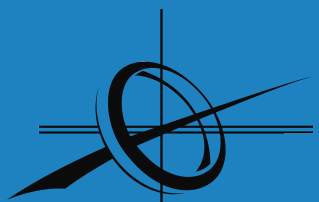
POLITÉCNICA



Universidad Politécnica de Madrid

ETSI SISTEMAS INFORMÁTICOS

Aplicación de técnicas de fine-tuning sobre un modelo GPT para la composición de música a piano.

Proyecto Fin de Grado

Grado en Ingeniería del Software

Autor:
David Ramos Archilla

Tutor:
Alejandro Martín García

4/7/2023

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Aplicación de técnicas de fine-tuning sobre un modelo GPT para la composición de música a piano.

Proyecto Fin de Grado

Grado en Ingeniería del Software

Curso académico 2022-2023

Autor:
David Ramos Archilla

Tutor:
Alejandro Martín García

Quisiera aprovechar este espacio para expresar mi más sincero agradecimiento a las personas que han sido fundamentales en la realización de mi trabajo de fin de grado. Este momento no habría sido posible sin el apoyo y el aliento de aquellos que me han acompañado en este viaje académico. En primer lugar, quiero agradecer profundamente mi familia, gracias a ellos he llegado hasta donde estoy hoy. Por último, quiero agradecer el apoyo que me ha dado mi tutor, que me ha guiado y ayudado durante toda la realización del proyecto.

Resumen

Este trabajo presenta un estudio sobre el desarrollo de un modelo de inteligencia artificial, basado en un Transformer, que ha sido reentrenado utilizando tres conjuntos de datos diferentes de estilos de música clásica. El objetivo de este modelo es generar composiciones musicales basadas en una melodía proporcionada como entrada, cuya continuación se asemejará a uno de los estilos sobre los que ha sido reentrenado el modelo. El proceso de reentrenamiento ha consistido en exponer al modelo a los tres conjuntos de datos, cada uno de los cuales representaba un estilo musical clásico distinto. Aprovechando la gran cantidad de datos disponibles en estos conjuntos, los modelos pudieron aprender los patrones y características propias de cada estilo, permitiendo así poder componer música de cada uno de ellos.

El desarrollo del trabajo ha consistido en varias etapas. En primer lugar, hay que elegir cuidadosamente los conjuntos de datos sobre los que se va a reentrenar el modelo. Esto se ha logrado a través de elegir un determinado número de obras de grandes compositores clásicos, que estaban contenidas en un conjunto de datos aún mayor y con una gran variedad de obras. Luego, se tuvo que realizar un preprocesamiento a estos datos, ya que las redes neuronales solo entienden números; cosa que se consiguió empleando un algoritmo de tokenización. Por último, una vez que se haya realizado todo lo anterior, llega el momento del entrenamiento. Partiendo de un modelo base preentrenado, se han empleado técnicas de fine-tuning, que consisten en ajustar algunos de los parámetros del modelo mediante un reentrenamiento con el objetivo de que el modelo aprenda los patrones y características propios de cada uno de los conjuntos de datos. Esto se hizo de manera independiente con cada uno de ellos.

Los resultados obtenidos muestran el potencial del reentrenamiento de modelos de inteligencia artificial para generar música en diferentes estilos clásicos. Los modelos reentrenados mostraron una notable capacidad para componer piezas musicales coherentes y específicas de cada género cuando se le proporcionaba una melodía como entrada. Adicionalmente, se ha creado un notebook `https://colab.research.google.com/github/DavidRamosArchilla/GIGA-Piano/blob/main/GIGA_Piano_Composer.ipynb`) en Google Colab para que cualquier persona, sin importar sus conocimientos técnicos, pueda probar por su cuenta lo que se ha desarrollado en este trabajo y componga su propia música.

Abstract

This work presents a study on the development of an artificial intelligence model, based on a Transformer, that has been retrained using three different datasets of classical music styles. The objective of this model is to generate musical compositions based on a provided melody as input, with the continuation resembling one of the styles on which the model has been retrained. The retraining process involved exposing the model to the three datasets, each representing a distinct classical music style. Leveraging the vast amount of data available in these sets, the models were able to learn the patterns and unique characteristics of each style, thus enabling the composition of music in each of them.

The development of the project has consisted of several stages. Firstly, it was necessary to carefully select the datasets on which the model would be retrained. This was achieved by choosing a specific number of works from prominent classical composers, which were part of a larger dataset containing a wide variety of compositions. Subsequently, data preprocessing was performed as neural networks only understand numbers. This was accomplished by employing a tokenization algorithm. Finally, once all the previous steps were completed, the training phase commenced. Starting from a pre-trained base model, fine-tuning techniques were utilized, which involved adjusting certain parameters of the model through retraining, aiming for the model to learn the specific patterns and characteristics of each dataset. This process was carried out independently for each dataset.

The obtained results demonstrate the potential of retraining artificial intelligence models to generate music in different classical styles. The retrained models exhibited a remarkable ability to compose coherent and genre-specific musical pieces when provided with a melody as input. Additionally, a Google Colab notebook(`https://colab.research.google.com/github/DavidRamosArchilla/GIGA-Piano/blob/main/GIGA_Piano_Composer.ipynb`) has been created to allow anyone, regardless of their technical knowledge, to independently test and compose their own music using the developments made in this work.

Índice

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto

En la última década, los avances en el campo del aprendizaje automático y las redes neuronales han dado lugar a una nueva ola de tecnologías capaces de producir contenido original en diversos dominios creativos. Entre ellas han destacado los large language models (LLM) [1], modelos de redes neuronales con una gran cantidad parámetros que utilizan la arquitectura Transformer y que han sido entrenados en grandes cantidades de texto sin etiquetar. En particular, modelos como GPT-3[2] y GPT-4[3] han demostrado una capacidad notable para comprender y producir texto coherente y de alta calidad en una amplia variedad de tareas y dominios. Por otra parte, estos últimos años han tenido un gran auge los modelos texto-a-imagen, que son modelo que a partir de una frase crean una imagen. Esto ha sido posible gracias a los modelos de difusión como Dall-e [4] o Stable Diffusion[5], modelos que generan resultados de buena calidad y que han supuesto una revolución en el arte generativo. Estos modelos tienen también una enorme cantidad de parámetros y han sido entrenados con grandes cantidades de imágenes de las que se tiene una breve descripción sobre ellas.

La capacidad de componer música de manera automática ha sido un desafío en el campo de la inteligencia artificial. Los avances mencionados en este campo, han abierto nuevas posibilidades en el ámbito de la generación de música, y es en este contexto en el que se plantea el desarrollo de un modelo de inteligencia artificial para la composición musical. Para ello, se pretende modificar un large language model preentrenado para que este sea capaz de componer música. Hay que tener en cuenta que aplicar este enfoque al dominio de la música presenta desafíos adicionales, ya que la música no se puede representar directamente en forma de texto. Para lograr esto, se crearán conjuntos de datos sobre tres estilos musicales diferentes, que contendrán ejemplos representativos del estilo correspondiente. Estos conjuntos de datos servirán para el entrenamiento del modelo y permitirán capturar las características distintivas de cada género musical. Se elegirá un modelo inicial preentrenado que sea capaz de generar música, considerando diferentes aspectos como por ejemplo su calidad. Posteriormente, se realizará un proceso de reentrenamiento del modelo utilizando los conjuntos de datos ya mencionados. Esto permitirá que el modelo aprenda las particularidades de cada género musical y pueda generar composiciones que se asemejen a los datos de entrenamiento.

1.2. Objetivos

En este trabajo se propone un modelo de inteligencia artificial que es capaz de componer música en base a una secuencia de notas proporcionada como

entrada. Se podrá elegir el estilo musical de las composiciones generadas de manera automática entre los estilos barroco, romántico y vanguardista.

Teniendo esto en cuenta, se definen los siguientes objetivos:

- Crear diferentes conjuntos de datos para cada uno de los estilos musicales sobre los que el modelo realizará composiciones.
- Elegir un modelo inicial preentrenado capaz de generar música de entre todos los modelos del estado del arte, adecuado para las limitaciones computacionales de este trabajo.
- Reentrenar un modelo capaz de generar música con conjuntos datos de diferentes estilos musicales con el fin de que las composiciones que haga cada modelo se asemejen a los datos de su conjunto de entrenamiento.
- Realizar un pequeño producto para que las personas sin conocimientos técnicos puedan experimentar componiendo música.

1.3. Estructura del documento

En esta sección se explica como está estructurado este documento.

- El capítulo 1 sirve de una breve introducción al trabajo, indicando el contexto en el que se elabora y como se estructura el reto del documento.
- El capítulo 2 ofrece una perspectiva del estado del arte y tecnologías que han sido usadas durante el desarrollo del trabajo.
- El capítulo 3 es una explicación detallada de cómo ha sido realizado el trabajo.
- El capítulo 4 muestra resultados que se han obtenido tras la elaboración del trabajo.
- El capítulo 5 contiene las conclusiones extraídas tras realizar el trabajo y posibles trabajos futuros.

Capítulo 2

Estado del arte

2.1. Inteligencia Artificial

La inteligencia artificial (IA) es un campo de estudio que busca desarrollar sistemas y programas capaces de imitar, simular o superar la inteligencia humana en tareas específicas. Se basa en el uso de algoritmos y modelos matemáticos complejos para procesar grandes cantidades de datos y aprender de ellos, permitiendo a las máquinas realizar tareas que normalmente requerirían de la intervención humana.

La IA se centra en la construcción de agentes inteligentes que pueden percibir su entorno, tomar decisiones racionales y actuar de manera autónoma para alcanzar objetivos específicos. Estos agentes pueden estar diseñados para realizar tareas muy diversas, como reconocimiento de voz, detección de objetos, traducción automática, diagnóstico médico, conducción autónoma, generación de nuevos datos, entre otras.

Figura 2.1: Mapa conceptual de la inteligencia artificial

Dentro de la inteligencia artificial hay una amplia gama de técnicas y enfoques, entre los que se destacan el aprendizaje automático (o machine learning), la lógica difusa, el procesamiento del lenguaje natural, la visión por ordenador, algoritmos genéticos y los sistemas expertos. Estas técnicas se combinan y adaptan según el problema específico a resolver y los datos disponibles.

A pesar de los avances significativos, la inteligencia artificial también plantea desafíos y preocupaciones. Entre ellos se incluyen la ética en la toma de decisiones automatizadas, la privacidad y la seguridad de los datos, así como el impacto en el empleo y la sociedad en general. Es fundamental abordar estos aspectos de manera responsable y garantizar que la IA se utilice para el beneficio humano y el progreso social y nunca con malos fines.

Con todo esto, la inteligencia artificial es un campo en constante evolución que busca desarrollar sistemas y programas que traten de simular y superar la inteligencia humana en tareas específicas. Mediante el uso de algoritmos y modelos complejos, la IA permite a las máquinas percibir, aprender y tomar decisiones autónomas, lo que ha llevado a avances significativos en diversos sectores y promete transformar nuestra sociedad en el futuro.

2.1.1. Redes neuronales

Las redes neuronales [6] son un tipo de modelo computacional inspirado en el funcionamiento del cerebro humano, que se utiliza en el campo de la inteligencia artificial y el aprendizaje automático. Estas redes están compuestas por unidades llamadas neuronas artificiales, que se organizan en capas y se conectan entre sí a través de conexiones ponderadas.

Una de las arquitecturas más simples y ampliamente utilizadas de las redes neuronales es el perceptrón multicapa. Esta arquitectura consta de una capa de entrada, una o varias capas ocultas y una capa de salida. Cada neurona en una capa oculta está conectada a todas las neuronas de la capa anterior y a todas las neuronas de la capa siguiente. Esta conectividad completa entre las capas proporciona flexibilidad y capacidad para aprender representaciones complejas de los datos de entrada. En esta arquitectura, perceptrón multicapa, la información fluye en una dirección, desde la capa de entrada a través de las capas ocultas hasta llegar a la capa de salida. Cada neurona en una capa oculta realiza una combinación lineal, cuyos coeficientes se conocen como pesos, de las salidas de las neuronas en la capa anterior y aplica una función de activación no lineal a esa combinación. Esto permite que la red neuronal capture y modele relaciones no lineales en los datos. Para que este aprendizaje sea posible, es necesario entrenar a la red neuronal. Durante este entrenamiento, los pesos de las conexiones entre las neuronas se ajustan de forma iterativa. Se utiliza un algoritmo de optimización, como la retropropagación del error (o backpropagation), para minimizar la diferencia entre los resultados obtenidos por la red y los resultados deseados. Este proceso de aprendizaje permite a la red ajustar sus conexiones y mejorar su capacidad para realizar predicciones o clasificar datos de manera precisa.

Es importante tener en cuenta que existen otras arquitecturas más complejas de redes neuronales, como las redes neuronales recurrentes (RNN) [7], las redes neuronales convolucionales (CNN) [8], las redes neuronales generativas adversarias (GAN) [9] y las redes Transformers. Estas arquitecturas están diseñadas para abordar desafíos específicos en el procesamiento de secuencias, el análisis de imágenes, la generación de datos y el procesamiento de lenguaje natural, respectivamente. Las redes neuronales recurrentes son adecuadas para el procesamiento de secuencias y pueden recordar información de iteraciones anteriores. Las redes neuronales convolucionales están optimizadas para el análisis de imágenes y se basan en operaciones de convolución para capturar

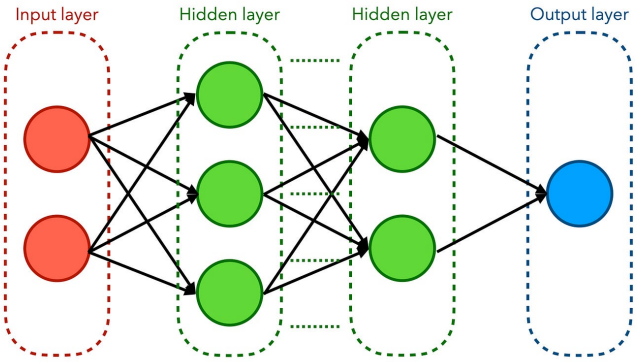


Figura 2.2: Red neuronal

características locales. Las redes neuronales generativas adversarias se utilizan para generar datos realistas y se componen de dos redes que compiten entre sí. Por otro lado, las redes Transformers son especialmente eficaces en el procesamiento de lenguaje natural y han revolucionado el campo. Utilizan mecanismos de atención para capturar las relaciones entre las palabras en una oración, lo que las hace altamente eficientes para tareas como la traducción automática, el resumen de texto y el procesamiento del lenguaje natural en general. Estas arquitecturas más complejas amplían las capacidades de las redes neuronales, permitiendo abordar problemas más sofisticados y obtener resultados de mayor calidad en tareas específicas.

Como se puede ver, las redes neuronales son capaces de aprender a partir de grandes cantidades de datos y pueden ser utilizadas para una amplia gama de tareas. Su capacidad para capturar relaciones complejas en los datos y su capacidad de generalización los convierten en un enfoque poderoso en el campo de la inteligencia artificial.

2.1.2. Machine Learning

El el machine learning [10] (aprendizaje automático) es una rama de la inteligencia artificial que se enfoca en el desarrollo de algoritmos y técnicas que permiten a las máquinas aprender de los datos y tomar decisiones o realizar predicciones sin ser explícitamente programadas. Se basa en la idea de que los sistemas informáticos pueden aprender patrones y relaciones en los datos y utilizar esta información para realizar tareas específicas. En este contexto, se trabaja con conjuntos de datos que contienen ejemplos de entrada (datos de entrada) y las salidas correspondientes (etiquetas o resultados deseados). El objetivo principal es entrenar un modelo de aprendizaje automático para que pueda generalizar a nuevos datos y producir salidas precisas y útiles.
El proceso de aprendizaje automático generalmente implica varias etapas. En primer lugar, se realiza una fase de preprocesamiento de los datos, donde se llevan a cabo tareas como la limpieza de datos, la selección de características relevantes y la normalización de los datos. A continuación, se selecciona un algoritmo de aprendizaje automático apropiado, que puede ser un algoritmo supervisado (donde el modelo se entrena utilizando pares de entrada-salida) o

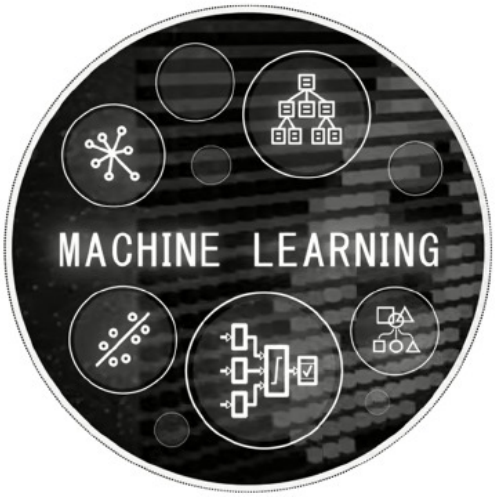


Figura 2.3: Mapa conceptual machine learning

no supervisado (donde el modelo busca patrones sin tener salidas etiquetadas). Algunos algoritmos comunes incluyen árboles de decisión [11], redes neuronales, máquinas de vectores de soporte (SVM) [12] y algoritmos de agrupamiento (clustering), entre otros.

Una vez seleccionado el algoritmo, se procede a la etapa de entrenamiento, donde el modelo ajusta sus parámetros utilizando los datos de entrada y salida proporcionados. Durante este proceso, el modelo busca ajustarse a los patrones y estructuras presentes en los datos de entrenamiento. Después del entrenamiento, se realiza una fase de evaluación utilizando datos adicionales llamados conjunto de prueba. Esto ayuda a medir el rendimiento y la precisión del modelo en datos no vistos previamente. Además, esto permite observar si el modelo tiene sobreajuste (overfitting), que es una situación en la que el modelo se adapta demasiado a los datos de entrenamiento y no generaliza bien a nuevos datos. Si el modelo no cumple con los criterios deseados, se pueden realizar ajustes adicionales, como la selección de diferentes hiperparámetros o la utilización de técnicas de regularización.

Por último, una vez que el modelo ha sido evaluado y satisface los requisitos deseados, se puede utilizar para hacer predicciones o tomar decisiones sobre nuevos datos. Esto se logra alimentando los datos de entrada al modelo y obteniendo las salidas predichas.

Figura 2.4: Etapas que se llevan a cabo en machine learning

2.1.3. Deep Learning

El deep learning [13] (aprendizaje profundo) es un enfoque dentro del campo de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos capaces de aprender y realizar tareas complejas a partir de grandes cantidades de datos. Este enfoque se inspira en la forma en que el cerebro humano procesa la información y extrae características para realizar tareas cognitivas. A diferencia de los enfoques tradicionales de aprendizaje automático (machine learning), que se basan en la extracción manual de características, el aprendizaje profundo busca aprender automáticamente las características relevantes directamente de los datos de entrada. Esto se logra mediante el uso de redes neuronales, y la profundidad en el aprendizaje profundo se refiere al uso de múltiples capas ocultas en una red neuronal. Estas capas ocultas permiten que la red aprenda representaciones jerárquicas de los datos de entrada, donde cada capa aprende características cada vez más abstractas y complejas. Este enfoque permite a las redes neuronales profundas aprender de manera más efectiva a medida que aumenta la cantidad de capas, lo que a su vez mejora su capacidad para resolver problemas complejos.

Una de las razones por las que el aprendizaje profundo ha ganado popularidad en los últimos años es su capacidad para abordar tareas desafiantes, como el reconocimiento de imágenes, el procesamiento del lenguaje natural y el reconocimiento de voz, con un rendimiento notable. Esto se debe a la capacidad de las redes neuronales profundas para capturar automáticamente patrones y características relevantes en los datos sin una extracción manual de características. Un buen ejemplo puede ser un problema en que se quiere identificar si una determinada imagen es un coche o no. Si se plantea el problema desde el punto de vista del machine learning y se utilizan enfoques tradicionales, sería necesario extraer manualmente características como la forma, el color o la textura de los coches para entrenar el modelo. Sin embargo, con el aprendizaje profundo, la red neuronal puede aprender automáticamente las características relevantes a partir de los datos de entrada, lo que permite una detección de coches más precisa y eficiente.

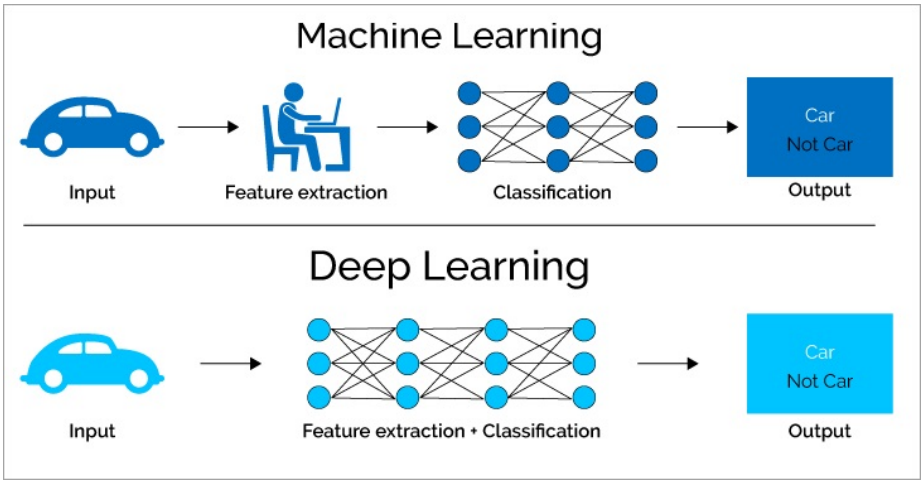


Figura 2.5: Comparación entre machine learning y deep learning

Además, el aprendizaje profundo se beneficia del uso de grandes conjuntos de datos y del aumento del poder de cómputo disponible en la actualidad. Estos avances han permitido entrenar redes neuronales profundas más grandes y complejas, lo que ha llevado a mejoras significativas en el rendimiento y la precisión en diversas aplicaciones.

2.2. Transformers

Los Transformers son un modelo de aprendizaje automático diseñado para tareas de procesamiento de lenguaje natural (NLP, por sus siglas en inglés) que revolucionó el campo desde su introducción en 2017 [14]. A diferencia de los enfoques tradicionales que se basaban en modelos recurrentes o convolucionales, los Transformers se destacan al utilizar exclusivamente mecanismos de atención para capturar las relaciones entre las palabras en un texto. Estos mecanismos de atención son la idea central detrás de los Transformers para el procesamiento de las secuencias de palabras. La atención permite que cada palabra en una secuencia interactúe con las demás, asignando una importancia diferente a cada una en función de su relevancia para la tarea en cuestión.

2.2.1. Arquitectura

Un Transformer se compone de dos componentes principales: el codificador (encoder) y el decodificador (decoder). El codificador se encarga de procesar la secuencia de entrada, mientras que el decodificador genera la secuencia de salida.

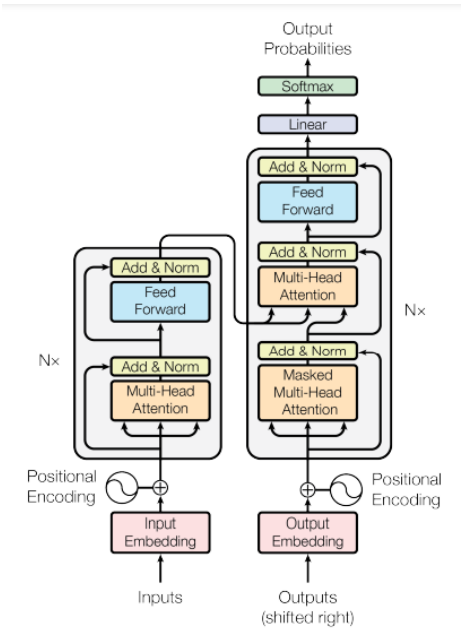


Figura 2.6: Arquitectura de un Transformer

Cada una de sus partes está formada por varios bloques apilados. Estos

bloques se componen de una capa de atención donde se calcula la atención para cada palabra en la secuencia, luego se aplica una técnica llamada residual connection"(conexión residual), que consiste básicamente en sumar la entrada de la capa de atención con su salida, y una normalización de capa conocida como "layer normalization"(normalización de capa) para estabilizar el entrenamiento del modelo y facilitar el flujo de información. Después de esto, se pasa la salida a otro pequeño bloque formado por una red neuronal completamente conectada, que procesa cada palabra de forma independiente.

Finalmente, tras el último bloque se añade una última capa completamente conectada y se aplica la función softmax, obteniendo así un vector de probabilidades sobre la siguiente palabra a ser generada. Además de las capas de atención y redes neuronales, los Transformers utilizan técnicas como el positional encoding (codificación posicional) para tener en cuenta el orden de las palabras en la secuencia y en la parte del decodificador se emplea "masked attention"que se basa en asegurar que cada palabra solo se base en palabras previas en la secuencia durante la generación, evitando que el modelo acceda a información futura en la secuencia.

2.2.2. Tipos de transformers

Se puede considerar que hay 3 tipos, transformers que solo tienen encoder, transformers que solo tienen decoder y transformers que tienen encoder y decoder:

- Solo encoder: Estos modelos se utilizan principalmente para tareas de clasificación de texto, reconocimiento de entidades y otras tareas similares. La parte codificadora de un Transformer se encarga de procesar una secuencia de palabras de entrada y convertirla en una representación de alta dimensionalidad, capturando la información semántica y contextual de cada palabra. Un ejemplo popular es BERT (Bidirectional Encoder Representations from Transformers) [15].

- Solo decoder: se utilizan principalmente en tareas de generación de texto, como la generación de respuestas a preguntas, subtítulos de imágenes o generación de texto a partir de un resumen, entre otras. Un ejemplo destacado es el modelo GPT (Generative Pre-trained Transformer)[16]. GPT se entrena para generar texto coherente y de alta calidad.

- Encoder y decoder: Estos modelos son muy utilizados en tareas de traducción automática, donde se requiere convertir un texto en un idioma de origen a un texto en un idioma de destino. La parte codificadora se encarga de procesar el texto en el idioma de origen y generar una representación vectorial, mientras que la parte decodificadora toma esa representación y genera una secuencia de palabras en el idioma de destino. Un ejemplo podría ser el modelo por Google en su sistema de traducción automática conocido como Google Translate.

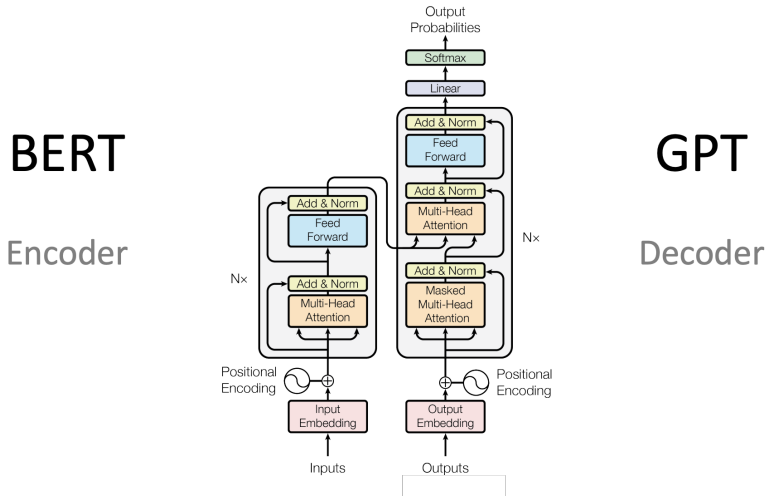


Figura 2.7: Comparativa entre BERT y GPT

2.2.3. ¿Por qué Transformers?

Como ya se ha mencionado anteriormente, los Transformers son muy útiles para trabajar con secuencias de datos. Estas secuencias de datos pueden ser de diferentes tipos. Por ejemplo, un caso muy común podría ser tomar oraciones como una secuencia de palabras y entrenar con ello al modelo. De un modo similar, se puede considerar una melodía como una secuencia de notas, logrando así una forma de entrenar un modelo con música. Al ser el objetivo de este trabajo generar música nueva, la arquitectura a usar será un Transformaer solo con decoder.

2.3. Procesamiento del lenguaje natural (NLP)

El Procesamiento del Lenguaje Natural (NLP) [17] es una rama de la inteligencia artificial que se centra en la interacción entre las computadoras y el lenguaje humano. Su objetivo principal es tratar de comprender, interpretar y generar texto o voz de manera similar a como lo hacen los seres humanos. El NLP abarca una amplia gama de tareas, desde la traducción automática hasta la generación de resúmenes y el análisis de sentimientos.

2.3.1. Tokenización

La tokenización [18] es un proceso esencial en NLP que consiste en dividir un texto en unidades más pequeñas llamadas ”tokens”. Estos tokens pueden ser palabras individuales, caracteres, subpalabras o incluso frases completas, dependiendo del nivel de granularidad deseado. La tokenización es un paso fundamental en el preprocesamiento de textos, ya que los modelos de no pueden trabajar directamente con texto, sino que necesitan recibir las entradas de manera numérica. En el campo de la música esto también puede ser aplicado, en lugar de un texto con palabras lo que se tiene es una melodía con notas

musicales. Al igual que en el NLP con texto, las melodías, compuesta por notas musicales, se dividen en tokens. Estos tokens representan elementos musicales individuales, como notas, acordes, ritmos o estructuras armónicas. OpenAI tiene una plataforma [19] en la que se puede probar como se tokenizaría una determinada frase. Un ejemplo ilustrativo podría ser el siguiente:

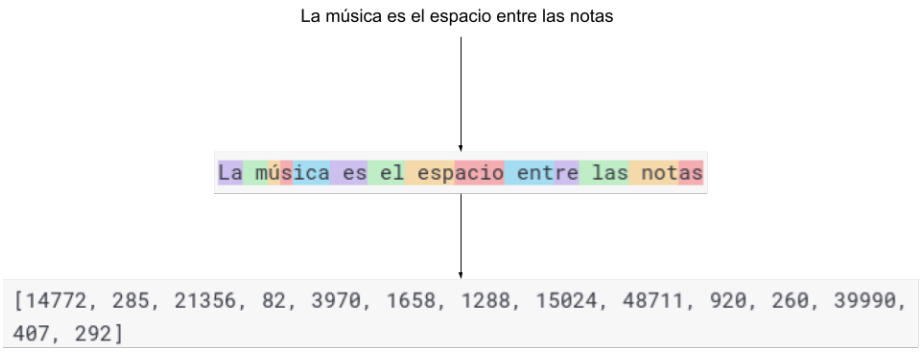


Figura 2.8: Proceso de tokenizacion

Ahora con las entradas tokenizadas se puede entrenar al modelo, sin embargo, la salida que generará serán tokens, por lo que habrá que convertir de vuelta estos tokens en palabras.

2.3.2. Formatos de ficheros de música

Dos de los formatos más ampliamente utilizados son el formato Waveform Audio File (.wav) y el formato Musical Instrument Digital Interface (.midi). Ambos formatos difieren en su estructura y características, así como en su capacidad para representar y transmitir información musical. El formato .wav se basa en una representación digital de la forma de onda de una señal de audio. Los archivos .wav contienen datos de audio sin comprimir, lo que significa que capturan cada muestra de sonido con precisión. Esto permite una reproducción fiel del sonido original y de gran calidad, pero también resulta en archivos más grandes en comparación con otros formatos de compresión de audio. Por otro lado, el formato .midi es una especificación estándar [20] para representar información musical en forma digital. A diferencia de los archivos .wav, los archivos .midi no contienen datos de audio real, sino instrucciones que describen cómo se debe interpretar la música. Estas instrucciones incluyen información sobre las notas, duraciones, intensidades y otros parámetros musicales, así como comandos para controlar dispositivos de reproducción musical, como sintetizadores o módulos de sonido. Los archivos .midi son relativamente pequeños debido a que solo contienen datos de control y no información de sonido real. Esto los hace ideales para el entrenamiento de un modelo generador de música, ya que debido a su formato estructurado se puede realizar fácilmente la conversión de una melodía a tokens.

2.4. Transfer learning

El transfer learning es una idea que se basa en aprovechar los conocimientos previos de un modelo preentrenado y adaptarlos a la nueva tarea en lugar de entrenar un modelo desde cero para dicha tarea. Esto permite entrenar un nuevo modelo en menos tiempo, usando menos recursos computacionales y menos datos de entrenamiento. Este modelo preentrenado se refiere a un modelo que ha sido entrenado previamente en una tarea utilizando un conjunto de datos grande y diverso. Dicho modelo ha adquirido conocimientos generales sobre la estructura y características de los datos, lo que lo convierte en una valiosa fuente de información para una tarea relacionada. La idea clave detrás del transfer learning es que este conocimiento previo puede ser transferido o adaptado al nuevo contexto de la tarea objetivo.

Esta técnica ha demostrado ser efectiva en una amplia gama de aplicaciones, como reconocimiento de imágenes, procesamiento de lenguaje natural, detección de objetos y muchas otras áreas donde el aprendizaje automático es aplicable.

2.4.1. Fine-tuning

El fine-tuning [21] se utiliza con frecuencia en el contexto del transfer learning, donde un modelo preentrenado se ajusta a una nueva tarea. Para ello, se selecciona un modelo preentrenado que haya sido entrenado en una tarea relacionada o en un conjunto de datos amplio y diverso. A continuación, se utiliza un conjunto de datos específico de la tarea que se desea lograr para ajustar el modelo preentrenado. Este conjunto de datos se compone de ejemplos etiquetados relacionados con dicha tarea en la que se busca mejorar el rendimiento. Finalmente, los pesos y parámetros del modelo se actualizan reentrenando este modelo, estableciendo una tasa de aprendizaje menor e incluso dejando algunos pesos como estaban para evitar que se modifiquen en exceso y se pierdan los conocimientos previos adquiridos durante el entrenamiento inicial.

2.5. Herramientas y librerías

Para la elaboración de este trabajo has sido empleadas las siguientes herramientas:

- Google Colab [22]: Google Colab es un producto de Google Research. Permite a cualquier usuario escribir y ejecutar código arbitrario de Python en el navegador. Es especialmente adecuado para tareas de aprendizaje automático, análisis de datos y educación. Además, Google Colab ofrece entornos gratuitos con GPU, pero con limitaciones de tiempo.

- pandas [23]: Pandas es una librería de Python especializada en la manipulación y el análisis de datos. Ofrece estructuras de datos y operaciones para manipular tablas numéricas y series temporales, es como el Excel de Python. Esto resulta muy útil a la hora de procesar el conjunto de datos.

- numpy [24]: NumPy es una librería para el lenguaje de programación Python que da soporte para crear vectores y matrices grandes

multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas eficientemente.

- pytorch [25]: PyTorch es un paquete de Python que proporciona funciones de alto nivel como el uso de tensores y de herramientas para operar con ellos con fuerte aceleración en GPU y la elaboración de redes neuronales profundas construidas sobre un sistema autograduado basado en cintas. Es con esta herramienta con la que está implementado el modelo usado en este trabajo.

- fastai[26]: Fastai es una librería de deep learning que proporciona a componentes de alto nivel que pueden ofrecer rápida y fácilmente resultados de vanguardia en dominios del deep learning, y proporciona a los investigadores componentes de bajo nivel que se pueden mezclar y combinar para construir nuevos enfoques. Esta librería ha sido usada para realizar el fine-tuning del modelo.

- matplotlib [27]: Matplotlib es una librería para la generación de gráficos en dos dimensiones, a partir de datos contenidos en listas o arrays en el lenguaje de programación Python. Ha sido de utilidad a la hora de crear los gráficos del error durante el entrenamiento del modelo.

- HoloViews [28]: HoloViews es una biblioteca Python de código abierto diseñada para que el análisis y la visualización de datos sean sencillos y fluidos. Se ha utilizado esta librería para crear diferentes gráficos.

- hvplot [29]: HvPlot es una librería construida sobre HoloViews que proporciona una API general y consistente para crear gráficos. Del mismo modo que matplotlib, ha sido útil para crear gráficos sobre los datos de entrenamiento.

- tqdm [30]: tqdm es una librería con la que se pueden crear barras de progreso. Estas barras de progreso se muestran durante el entrenamiento del modelo y además ofrece una estimación del tiempo de ejecución.

- FluidSynth [31]: FluidSynth, es un Sintetizador de software de código abierto que convierte notas midi en una señal de audio usando tecnología SoundFont sin la necesidad de una tarjeta de sonido compatible con SoundFont. Con esto, las composiciones generados por el modelo se pueden escuchar en un formato que suena más natural y agradable para el oído.

2.6. Trabajos similares

Hay otros trabajos realizados que tienen un objetivo similar a este, que es el de componer música. Entre ellos, se puede mencionar:

- Museformer [32]: Museformer es un modelo desarrollado por Muzic, un proyecto de investigación de Microsoft. Se basa en un Transformer que utiliza una técnica de atención novedosa a la que llaman "fine and coarse-grained attention"para componer música. Específicamente, con la fine-grained attention, un token de un compás específico atiende directamente a todos los tokens de los compases que son más relevantes

para las estructuras musicales (por ejemplo, los compases anteriores 1º, 2º, 4º y 8º, seleccionados mediante estadísticas de similitud); con la coarse-grained attention, un token sólo atiende al resumen de los otros compases en lugar de a cada token de ellos para reducir el coste computacional.

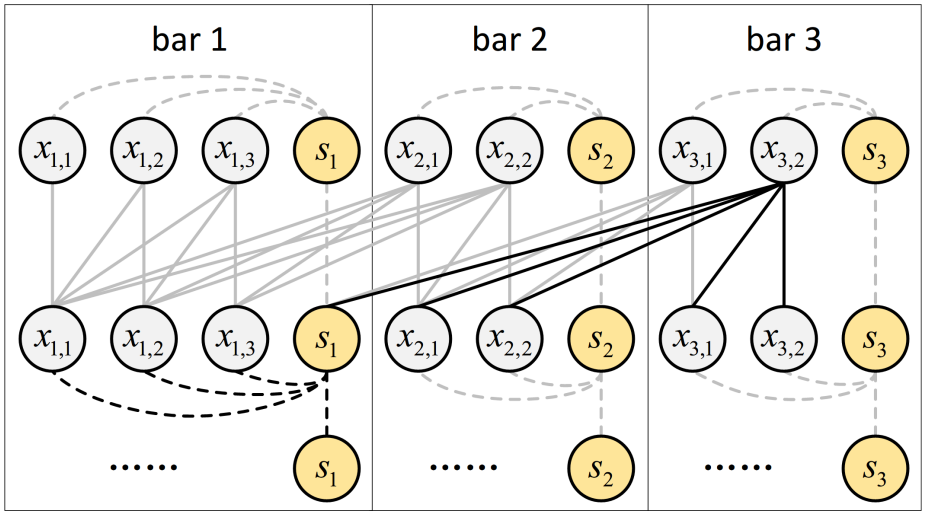


Figura 2.9: Flujo de información de FC-Attention

- Music Transformer [33]: Music Transformer ha sido desarrollado por Magenta, un grupo creado por Google. Se trata de una red neuronal basada en la atención que puede generar música con una coherencia a largo plazo mejorada. Se emplea atención relativa como mecanismo de atención, que modula explícitamente la atención en función de la distancia entre dos tokens, el modelo es capaz de centrarse más en las características relacionales. La autoatención relativa (relative self-attention) también permite al modelo generalizar más allá de la longitud de los ejemplos de entrenamiento.

- Dance-Diffusion [34]: Dance-Diffusion es un producto desarrollado por Harmonai. Dance-Diffusion es un modelo de difusión entrenado para componer música. Este es el más diferente al resto, ya que se basa en un modelo de difusión. Además, cabe destacar que la salida que produce es en formato .wav y no en .midi como lo hace el resto de modelos de esta lista.

- Theme Transformer [35]: se propone un enfoque de condicionamiento alternativo, denominado condicionamiento basado en temas, que entrena explícitamente al Transformer para que trate la secuencia de condicionamiento como un material temático que tiene que manifestarse múltiples veces en su resultado de generación. Esto se consigue con dos contribuciones técnicas principales. En primer lugar, se propone un enfoque basado en el aprendizaje profundo que utiliza el aprendizaje de representación contrastiva y la agrupación para recuperar automáticamente materiales temáticos de piezas musicales en los datos de

entrenamiento. En segundo lugar, se propone un nuevo módulo de atención paralela que se utilizará en una arquitectura de codificador/decodificador secuencia a secuencia (seq2seq) para tener en cuenta de forma más eficaz un determinado material temático condicionante en el proceso de generación del decodificador Transformer.

- MT-GPT-2 [36]: se propone el modelo MT-GPT-2(music textual GPT-2) que se utiliza en la generación de melodías musicales basado en el modelo GPT-2 y el aprendizaje por transferencia (transfer leraning). Además, se propone un método de evaluación musical simbólica mediante la combinación de estadística matemática, conocimientos de teoría musical y métodos de procesamiento de señales, que es más objetivo que el método de evaluación manual.

Lo novedoso que se propone en este trabajo que no realizan los anteriores mencionados, es el reentrenamiendo de un modelo base con diferentes conjuntos de datos de estilos de música clásica, con el objetivo de que las composiciones generadas parezca que pertenecen a dichos estilos musicales.

Capítulo 3

Desarrollo del proyecto

3.1. Descripción del proyecto

La idea principal de este trabajo es lograr componer música a partir de una breve melodía. Esta composición generada de manera automática contendrá características y patrones propios de tres estilos de música clásica diferentes: barroco, romanticismo y vanguardias. A partir de esto, lo que se propone hacer es buscar un modelo preenrtrenado que sirva como base y reentrenarlo con los diferentes conjuntos de datos, creando varias versiones, cada una de ellas capaz de generar composiciones propias del estilo sobre el que han sido reentrenadas. Para crear estos tres conjuntos de datos, lo que se ha hecho ha sido buscar un conjunto de datos de ficheros .midi lo suficientemente grande como para crear a partir de él otros conjuntos más pequeños mediante una selección de una serie de piezas de tres autores: Johann Sebastian Bach, Frédéric Chopin y Sergei Rachmaninoff.

Una vez creados los conjuntos de datos, hay que realizar un preprocesamiento de estos para poder alimentar al modelo con estos datos. El proceso que se ha llevado a cabo ha consistido en aplicar un algoritmo de tokenización. Básicamente, este algoritmo se encarga de convertir los ficheros midi en representaciones vectoriales. Además, debe ser posible hacer la conversión a la inversa, es decir, a partir de una representación vectorial, crear un fichero midi. De esta forma, será posible convertir las generaciones del modelo en nuevas composiciones en formato midi.

El modelo que ha sido elegido tiene una arquitectura bastante similar a GPT-2 y puede ser tratado como un modelo de lenguaje. Para realizar el reentrenamiento, se ha hecho uso de la librería fastai, la cual es adecuada para este tipo de problemas. También se realizaron experimentos para reentrenar el modelo usando simplemente la librería pytorch, pero los resultados fueron descartados porque no eran lo suficientemente buenos.

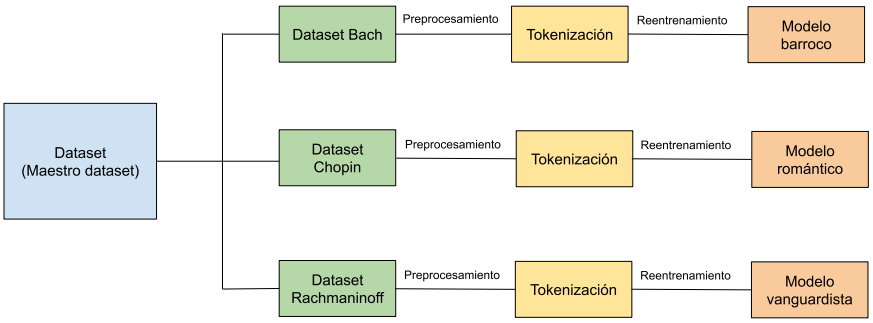


Figura 3.1: Partes del desarrollo del trabajo

3.2. Elección del modelo

Para comenzar con la tarea de generación de música primero ha sido necesario encontrar un modelo base ya preentrenado. Esto debe de ser así ya que el tipo de modelo que se estaba buscando era un Transformer, y entrenar un Transformer desde cero con la intención de producir buenos resultados no era viable por la exigencia de recesos computacionales. Tras una larga investigación y un largo periodo probando diferente modelos que acabaron siendo descartados por diferentes razones, como pueden ser los ya mencionados en una sección anterior Museformer de Microsoft, Music Transformer del grupo Magenta de Google, Dance-Diffusion de Harmonai, y Theme-Transformer; se acabó eligiendo como modelo inicial Giga-Piano [37], un modelo que ha sido preentrenado para generar obras de piano. Este proyecto proporcionaba tanto el código de entrenamiento, como los pesos del modelo ya entrenado, como un notebook en Google Colab para la inferencia del modelo. Este notebook ofrece varias opciones, como la generación de una continuación de una secuencia de notas proporcionada como entrada, generación de nueva música sin ninguna secuencia condicionante y generación de progresiones de acordes y de notas. Además, se ha hecho un fork (`https://github.com/DavidRamosArchilla/GIGA-Piano`) a este proyecto donde está todo el código y demás artefactos empleados para el desarrollo de este trabajo.

En cuanto a la arquitectura del modelo, es una versión de GPT y el la porción de código A.0.1 muestra como ha sido construido. Sus principales elementos son:

- Una capa de embeddings que asigna los tokens de entrada a representaciones vectoriales. De forma que si recibe como entrada un tensor de dimensiones (B,T) devuelve un tensor con dimensiones (B,T,E), siendo E la dimensión de embedding.
- Una capa con los embeddings posicionales (positional embeddings) que proporciona información sobre las posiciones relativas de los tokens en la secuencia de entrada. Estos embeddings se sumarán con los mencionados anteriormente y será lo primer en realizarse.
- Una capa de dropout conectada a la salida de los embeddings.
- Una secuencia de bloques transformer (explicados en detalle a continuación).
- Una capa de normalización que se aplica a la salida del último bloque.
- Una capa lineal que proyecta los embeddings al tamaño del vocabulario y que es la encargada de realizar la predicción del siguiente token, a la cual se le aplica una función de activación softmax para obtener una distribución de probabilidad sobre el vocabulario.

Los bloques están formados de la siguiente manera:

- Dos capas de normalización, una que se aplica a los embeddings de entrada y la otra a la salida del mecanismo de atención.

- Una capa de atención (MultiHead Attention). Esta capa utiliza un mecanismo de atención llamado Self-Attention with Relative Position Representations [38], lo cual proporciona mejores resultados para secuencias largas que el mecanismo de Self-Attention.
- Una red neuronal (feed-forward) que consta de dos capas lineales con una función de activación GELU y dropout.

Por último, los hiperparámetros con los que el modelo ha sido construido son los siguientes:

- Tamaño de vocabulario: 128. Esto hace referencia al número total de tokens diferentes que hay.
- Tamaño de bloque: 1024. Representa la longitud máxima de la secuencia de entrada.
- Dimensión de feed forward: 1024. Es el número de neuronas de la capa que forma el bloque feed forward.
- Número de bloques: 16. Es el número de bloques que tiene el Transformer.
- Número de cabezas: 16. Es el número de cabezas que tiene cada bloque de atención (MultiHeadAttention).
- Dimensión embedding: 1024. Es el tamaño de la dimensión de la capa de token embedding.
- Probabilidad dropout: 0.1. Es el parámetro de la capa de dropout. Este parámetro controla la probabilidad de que se aplique el dropout.

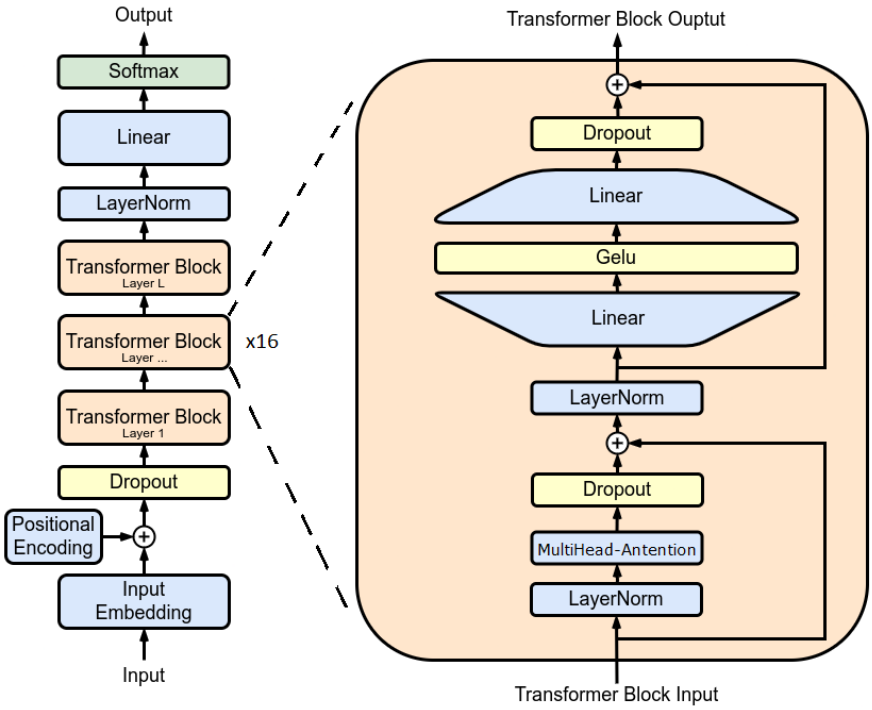


Figura 3.2: Arquitectura del modelo

3.3. Dataset

Con el objetivo de reentrenar el modelo para diferentes estilos, es necesario primero obtener un dataset inicial. Además, este dataset debe contener una gran cantidad de ficheros en formato midi y que sean de piano, ya que el modelo inicial solamente fue entrenado con obras de piano. Ante esta necesidad, hay varias alternativas como pueden ser the million song dataset [39] o el FMA dataset [40], pero finalmente el que se eligió fue el MAESTRO dataset [41]. MAESTRO (MIDI and Audio Edited for Synchronous TRacks and Organization) es un dataset formado por más de 200 horas de grabaciones de audio y MIDI de

interpretaciones de piezas de piano. Estas grabaciones fueron realizadas durante varias ediciones del Ïnternational Piano-e-Competitionçon la ayuda de sus organizadores. Los datos MIDI, que son los de interés, incluyen velocidades de pulsación de teclas y posiciones de los tres pedales (sustain, sostenuto y una corda). Los archivos están alineados con una precisión de aproximadamente 3 ms y se dividen en piezas musicales individuales, que están anotadas con el compositor, el título y el año de la interpretación. El audio sin comprimir es de calidad de CD o superior (44.1-48 kHz 16-bit PCM estéreo).

Uno de los aspectos positivos del MAESTRO dataset, es que incluye un fichero csv con metadatos del dataset. Estos metadatos vienen recogidos en la siguiente tabla:

Campo	**Descripción**
canonical_composer	Compositor de la pieza. Se ha intentado estandarizar una única ortografía para cada nombre.
canonical_title	Título de la pieza. No se garantiza una representación única.
split	División sugerida entre entrenamiento, validación y prueba.
year	Año de la actuación.
midi_filename	Nombre de archivo MIDI.
audio_filename	Nombre de archivo WAV.
duration	Duración en segundos, basada en el archivo MIDI.

Tabla 3.1: Descripción de campos

3.3.1. Elección de los compositores

Ahora, el objetivo es extraer 3 subconjuntos de datos de este dataset, cada uno de un estilo diferente, para reentrenar con cada uno de ellos el modelo. Con la ayuda de estos metadatos, es sencillo obtener dichos subconjuntos ya que en los metadatos aparece el nombre del compositor de cada una de las piezas.

Se ha hecho un breve análisis para conocer cuántas obras había de cada compositor y para tener una idea de sus duraciones, ya que las obras más largas se van a convertir en una secuencia de tokens más largas a la hora de la tokenización. Para ello se han obtenido las siguientes gráficas, que muestran información sobre los 10 compositores con mas piezas en el dataset:

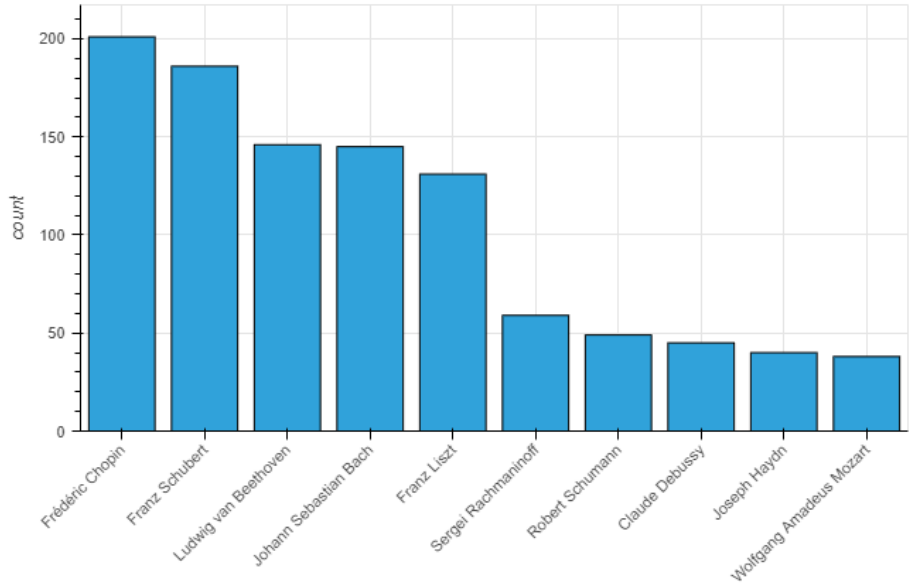


Figura 3.3: Cantidad de piezas de cada compositor

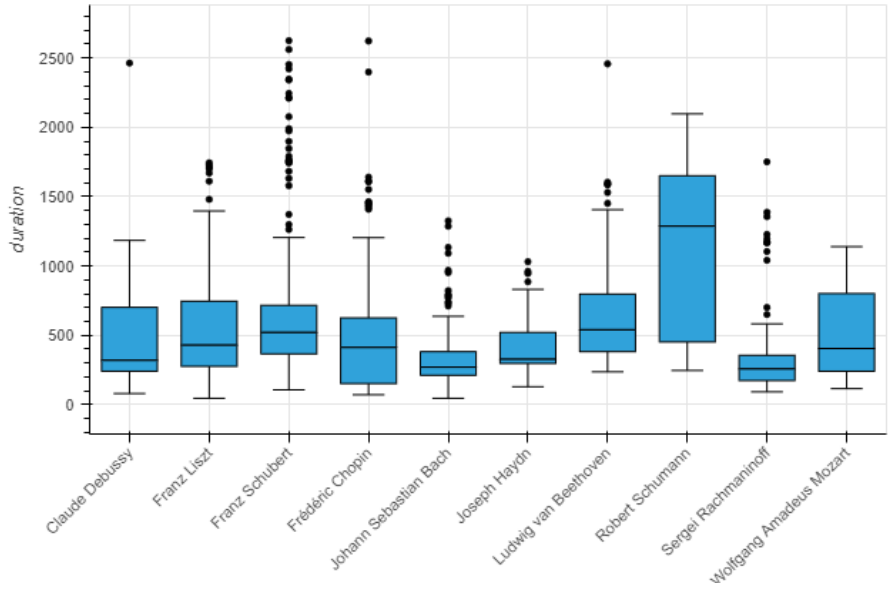


Figura 3.4: Diagrama de cajas y bigotes sobre la duración de las piezas

Con esta información, se eligen 3 compositores, cada uno de ellos representado un estilo diferente. Los compositores elegidos han sido Johann Sebastian Bach, Frédéric Chopin y Sergei Rachmaninoff y sus estilos son barroco, romanticismo y vanguardias respectivamente. De cada compositor se han elegido 50 piezas aleatoriamente para reentrenar al modelo. Para ello se ha usado el script A.0.2:

3.3.2. Cargar los datasets para ser usados

Una vez se tienen los datasets, hay que convertir los ficheros midi en tokens para poder entrenar al modelo. Para ello, se emplea un algoritmo para tokenizar ficheros midi que está ya implementado en el proyecto Giga-Piano. Este algoritmo convierte a los midis en secuencias de enteros en un rango de 0 a 127, por lo que el tamaño de vocabulario es 128. Una vez se tokenizan los ficheros midi, hay que construir el dataset de forma que se pueda entrenar al modelo. Para ello, el modelo debe recibir como entrada una secuencia de tokens, siendo su longitud máxima el tamaño de bloque, y su salida será una predicción de cuál es el siguiente token en la secuencia. Además, durante el entrenamiento también se debe indicar cuál es el siguiente token de la secuencia de entrada, para que el modelo pueda calcular el error en su predicción. Con la el código de A.0.3 se puede lograr esto.

En este código, la variable train_data1 ha sido calculada en otra celda del notebook y es una lista que contiene todos los tokens de todos los ficheros midis elegidos para el dataset. Además, se divide el dataset en 2 partes, entrenamiento y validación. El conjunto de entrenamiento serán los datos con los que se entrenará al modelo, mientras que el conjunto de validación son datos con los que no se entrenará al modelo y se utilizará para evaluarlo. De este modo, se puede determinar si el modelo está entrenando correctamente o si, por el contrario, tiene overfitting. Por último, en esta parte es donde se definen los valores del tamaño de batch y de la longitud de secuencia, que son 16 y 128 respectivamente.

3.4. Experimentos realizados

3.4.1. Entrenamiento utilizando solamente pytorch

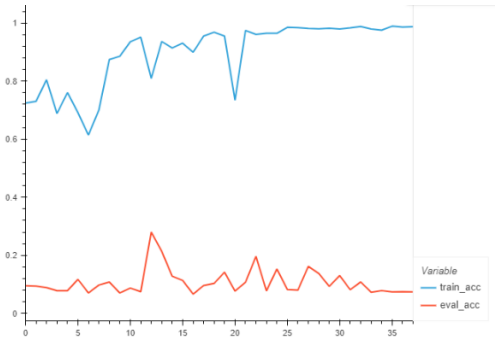
Se han llevado a cabo varios experimentos para tratar de reentrenar el modelo. Inicialmente, se probó a congelar todo el modelo salvo la última capa. Esto se refiere a que durante el entrenamiento lo únicos pesos que se actualizan son los de la última capa. Esto se puede plantear de 2 formas, o bien se reinician los pesos por completo de la última capa, o bien se dejan los pesos que ya había aprendido el modelo en su preentrenamiento. Finalmente se optó por la opción de dejar los pesos que ya había aprendidos, ya que el modelo podría tardar demasiado en aprender unos nuevos si estos reiniciaban. Además de esto, también se hicieron pruebas sin congelar ningún peso, es decir, reentrenando todo el modelo.

Para estos experimentos el código que se utilizó fue A.0.4.

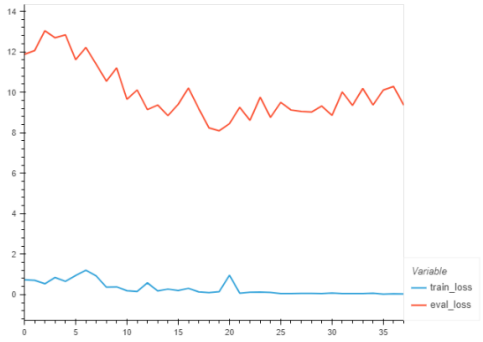
Aquí, la función train estaba ya implementada en el proyecto Giga-Piano, pero se le han hecho ligeras modificaciones. En particular, se ha añadido la función estimate_loss para poder evaluar el modelo durante el entrenamiento, ya que esto en el proyecto inicial no se estaba haciendo.

A partir de este código, se han realizado varios entrenamientos con diferentes configuraciones, de tal forma que se han probado diferentes valores para la tasa de aprendizaje y se ha probado tanto como entrenar a todo el modelo como solo la última capa. Para el entrenamiento se ha usado uno de los 3 conjuntos de datos para ver como se comportaba el modelo. El conjunto de datos con el que están realizados estos experimentos es el que tiene las obras de Chopin.

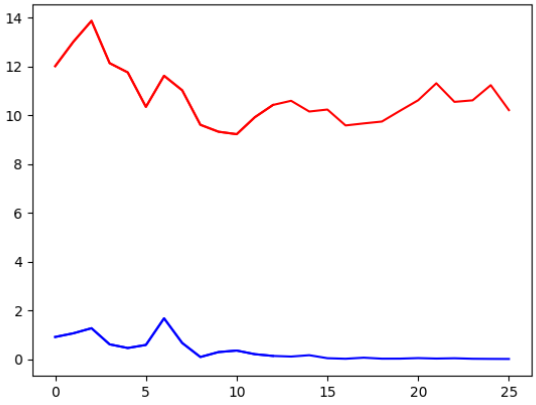
Por último, el tamaño del batch se ha fijado en 16 y se ha entrenado para un solo epoch y 16000 iteraciones, que en tiempo ha resultado en aproximadamente 2 horas cada entrenamiento. Las gráficas sobre los datos de entrenamiento de dichos experimentos son la siguientes:



Precisión con tasa de aprendizaje: 0.001



Error con tasa de aprendizaje: 0.001

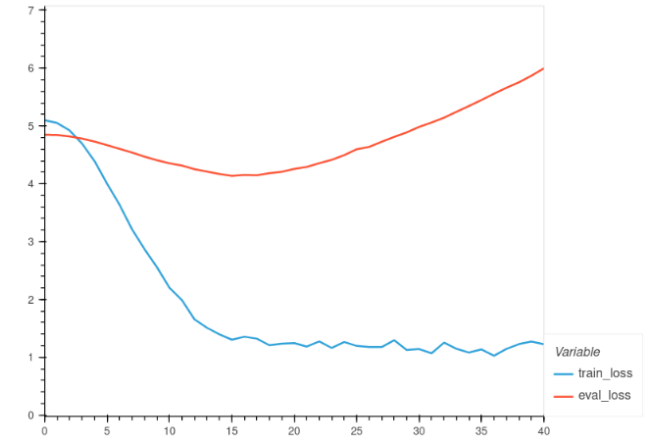


Error con tasa de aprendizaje: 1.0

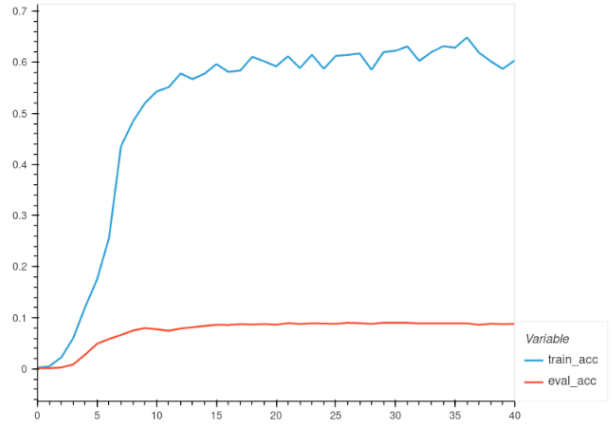
Figura 3.5: Entrenamiento del modelo por completo.

Estas gráficas corresponden a los datos de entrenamiento cuando se estaban reentrenando todos los pesos del modelo, primero cuando se utilizó una tasa de entrenamiento de 0.001 y luego cuando se empleó 1.0. A partir de estas gráficas se puede obtener información acerca de los entrenamientos. Como se puede apreciar en las tres gráficas, el error en validación es demasiado alto y no decrece. Además, la precisión en entrenamiento es muy elevada y el error muy bajo, lo que indica que en estos casos se está produciendo overfitting. Todo esto indica que los resultados de estos entrenamiento no han sido buenos.

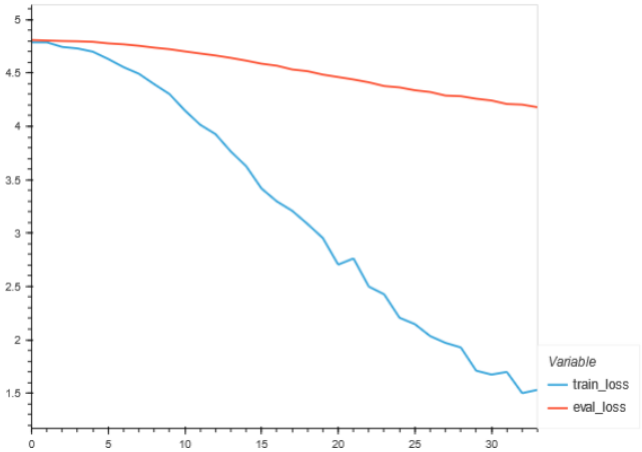
A continuación, se muestran gráficas para casos en los que solo se han reentrenado los pesos de la última capa del modelo.



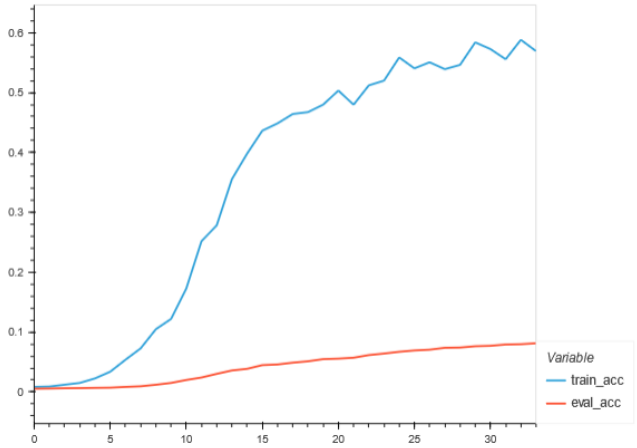
Error tasa de aprendizaje: 1.0



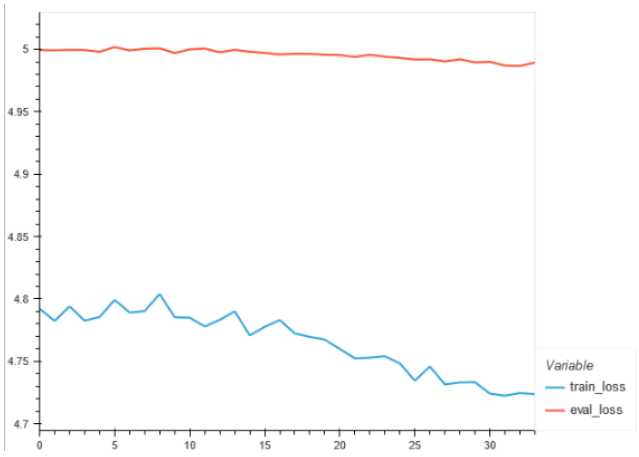
Precisión tasa de aprendizaje: 1.0



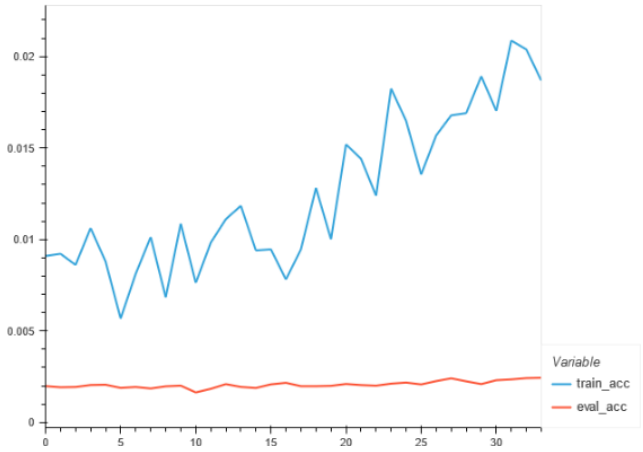
Error tasa de aprendizaje: 0.1



Precisión tasa de aprendizaje: 0.1



Error tasa de aprendizaje: 0.001



Precisión tasa de aprendizaje: 0.001

Figura 3.6: Entrenamiento del solo la última capa del modelo.

Para estos casos, cuando la tasa de aprendizaje era 1.0, hay un punto en el que el error en validación comienza a ser creciente y la precisión en validación deja de crecer, por lo que en esta situación se produce overfitting. En el caso en el que la tasa de aprendizaje es 0.1, las curvas del error divergen bastante, cosa que indica que también se está produciendo overfitting. Por último, en la situación en la que la tasa de aprendizaje es 0.001, el error tanto como para el conjunto de entrenamiento como para el de validación decrece muy lentamente, y además, la precisión en validación casi no varía, por tanto, este entrenamiento tampoco es válido.

Después de observar estos resultados, se ha optado por descartarlos y emplear otro enfoque diferente. Esto está explicado en detalle en la siguiente sección.

3.4.2. Entrenamiento utilizando fastai

Tras estos experimentos, se ha optado por utilizar otra aproximación, que ha sido utilizar la librería de fastai. Siguiendo los pasos indicados en el post "Faster than training from scratch - Fine-tuning the English GPT-2 in any language with Hugging Face and fastai v2 (practical case with Portuguese)"[42], se procede a aplicar una técnica similar para reentrenar al modelo. En este post, se indican unos pasos para reenetrenar GPT-2 [43] para generar texto en portugués. Al ser el modelo de este trabajo muy similar a GPT-2, es posible realizar el mismo tipo de entrenamiento que el que se muestra, siendo prácticamente la única diferencia la forma de tokenizar los datos, debido a la naturaleza de los mismos para cada situación. Para empezar, hay que crear otro dataloader porque con el que había creado anteriormente no funciona. Para ello se ha usado el código A.0.5.

En este caso, ventana hace referencia a la longitud de la secuencia de entrada y el tamaño de batch que se está usando es 32. Del mismo modo que en los experimentos anteriores se están creando un dataloader para los datos de entrenamiento y otro para los de validación. Ambos dataloaders se agrupan bajo un objeto de la clase Dataloaders de fastai, que los agrupa y posteriormente durante el entrenamiento los gestiona de manera automática.

Ahora, hay que crear varios grupos con las capas del modelo. La idea detrás de esto es congelar todo el modelo e ir entrenando el modelo descongelando cada vez un grupo más. No obstante, solamente se ha entrenado el último grupo porque de esta forma ya se producían buenos resultados y el hecho de entrenar añadiendo más grupos no ha resultado en una mejora del modelo. La forma de crear los grupos es con la función splitter A.0.6, que será un parámetro posteriormente a la hora de entrenar.

Por último, solo falta crear un objeto de la clase Learner de fastai, que utilizará todo lo creado hasta ahora. El código empleado es A.0.7.

Como el modelo devuelve una tupla de valores (unas predicciones y el error), hay que crear un callback para que el learner pueda usar las predicciones realizadas por el modelo. Esta callback es la clase DropOutput, que define un evento after_pred que se ejecuta después de que el modelo realiza una predicción. Lo que se consigue hacer con esto es reemplazar self.learn.pred (que contiene las predicciones con las que se calculará el error) por simplemente su primer elemento. Además, en esta parte del código se definen tanto la función de error a usar como las métricas adicionales que se desean calcular. En este caso, la función de error que se utiliza es la entropía cruzada y las métricas adicionales son la precisión y la perplejidad.

Teniendo esto preparado, es momento de entrenar al modelo con cada uno de los datasets. El código que se emplea para ello es A.0.8.

En esta porción de código, el método learner.lr_find() devuelve un valor para la tasa de aprendizaje sugerido. Además, se genera una gráfica de como se ha hallado este valor. Para el entrenamiento, se utiliza un valor ligeramente inferior al sugerido, concretamente, la mitad.

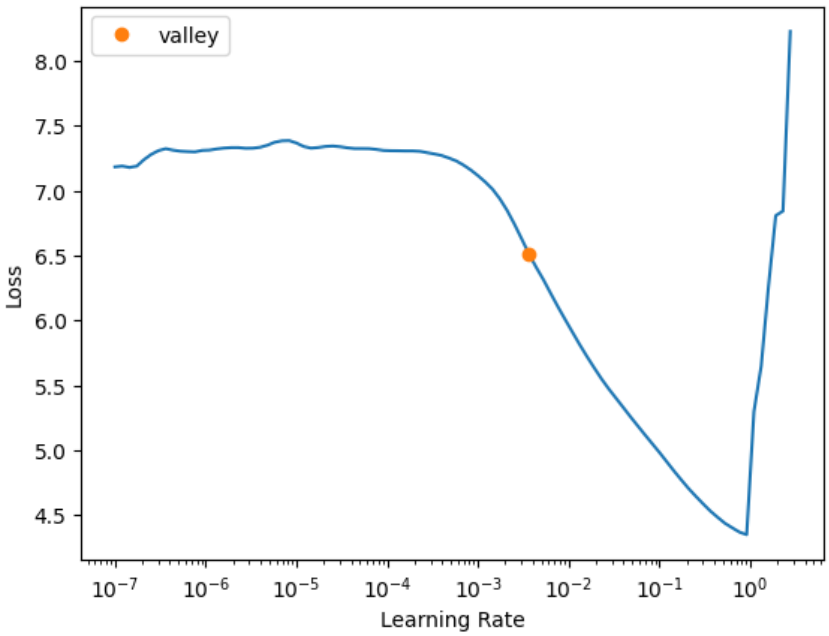


Figura 3.7: Ejemplo de gráfica generada por learner.lr_find()

Estos son los datos de entrenamiento obtenidos para cada uno de los datasets:

epoch	train_loss	valid_loss	accuracy	perplexity	time
0	1.594135	1.612138	0.521292	5.013518	1:54:02

Tabla 3.2: Resultados del entrenamiento con el dataset de estilo vanguardista

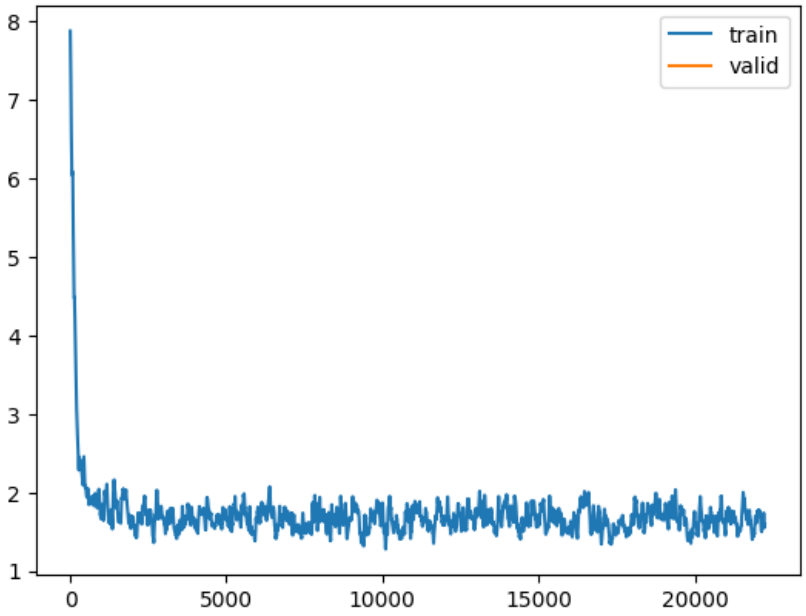


Figura 3.8: Gráfica del error durante el entrenamiento con el dataset de estilo vanguardista

epoch	train_loss	valid_loss	accuracy	perplexity	time
0	1.435151	1.489983	0.532048	4.437022	1:01:16

Tabla 3.3: Resultados del entrenamiento con el dataset de estilo barroco

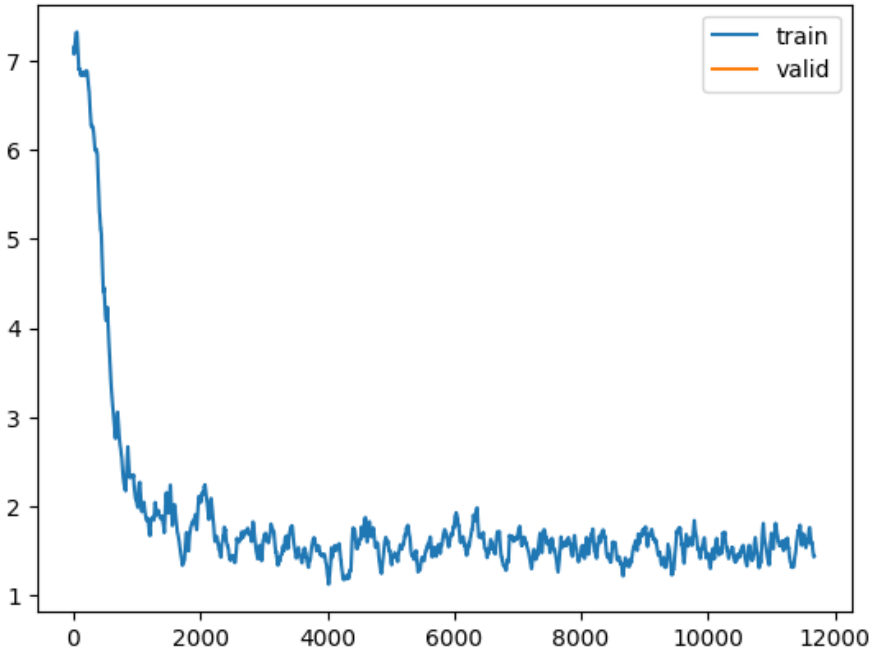


Figura 3.9: Gráfica del error durante el entrenamiento con el dataset de estilo barroco

epoch	train_loss	valid_loss	accuracy	perplexity	time
0	1.586708	1.410434	0.562164	4.097733	2:13:29

Tabla 3.4: Resultados del entrenamiento con el dataset de estilo romántico

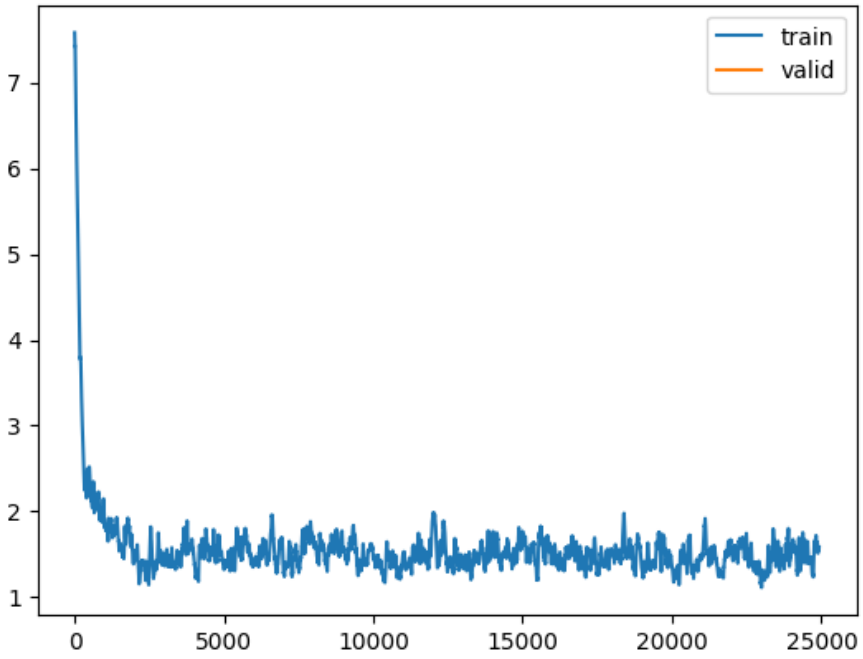


Figura 3.10: Gráfica del error durante el entrenamiento con el dataset de estilo romántico

Como se puede apreciar en las tablas 3.2, 3.3 y 3.4, los valores del error para entrenamiento y validación (train loss y valid loss respectivamente) son bastante cercanos, siendo en algunos casos inferior el error en validación. Además, los valores de la precisión (accuracy) son adecuados, ya que todos ellos superan una precisión del 50Por estos motivos, se puede concluir que ya no aparecen los mismos problemas mencionados en los otros experimentos, por lo que estos modelos se consideran válidos.

Capítulo 4

Resultados

4.1. Resultados obtenidos

Una vez se tienen entrenados los diferentes modelos, es posible empezar a componer con ellos. Esta sección muestra brevemente algunas de estas composiciones, que pueden ser descargadas para escucharlas desde (`https://github.com/DavidRamosArchilla/GIGA-Piano/tree/main/Resultados_ejemplo`). A continuación, se muestran composiciones de cada uno de los modelos cuando se les proporcionó como entrada el inicio de la conocida obra de Beethoven, Para Elisa:

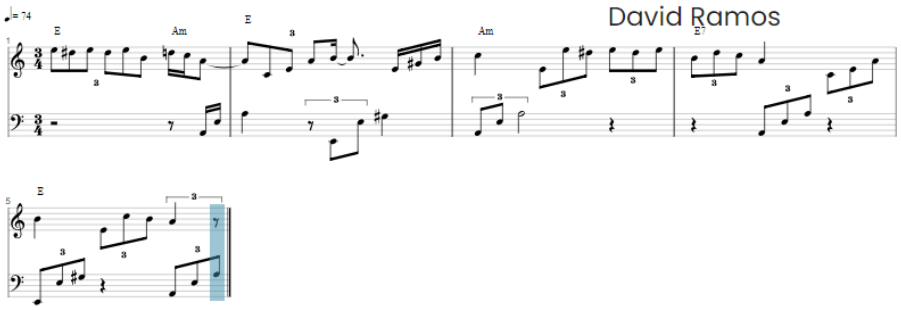


Figura 4.1: Parte inicial de "Para Elisa"

La forma de funcionamiento es la siguiente: a la secuencia proporcionada como entrada se le aplica una conversión a tokens. Una vez hecho esto, se utilizan estos tokens como entrada al modelo y este realizará una predicción del siguiente token para la secuencia de entrada. Luego, ese token predicho, se añade a la secuencia inicial y esta nueva secuencia de token se emplea de nuevo como entrada para el modelo. Ese proceso se repite hasta generar una secuencia de tokens con la longitud deseada. Por último, se transforma la secuencia de tokens final en un fichero midi. El siguiente diagrama muestra cómo es este proceso, con la ligera diferencia de que es para generar texto:

Figura 4.2: Proceso de generación de tokens

4.1.1. Modelo de estilo barroco

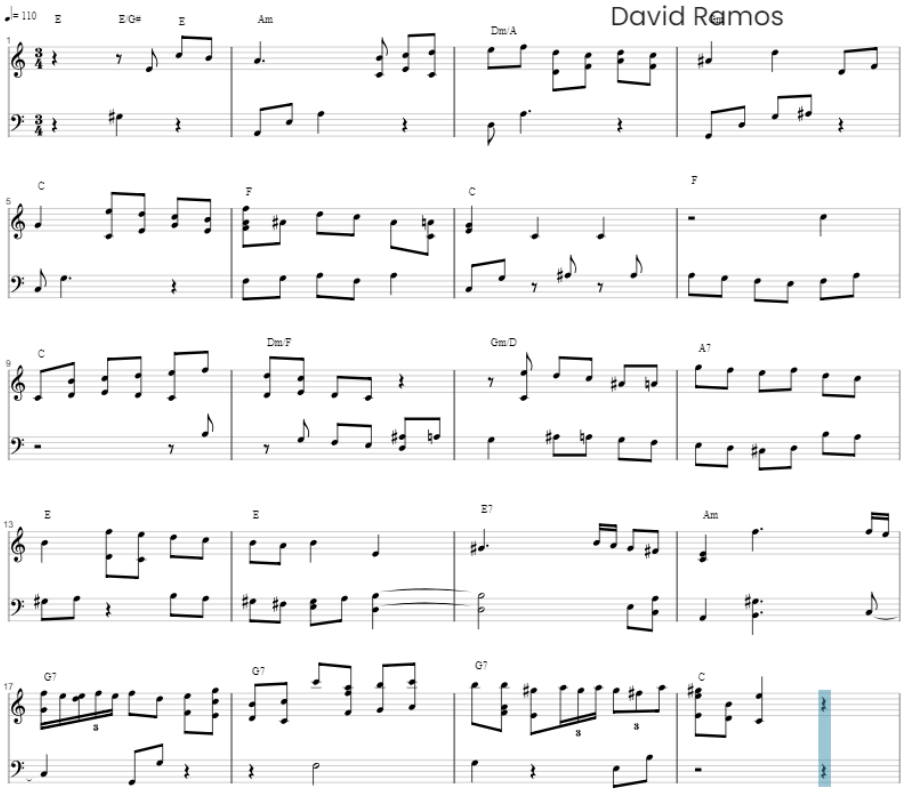


Figura 4.3: Composición de una continuación de Para Elisa con un estilo barroco

4.1.2. Modelo de estilo romántico

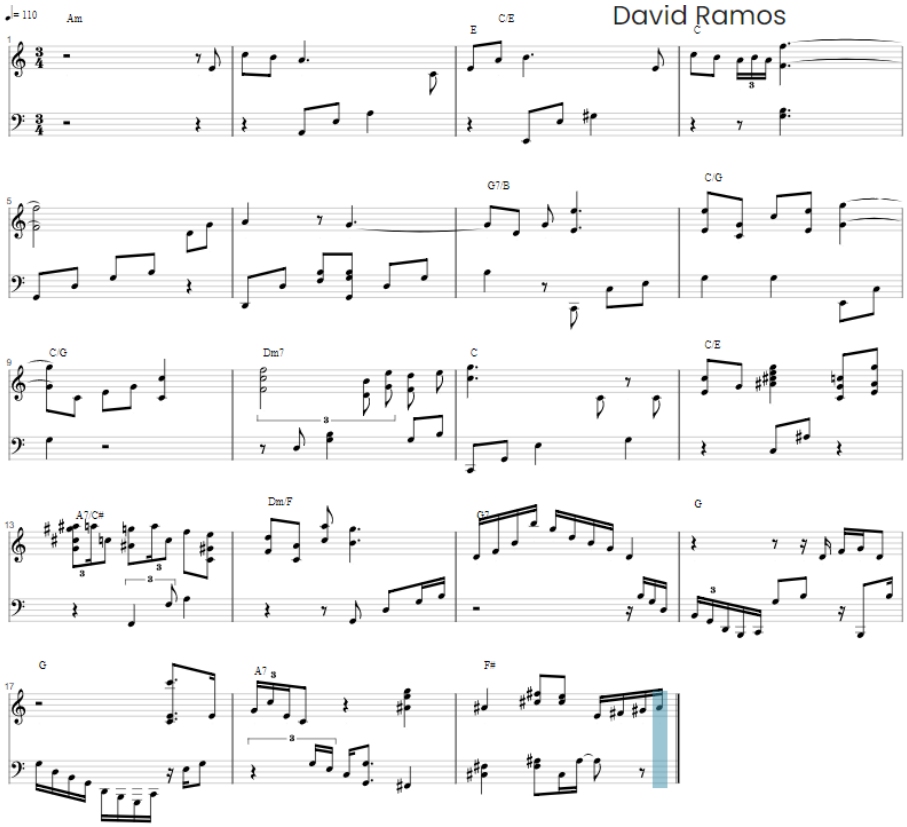


Figura 4.4: Composición de una continuación de Para Elisa con un estilo romántico

4.1.3. Modelo de estilo vanguardista

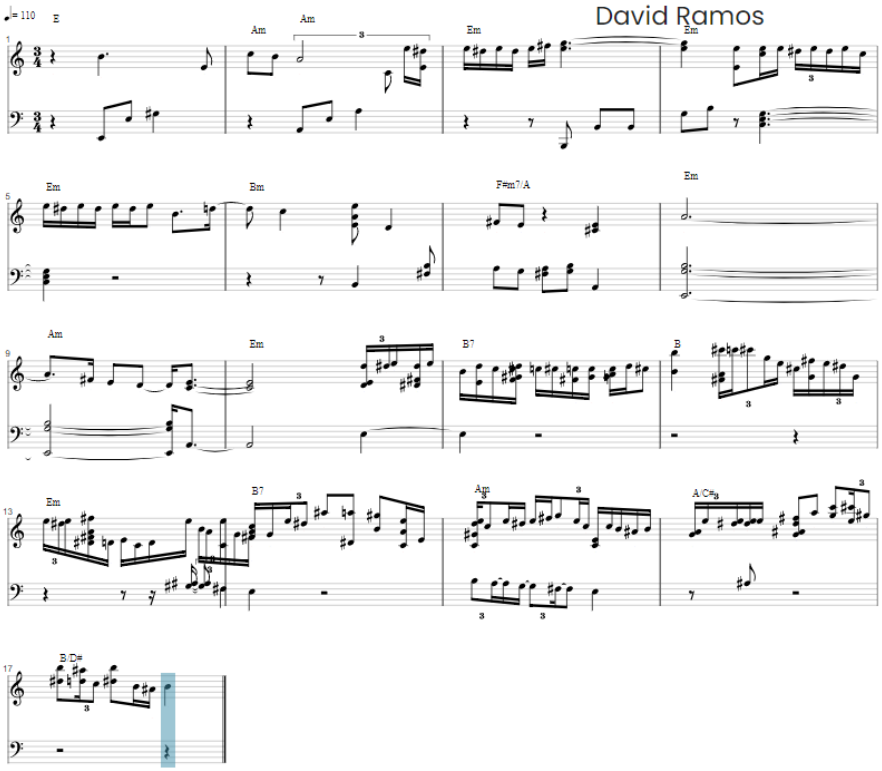


Figura 4.5: Composición de una continuación de Para Elisa con un estilo vanguardista

4.2. Objetivos logrados

En base a los resultados obtenidos, se puede observar que el modelo logra adaptar las composiciones a los estilos con los que ha sido reentrenado. Los modelos han sido capaces de aprender y replicar los patrones característicos de los estilos, algo que inicialmente no parecía algo sencillo. Algunos ejemplos de esto pueden ser la presencia de ornamentos (en especial trinos) como se puede ver en el ejemplo de composición barroca, y largos arpegios, como se puede apreciar en la composición romántica.

Otro de los aspectos positivos del modelo es que tiene coherencia. Esto es algo que al oído es muy importante, porque de otra forma puede crear sonidos desagradables. Unos de los principales aspectos respecto a esto, es que el modelo, en la gran mayoría de las ocasiones, genera la composición en la misma tonalidad en la que está la secuencia que se proporciona como entrada. Además, el modelo es capaz de realizar modulaciones. Esto es algo que ha aprendido de los propios datos de entrenamiento, ya que no se le ha proporcionado ninguna información sobre teoría de la música ni nada similar.

Por otro lado, se ha logrado hacer un notebook en Google Colab muy sencillo de usar y que o requiere ningún conocimiento técnico. Una de las grandes ventajas que tiene, es que es una aplicación web y puede ser fácilmente accedida desde un enlace y no es necesario registrarse en ningún sitio.

4.3. Problemas encontrados

- Limitaciones de recursos computacionales. Inicialmente, se barajó la posibilidad de usar otros modelos que emplean otras técnicas más novedosas y son más grandes, lo que hace producir mejores resultados. El problema de esto es que reentrenar la mayoría de estos modelos es computacionalmente muy costoso, por lo que se acabó eligiendo otro modelo más sencillo.

- Sobreajuste. Tras los primeros experimentos, se observó que esa forma de entrenamiento esta produciendo overfitting, por lo que no se podía considerara como válida. Finalmente, se optó por emplear otra estrategia, que fue a través del uso de la librería fastai, con la que se acabó consiguiendo los resultados deseados.

- Validación del modelo entrenado. El proyecto del que se partió como base, no incluía nada para evaluar el modelo durante el entrenamiento con unos datos diferentes a los que se usaban para entrenar. Esto es un gran problema, ya que no se puede evaluar realmente el rendimiento del modelo por el posible hecho de que se haya producido overfitting. Esto se solucionó añadiendo nuevo código al que ya se usaba para entrenar, y de este modo, se pudo observar que efectivamente, el modelo estaba teniendo overfitting.

Capítulo 5

Conclusiones y trabajos futuros

5.1. Conclusiones

Tras el proceso de reentrenamiento del modelo, este ha logrado capturar las características distintivas de los tres estilos de música clásica utilizados en los conjuntos de datos, así como generar música que se asemeja al estilo de los datos de entrenamiento. Esto indica que el modelo es capaz de aprender y adaptarse a diferentes estilos musicales, y por ello, se obtiene que las técnicas empleadas han dado los resultados esperados y han sido buenas.

Sin embargo, también has identificado ciertas limitaciones en el proceso de generación de música utilizando el modelo reentrenado. A pesar de que el modelo ha demostrado poder producir música coherente, algunos casos puntuales en los que las composiciones generadas parezca una secuencia aleatoria de notas, que aunque se respete la armonía, no llega a ser un resultado satisfactorio.

En conclusión, este trabajo ha mostrado una forma de generar música utilizando inteligencia artificial y cómo esta inteligencia artificial se ha podido adaptarse a diferentes estilos de música clásica.

5.2. Impacto social y medioambiental

La creación y difusión de música generada por inteligencia artificial pueden tener un impacto significativo en la sociedad. En primer lugar, la disponibilidad de herramientas de generación musical accesibles y fáciles de usar puede fomentar la creación musical, permitiendo que personas con diferentes niveles de experiencia y habilidades puedan adentrase en el mundo de la composición musical.

Además, la generación de música con un Transformer como el que se ha desarrollado puede contribuir a la revitalización de estilos musicales clásicos. Al entrenar el modelo con conjuntos de datos de música clásica, los usuarios pueden disfrutar de nuevas composiciones que evocan a los estilos musicales de épocas pasadas o incluso descubrir cómo serían sus canciones favoritas en uno de estos estilos.

5.3. Lineas futuras

Unas posibles líneas futuras de investigación pueden ser las siguientes:

- Utilizar otro algoritmo para tokenizar. A pesar de que el proyecto del que se ha partido ya contaba con un tokenizador, hay otros métodos más novedosos para tokenizar midis que pueden llegar a producir mejores

resultados. Una posible alternativa puede ser utilizar la librería MidiTok [44], que implementa diferentes algoritmos bien conocidos como pueden ser REMI [45] o Octuple [46].

- Crear otros conjuntos de datos sobre otros estilos musicales diferentes a los que se ha probado.
- Emplear un Transformer más grande, es decir, con más parámetros. Esto puede hacer que los resultados mejoren.
- Realizar un entrenamiento más largo, ya sea entrenando el modelo durante muchos maś epochs o entrenando más capas cuyos pesos quedaron congelados durante entrenamientos previos.
- Entrenar una GAN para que aprenda a detectar composiciones de mala calidad generadas por el modelo.
- Emplear métricas para la evaluación automática de los resultados generados por el modelo.

Bibliografía

[1] Wikipedia contributors, "Llm (modelo grande de lenguaje)," 2023, [Online; accessed 2-June-2023]. [Online]. Available: https://es.wikipedia.org/wiki/LLM_(modelo_grande_de_lenguaje)

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] OpenAI, "Gpt-4 technical report," 2023.

[4] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*. PMLR, 2021, pp. 8821–8831.

[5] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 684–10 695.

[6] Z. Zhang and Z. Zhang, "Artificial neural network," *Multivariate time series analysis in climate and environmental research*, pp. 1–35, 2018.

[7] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, mar 2020. [Online]. Available: https://doi.org/10.1016%2Fj.physd.2019.132306

[8] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[10] Z.-H. Zhou, *Machine learning*. Springer Nature, 2021.

[11] C. Kingsford and S. L. Salzberg, "What are decision trees?" *Nature biotechnology*, vol. 26, no. 9, pp. 1011–1013, 2008.

[12] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[14] 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.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[16] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[17] V. Raina, S. Krishnamurthy, V. Raina, and S. Krishnamurthy, "Natural language processing," *Building an Effective Data Science Practice: A Framework to Bootstrap and Manage a Successful Data Science Practice*, pp. 63–73, 2022.

[18] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot *et al.*, "Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp," *arXiv preprint arXiv:2112.10508*, 2021.

[19] OpenAI, "OpenAI Tokenizer," https://platform.openai.com/tokenizer, [Online; accessed 17-May-2023].

[20] M. M. Association, *Complete MIDI 1.0 Detailed Specification*, 1999/2008. [Online]. Available: http://www.midi.org/techspecs/gm.php

[21] Huggingface, "Fine-tuning a un modelo pre-entrenado," https://huggingface.co/docs/transformers/v4.19.0/es/training, [Online; accessed 2-June-2023].

[22] Google, "Google Colab," https://colab.research.google.com/, [Online; accessed 17-May-2023].

[23] W. McKinney, "Pandas," https://pandas.pydata.org/, [Online; accessed 17-May-2023].

[24] T. Oliphant, "Numpy," https://numpy.org/, [Online; accessed 17-May-2023].

[25] F. A. R. lab, "Pytorch," https://pytorch.org/, [Online; accessed 17-May-2023].

[26] R. T. Jeremy Howard, "Fastai," https://docs.fast.ai/, [Online; accessed 17-May-2023].

[27] J. D. Hunter, "Matplotlib," https://matplotlib.org/, [Online; accessed 17-May-2023].

[28] HoloViz, "HoloViews," https://holoviews.org/, [Online; accessed 17-May-2023].

[29] ——, "HvPlot," https://hvplot.holoviz.org/, [Online; accessed 17-May-2023].

[30] "tqdm," https://tqdm.github.io/, [Online; accessed 17-May-2023].

[31] P. Hanappe, "Fluidsynth," https://www.fluidsynth.org/, [Online; accessed 2-June-2023].

[32] B. Yu, P. Lu, R. Wang, W. Hu, X. Tan, W. Ye, S. Zhang, T. Qin, and T.-Y. Liu, "Museformer: Transformer with fine-and coarse-grained attention for music generation," *arXiv preprint arXiv:2210.10349*, 2022.

[33] 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.

[34] Z. Evans, "Dance Diffusion," https://github.com/pollinations/dance-diffusion, [Online; accessed 21-May-2023].

[35] Y.-J. Shih, S.-L. Wu, F. Zalkow, M. Muller, and Y.-H. Yang, "Theme transformer: Symbolic music generation with theme-conditioned transformer," *IEEE Transactions on Multimedia*, 2022.

[36] Y. Guo, Y. Liu, T. Zhou, L. Xu, and Q. Zhang, "An automatic music generation and evaluation method based on transfer learning," *Plos one*, vol. 18, no. 5, p. e0283103, 2023.

[37] asigalov61, "Giga-Piano," https://github.com/asigalov61/GIGA-Piano, [Online; accessed 21-May-2023].

[38] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," *arXiv preprint arXiv:1803.02155*, 2018.

[39] 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.

[40] M. Defferrard, K. Benzi, P. Vandergheynst, and X. Bresson, "FMA: A dataset for music analysis," in *18th International Society for Music Information Retrieval Conference (ISMIR)*, 2017. [Online]. Available: https://arxiv.org/abs/1612.01840

[41] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck, "Enabling factorized piano music modeling and generation with the MAESTRO dataset," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=r1lYRjC9F7

[42] P. Guillou, "Faster than training from scratch — Fine-tuning the English GPT-2 in any language with Hugging Face and fastai v2 (practical case with Portuguese)," https://medium.com/@pierre_guillou/faster-than-training-from-scratch-fine-tuning-the-english-gpt-2-in-any-language-with-hugging-f2ec05c98787, [Online; accessed 24-May-2023].

[43] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[44] N. Fradet, "MidiTok," https://github.com/Natooz/MidiTok, [Online; accessed 1-June-2023].

[45] 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*, 2020, pp. 1180–1188.

[46] M. Zeng, X. Tan, R. Wang, Z. Ju, T. Qin, and T.-Y. Liu, "Musicbert: Symbolic music understanding with large-scale pre-training," *arXiv preprint arXiv:2106.05630*, 2021.

[47] V. M. Dominguez Rivas, *Plantilla TFG ETSISI UPM.* ETSISI, 2020.

Anexos

Apéndice A

Código fuente del proyecto

A.0.1. arquitectura_modelo.py

```
class GPT(nn.Module):
    """  the full GPT language model, with a context size of block_size """

    def __init__(self, config):
        super().__init__()

        # input embedding stem
        self.tok_emb = nn.Embedding(config.vocab_size, config.n_embd)
        self.pos_emb = nn.Parameter(torch.zeros(1, config.block_size, config.n_embd))
        self.drop = nn.Dropout(config.embd_pdrop)
        # transformer
        self.blocks = nn.Sequential(*[Block(config) for _ in range(config.n_layer)])
        # decoder head
        self.ln_f = nn.LayerNorm(config.n_embd)
        self.head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        self.softmax = nn.Softmax(dim=-1)
        self.enable_rpr = config.enable_rpr

        self.block_size = config.block_size
        self.apply(self._init_weights)

class Block(nn.Module):
    """ an unassuming Transformer block """

    def __init__(self, config):
        super().__init__()
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.ln2 = nn.LayerNorm(config.n_embd)
        self.enable_rpr = config.enable_rpr
        if config.enable_rpr:
            self.attn = MultiheadAttentionRPR(config.n_embd, config.n_head,
                                config.attn_pdrop, er_len=config.er_len)
        else:
            self.attn = CausalSelfAttention(config)
        self.mlp = nn.Sequential(
            nn.Linear(config.n_embd, config.dim_feedforward),
            nn.GELU(),
            nn.Linear(config.dim_feedforward, config.n_embd),
            nn.Dropout(config.resid_pdrop),
        )
```

A.0.2. creacion_datasets.py

```
import urllib.request
import zipfile
import os

import random
from random import sample

import pandas as pd
```

```
random.seed(10)
RUTA_DATASET = './Dataset'
LISTA_COMPOSITORES = [['Johann Sebastian Bach'], ['Sergei Rachmaninoff'], ['Frédéric Chopin']]
nombres_zip = ['midis_barroco.zip', 'midis_vanguardias.zip', 'midis_romanticistmo.zip']
url = "https://storage.googleapis.com/magentadata/datasets/maestro/v3.0.0/maestro-v3.0.0-midi.zip"

nombre_archivo = os.path.basename(url)
ruta_archivo = os.path.join(RUTA_DATASET, nombre_archivo)

if not os.path.exists(RUTA_DATASET):
    os.makedirs(RUTA_DATASET)
urllib.request.urlretrieve(url, ruta_archivo)

with zipfile.ZipFile(ruta_archivo, 'r') as zip_ref:
    zip_ref.extractall(RUTA_DATASET)


def comprimir_archivos_en_zip(lista_archivos, nombre_zip):
    with zipfile.ZipFile(nombre_zip, 'w', zipfile.ZIP_DEFLATED) as archivo_zip:
        for archivo in lista_archivos:
            nombre_base = os.path.basename(archivo)
            archivo_zip.write(archivo, nombre_base)


def obtener_lista_ficheros(compositores, tam_lista):
    df = pd.read_csv(RUTA_DATASET + '/maestro-v3.0.0/maestro-v3.0.0.csv')
    ficheros_obras_compositores = df[df.canonical_composer.isin(
        compositores)]['midi_filename']

    ficheros_obras_compositores = ficheros_obras_compositores.values
    ficheros_obras_compositores = RUTA_DATASET + \
        '/maestro-v3.0.0/' + ficheros_obras_compositores
    ficheros_obras_compositores = sample(
        ficheros_obras_compositores.tolist(), tam_lista)
    return ficheros_obras_compositores



for nombre_zip, compositores_estilo in zip(nombres_zip, LISTA_COMPOSITORES):
    ficheros_obras_compositores = obtener_lista_ficheros(
        compositores_estilo, 50)
    comprimir_archivos_en_zip(ficheros_obras_compositores, nombre_zip)
```

A.0.3. dataset_dataloaders.py

```
SEQ_LEN = 128  # longitud de la secuencia de entrada
BATCH_SIZE = 16  # tamaño del batch

porcentaje_train = 0.9
n = int(len(train_data1) * porcentaje_train)

# división de entrenamiento y validación
data_train, data_val = torch.LongTensor(train_data1[:n]), torch.LongTensor(
    train_data1[n:]
)


class MusicSamplerDataset(torch.utils.data.Dataset):
    def __init__(self, data, seq_len):
        super().__init__()
        self.data = data
```

```
        self.seq_len = seq_len

    def __getitem__(self, index):
        idx = index * self.seq_len

        x = self.data[idx : idx + self.seq_len].long()
        trg = self.data[(idx + 1) : (idx + 1) + self.seq_len].long()

        return x, trg

    def __len__(self):
        return self.data.size(0)


train_dataset = MusicSamplerDataset(data_train, SEQ_LEN)
val_dataset = MusicSamplerDataset(data_val, SEQ_LEN)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=BATCH_SIZE)
```

A.0.4. entrenamiento.py

```
@torch.no_grad()
def estimate_loss(model, dataloader_test, batch_num, eval_iters):
    model.eval()
    losses = torch.zeros(eval_iters)
    accs = torch.zeros(eval_iters)
    print(f"\nValidating iter: {batch_num}")
    for k in range(eval_iters):
        batch = next(iter(dataloader_test))
        x = batch[0].to(get_device())
        tgt = batch[1].to(get_device())
        y, loss = model(x, tgt)
        acc = float(compute_epiano_accuracy(y, tgt))
        losses[k] = loss.item()
        accs[k] = acc
    loss_out = losses.mean()
    acc_out = accs.mean()
    print(f"End validation iter: {batch_num}  Loss: {loss_out}  -  Accuracy: {acc_out}")
    model.train()
    return loss_out, acc_out


def train(
    cur_epoch,
    model,
    dataloader,
    dataloader_test,
    loss,
    opt,
    lr_scheduler=None,
    num_iters=-1,
    save_checkpoint_steps=1000,
    eval_interval=50,
    eval_iters=75,
):
    loss_hist_train = []
    loss_hist_val = []
    acc_hist_train = []
    acc_hist_val = []
    save_steps = 0
    out = -1
    model.train()
```

```
with tqdm(total=len(dataloader)) as bar_train:
    for batch_num, batch in enumerate(dataloader):
        x = batch[0].to(get_device())
        tgt = batch[1].to(get_device())
        y, out = model(x, tgt)
        out.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)
        opt.step()
        opt.zero_grad()

        if lr_scheduler is not None:
            lr_scheduler.step()

        lr = opt.param_groups[0]["lr"]
        bar_train.set_description(
            f"Epoch: {cur_epoch} Loss: {float(out):.4} LR: {float(lr):.8}"
        )
        bar_train.update(1)
        if batch_num % eval_interval == 0:
            loss_hist_train.append(out.item())
            val_loss, val_acc = estimate_loss(
                model, dataloader_test, batch_num, eval_iters
            )
            loss_hist_val.append(val_loss)
            acc_train = float(compute_epiano_accuracy(y, tgt))
            acc_hist_train.append(acc_train)
            acc_hist_val.append(val_acc)

        if save_steps % save_checkpoint_steps == 0:
            print("Saving model progress. Please wait...")
            print(
                "gpt2_rpr_checkpoint_"
                + str(cur_epoch)
                + "_epoch_"
                + str(save_steps)
                + "_steps_"
                + str(round(float(out), 4))
                + "_loss.pth"
            )
            torch.save(
                model.state_dict(),
                "gpt2_rpr_checkpoint_"
                + str(cur_epoch)
                + "_epoch_"
                + str(save_steps)
                + "_steps_"
                + str(round(float(out), 4))
                + "_loss.pth",
            )
            print("Done!")
            print("Saving training loss graph...")
            plt.plot(
                list(range(len(loss_hist_train))),
                loss_hist_train,
                color="blue",
                label="train",
            )
            plt.plot(
                list(range(len(loss_hist_val))),
                loss_hist_val,
                color="red",
                label="val",
```

```
                )
                plt.savefig("gpt2_rpr_checkpoint_training_loss_graph.png")
            save_steps += 1

            if batch_num == num_iters:
                break

    return loss_hist_train, loss_hist_val, acc_hist_train, acc_hist_val
```

A.0.5. dataloader_fastai.py

```
from fastai.text.all import *
from fastai.data.load import DataLoader


def collate_fn(batch):
    x, trg = zip(*batch)
    x = torch.stack(x)
    trg = torch.stack(trg)
    return x, trg


ventana = 128
data = torch.tensor(train_data1)
samples = [data[i : i + ventana] for i in range(len(data) - ventana)]
target = [data[i + 1 : i + ventana + 1] for i in range(len(data) - ventana)]

X = torch.stack(samples)
targets = torch.stack(target)

longitud_total = X.shape[0]
porcentaje_train = 0.9
n = int(longitud_total * porcentaje_train)
train_dl = DataLoader(
    zip(X[:n], targets[:n]), bs=32, collate_fn=collate_fn, n=n
)  # el parámetro n es el tamaño del dataset y bs el tamaño de batch
valid_dl = DataLoader(
    zip(X[n:], targets[n:]), bs=32, collate_fn=collate_fn, n=longitud_total - n
)

dls = DataLoaders(train_dl, valid_dl, device=device)
```

A.0.6. splitter.py

```
def splitter(model):
    "Split a GPT2 `model` in 4 groups for differential learning rates."

    modules = model.blocks[:4]
    groups = [nn.Sequential(*modules)]

    modules = model.blocks[4:8]
    groups = L(groups + [nn.Sequential(*modules)])

    modules = model.blocks[8:12]
    groups = L(groups + [nn.Sequential(*modules)])

    modules = model.blocks[12:]
    groups = L(groups + [nn.Sequential(*modules)])

    groups = L(groups + [nn.Sequential(model.tok_emb, model.ln_f)])
```

```
    return groups.map(params)
```

A.0.7. learner.py

```
class DropOutput(Callback):
    def after_pred(self):
        self.learn.pred = self.pred[0]


learn = Learner(
    dls,
    model,
    loss_func=CrossEntropyLossFlat(),
    splitter=splitter,
    cbs=[DropOutput],
    metrics=[accuracy, Perplexity()],
).to_fp16()
```

A.0.8. entrenamiento_fastai.py

```
lr_found = learn.lr_find()
valley = lr_found.valley

learn.freeze()
learn.fit_one_cycle(1, valley / 2)

learn.save(RUTA_GUARDAR_MODELO)
```