la confianza de que la imagen sea real.

```python
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), padding=0),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
        )

    def forward(self, x):
        return self.net(x)
```

Figura 3.1. Clase para el discriminador del GAN.

Entrenamiento

Para el entrenamiento hemos separado la lógica de este ultimo con la de los modelos del generador y el discriminador. Lo primero que hacemos es cargar el dataset y utilizamos un dataloader para iterar en lotes, barajando los datos cada época, e inicializamos los modelos. Creamos dos optimizadores Adam para el generador y el discriminador, y definimos una función de pérdida por medio de una variable criterion en el cual utilizamos la pérdida binaria con logits, la cual es adecuada para un GAN como el que estamos haciendo.

```python
dataset = MusicDataset(DATASET_PATH)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

G = Generator().to(DEVICE)
D = Discriminator().to(DEVICE)

opt_g = optim.Adam(G.parameters(), lr=LEARNING_RATE, betas=(0.5, 0.999))
opt_d = optim.Adam(D.parameters(), lr=LEARNING_RATE, betas=(0.5, 0.999))
criterion = nn.BCEWithLogitsLoss()
```

Figura 3.2. Inicialización de modelos, optimizadores y pérdida.

Para cada época se entrena el discriminador con un batch de espectrogramas reales, ruido aleatorio como entrada del generador e imágenes falsas generadas. Generamos 2 variables, una para las predicciones del discriminador sobre imágenes reales y otra para las imágenes falsas y utilizamos como etiquetas el valor 0.9 para los reales y 0 para falsas, ya que nos ayuda a estabilizar el entrenamiento. Calculamos las pérdidas, las sumamos y se actualiza el discriminador.

```
real = real.to(DEVICE)
batch_size = real.size(0)

noise = torch.randn(batch_size, NOISE_DIM, 1, 1).to(DEVICE)
fake = G(noise)

d_real = D(real)
d_fake = D(fake.detach())

real_labels = torch.ones_like(d_real) * 0.9
fake_labels = torch.zeros_like(d_fake)

loss_d_real = criterion(d_real, real_labels)
loss_d_fake = criterion(d_fake, fake_labels)
loss_d = loss_d_real + loss_d_fake

D.zero_grad()
loss_d.backward()
opt_d.step()
```

Figura 3.3. Entrenamiento del discriminador.

Para el generador se busca engañar al discriminador buscando que una imagen falsa sea cercana a una real. En cada epoch se busca que el generador construya una imagen que se acerque más a una imagen real, de modo que al discriminador le cueste identificar que la imagen es falsa, para eso, al igual que el discriminador se va actualizando el generador en cada epoch.

```
d_fake_for_g = D(fake)
loss_g = criterion(d_fake_for_g, real_labels)

G.zero_grad()
loss_g.backward()
opt_g.step()
```

Figura 3.4. Entrenamiento del generador.

Por último, mostramos el resultado en cada época mostrando la pérdida tanto para el generador como el discriminador. Y guardamos el modelo al finalizar el entrenamiento.

Para un mejor monitoreo de los resultados, implementamos una funcionalidad para ir guardando el modelo cada ciertos epochs y genere una muestra de un espectrograma y su respectiva conversión en audio para ver el progreso del entrenamiento del generador.

```python
generator.eval()
with torch.no_grad():
    noise = torch.randn(1, NOISE_DIM, 1, 1).to(DEVICE)
    fake = generator(noise).cpu().squeeze(0)

    fake = F.interpolate(fake.unsqueeze(0), size=(64, 662), mode='bilinear', align_corners=False).squeeze(0)

    img = (fake + 1) / 2
    plt.figure(figsize=(10, 4))
    plt.imshow(img.squeeze(0), aspect='auto', origin='lower', cmap='magma')
    plt.colorbar()
    plt.title(f"Generated Spectrogram - Epoch {step}")
    plt.tight_layout()
    plt.savefig(f"{OUTPUT_DIR}/sample_{step}.png")
    plt.close()
```

Figura 3.5. Generación de espectrograma.

```python
mel_spec = img.squeeze(0)
mel_spec_db = mel_spec * 80.0 - 80.0
mel_spec_amp = torch.pow(10.0, mel_spec_db / 20.0)

mel_scale = torchaudio.transforms.MelScale(
    n_mels=64,
    sample_rate=22050,
    n_stft=512 + 1
)
mel_basis = mel_scale.fb
inv_mel_basis = torch.linalg.pinv(mel_basis.T)
linear_spec = torch.matmul(inv_mel_basis, mel_spec_amp)
linear_spec = torch.clamp(linear_spec, min=1e-5)
linear_spec = linear_spec / (linear_spec.max() + 1e-6)

griffin = torchaudio.transforms.GriffinLim(n_fft=1024, hop_length=512)
audio = griffin(linear_spec)
audio = torch.nan_to_num(audio, nan=0.0)

torchaudio.save(f"{OUTPUT_DIR}/sample_{step}.wav", audio.unsqueeze(0) if audio.ndim == 1 else audio, 22050)
```

Figura 3.6. Conversión de espectrograma a audio.

Con esto definido podemos realizar el entrenamiento y visualizar los resultados. En la siguiente sección hablaremos a detalle sobre esto ultimo y realizaremos un análisis sobre estos mismos para poder llegar a una conclusión sobre esta primera implementación.

Análisis de Resultados

Para el entrenamiento del GAN hemos utilizado un total de 200 epochs, y un batch_size de 8, el cual establece el tamaño de la cantidad de muestras que se procesan antes de actualizar los modelos. Es importante aclarar que en un modelo especifico como el que estamos realizando, no se puede evaluar la precisión o si el modelo presenta overfiiting o underfitting de forma tan directa. En este caso, se necesita más de evaluación auditiva para determinar la capacidad del modelo. Aún así podemos visualizar los resultados del progreso del entrenamiento y la pérdida.

Echemos un vistazo a los resultados.

Evolución del entrenamiento

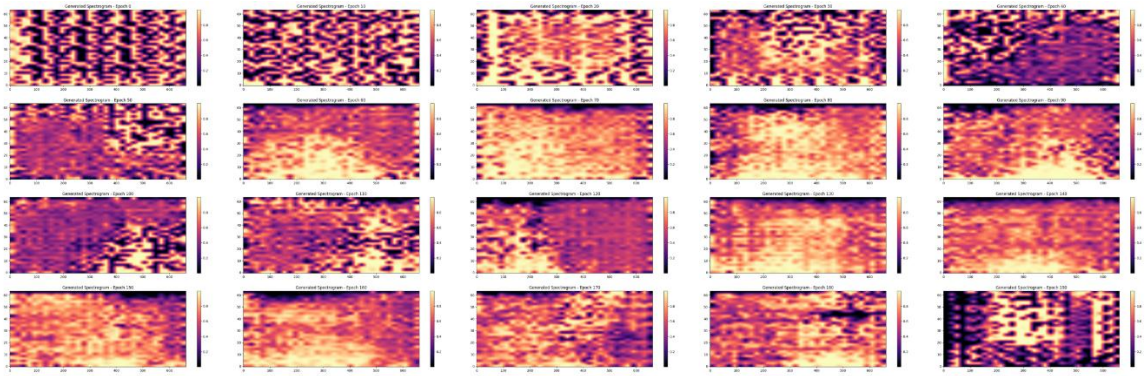

Figura 4.0. Evolución de los espectrogramas del entrenamiento.

En esta línea del tiempo podemos ver cómo evoluciona la generación de nuevos espectrogramas a lo largo del entrenamiento. En los primeros epochs, los espectrogramas se ven como ruido totalmente aleatorio, es decir, bloques abruptos sin estructura, lo cual es esperado ya que el generador apenas empieza a aprender. En los epochs 50 al 90 podemos ver que empieza a aparecer una mayor homogenidad, es decir, patrones más suaves y transición en los colores. Aún son algo aleatorios, pero se puede ver que el modelo está captando la textura global del espectrograma. Esto indica un sonido un poco menos caótico, pero todavía muy primitivo. En los epochs del 100 al 150 se ve una mejora en la distribución del color, hay menos zonas negras y gradientes más suaves. Empieza a haber bandas horizontales, las cuales pueden representar frecuencias estables o sostenidas. Pero

en los epochs del 160 al 190 se ve una tendencia a un colapso parcial, ya que las imágenes empiezan a parecerse entre sí, en el epoch 190 se puede ver un patron raro, indicando que el modelo colapso y empezó a generar un tipo diferente de espectrograma. Esto puede significar una pérdida de diversidad o un mal salto en el entrenamiento.

En el caso de los archivos de audio, no podemos visualizar los resultados directamente, ya que se tratan de archivos de audio, sin embargo, a oído simple, se puede escuchar una ligera diferencia entre los archivos, pero en este primer modelo, se puede decir que el modelo no ha tenido éxito ya que se escucha principalmente estática, el sonido no tiene coherencia ni estructura. Aunque se puede escuchar cómo va variando la intensidad del sonido, cómo si el modelo estuviera intentando generar algo, pero aún no puede. Dado a que se trata de audios, se puede acceder a ellos mediante este link, para que ustedes puedan escuchar los resultados.

Pérdida

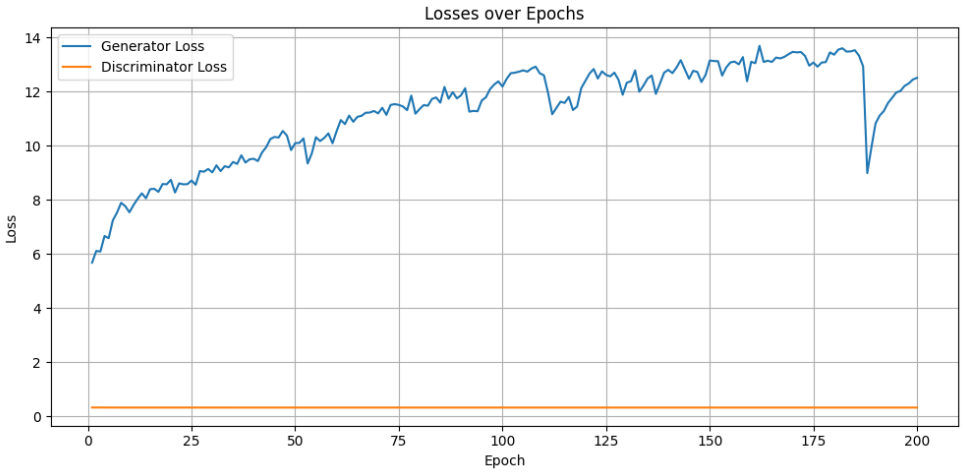

Figura 4.1. Representación de la pérdida en ambos modelos.

En la gráfica de las perdidas podemos ver que el discriminador se mantiene casi constante y muy baja con valores cercanos a 0 durante todo el entrenamiento. Lo que sugiere que el discriminador está demasiado seguro al distinguir entre cosas reales y falsas, esto nos dice que el discriminador no está aprendiendo nada nuevo.

En el generador, podemos ver un aumento que empieza aproximadamente en 6 y va subiendo hasta 12 o 14 con varias oscilaciones y sin convergencia clara. Esto nos dice que el generador no está mejorando y el discriminador lo está aplastando desde el inicio del entrenamiento. Estos resultados reflejan un fallo en el balance del GAN ya que el discriminador es demasiado fuerte desde el inicio y el generador no logra aprender, intenta lanzar cosas al azar esperando colar alguna. Y como no recibe un buen “feedback” del discriminador, el generador no sabe por dónde mejorar. Lo que en un resultado más directo nos dice que estamos teniendo underfitting en el generador.

Siguientes pasos

Habiendo visualizado los resultados obtenidos tanto en la generación de música y en la función de pérdida, podemos concluir que el modelo no es efectivo para la generación de música del género metal, ya que los resultados indican que el modelo no es capaz de aprender a cómo construir música que sea parecida a una canción real, y los resultados de los archivos de audio, si bien, podemos decir que el modelo fue capaz de generar audio nuevo, estos últimos son estática pura, y no se alcanza a percibir un sonido legible que pueda simular el género metal.

Sin embargo, aún es posible intentar mejorar el resultado el modelo y tratar de que genere algo de música, para el modelo podemos intentar normalizar la salida del discriminador o cambiar el loss. Para prevenir que el discriminador sea demasiado fuerte podemos intentar agregar un dropout al discriminador. Quizás pueda funcionar entrenar el discriminador cada determinado paso, ya que el discriminador aplasta completamente al generador y realmente no mejora ninguno ya que no están aprendiendo, o intentar reducir el learning rate al discriminador.