Universidad Zaragoza
1542

Trabajo Fin de Grado

Generación automática de ritmos de percusión mediante modelos de redes neuronales profundas basados en Transformers

Automatic generation of percussive rhythms using Transformer-based deep neural network models

Autor

Sergio Ferraz Laplana

Directores

Jose Ramón Beltrán Blázquez

Carlos Hernández Oliván

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2023

AGRADECIMIENTOS

Agradezco este Trabajo de Fin de Grado a mis tutores. A Carlos Hernández por toda su ayuda y paciencia durante todo este tiempo, aún estando en Japón. Y a Jose Ramón Beltrán por ofrecerme esta oportunidad para realizar mi TFG sobre un tema que me encanta y sobre el cual tenía muchas ganas de aprender.

Gracias a mis padres por estar siempre ahí apoyándome y sobre todo por darme la oportunidad de estudiar la carrera que yo quería. A mi hermano, por toda su ayuda y apoyo. A mis amigos de Monzón que han estado ahí conmigo a pesar de yo no haber estado al 100 % por estar estudiando. A mis amigos de Villanueva que aunque nos veamos poco siempre puedo contar con ellos. A mis amigos de clase, con los que hemos formado un grupo increíble. A mis antiguos compañeros de piso Sergio y Alejandro, y a los de ahora Manuel y Adolfo, por aguantarme y hacer que la convivencia sea buenísima. Gracias también a mis compañeras de la rama de Sonido e Imagen. Carlota, María I., Xoni y María P, con las que me lo he pasado súper bien yendo a clase.

RESUMEN

La invención de las arquitecturas de modelos de Deep Learning basados en Transformers dentro del ámbito de la Inteligencia Artificial ha tenido mucho impacto en la composición musical automática. En este trabajo de Fin de Grado se ha realizado el análisis de la generación de ritmos de percusión, entrenando un modelo Transformer Decoder con una base de datos de archivos MIDI representados mediante el modelo de tokenización Compound Word. Se ha implementado totalmente este tipo de representación de la música en tokens en Python y se han realizado múltiples experimentos tanto con la base de datos completa como con unos pocos archivos, entrenando el modelo intentando optimizarlo al máximo posible para minimizar la función de pérdida. En el mejor experimento realizado, se ha llegado a obtener una pérdida de 0,22 en entrenamineto (0,33 en test) y se han generado archivos MIDI, que aunque no son como cabría esperar, estos son escuchables y se han apreciado indicios de que la red parece generar un ritmo acompasado.

ABSTRACT

The invention of Transformers-based Deep Learning model architectures within the field of Artificial Intelligence has had a great impact on automatic music composition. In this Final Degree project, the analysis of the generation of percussion rhythms has been carried out, training a Transformer Decoder model with a database of MIDI files represented by the Compound Word tokenization model. This type of representation of music in tokens has been fully implemented in Python and multiple experiments have been carried out both with the complete database and with a few files, training the model trying to optimize it as much as possible to minimize the loss function. In the best experiment carried out, a loss of 0,22 in training (0,33 in test) has been obtained and MIDI files have been generated, which, although they are not as expected, they are listenable and there have been signs that the network seems to generate a measured rhythm.

Índice

Capítulo 1

Introducción y objetivos

1.1. Motivación del proyecto

En los últimos años ha habido un gran avance en el ámbito de la inteligencia artificial, tanto que han aparecido redes neuronales generativas que son capaces de crear contenido que puede considerarse artístico, como pueden ser DALL-E 2[1] o MidJourney[2], redes capaces de generar imágenes a partir de unas indicaciones de texto.

Este avance ha ido más allá de la imagen y ha llegado al mundo de la música. El hecho de poder crear composiciones musicales de forma totalmente automatizada son de gran interés para las empresas, como puede ser Spotify[3] (plataforma de música en streaming, que ofrece acceso a millones de canciones y podcasts de todo el mundo) la cual ya hace uso de la inteligencia artificial para tareas como recomendación musical. También es una gran herramienta para artistas con pocos recursos o con pocos conocimientos sobre teoría musical.

La organización más grande sobre este tema es ISMIR[4] (The International Society for Music Information Retrieval), una sociedad profesional sin ánimo de lucro que se centra en desarrollar herramientas y técnicas para el análisis, la organización y la recuperación de datos relacionados con la música.

1.2. Objetivos

Se han definido los siguientes objetivos a cumplir en este trabajo:

– Estudio del estado del arte de técnicas de composición musical con Machine Learning, en concreto, Deep Learning.

[1] https://openai.com/product/dall-e-2
[2] https://www.midjourney.com/
[3] https://open.spotify.com/
[4] https://www.ismir.net/

– Tokenizar una base de datos de archivos MIDI con la representación Compound Word.

– Entrenar con estos datos una arquitectura basada en Deep Learning, en concreto modelos de *transformers*.

– Generar de forma automática desde cero pistas de audio MIDI que contengan ritmos de percusión y que estos tengan una coherencia musical.

El desarrollo de programación se realizará en el lenguaje Python y en Jupyter Notebook, en la herramienta Google Colab.

1.3. Estructura de la memoria

Esta memoria está separada en cinco capítulos. Este primero contiene la introducción y los objetivos del trabajo a realizar. En el segundo capítulo se hace un estudio del estado del arte, explicando los conceptos necesarios e ilustrando el trabajo relacionado. El capítulo tres trata sobre el desarrollo y la implementación de las tareas realizadas. En el cuarto capítulo se comentan los experimentos llevados a cabo y los resultados obtenidos. Por último, en el quinto capítulo, se exponen las conclusiones y posibles líneas futuras de este trabajo.

Capítulo 2

Estado del Arte

2.1. Inteligencia Artificial

La inteligencia artificial (IA) es una disciplina cuyo propósito es la creación de máquinas que imiten la inteligencia humana para realizar tareas, y que pueden mejorar conforme recopilan información. Existen dos tipos de IA, la débil o estrecha y la general o fuerte. La IA débil se refiere a sistemas diseñados para realizar tareas específicas y limitadas. Estos sistemas se especializan en un área particular pero carecen de la capacidad de comprender o aplicar conocimientos fuera de su dominio específico. Ejemplos incluyen sistemas de reconocimiento de voz, chatbots y motores de recomendación. En cambio, la IA general o fuerte se refiere a sistemas con un nivel de inteligencia similar o superior al humano en todas las áreas cognitivas. Tiene la capacidad de razonar, aprender y comprender información de manera similar a los seres humanos. Sin embargo, este tipo de IA todavía no se ha logrado desarrollar por completo y está en proceso de investigación.

La arquitectura de las inteligencias artificiales y los procesos por los cuales aprenden, se mejoran y se implementan dependiendo del área de interés y de la utilidad que se les quiera dar. Estos van desde la ejecución de sencillos algoritmos hasta la interconexión de complejas redes neuronales artificiales que intentan replicar los circuitos neuronales del cerebro humano y que aprenden mediante diferentes modelos de aprendizaje tales como el aprendizaje automático (Machine Learning) o el aprendizaje profundo (Deep Learning).

2.1.1. Redes Neuronales

Las redes neuronales artificiales consisten en un conjunto de unidades, llamadas neuronas, conectadas entre sí para transmitirse señales (ver Figura 2.1). Estos sistemas de redes neuronales, en lugar de ser programados de forma explícita, aprenden y se adaptan a sí mismos y sobresalen en áreas donde la detección de soluciones o

características es difícil de expresar con la programación convencional.

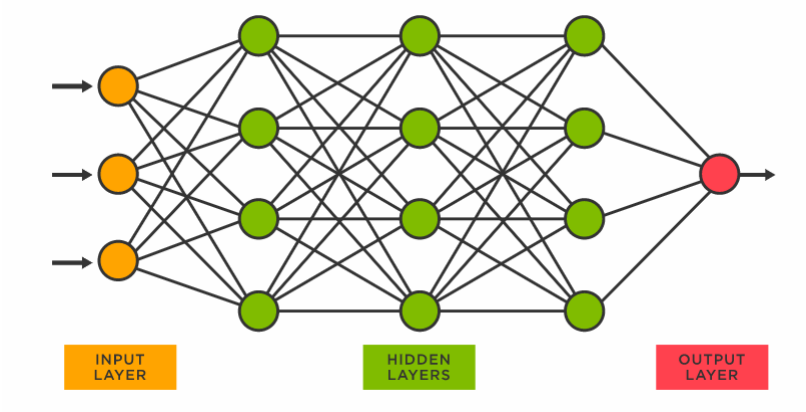


Figura 2.1: Estructura de una red neuronal. La red se organiza en capas: la capa de entrada recibe los datos, las capas ocultas son las que se adapatan para la tarea concreta y la capa de salida proporciona la salida deseada. [Fuente: `https://www.tibco.com/es/reference-center/what-is-a-neural-network`]

Cada neurona está conectada con otras a través de enlaces, en los cuales el valor de salida de la neurona anterior es multiplicado por un valor llamado peso (w). Estos pesos pueden incrementar o inhibir el estado de activación de las neuronas adyacentes. Además se añade un término de sesgo o *bias* (b) que permite controlar el nivel de activación de la neurona. Del mismo modo, a la salida de la neurona, puede existir una función limitadora o umbral, que modifica el valor resultado o impone un límite que no se debe sobrepasar antes de propagarse a otra neurona, conocida como función de activación. Las ecuaciones (2.1) y (2.2) representan el cálculo de la salida a de una neurona, siendo x_i las entradas, w_i los pesos, b es el sesgo de la neurona y $g(x)$ la función de activación.

$$z = \sum_{i=1}^{n} w_i \cdot x_i + b \tag{2.1}$$

$$a = g(z) \tag{2.2}$$

La salida final de una red neuronal $\hat{y}$ se denomina predicción, y se calcula en la capa de salida de la misma forma que en las demás neuronas. Por lo que, en la última capa, $\hat{y}$ será igual a a.

Para realizar el aprendizaje automático, se intenta minimizar una función de pérdida, que se utiliza para medir el error entre la predicción del modelo $\hat{y}$ y el valor real y que se ha pasado como entrada. El sesgo y los pesos en los enlaces de las neuronas se van actualizando buscando reducir el valor de dicha función de pérdida. Este proceso se realiza mediante los algoritmos de propagación hacia atrás (*backpropagation*).

Uno de los tipos de red neuronal más importante y sobre la cuál se va a hablar en este trabajo es la red neuronal *feed-forward*. En este tipo de redes la información fluye en una sola dirección, desde la capa de entrada hacia la capa de salida, sin hacer bucles o conexiones de realimentación. Estas redes se componen de múltiples capas de neuronas, con cada capa completamente conectada a la siguiente. Un ejemplo de este tipo de redes se puede apreciar en la Figura 2.1.

En las redes neuronales con múltiples capas o también llamadas profundas, se divide el cálculo por capas. En las ecuaciones (2.3) y (2.4) se define la salida de la capa l de la red, siendo $W^{[l]}$ la matriz de pesos correspondientes a las neuronas de la capa, $b^{[l]}$ es el vector de sesgos y $A^{[l-1]}$ es la entrada de la capa actual y la salida de la capa anterior. Además $A^{[0]} = X$, donde X es la matriz de entrada a la red, la cual contiene m ejemplos de entrenamiento.

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \tag{2.3}$$

$$A^{[l]} = g(Z^{[l]}) = \hat{Y} \tag{2.4}$$

Una vez obtenida la predicción de la red $\hat{Y}$, se calcula la función de pérdida $\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$, la cual mide la discrepancia entre la salida predicha por la red neuronal y la salida real[1] para un solo ejemplo de entrenamiento. Para obtener el error promedio del entrenamiento completo se define la función de coste $\mathcal{J}$ como el promedio de las funciones de pérdida de cada ejemplo de entrenamiento (2.5).

$$\mathcal{J} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{Y}^{[l](i)}, Y^{[l](i)}) \tag{2.5}$$

Para optimizar la red, se debe minimizar el error obtenido. Para ello, se actualizan los parámetros del modelo (pesos y sesgos) mediante el algoritmo *backpropagation*. Dicho error se propaga capa por capa a través de la red hacia atrás, para ajustar los pesos de las conexiones de manera que reduzcan el error. El ajuste de los pesos se realiza mediante el algoritmo de gradiente descendente. El descenso del gradiente o *gradient descent* se basa en la idea de que, al calcular el gradiente de la función objetivo en un punto dado, podemos determinar la dirección de máximo descenso de la función en ese punto. El gradiente es un vector que indica la dirección de máximo crecimiento de la función en un punto específico. Se aplica de manera iterativa, actualizando los parámetros del modelo (pesos) en la dirección opuesta al gradiente de la función objetivo, con el objetivo de alcanzar el mínimo.

[1] La salida real de una red neuronal Y se proporciona a la entrada y contiene los valores esperados tras el entrenamiento. Comúnmente es igual que la entrada X.

En la propagación hacia atrás, comenzando desde la capa de salida, se calcula el gradiente del error con respecto a los pesos de esa capa utilizando la regla de la cadena. La entrada es el gradiente de la función de pérdida respecto a la salida de la capa de salida y se denota $dA^{[l]}$. Las salidas de cada capa serán: $dA^{[l-1]}$ (2.8), $dW^{[l]}$ (2.6) y $db^{[l]}$ (2.7).

$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{2.6}$$

$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{2.7}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{2.8}$$

$$dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}} = dA^{[l]} * g'(Z^{[l]}) \tag{2.9}$$

El gradiente del error se propaga hacia atrás a través de las capas ocultas, calculando los gradientes de error en cada capa y actualizando los pesos correspondientes.

Una vez que se han calculado los gradientes de error para todos los pesos en la red neuronal, se actualizan los parámetros:

$$W^{[l]} = W^{[l]} - \alpha \; dW^{[l]} \tag{2.10}$$

$$b^{[l]} = b^{[l]} - \alpha \; db^{[l]} \tag{2.11}$$

Donde α es la tasa de aprendizaje o *learning rate*, un hiperparámetro del modelo que determina la magnitud del ajuste realizado en los parámetros en cada paso de actualización. Controla la velocidad a la que se actualizan los parámetros. La elección de la tasa de aprendizaje adecuada es crucial para obtener un buen rendimiento en el entrenamiento de una red neuronal. Hay que tener en cuenta varios factores como el tamaño del conjunto de datos de entrenamiento o el gardiente de la pérdida, cuanto mayores sean estos, una tasa más pequeña será apropiada.

2.1.2. Deep Learning

El Deep Learning [1] o aprendizaje profundo es una clase de algoritmos de aprendizaje automáticos basados en el uso de redes neuronales con múltiples capas de procesamiento no lineal. Cada capa utiliza la salida de la capa anterior como entrada, lo cual ayuda a que cada capa aprenda a transformar sus datos de entrada en una representación un poco más abstracta y compuesta. (ver Figura 2.2)

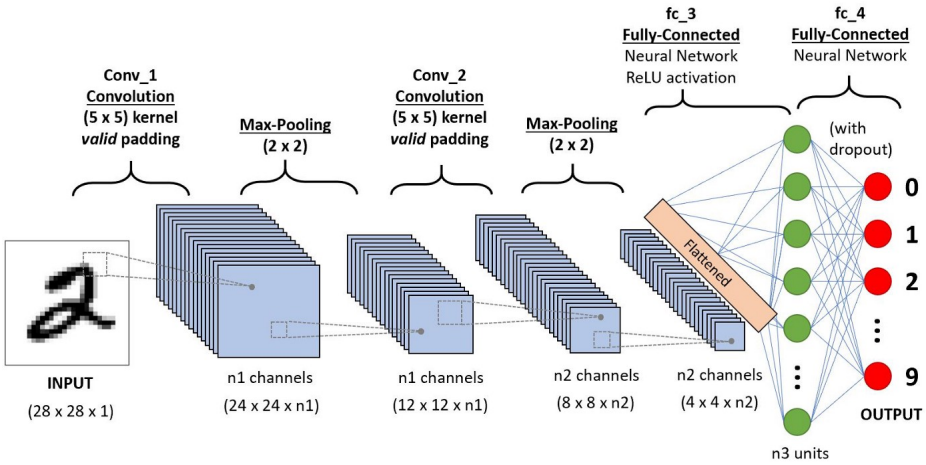


Figura 2.2: Modelo de capas de Deep Learning. [Fuente: `https://www.analyticsvidhya.com/blog/2022/03/basic-introduction-to-convolutional-neural-network-in-deep-learning/`]

Las redes neuronales profundas son de gran utilidad cuando se tiene una gran base de datos, pues son capaces de aprender características más específicas en el proceso de aprendizaje automático, lo cual permite que realicen tareas más complejas y precisas como las anteriormente mencionadas.

Las arquitecturas basadas en Deep Learning han sido de gran utilidad en tareas como el reconocimiento de imágenes, el procesamiento del lenguaje natural (NLP: *Natural Language Processing*), en medicina e incluso en composición musical, que es el tema principal de este trabajo.

2.1.3. Modelos de Deep Learning

Existen diferentes modelos de Deep Learning que utlizan redes neuronales profundas como la red *feed-forward* mencionada anteriormente.

Uno de los tipos de redes más comunes y utilizadas son las redes neuronales convolucionales (CNN). Estas redes procesan datos en forma de matrices o tensores multidimensionales, como imágenes. Son ampliamente utilizadas en tareas como reconocimiento de imágenes o detección de rostros. Utilizan la convolución para aplicar filtros a los datos de entrada y extraer características relevantes. Estos filtros son pequeñas matrices que se deslizan sobre la imagen original, multiplicándose elemento por elemento y sumándose para generar un mapa de características. Además, se utilizan capas de *pooling*, que reducen el tamaño de las representaciones obtenidas. (ver Figura 2.3)

Las redes neuronales recurrentes (RNN) son utilizadas para procesar datos de entrada secuenciales, como texto. A diferencia de las CNN, que se centran en datos

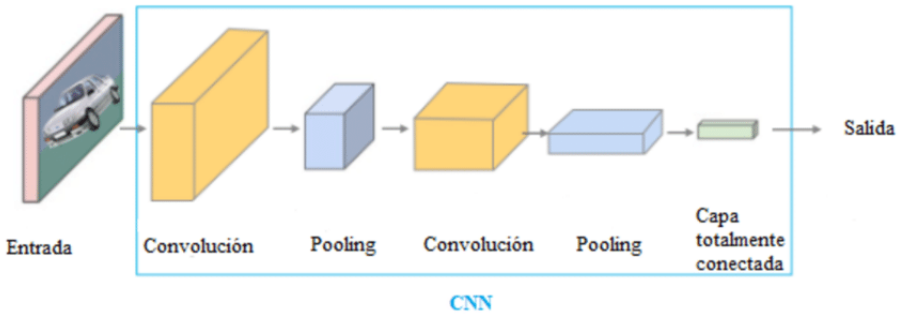


Figura 2.3: Ejemplo básico de red neurnal convolucional [2]

con estructuras matriciales como imágenes, las RNN se especializan en datos con estructuras temporales (como la música). Se distinguen por su "memoria", ya que obtienen información de entradas anteriores para influir en la entrada y salida actuales, lo cual les permite capturar la información contextual a lo largo del tiempo. Una de las variantes más comunes de las RNN es la llamada *Long Short-Term Memory* (LSTM) [3]. Estas permiten a las RNN recordar sus entradas durante un largo periodo de tiempo. (ver Figura 2.4)

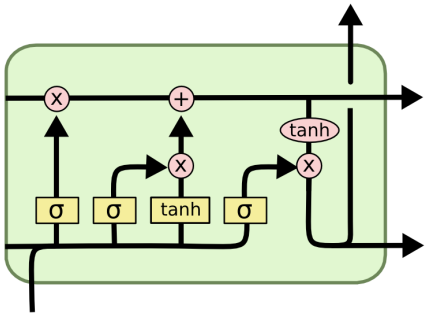


Figura 2.4: *Long Short-Term Memory* (LSTM) [Fuente: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`]

Los *Transformers* son modelos basados en la atención que han demostrado un gran éxito en tareas de procesamiento de lenguaje natural (NLP), como la traducción automática. Han superado a las RNN en muchos casos debido a su capacidad para capturar relaciones de largo alcance en secuencias. En el siguiente apartado se explican con más profundidad.

2.1.4. Transformers

Los *Transformers* [4] son modelos basados en Deep Learning que se caracterizan por utilizar una estructura de auto-atención para procesar los datos de entrada. La invención de estos ha sido clave en el procesamiento del lenguaje natural.

Al igual que las redes neuronales recurrentes (RNN), los transformers están diseñados para procesar datos de entrada secuenciales, como el lenguaje natural, con aplicaciones para tareas como la traducción y el resumen de texto. Sin embargo, a diferencia de las RNN que procesan los datos de entrada de uno en uno, los *transformers* procesan toda la información de entrada a la vez. El mecanismo de auto-atención o *self-attention* proporciona contexto para cualquier posición en la secuencia de entrada. Por ejemplo, si los datos de entrada son frases de lenguaje natural, el *transformer* no tiene que procesar una palabra cada vez. Esto permite una mayor paralelización (tipo de computación en la que muchos cálculos o procesos se llevan a cabo simultáneamente) que las RNN y, por lo tanto, reduce los tiempos de entrenamiento considerablemente.

Como se ha comentado, la entrada son secuencias de datos que se representan mediante palabras o *tokens*. Cada palabra se representa mediante vectores númericos que se denominan *embeddings*. En lugar de tratar las palabras como unidades discretas, se asigna a cada una un vector numérico denso[2] donde cada elemento del vector representa una característica específica. Estos vectores capturan información semántica y estructural sobre las palabras en función de su contexto y relaciones con otras palabras en la base de datos de entrenamiento.

La auto-atención o *self-attention* es un mecanismo de atención que relaciona diferentes posiciones de una sola secuencia para calcular las relaciones que tiene cada elemento de la secuencia con los demás. Se basa en tres conceptos principales:

- Consultas o *Queries*: es una representación de la palabra (token) que se está procesando que se utiliza para obtener información relevante respecto a los demás palabras.
- Claves o *Keys*: son "etiquetas" para todas las palabras de la secuencia de entrada
- Valores o *Values*: representan la información contenida en las palabras de la secuencia de entrada.

Cada valor de entrada se transforma en cada una de estas tres representaciones vectoriales: una consulta, una clave y un valor. Una vez hecho esto, se calcula la atención a partir del producto escalar entre cada consulta (Q) y todos los vectores de claves (K). Estos productos escalares se escalan por la raíz cuadrada de la dimensión de las consultas (d_k) para así obtener gradientes más estables. A continuación, se aplica una función de activación *softmax* a los valores resultantes para obtener los pesos de

[2]Un vector numérico denso se refiere a un vector que contiene una gran cantidad de elementos, y cada uno es un número real. Un vector numérico denso tiene valores significativos en la mayoría de sus elementos.

atención normalizados, de manera que sean todos positivos y su suma igual a la unidad. Estos pesos indican la importancia relativa de cada palabra en función de la consulta (posición) actual. Por último, se multiplica cada vector de valores (V) por los pesos obtenidos y se realiza la suma del resultado, para obtener finalmente la salida de esta capa de auto-atención. Este proceso se realiza para cada posición o valor de entrada.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V \tag{2.12}$$

Los Trasformers utilizan un modelo *encoder-decoder* utilizando el mecanismo de auto-atención de forma apilada (Figura 2.5).

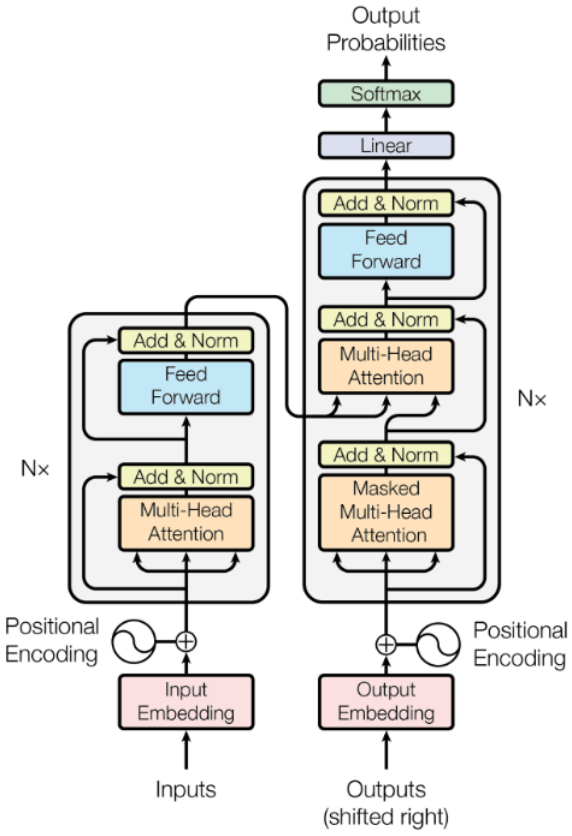


Figura 2.5: Arquitectura del modelo Transformer [4].

En el caso de los *transformers* el orden de las palabras en la secuencia de datos de entrada es muy relevante. A diferencia de las redes neuronales convolucionales o recurrentes, los *transformers* no tienen una noción de la posición en una secuencia, ya que operan de manera independiente en cada token. Por lo tanto, se agrega la codificación posicional o *positional encoding* para capturar la información de posición de cada token en la secuencia de entrada. Esto se realiza sumando un vector a cada uno de los *embeddings* de entrada. Estos vectores siguen un patrón específico, el cual el modelo aprende y le ayuda a determinar la posición de cada token o la distancia entre

diferentes tokens. La forma común de realizar esta codificación posicional es mediante senos y cosenos con diferentes frecuencias.

El codificador está formado por una pila de seis capas idénticas. Cada capa está formada por un mecanismo de auto-atención *multihead*, y una red *feed forward* totalmente conectada. Cada capa utiliza como entrada la salida de la capa anterior.

La capa de atención *multi-head* permite que el modelo preste atención a diferentes partes de la secuencia de entrada de manera simultánea y en paralelo, mejorando así su capacidad de capturar relaciones complejas. Se realizan múltiples transformaciones para obtener varios conjuntos de matrices Consultas/Claves/Valores. Cada uno de estos conjuntos matriciales se denomina cabeza de atención o *attention head*. Para cada *attention head*, se aplica el mecanismo de auto-atención descrito anteriormente para obtener su propio conjunto de pesos aprendibles (que se inicializan de manera aleatoria), lo que permite que el modelo enfoque diferentes aspectos de la secuencia de entrada. Los *Transformers* utilizan ocho *attention heads*. Por último, las ocho salidas se concatenan y se multiplican por una matriz de pesos adicional (que ha sido entrenado juntamente con el modelo).

La capa *feed forward* es un red neuronal completamente conectada que trata cada posición por separado y de forma idéntica. Consiste en una transformación lineal y una función de activación *Relu*.

El decodificador también está formado por una pila de seis capas idénticas. Además de las dos subcapas que también tiene el codificador, el decodificador introduce una tercera, que realiza auto-atención enmascarada o *masked multi-head attention* sobre la salida del codificador.

Esta capa de *masked multi-head attention* es prácticamente idéntica a la capa de auto-atención *multi-head*. La diferencia está en que se aplica una máscara a los pesos de atención calculados entre las consultas y las claves. Esta máscara es una matriz llamada matriz de atención, donde los elementos en las filas superiores a la diagonal de la matriz se establecen en menos infinito, es decir, las posiciones correspondientes a los tokens futuros. La máscara de atención asegura que solo se tenga en cuenta la información anterior o actual al generar cada token, evitando así la dependencia directa con los tokens futuros. Esto es importante para que el modelo no "adivine" o prediga las palabras que aún no han sido generadas, garantizando una generación coherente.

La salida del decodificador son vectores de numeros reales que pasan por las capas *Linear* y *Softmax* para convertir estos vectores en tokens.

La capa lineal es una red neuronal totalmente conectada que convierte los vectores de salida del decodificador en vectores mucho más grandes, en concreto del tamaño igual al número de tokens únicos (vocabulario). Las posiciones del vector contienen los valores

correspondientes a todos los tokens. Por último, estos se convierten a probabilidades mediante una función *softmax*. La posición con mayor probabilidad será el token de salida.

2.2. MIDI

MIDI[3] ("Musical Instrument Digital Interface") es un estándar tecnológico que describe un protocolo, una interfaz digital y conectores que permite a instrumentos musicales electrónicos, ordenadores y otros dispositivos intercambiar información musical. MIDI no transmite señales de audio, sino datos de control, como notas, volumen, efectos y otros parámetros, que son enviados mediante un cable MIDI a otros dispositivos que controlan la generación de sonidos u otras características. Una simple conexión MIDI puede transmitir hasta dieciséis canales de información que pueden ser conectados a diferentes dispositivos cada uno. Cuando una nota es tocada en un instrumento MIDI, esta genera una señal digital que puede ser usada para activar la misma nota en otro instrumento. Las ventajas del uso de MIDI incluyen el pequeño tamaño de los ficheros y la fácil manipulación, modificación y selección de los instrumentos.

El sonido del instrumento (o programa) para cada uno de los 16 canales posibles es seleccionado a partir de un mensaje de cambio de programa (*Program Change*), el cual tiene como parámetro el número de programa (*Program Number*). La especificación General MIDI define los sonidos de instrumentos para cada uno de los 128 sonidos posibles[4].

Un archivo MIDI estándar (*Standard MIDI File Format* o SMF[5]) es un archivo binario que almacena toda la información necesaria para reproducir todos los eventos soportados por MIDI (notas, velocidad, cambio de programa). Cada evento va asociado a una etiqueta de tiempo que indica al secuenciador MIDI cuando debe reproducir dicho evento. Este tipo de archivos utilizan la extensión `.mid` al final del nombre del archivo para indicar que es un archivo MIDI estándar[6]. Este tipo de archivos MIDI serán los utilizados en este trabajo.

Respecto a lo que incumbe a este trabajo que es la percusión, el canal 10 es el reservado para ésta. Dentro de este canal cada número de nota distinto especifica un instrumento de percusión único, en lugar del tono o *pitch* del sonido. El estándar General MIDI incluye 47 sonidos de percusión (Figura 2.6), utilizando los números de

[3] `https://www.midi.org/`
[4] `https://www.midi.org/specifications-old/item/gm-level-1-sound-set`
[5] `https://www.midi.org/specifications-old/item/standard-midi-files-smf`
[6] `https://blog.karaoplay.com/que-es-un-archivo-midi/`

nota 35-81 (de los 128 números posibles del 0 al 127).

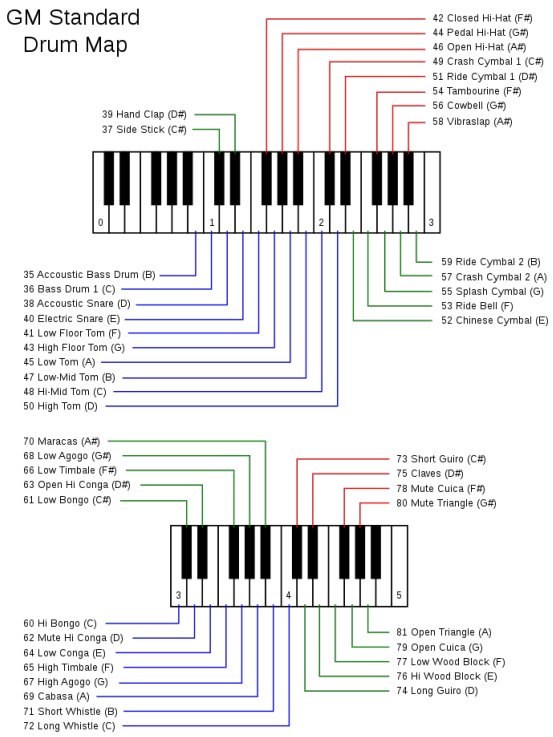


Figura 2.6: Distribución de la percusión en MIDI. [Fuente: Manudiclemente - text editor, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=19937484`]

2.3. Tokenización

La tokenización es una técnica muy útil en el procesamiento del lenguaje natural y se utiliza en muchas aplicaciones, como pueden ser la traducción automática o la generación de texto. Consiste en dividir un texto en unidades más pequeñas llamadas tokens. Estos tokens pueden ser palabras, números, signos de puntuación, caracteres o cualquier otro elemento que se considere relevante para el análisis del texto. La Figura 2.7 ilustra un ejemplo sencillo de tokenización del lenguaje, en el que dada una frase, cada palabra representa un token. Existen múltiples modelos de tokenización, y elegir el adecuado para la tarea en la que se está trabajando, es crucial.

En el contexto de generación musical, la tokenización se refiere al proceso de convertir una pieza musical en una secuencia de tokens que pueden ser notas, acordes, silencios, ritmos o cualquier otro elemento o parámetro que tenga un sentido en el análisis musical.

La tokenización de la música es esencial para entrenar modelos de aprendizaje automático que puedan generar nueva música de forma autónoma. Al transformar la

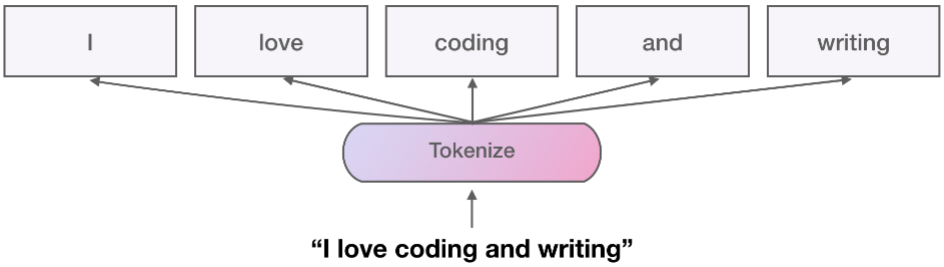


Figura 2.7: Ejemplo de Tokenización de lenguaje. [Fuente: `https://github.com/dair-ai/nlp_fundamentals/blob/master/1_nlp_basics_tokenization_segmentation.ipynb`]

música en una secuencia de tokens, se crea una representación numérica que puede ser procesada por el modelo para generar nueva música.

Existen diferentes modelos de tokenización musical como REMI [5] o MIDI-like [6], en este trabajo se ha optado por implementar la representación Compound Word [7].

Se puede destacar la librería de MidiTok [8] que ha servido de ayuda para la implementación de la tokenización. Es una herramienta python que implementa la mayoría de los tipos de tokenización existentes hasta la fecha.

2.4. Trabajo relacionado

La composición musical automática mediante inteligencia artificial ha evolucionado mucho con el paso de los años, y con el reciente descubrimiento de los *Transformers*, este ámbito ha despegado.

Uno de los trabajos más importantes dentro de este ámbito ha sido *Multi-Track Music Machine* [9], un sistema que utiliza una arquitectura basada en *Transformers* para la generación musical de varias pistas (*multi-track*). Proponen un nuevo tipo de tokenización (MMM), que se basa en la concatenación de secuencias ordenadas temporalmente de eventos musicales de cada una de las pistas. Ofrece un alto grado de control en la generación sobre los instrumentos y la densidad de las notas de cada pista. Utilizan un modelo *Transformer-XL* y la base de datos completa Lakh Midi para el entrenamiento. Una de las limitaciones que encontraron es que el modelo solo acepta canciones con un número fijo de compases que pueden ser tokenizados.

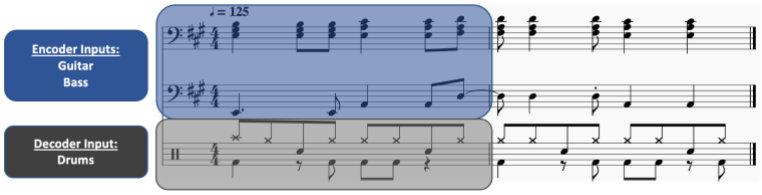
Pop Music Transformer [5] es un sistema de generación musical que hace uso del modelo *Transformer-XL*. Proponen el metodo de tokenización REMI (derivado de MIDI) en los datos de entrada, imponiendo una estructura métrica que permite el control del ritmo y de la armonía. De esta manera, el modelo se beneficia de los conocimientos musicales humanos. Con ello, consiguen generar piezas de piano con

buenos resultados.

Los mismos autores de la mencionada representación REMI, reevaluaron dicha representación y la modificaron creando la tokenización Compound Word [7]. Este trabajo consiste en la generación de piano desde cero utilizando un modelo de *transformer* lineal. Se analiza la mejora que supone este tipo de representación de agrupación de los tokens. Este trabajo se explicará con más detalle en el siguiente capítulo, puesto que la tokenización elegida que se va a implementar es Compound Word,

Aunque existen diversos trabajos de generación multi-track como el comentado anteriormente, pocos de ellos incluyen la generación de percusión.

Uno de los trabajos que consiste en la generación automática de percusión, y sobre el cual se ha inspirado este trabajo, es *Conditional Drums Generation using Compound Word Representations* [10]. En este trabajo se aborda la tarea de generación de percusión de manera condicional utilizando Compound Word como método de tokenización de la música. Inspirándose en la representación original de Compound Word, proponen una variante de esta tokenización de manera que se dividen los datos de entrada en dos partes: las pistas de acompañamiento de guitarra y bajo, y su correspondiente percusión. (ver Figura 2.8)



Onset	Group	Type	Duration	Value
0.0	High-level	Bar	Bar	Bar
0.0	High-level	Tempo	Bar	125
0.0	High-level	Time Signature	Bar	4/4
0.0	Bass	Note	1.5	NaN
0.0	Guitar	Chord	1.0	NaN
...	...	...	...	...
3.5	Guitar	Chord	0.5	NaN
0.0	High-level	Bar	Bar	Bar

Onset	Drums
Bar	Bar
0.0	35
0.0	49
0.5	42
1.0	38
...	...
3.5	42
Bar	Bar

Figura 2.8: Modelo de Tokenización propuesto por [10]

En azul se resalta la representación de las pistas de acompañamiento y en gris la de la percusión. El token común en ambas representaciones es *Onset*, el cual representa el inicio de un evento medido respecto a notas negras en un compás. Respecto a la percusión que es lo que nos interesa, solamente se indica el *pitch* que especifica el instrumento de percusión como ya se ha mencionado.

En cuanto al modelo, utilizan una arquitectura basada en un modelo

Encoder-Decoder. Los datos de entrada de las pistas de acompañamiento se pasan por el codificador, que es un modelo LSTM Bidireccional. En cuanto a la percusión, se procesa de la misma manera que se va a hacer en este trabajo, la cual es la utilizada por los autores de Compound Word y que se explicará en el siguiente capítulo. Ambas entradas procesadas se pasan al decodificador, el cual consiste en un *Transformer Decoder with Relative Global Attention* [11]. La atención relativa [12] añade las posiciones realtivas, que capturan la distancia entre tokens de la secuencia. De esta manera, se captan mejor las dependencias entre tokens de diferentes posiciones.

Para el entrenamiento, utilizan la base de datos Lakh MIDI, seleccionando solamente las pistas de los géneros musicales Rock y Metal. Además, cortan todas las pistas para que tengan 16 compases para prevenir de secuencias demasiado largas. De esta manera obtienen 2.121 archivos MIDI, que dividen en un ratio de 8:1:1 para entrenamiento/validación/test respectivamente.

El codificador consiste en 3 capas BiLSTM con 512 unidades ocultas por cada capa. El decodificador consiste en un total de 4 capas de auto-atención con 8 cabezas de atención. Emplena una tasa de *dropout* de 0,3 para evitar sobreajuste.

Capítulo 3

Desarrollo e implementación del modelo

En el capítulo anterior se han explicado los conceptos más importantes en los que se basa este trabajo. En este capítulo se va a hacer incapié en el desarrollo de la implementación de la tokenización Compound Word, así como el modelo que se ha utilizado para el entrenamiento.

Como bien se ha mencionado, Compound Word [7] es una extensión de la representación REMI [5]. Consiste en agrupar los tokens consecutivos y relacionados entre sí que definen un evento musical en un "súper-token" o evento (*compound word*). Estos eventos tienen en cuenta el tipo de los tokens que incluyen, es decir, pueden definir los parámetros de una nota o la métrica de la canción marcando el inicio de un compás o de un tiempo. Dicho esto, se pueden definir dos tipos de eventos:

- Métrica o *Metrical*: marca el inicio de un nuevo compás (*Bar*) o de un tiempo (*Beat*[1]).
- Nota o *Note*: contiene las características de una nota que comienza en el *beat* actual.

En la tokenización REMI, una nota está compuesta por los tokens pitch, duración y velocidad, los cuales agruparemos en un súper-token (evento). Los cambios de tempo y de acorde solamente se indican al comienzo de un nuevo *beat*, por lo que juntaremos estos en otro súper-token.

Cada evento tendrá el mismo número de tokens, pero dependiendo del tipo solo serán relevantes algunos de ellos, de manera que los tokens no relevantes en ese evento

[1]El *beat* o tiempo de una canción se refiere al pulso o ritmo fundamental. Proporciona el tiempo y la estructura para el resto de los elementos musicales. Se percibe como un "latido" regular y suele mantener a lo largo de la canción para mantener el ritmo.

se les da el valor "*NONE*", que quiere decir que ese token se debe ignorar. Además, se añade un token adicional que indica el tipo del evento, si es métrica (M) o si es una nota (N).

En la figura 3.1 se representa el paso de la representación REMI a Compound Word tal y como se acaba de explicar. En (a) se representa una secuencia de tokens utilizando REMI. Seguidamente se agrupan los tokens en eventos (b). Finalmente, en (c) se pasa a la representación Compound Word, añadiendo los tokens restantes en cada evento, siendo los que aparecen en blanco los que serán irrelevantes dentro del evento.

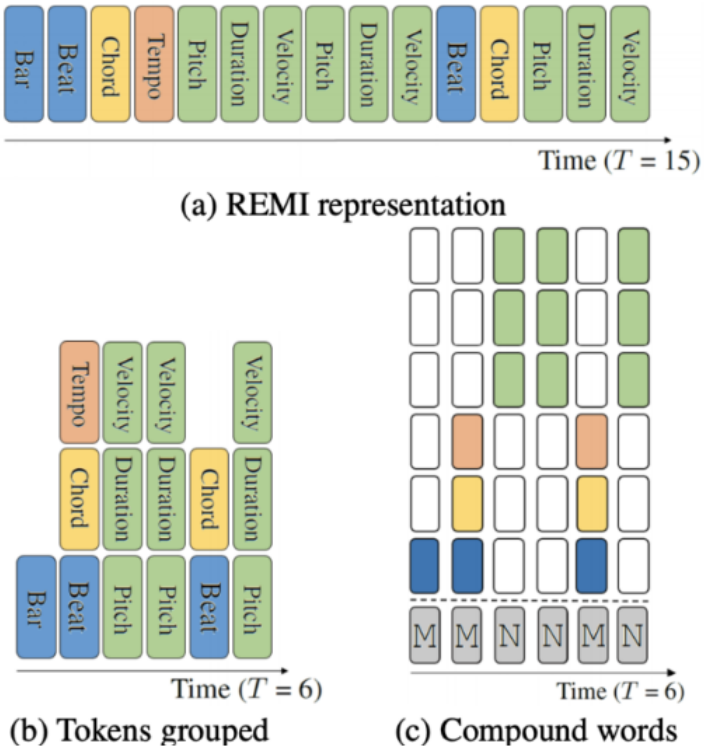


Figura 3.1: REMI a Compound Word [7]

3.1. Implementación de Compound Word

Para poder tokenizar un archivo MIDI, primero se deben obtener sus datos. Para ello, se ha hecho uso de la librería musicaiz [13], que proporciona los componentes necesarios para la generación, el análisis o la evaluación de música simbólica (MIDI).

La primera fase del desarrollo consiste en implementar la tokenización Compound Word en la librería musicaiz[2]. Mediante la ayuda de esta librería podemos obtener los compases, los instrumentos y las notas que contiene un archivo MIDI.

[2] `https://github.com/carlosholivan/musicaiz`

Lo primero de todo es definir los tokens que va a contener cada evento, tal y como se representa en la Figura 3.2.

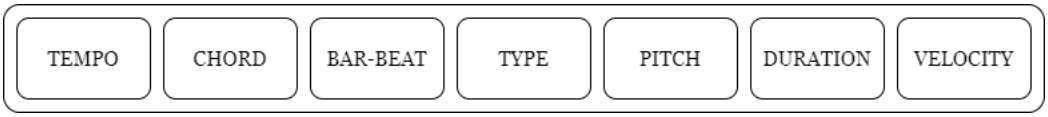


Figura 3.2: Tokens de un evento

Además al final de cada canción tokenizada se añadirá un evento que marcará el final tambien conocido como fin de secuencia o *end of sequence (EOS)*. Este evento tendrá todos los tokens con valor *NONE* excepto el que marca el tipo (*Type*) que tomará el valor *EOS*.

Cabe destacar que para una mayor precisión en cuanto a la tokenización de la duración de las notas, se ha optado por dividir cada compás en subtiempos (*subbeats*), en vez de en tiempos (*beats*). El número de *subbeats* en el que se dividirá cada compás será un parámetro que se podrá escoger antes de tokenizar un archivo, por defecto cada compás tendrá dieciséis *subbeats*.

Tokens	Descripción
Tempo	Indica el tempo[3] del subbeat que da comienzo.
Chord	Acorde del subbeat actual (en percusión se ignora siempre)
Bar-Beat	Marca el inicio de un nuevo compás (Bar) o de un subtiempo (Subbeat)
Type	Indica el tipo de evento (Nota, Métrica o EOS)
Pitch	Representa el pitch[4] de la nota, en percusión indica el instrumento
Duration	Se refiere a la duración de la nota. Esta se obtiene respecto a la duración de un subbeat, es decir, indica cuantos subbeats dura la nota.
Velocity	Indica la intensidad[5] con que se ha tocado la nota

Tabla 3.1: Descripción de los tokens

Una vez definidos y explicados en la Tabla 3.1 los tokens con los que se va a trabajar, el siguiente paso es analizar como la librería musicaiz devuelve los datos del archivo

[3]El tempo es la velocidad o ritmo a la cual se tocan los instrumentos. Se mide en pulsaciones por minuto (BPM).

[4]El pitch de una nota es su tono. Determina si una nota suena más grave o más aguda. Cada nota tiene una frecuencia de vibración asociada, la cual determina su pitch.

[5]La intensidad de una nota en MIDI se refiere a la velocidad de pulsación. Indica la fuerza con la que se toca una nota en un teclado MIDI.

MIDI.

Respecto a los instrumentos, se representan de la siguiente manera: *Instrument(program=10, is_drum=True, name='met_drum', family=unknown)*. Estos se definen principalmente por su número de programa que dependiendo de General MIDI tendrán un sonido predefinido. Como solamente vamos a tokenizar los intrumentos de percusión para entrenar la red, tokenizaremos únicamente las notas que pertenezcan al intrumento cuyo parámetro booleano *is_drum* sea *True*. Este parámetro existe ya que, como se ha explicado en el apartado de MIDI, los instrumentos de percusión son independientes de los demás y sólo se transmiten por el canal diez.

En cuanto a las notas, se tiene la siguiente información: *Note(pitch=60, name=C, start_sec=0.000000, end_sec=0.600000, start_ticks=0, end_ticks=120, symbolic, velocity=90, ligated=False, instrument_prog=0, bar_idx=0, beat_idx=0, subbeat_idx=0)*. Los parámetros más importantes son:

- *instrument_prog*: indica el número de programa del instrumento. Este nos servirá para filtrar las notas que sean tocadas por instrumentos de percusión, tal y como se ha explicado en el párrafo anterior.

- *bar_idx / beat_idx / subbeat_idx*: indican el compás, *beat* y *subbeat* al que pertenece la nota respectivamente.

- *symbolic*: representa la duración de la nota de forma simbólica. A partir de esta y de la duración de un *subbeat* se podrá obtener la duración de la nota.

Por último, podemos obtener los compases y los *subbeats*, que utilizaremos para indicar en los tokens el número de *subbeat* dentro de la canción. La representación de un *subbeat* es: *Subdivision(time_signature=TimeSig(num=4, den=4), bpm=120.0, start_ticks=200 end_ticks=225 start_sec=1.0 end_sec=1.125 global_idx=8 bar_idx=0 beat_idx=2)*. Como se puede observar también nos proporciona el tempo, parámetro musical muy importante, que también se usará para el cálculo de la duración.

Una vez estudiados todos los parámetros, se ha implementado la tokenización Compound Word[6] en la librería musicaiz utilizando Python como lenguaje de programación. Cabe destacar que esta implementación añade algunos tipos más de tokens que los mencionados, pero que se ha decidido no utilizar para adaptarse al modelo de los autores originales de Compound Word y poder utilizar su repositorio. Estos tokens adicionales son:

[6] `https://github.com/carlosholivan/musicaiz/blob/main/musicaiz/tokenizers/cpword.py`

– Program: indica el instrumento o número de programa de General MIDI que ha tocado la correspondiente nota. Este token solamente se utiliza en los eventos de tipo nota.

– Time Signature: representa la estructura rítmica y la organización del tiempo en una composición. Se representa mediante una fracción, en la que el numerador se refiere a la cantidad de tiempos o *beats* que se deben contar en cada compás, y el numerador indica la figura musical que representa un *beat*. Por ejemplo, un 4/4 indica que cada compás contiene cuatro tiempos, y la negra (cuarto de nota) representa un tiempo, es decir, en cada compás se deben contar cuatro negras.

– Rest: Indica los silencios (este token no se llegó a implementar por lo que se dejó siempre en *NONE*).

La clase implementada es *CPWordTokenizer* y en ella se han implementado tres funciones para dividir el proceso de tokenización:

– La función *tokenize_position* tokeniza un único *subbeat* dado con sus correspondientes notas.

– Mediante la función *tokenize_bar* se tokeniza un compás procesando cada *subbeat* por separado.

– Por último, para tokenizar un archivo MIDI, se utiliza la función *tokenize_file*, compás por compás.

3.2. Base de Datos

La base de datos con la que se ha trabajado ha sido *Lakh MIDI Dataset*[7] [14], consiste en una colección de 176,581 archivos MIDI únicos, de los cuales 45,129 han sido relacionados y alineados con entradas de la base de datos *Million Song Dataset* [15]. Su objetivo es facilitar la recuperación de información musical a gran escala, tanto simbólica (usando solo los archivos MIDI) como basada en contenido de audio (usando información extraída de los archivos MIDI).

En concreto, se ha utilzado el subconjunto *Clean*. Este consiste en un conjunto de archivos MIDI cada uno etiquetados con el nombre de la canción que contienen, además de estar separados por artistas. Esta sub-base de datos contiene 17,232 archivos MIDI.

Para entender un poco mejor la percusión en MIDI, se han extraído datos de todos estos archivos. La Figura 3.3 representa el número de veces que se repite cada pitch

[7] `https://colinraffel.com/projects/lmd/`

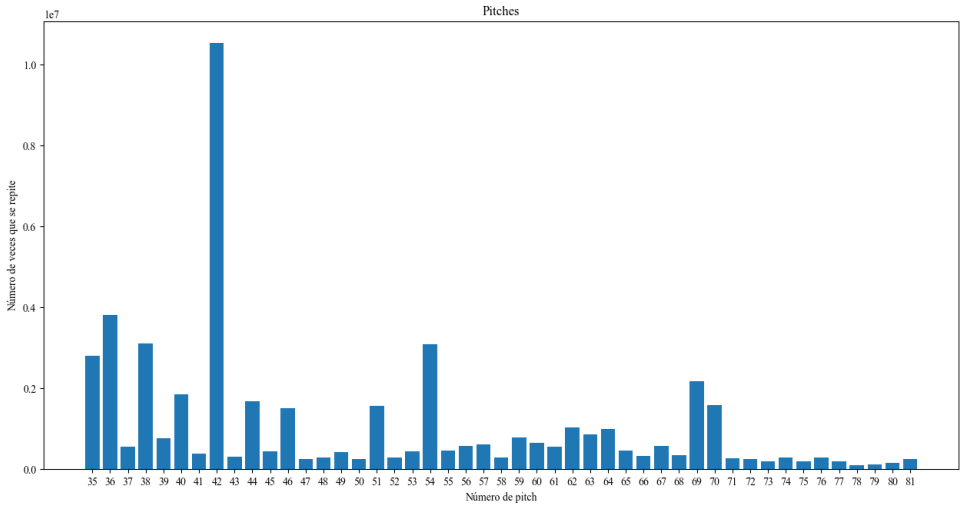


Figura 3.3: Número de pitches

en todos los archivos tokenizados. Como se ha comentado anteriormente, la percusión en MIDI se transmite solo por el canal diez y el número de pitch representa un único instrumento de percusión. Por lo tanto, si nos fijamos en el gráfico, los números que más se repiten son el 35, 36, 38 y 43, que si volvemos a la figura 2.6 vemos que corresponden respectivamente con el bombo (*bass*) ambos 35 y 36, la caja (*snare*) y el *hit-hat* cerrado. Esto tiene sentido ya que estos tres componentes de la batería son los que más se tocan a lo largo de una canción.

3.3. Modelo de Entrenamiento

Una vez se ha tokenizado la base de datos, el siguiente paso es entrenar. El modelo de *transformer* que se utiliza es el *Transformer Decoder*, ilustrado en la Figura 3.4. A diferencia del modelo *Transformer* convencional explicado en el capítulo anterior, el cual consta de un codificador y un decodificador, este modelo *Transformer Decoder* se enfoca únicamente en la parte del decodificador.

A la izquierda se realiza el procesamiento de los datos de entrada, los cuales serán la entrada de la capa de auto-atención. La entrada es una combinación de vectores de *embeddings*, uno por cada tipo de token. Una de las principales características de Compund Word es que se pueden adaptar las características a cada tipo de token individualmente. Esto permite utilizar diferentes tamaños de *embedding* para diferentes tipos de tokens. Seguidamente, estos vectores son concatenados y se pasan por un capa lineal, realizando la posterior codificación posicional explicada en el capítulo anterior.

De esta manera se obtienen los vectores de entrada a las de capas de *multi-head*

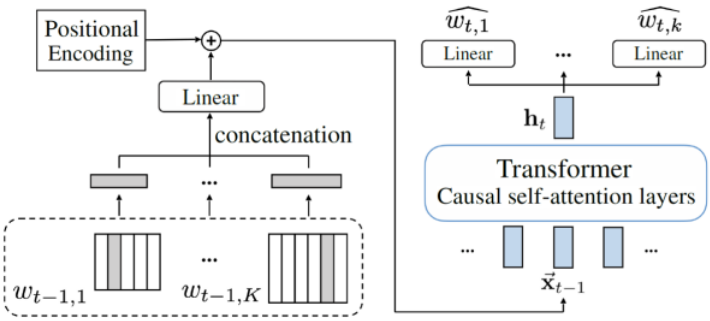


Figura 3.4: Modelo de la red utilizada [7]

self-attention. Una de las principales premisas de Compound Word es el uso de tantas cabezas de atención como tipos de tokens tenga la representación. Primero se predice el token que define el tipo de evento y a partir de este se generan los demás. En este caso las capas de auto-atención son causales, que son exactamente iguales que las que utilizan máscaras explicadas en el capítulo anterior. Por último, se pasa la salida de las capas de auto-atención por un capa lineal (una por cada tipo de token) que la convierte a probabilidades.

El modelo está configurado de manera que se pueden modificar los hiperparámetros que hacen entrenar la red neuronal, ya que no hay una sola manera correcta de hacerlo ni unos valores óptimos para todos los tipos de entrenamiento. Estos hiperparámetros son:

- El número de capas de auto-atención. Cuanto mayor sea el número de capas más compleja podrá ser la representación, pero también puede aumentar la complejidad computacional y el riesgo de sobreajuste o *overfitting*[8].

- La dimensión de los *embeddings*. Un número elevado permite una representación más detallada de los elementos del vocabulario, lo que conlleva a poder capturar mejor las relaciones semánticas y contextuales. Sin embargo, también implica un aumento en la cantidad de parámetros y puede requerir más recursos computacionales y memoria.

- El número de cabezas de atención. Cada cabeza se enfoca en diferentes partes de la secuencia y luego se combinan.

- La tasa de aprendizaje o *learning rate*. Es la velocidad a la que el modelo ajusta sus parámetros durante el entrenamiento. Es crucial escoger un valor adecuado

[8]El *overfitting* ocurre cuando un modelo de aprendizaje automático se ajusta demasiado a los datos de entrenamiento y no generaliza bien para nuevos datos. En otras palabras, el modelo memoriza los ejemplos de entrenamiento en lugar de capturar patrones que se pueden aplicar a datos no vistos.

para garantizar una convergencia estable y un buen rendimiento.

- *Hidden size*. Es el tamaño de la dimensión oculta de los vectores de representación en el modelo *Transformer*. Es el número de dimensiones que se utiliza para codificar la información en cada posición de la secuencia de entrada.
- El número de capas ocultas en la red neuronal *feed-forward*. Indica el tamaño de la capa oculta en la subcapa de alimentación directa en cada bloque de atención.
- *Batch size*. Indica cuántas muestras se utilizarán en cada paso de entrenamiento. Con un valor alto se puede aprovechar mejor el paralelismo y acelerar el entrenamiento, pero también puede requerir más memoria y poder de cómputo.
- La longitud máxima de la secuencia de entrada. Indicará el número de eventos máximo por canción. Si no se llega a este número se añadirá padding con el evento de fin de secuencia, y si se excede se cortará en este número, añadiendo el pertinente evento de fin de secuencia.
- Tasa de *Dropout*. Controla la proporción de neuronas o conexiones que se desactivan aleatoriamente durante el entrenamiento para evitar el *overfitting*.
- El número de épocas de entrenamiento. Se refiere a la cantidad de veces que se recorre todo el conjunto de datos durante el proceso de entrenamiento. Durante cada época, el modelo se ajusta a los datos de entrenamiento mediante la actualización de los parámetros del modelo.

Como se ha podido observar, escoger números apropiados para los hiperparámetros es muy importante a la hora de entrenar el modelo, ya que influye directamente en su comportamiento. Además, hay que tener en cuenta el número de parámetros totales, pues tiene un impacto significativo en su capacidad de representación y en su capacidad para aprender patrones y características de los datos. A medida que aumenta el número de parámetros, el modelo puede volverse más propenso al sobreajuste. También tiene un impacto en el coste computacional, cuantos más parámetros se requerirá más memoria y poder de procesamiento, además de que el entrenamiento será más lento. Es importante encontrar un equilibrio adecuado entre la capacidad de representación y la generalización del modelo, considerando los recursos disponibles y las restricciones del entorno de implementación.

Para valorar el entrenamiento se define la función de pérdida, que representa el error entre las predicciones generadas por el modelo y los valores reales o etiquetas asociados a los datos de entrenamiento. Para optimizar los resultados obtenidos por el modelo se debe minimizar esta pérdida.

La función de pérdida que se utiliza en este modelo es la *Cross-Entropy Loss* de *Pytorch*[9], representada en la ecuación (3.1).

$$L = -\sum_{i=1}^{N} w_{y_i} \log \left(\frac{\exp(x_{i,y_i})}{\sum_{c=1}^{C} \exp(x_{i,c})} \right) \tag{3.1}$$

Donde x es la entrada, y es el objetivo, w es el peso, C es el número de clases (numero de tokens en el vocabulario) y N representa el *batch size*.

[9] `https://pytorch.org/`

Capítulo 4

Experimentos y resultados

Una vez definida la tokenización y la base de datos con la que se va a entrenar al modelo, el siguiente paso en realizar experimentos a partir de diferentes entrenamientos.

El proceso tanto de entrenamiento como de generación se ha realizado mediante la herramienta Google Colaboratory (Colab), una herramienta para escribir y ejecutar cuadernos de Python en la nube. En ella se hizo una copia del cuaderno ofrecido por los autores de Compound Word, en el cual ya está implementado el modelo de entrenamiento. Mediante este cuaderno podemos escoger el número de épocas que queremos entrenar el modelo y este nos va informando de la pérdida en cada una. De esta manera podemos optimizarlo para minimizar dicha pérdida. Además, se ha modificado ligeramente el código original para adaptarlo mejor a la implementación. Lo mejor que ofrece es el hecho de poder entrenar a partir del punto anterior donde lo dejamos, gracias a su posibilidad de conectarlo a Google Drive, pués se van guardando los parámetros de cada época entrenada. Estos serán a los que se accederá posteriormente para las fases de test y generación. Sin embargo, la versión gratuita de Colab no fue suficiente debido a sus limitaciones en los cortos tiempos de ejecución que permitía, por lo que se compró una suscripción de 1 mes a Colab Pro, pudiendo tener acceso a GPUs más rápidas como la GPU A100 o P100, más memoria y tiempos de ejecución más largos.

4.1. Experimentos

En el modelo de entrenamiento utilizado para hacer los diferentes experimentos que se van a explicar, se han utilizado los mismos hiperparámetros que utilizan los autores de Compound Word:

- Número de capas de auto-atención = 12.
- Número de cabezas de atención = 8

- *Hidden Size* = 512
- La dimensión de los *embeddings* depende del tamaño del vocabulario para cada tipo de token.
- Número de capas ocultas en la red *feed-forward* = 2,048
- Tasa de *dropout* = 0.1
- Tasa de aprendizaje = 0.0001
- Tamaño máximo de secuencia de entrada = 3,584
- *Batch size* = la mitad del número de datos de entrada.
- El número de épocas de entrenamiento ha ido variando dependiendo del experimento y del resultado que se iba obteniendo.

Un concepto importante que hay que tener en cuenta a la hora de entrenar una red neuronal es la división de los datos en dos conjuntos: los que se utilizan para entrenar y que la red aprenda, llamados datos de entrenamiento; y los que se utilizan para validar si realmente ha aprendido o no, llamados datos de test. Esto se hace porque se necesita que la red aprenda para un conjunto de datos que muestren las mismas características y no para un conjunto de datos reducido. De esta manera, si las pérdidas son similares en ambos, significa que la red está aprendiendo de manera correcta. Por lo contrario, la red puede presentar sobreajuste, lo que puede significar una falta de datos de entrenamiento, ya que la pérdida de entrenamiento sigue bajando pero la de test no. Una buena elección es realizar una división de 80/20, es decir, el 80 % para entrenamiento y el 20 % para test.

4.1.1. Primer Experimento

Como primer experimento se optó por tokenizar una pequeña parte de base de datos y entrenar a la red con estos datos para ver su comportamiento. Se seleccionaron unos cuantos artistas de la base de datos, obteniendo así 2.576 archivos. El tiempo necesario para tokenizar estos archivos fue de unas siete horas, aproximadamente. Una vez terminada la tokenización, se procesaron correctamente 1.844 archivos. Al realizar la división (aleatoriamente) se obtuvieron 1.475 archivos para el entrenamiento (80 %) y 369 para test (20 %). Posteriormente, se entrenó la red con un tamaño de *batch* de 737 y 39 millones de parámetros, llegando a obtener una pérdida igual a 0,30 en entrenamiento. Sin embargo, al realizar el test de los parámetros obtenidos, se obtuvo

una pérdida bastante mayor, en concreto 0,766. Esto significa que se produjo en este primer experimento un sobreajuste durante el proceso de entrenamiento.

Al intentar generar música, se obtuvieron secuencias de datos incorrectas ya que los *subbeats* estaban generados de forma desordenada. Este problema se puede deber a que se tokenizan solamente los *subbeats* que contienen notas, lo cual hace que al modelo le cueste más entender que existe un orden. Es decir, si en una canción la percusión comienza en un compás en el que las notas se encuentran en el *subbeat* siete, dicha canción tokenizada empezará con un evento en el que token *Subbeat* tome el valor *Subbeat_7* y, si en otra canción diferente la percusión comienza en el *subbeat* cero, esta canción tokenizada empezará con un evento con el token *Subbeat* con el valor *Subbeat_0*. Este problema podría solucionarse entrenando durante más tiempo la red para obtener una pérdida menor o también con mayor cantidad de datos de entrada.

4.1.2. Segundo Experimento

En un segundo experimento se optó por tokenizar la base de datos completa y entrenar hasta obtener una pérdida decente. El tiempo que costó tokenizar la base de datos completa fue de tres días completos, aproximadamente. De los 17.232 archivos MIDI que contiene la base de datos, se procesaron correctamente 12.217, que separando de nuevo en un 80/20 %, se tienen 9.773 archivos para el entrenamiento y 2.444 para test. Al aumentar considerablemente los datos de entrenamiento, se pudo observar que el tamaño del *batch* aumentó a 4.886, así como el número de parámetros en 100 mil más que antes, lo que conlleva a que el tiempo de entrenamiento también aumente considerablemente. Debido a esto, se obtuvo una pérdida de 0,40, que en test es del 0,529, lo cual quiere decir que se consiguió una mejora respecto al experimento anterior. De nuevo se ha intentado generar música, pero sin éxito, ya que surge el mismo problema anteriormente mencionado: los *subbeats* se generan de forma desordenada. Esto quiere decir que con aumentar la cantidad de datos no es suficiente por lo que se debe buscar otra estrategia.

4.1.3. Tercer Experimento

Debido a que no se consiguieron los resultados esperados, se optó por revisar el método de tokenización. Como se ha explicado anteriormente, uno de los tokens que puede haber en un evento es el token *Subbeat*, el cual toma el valor del *subbeat* que contiene notas. Los *subbeats* se numeran de forma global respecto a la pista completa, es decir, el token puede tomar valores desde 0 hasta (B * 16 - 1) siendo B el número de compases totales de la pista. Esto hace que en el vocabulario, el número de tokens

únicos que representan los *subbeats* sea demasiado grande, lo cual hace que haya más parámetros en el modelo y le sea muy complicado aprender el orden. Por esto, se optó por numerar los *subbeats* respecto a cada compás, es decir, cada compás de la pista tendrá *subbeats* del 0 al 15 (al dividirlo en 16 partes). Al realizar este cambio, se minimiza mucho el tamaño del vocabulario, y por tanto, el número de parámetros. De esta manera se consigue que el modelo comprenda mejor el orden de los *subbeats*.

Para comprobar que este cambio podría mejorar el entrenamiento, se modificaron los archivos utilizados en el primer experimento y se entrenó el modelo, llegando a obtener una pérdida de 0,18. Aunque esta es bastante baja, en el proceso de test se obtiene una pérdida de 0,472 con estos parámetros. Sin embargo, al intentar generar música, se observó de nuevo que algunos subbeats se generan desordenados, pero en este caso dicho problema no afecta tanto y se han podido obtener archivos MIDI, realizando el proceso inverso de tokens a MIDI utilizando la librería *miditoolkit*[1].

4.1.4. Cuarto Experimento

Con el objetivo de mejorar los resultados, se realizó un último experimento en el que se aplicó el cambio explicado en el experimento 3 a la base de datos completa y se volvió a entrenar el modelo. La pérdida a la que se llegado en el entrenamiento es de 0,22, que aunque sea mayor que en el anterior experimento, es mejor ya que representa un 0,331 en test.

4.2. Resultados

Para realizar una valoración global del entrenamiento de cada experimento. Se han plasmado las pérdidas obtenidas en los experimentos anteriores en la tabla 4.1, así como su equivalente en la fase de test.

	Pérdida Entrenamiento	Pérdida Test	Nº Datos Entrenamiento	Subbeat
Experimento 1	0,35 0,30	0,759 0,766	1.475	Global
Experimento 2	0,50 0,40	0,587 0,529	9.773	Global
Experimento 3	0,20 0,18	0,476 0,472	1.475	Compás
Experimento 4	0,25 0,22	0,351 0,331	9.773	Compás

Tabla 4.1: Mínimas pérdidas obtenidas en cada experimento

[1] `https://github.com/YatingMusic/miditoolkit`

En el experimento 1 que se refiere al entrenamiento con solamente una parte de la base de datos y *subbeats* globales en la tokenización, se puede observar que ha habido claramente un sobreajuste, ya que la pérdida de test es mucho mayor que la de entrenamiento. Esto se puede solucionar agregando más datos, por lo que para el experimento 2 se emplea la base de datos completa. Ha habido una mejora considerable, pero como se ha comentado los resultados siguen sin ser los esperados. Para intentar mejorar el comportamiento del modelo, se ha aplicado el cambio explicado en los *subbeats*. En el experimento 3 se utiliza de nuevo solo una parte de la base de datos, pero aplicando dicho cambio en la tokenización. Si lo comparamos con el primer experimento, la mejora es evidente, ya que no hay tanto *overfitting*, pero todavía se puede mejorar. Por lo que, por último, en el experimento 4 se utiliza la base de datos completa con dicho cambio, en el cual se aprecia que la diferencia entre la pérdida de test y la de entrenamiento es mucho más baja, y que, por lo tanto, el entrenamiento ha mejorado y los resultados obtenidos serán mejores.

Gracias al cambio realizado en la tokenización, se han podido obtener archivos MIDI que se pueden escuchar. Además, ha disminuido el tamaño del vocabulario, el número de parámetros, y con ello el tiempo de entrenamiento. Esto hace que para obtener la misma pérdida que en los dos primeros experimentos, se pueda entrenar menos épocas.

A la hora de generar, se han generado 10 archivos en cada experimento. En la tabla 4.2 se plasman los tiempos promedios que se tarda en cada experimento en generar cada canción. Se puede apreciar que cuanto mejor es el entrenamiento, mayor es el tiempo que le cuesta generar una canción. Esto es debido a que intenta generar mejores ritmos y canciones más largas.

	Pérdida	Song Time (segundos)
Experimento 3	0,18	63,2624
	0,20	46,7834
	0,25	26,0573
Experimento 4	0,22	97,8012
	0,30	99,2383

Tabla 4.2: Resultados temporales en la generación

En la Figura 4.1 se puede observar la representación *pianoroll* de la percusión de un archivo de la base de datos. Se puede apreciar como se mantiene un patrón en los distintos instrumentos, lo cual es normal en percusión.

En el experimento 3 se han generado archivos y se han escuchado, siendo el que mejor resultado ofrece el representado en la Figura 4.2. Como se puede observar puede llegar a parecerse a un ritmo convencional, pero sigue siendo un resultado muy pobre.

Para intentar mejorar se han aumentado los datos, llevando a cabo el experimento

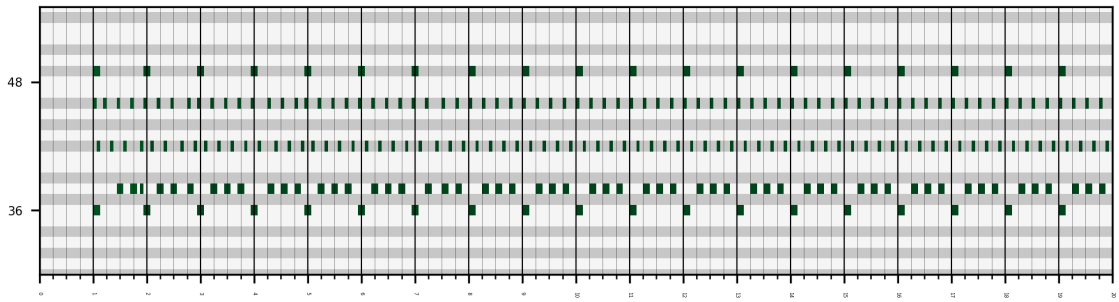


Figura 4.1: Representación *pianoroll* de un archivo MIDI de la base de datos

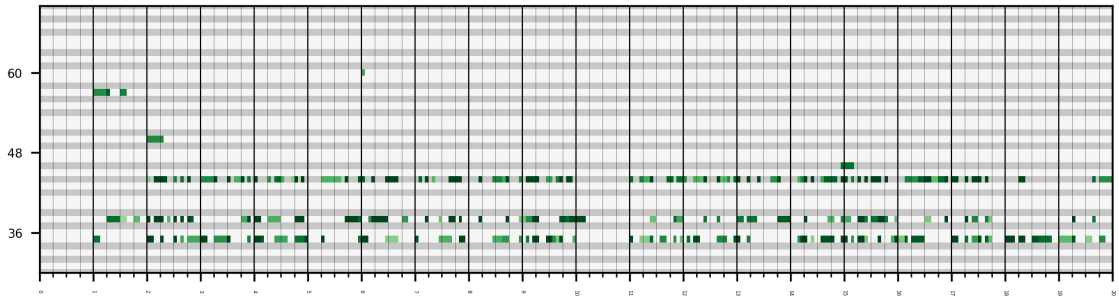


Figura 4.2: Representación *pianoroll* de un archivo MIDI generado durante el experimento 3

4. En la Figura 4.3 se puede observar la representación *pianoroll* de un archivo generado medianamente decente. En este se observa que se intenta llevar un ritmo, pero no acaba de tener éxito.

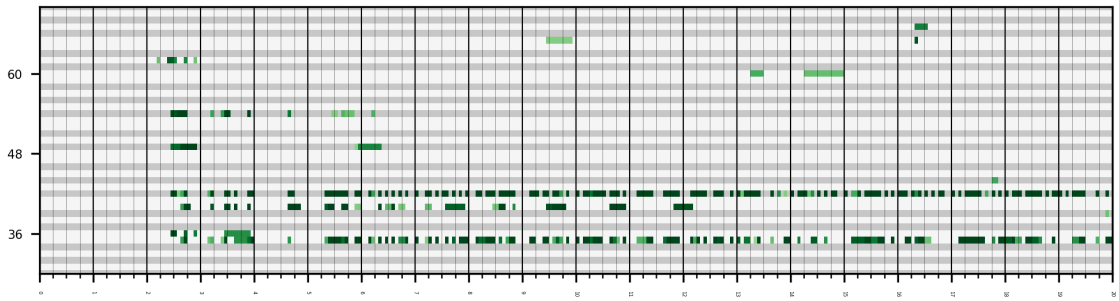


Figura 4.3: Representación *pianoroll* de un archivo MIDI generado durante el experimento 4

Una vez escuchados los archivos generados se ha hecho una valoración final. Los resultados obtenidos no son tan buenos como cabía esperar. Este problema puede ser ocasionado por diversos factores: por no estar utilizando el modelo apropiado, por falta o calidad de datos de entrenamiento o por no ser Compound Word la tokenización adecuada para percusión.

Como ya se ha comentado, la percusión es una parte de la música que prácticamente

no se ha trabajado en la generación automática, por lo que es difícil realizar diversas comparaciones con otros trabajos. Como ya se ha mencionado, el trabajo *Conditional Drums Generation using Compound Word Representations* ha sido el que ha servido de inspiración para este trabajo por lo que se va a realizar un análisis respecto a este.

Lo primero que hay que destacar es que en dicho trabajo se realiza generación condicionada por acompañamiento de entrada. Eso quiere decir, que la generación de la percusión se adapta a las pistas de una guitarra y un bajo de entrada. En nuestro caso la generación es desde cero, sin ningún tipo de condicionamiento.

El modelo y los hiperparámetros que utilizan se han explicado previamente en el apartado 2.4.

En la Figura 4.4 se representa una de las piezas que generan (aislando la percusión) y que se pueden encoontrar en su repositorio[2]. En esta sí que se puede apreciar el ritmo.

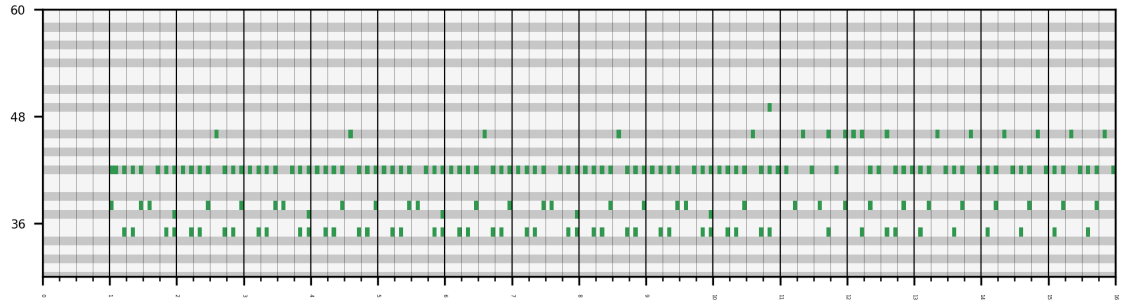


Figura 4.4: Representación *pianoroll* de un archivo MIDI generado por los autores de *Conditional Drums Generation using Compound Word Representations*

Cabe destacar que el modelo utilizado para el entrenamiento por lo autores de Compound Word está enfocado a la generación de piano, y por la tanto, tiende a generar pistas más melódicas que rítmicas. Esto en gran parte es lo que produce que las pistas generadas pierdan el ritmo que se busca, y también por esto, al escucharlas, suenen estilo jazz.

[2] `https://github.com/melkor169/CP_Drums_Generation`

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

A lo largo del desarrollo de este trabajo, se ha implementado por completo y desde cero la tokenización Compound Word en la librería *musicaiz*. Se ha estudiado el modelo de entrenamiento utilizado y su implementación por sus autores, realizando pequeños cambios en este, y se ha entrenado con una base de datos previamente procesada.

El objetivo principal de este trabajo era conseguir generar ritmos de percusión automáticamente desde cero. Valorando los resultados obtenidos se puede decir que se ha cumplido este objetivo pues se han obtenido archivos MIDI reproducibles aunque carecen de la estructura rítmica deseada y no son musicalmente adecuados.

La generación de percusión es una tarea muy novedosa y que tiene una gran complejidad, pues requiere una forma de interpretación musical muy específica en la que se requiere que la red entienda los diferentes patrones rítmicos y sea capaz de encadenarlos adecuadamente.

Para poder entrenar durante más épocas con el objetivo de disminuir todavía más la pérdida y comprobar así si el resultado obtenido mejora, serían necesarios más recursos de los que se ha dispuesto para la elaboración de este trabajo.

Por otro lado, si se quisiera utilizar la base de datos Lakh MIDI totalmente completa y no los subconjuntos que se han utilizado, se tardaría casi 10 veces más de tiempo en tokenizarse, es decir, un mes completo, y no se ha dispuesto ni de ese tiempo ni de mejores recursos para agilizar el proceso. Además en este caso, Google Colab no sería una buena herramienta para el entrenamiento, debido a los límites de memoria y de procesado, y si se quisiese tener acceso a mejores recursos en Colab, es necesario pagarlos.

Con mejores recursos y, sobre todo, con mucho más tiempo, se podría haber realizado un estudio con diferentes tipos de tokenización y modelos de entrenamiento para ver cual funciona mejor.

Para concluir, el desarrollo del trabajo ha sido provechoso pues se ha aprendido sobre nuevos conceptos como la tokenización, MIDI, Deep Learning o los Transformers. Se ha llegado a un punto en el que aunque los resultados obtenidos no son los que cabría esperar, estos son escuchables y se han apreciado indicios de que la red parece generar un ritmo acompasado. Esto lleva a la conclusión de que en estudios futuros se puedan mejorar los resultados considerablemente.

5.2. Líneas Futuras

Como trabajo futuro se propone realizar un nuevo entrenamiento, en el que se disminuyan el número de capas de atención del Transformer y también se disminuya el tamaño máximo de las secuencias de los datos de entrada. De esta manera se buscará que la red se concentre en pasajes más cortos que se adaptan mejor a la realidad de la creación musical de ritmos.

Para el entrenamiento se ha utilizado un subconjunto de la base de datos Lakh MIDI la cual contiene diferentes estilos musicales. En los ritmos de percusión, el género musical tiene mucho que ver tanto en la manera de tocar como en el propio ritmo, es decir, no tiene nada que ver un ritmo de jazz con uno de rock por ejemplo. Una idea para trabajo futuro sería realizar el entrenamiento con canciones de un género en específico, incluyendo en género en la tokenización, para poder condicionar la red en el proceso de generación.

También se propone hacer un estudio a partir de múltiples experimentos para diferentes modelos de entrenamiento, utilizando diferentes tipos de tokenización para analizar cuál es la más adecuada para la percusión.

Por último, en cuanto a la base de datos, realizar un procesado más exhaustivo, filtrando las canciones por géneros musicales y haciendo la segmentación de cada una de ellas dividiendo ritmos correspondientes a la introducción, a las estrofas, estribillos, puentes, etc. Este trabajo debería realizarse a mano, ya que todavía no existen herramientas fiables de segmentación musical y detección de estructura. De esta manera, el modelo entendería mucho mejor las diferentes partes de una canción y se podrían generar ritmos de percusión con estructura.

Capítulo 6

Materiales

En este capítulo se van a enumerar recursos adicionales que han sido de gran utilidad para el desarrollo de este trabajo.

- Especialización Deep Learning en Coursera, instruído por Andrew Ng de Deeplearning.ai. `https://www.coursera.org/specializations/deep-learning`
- La librería MidiTok de python, de la cual se ha hecho uso para entender la tokenización. `https://github.com/Natooz/MidiTok`
- Blog de Jay Alammar en el que explica detalladamente los Transformers. `http://jalammar.github.io/illustrated-transformer/`
- Wikipedia para definir conceptos básicos como MIDI. `https://es.wikipedia.org/wiki/Wikipedia:Portada`

Capítulo 7

Bibliografía

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] H. Andrade, S. Sinche, and P. Hidalgo, "Modelo para detectar el uso correcto de mascarillas en tiempo real utilizando redes neuronales convolucionales," *Revista de Investigación en Tecnologías de la Información*, vol. 9, pp. 111–120, 01 2021.

[3] J. Schmidhuber, S. Hochreiter, *et al.*, "Long short-term memory," *Neural Comput*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[5] Y.-S. Huang and Y.-H. Yang, "Pop music transformer: Beat-based modeling and generation of expressive pop piano compositions," in *Proceedings of the 28th ACM International Conference on Multimedia*, pp. 1180–1188, 2020.

[6] S. Oore, I. Simon, S. Dieleman, D. Eck, and K. Simonyan, "This time with feeling: Learning expressive musical performance," *Neural Computing and Applications*, vol. 32, pp. 955–967, 2020.

[7] W.-Y. Hsiao, J.-Y. Liu, Y.-C. Yeh, and Y.-H. Yang, "Compound word transformer: Learning to compose full-song music over dynamic directed hypergraphs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 178–186, 2021.

[8] N. Fradet, J.-P. Briot, F. Chhel, A. El Fallah Seghrouchni, and N. Gutowski, "MidiTok: A python package for MIDI file tokenization," in *Extended Abstracts for the Late-Breaking Demo Session of the 22nd International Society for Music Information Retrieval Conference*, 2021.

[9] J. Ens and P. Pasquier, "Mmm: Exploring conditional multi-track music generation with the transformer," *arXiv preprint arXiv:2008.06048*, 2020.

[10] D. Makris, G. Zixun, M. Kaliakatsos-Papakostas, and D. Herremans, "Conditional drums generation using compound word representations," in *Artificial Intelligence in Music, Sound, Art and Design: 11th International Conference, EvoMUSART 2022, Held as Part of EvoStar 2022, Madrid, Spain, April 20–22, 2022, Proceedings*, pp. 179–194, Springer, 2022.

[11] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, C. Hawthorne, A. M. Dai, M. D. Hoffman, and D. Eck, "Music transformer: Generating music with long-term structure," *arXiv preprint arXiv:1809.04281*, 2018.

[12] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," 2018.

[13] C. Hernandez-Olivan and J. R. Beltran, "Musicaiz: A python library for symbolic music generation, analysis and visualization," *SoftwareX*, vol. 22, p. 101365, 2023.

[14] C. Raffel, *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching. 331 Ph. D.* PhD thesis, thesis, Columbia University, 2016.

[15] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

Lista de Figuras

Lista de Tablas