



Universidad Internacional de La Rioja
Facultad de Ciencias Sociales y Humanidades

Máster Universitario en Investigación Musical

All You Need Is AI Muse: hacia el desarrollo de interfaces interactivas con grandes modelos de lenguaje para la creación y performance musical algorítmica

| | |
|---|---------------------------|
| Trabajo de fin de estudio presentado por: | Autor |
| Tipo de trabajo: | Investigación |
| Director: | Luis Miguel Morales Nieto |
| Fecha: | 13 de febrero de 2024 |

*A mis hijos, Ana y Daniel,
para que nunca pierdan la curiosidad
y la capacidad de admiración
por el mundo que les rodea.*

Resumen

Este trabajo se centra en la interacción integral de Modelos de Lenguaje a Gran Escala (LLM) con la creación de música algorítmica, examinando cómo estos modelos pueden influir en el proceso creativo. La investigación profundiza en la capacidad de los LLM para generar código de programación musical en diversas interfaces humano-máquina. *AI Muse* es una herramienta específicamente diseñada para este estudio, que optimiza la ejecución de código en entornos de *live coding* con SuperCollider y Tidal Cycles, facilitando una comunicación fluida y directa. El estudio explora la generación de estructuras sonoras y patrones rítmicos a través de un proceso iterativo que busca maximizar la precisión y la creatividad de las respuestas de los LLM. *AlgorAI* es una obra de arte sonoro compuesta en la investigación, e ilustra tanto el potencial creativo como las limitaciones de los LLM, resaltando la importancia de la colaboración humano-máquina para superar desafíos en la generación de estructuras complejas. Los resultados indican que los LLM pueden desempeñar un rol significativo como asistentes creativos, herramientas de exploración y fuentes de inspiración.

Palabras clave:

Modelos de lenguaje a gran escala, aprendizaje profundo, creación musical, interacción humano-máquina, *live coding*.

Abstract

This study delves into the comprehensive interaction between Large Language Models (LLMs) and the creation of algorithmic music, investigating the impact these models can have on the creative process. It examines the capability of LLMs to produce musical programming code across various human-machine interfaces. *AI Muse*, a tool specifically developed for this research, enhances code execution in live coding environments utilizing SuperCollider and Tidal Cycles, thereby ensuring smooth and direct communication. The research focuses on the iterative generation of sound structures and rhythmic patterns, aiming to optimize both the accuracy and creativity of LLM responses. *AlgorAI*, a piece of sound art created during the study, showcases the creative potential and limitations of LLMs, emphasizing the crucial role of human-machine collaboration in overcoming challenges associated with generating complex structures. The findings indicate that LLMs can serve as creative assistants, exploration tools, and sources of inspiration.

Keywords:

Large Language Models, Deep Learning, Music Creation, Human-Machine Interaction, Live Coding.

Índice de contenidos

| | |
|---|----|
| Índice de contenidos | 5 |
| Índice de figuras | 10 |
| Índice de tablas | 12 |
| Índice de acrónimos | 13 |
| 1 Introducción | 14 |
| 1.1 Consideraciones | 14 |
| 1.2 Justificación | 16 |
| 1.3 Objetivos de la investigación | 18 |
| 2 Marco teórico y estado de la cuestión | 19 |
| 2.1 Inteligencia artificial, machine learning, deep learning e inteligencia artificial generativa | 19 |
| 2.1.1 La inteligencia artificial y sus precursores | 19 |
| 2.1.2 <i>Machine learning</i> | 21 |
| 2.1.3 <i>Deep learning</i> | 23 |
| 2.1.3.1 El concepto de <i>neurona artificial</i> | 23 |
| 2.1.3.2 Redes neuronales artificiales | 25 |
| 2.1.3.3 Redes neuronales recurrentes | 26 |
| 2.1.3.4 La arquitectura <i>transformer</i> | 27 |
| 2.2 Modelos de lenguaje (LM) | 28 |
| 2.2.1 Grandes modelos de lenguaje (LLM) | 30 |
| 2.2.2 Modelos preentrenados y <i>fine-tuning</i> | 31 |
| 2.2.3 Hiperparámetros y parámetros del modelo | 32 |
| 2.2.4 Temperatura, top-p y top-k | 34 |
| 2.3 <i>Prompting engineering</i> | 35 |

| | | |
|---------|---|----|
| 2.3.1 | Técnicas más importantes de <i>prompting engineering</i> | 36 |
| 2.3.1.1 | <i>Few-shot prompting</i> | 37 |
| 2.3.1.2 | <i>Chain of thoughts</i> | 37 |
| 2.3.1.3 | <i>Self consistency of chain of thoughts</i> | 38 |
| 2.3.1.4 | <i>Tree of thoughts</i> | 39 |
| 2.3.2 | <i>Retrieval-augmented generation</i> | 40 |
| 2.3.2.1 | <i>Prompting</i> versus <i>fine-tuning</i> versus <i>Retrieval-Augmented Generation</i> | 42 |
| 2.4 | Limitaciones conocidas de los LLM | 42 |
| 2.4.1 | Corte de conocimiento | 43 |
| 2.4.2 | Alucinaciones | 43 |
| 2.4.3 | Ineficiencia y coste | 43 |
| 2.4.4 | Modelos de <i>caja negra</i> , con sesgos y riesgos | 45 |
| 2.5 | Estado de la cuestión | 45 |
| 2.5.1 | Modelos de IA generativa aplicados a la música | 46 |
| 2.5.2 | LLM como asistentes para la creación de música simbólica | 47 |
| 2.5.3 | LLM en la creación de código de programación | 48 |
| 3 | Metodología | 50 |
| 3.1 | Revisión bibliográfica | 50 |
| 3.1.1 | Objetivos de la revisión | 50 |
| 3.1.2 | Fuentes y búsqueda | 50 |
| 3.1.3 | Selección y evaluación | 50 |
| 3.1.4 | Resultados de la revisión | 51 |
| 3.2 | Enfoque, alcance y diseño | 51 |
| 3.2.1 | Enfoque | 51 |
| 3.2.2 | Alcance | 51 |
| 3.2.3 | Diseño | 52 |
| 3.3 | Herramientas y recursos utilizados | 52 |
| 3.4 | Desarrollo y aplicación | 52 |
| 3.4.1 | Discusión de los criterios de elección de los lenguajes de programación musical | 52 |
| 3.4.2 | Elección de los sistemas de <i>large language model</i> (LLM) a utilizar | 53 |

| | |
|---|----|
| 3.4.3 Exploración de posibles interfaces de interacción con sistemas de LLM | 53 |
| 3.4.4 Documentación de las interacciones | 53 |
| 3.4.5 Creación de una pieza de arte sonoro | 53 |
| 3.4.6 Análisis cualitativo y reflexión | 53 |
| 3.4.7 Reflexión de las limitaciones y prospectivas | 53 |
| 3.4.8 Disponibilidad y publicación de los scripts del trabajo | 54 |
| 4 Elección de lenguajes de programación musical para este trabajo | 55 |
| 4.1 Música en código de programación | 55 |
| 4.2 El problema de componer con notas musicales | 57 |
| 4.3 Lenguajes orientados a la síntesis de sonido | 59 |
| 4.4 Criterios de selección de lenguajes de programación musical para esta investigación | 60 |
| 4.4.1 Comunidad activa de usuarios y desarrolladores | 60 |
| 4.4.2 Documentación extensa y accesible | 60 |
| 4.4.3 Capacidad de trabajo en tiempo real | 61 |
| 4.4.4 Fácil gestión de errores | 62 |
| 4.4.5 Independencia del IDE | 63 |
| 4.4.6 Lenguajes dominados por el investigador. | 63 |
| 4.4.7 SuperCollider y Tidal Cycles | 63 |
| 5 Análisis y evaluación de la práctica con diferentes interfaces en OpenAI | 65 |
| 5.1 ChatGPT, como primer banco de pruebas | 65 |
| 5.1.1 GPT-4, modelo utilizado en todos los experimentos | 66 |
| 5.2 Rendimiento de ChatGPT en tareas de generación de código sonoro | 66 |
| 5.2.1 Creación de esbozos generales de una obra | 67 |
| 5.2.2 Creación de bloques de código sencillos | 70 |
| 5.2.2.1 Alucinaciones recurrentes | 71 |
| 5.2.2.2 Tendencia a valores por defecto | 73 |
| 5.2.2.3 Introducción de errores de sintaxis en código previamente correcto | 74 |
| 5.2.2.4 Ocasionales errores de sintaxis que escapan a la autocorrección . | 74 |
| 5.2.2.5 Gran distancia entre el sonido producido y el sonido esperado . | 75 |
| 5.2.3 Influjo del uso de técnicas de prompting en ChatGPT | 76 |
| 5.2.3.1 <i>Zero-shot</i> versus <i>few-shot</i> | 76 |
| 5.2.3.2 <i>Chain of Thoughts</i> (CoT) y <i>Structured Chain of Thoughts</i> (SCoT) . | 77 |

| | |
|--|-----|
| 5.2.3.3 Técnica de <i>self-debugging</i> | 79 |
| 5.3 Mayor control de parámetros con el Playground de OpenAI | 79 |
| 5.4 Ampliando el <i>Knowledge: retrieval-augmented generation</i> | 80 |
| 5.5 Programando con la API de OpenAI | 81 |
| 5.5.1 Generación automática de código en Tidal Cycles | 81 |
| 6 <i>AI Muse</i> , herramienta en Python para la interacción en vivo con LLM en entornos de <i>live coding</i> | 83 |
| 6.1 Descripción general de la herramienta | 83 |
| 6.2 ¿Por qué un entorno de <i>live coding</i> ? | 83 |
| 6.3 Integración de la API de OpenAI | 84 |
| 6.4 Lenguajes de programación musical disponibles | 84 |
| 6.5 Flujo de trabajo | 84 |
| 6.6 Archivo de configuración y comandos | 86 |
| 6.7 Resultados del uso de <i>AI Muse</i> | 87 |
| 7 <i>AlgorAI</i> : creación de una pieza de arte sonoro generado con ayuda de GPT-4 | 90 |
| 7.1 Proceso y criterios de selección de material sonoro | 90 |
| 7.2 Descripción de la pieza | 91 |
| 7.3 Técnicas de síntesis sonora utilizadas en los materiales sonoros | 92 |
| 7.4 Interacción humano-LLM en el proceso de composición | 94 |
| 8 Resultados y discusión | 96 |
| 8.1 Resultados | 96 |
| 8.2 Discusión crítica de los resultados | 97 |
| 9 Conclusiones | 99 |
| 10 Limitaciones y prospectiva | 101 |
| 10.1 Limitaciones | 101 |
| 10.2 Prospectiva | 102 |
| Referencias bibliográficas | 104 |
| Anexo A Materiales sonoros generados por GPT-4 utilizados en la composición <i>AlgorAI</i> | 115 |
| A.1 Códigos de materiales sonoros escritos en SuperCollider | 115 |

| | |
|--|-----|
| A.2 Códigos de materiales sonoros escritos en Tidal Cycles | 127 |
| Anexo B Repositorio con los materiales de la investigación | 133 |

Índice de figuras

| | |
|--|----|
| 1.1 Número de publicaciones científicas del campo de la inteligencia artificial por mes que contienen la expresión «All You Need» en su título | 15 |
| 2.1 Relación entre inteligencia artificial, machine learning, deep learning e inteligencia artificial generativa. | 19 |
| 2.2 Test de Turing | 20 |
| 2.3 Problema de clasificación versus regresión | 22 |
| 2.4 Esquema de aprendizaje supervisado | 23 |
| 2.5 Neurona biológica y neurona artificial | 24 |
| 2.6 Estructura de capas de una posible red neuronal artificial | 25 |
| 2.7 Diagrama de entrenamiento de una red neuronal artificial | 26 |
| 2.8 Arquitectura de una red neuronal recurrente | 27 |
| 2.9 Matriz de <i>atención</i> de un <i>transformer</i> entrenado con textos en inglés | 28 |
| 2.10 Inferencia de token de un LLM | 29 |
| 2.11 Generación de un texto completo por medio de la autorregresión por un LLM . . | 29 |
| 2.12 Generación de textos por <i>GPT-2</i> | 30 |
| 2.13 Gráfico comparativo de tamaños de LLM | 31 |
| 2.14 Ejemplos intuitivos de transfer learning en LLM | 32 |
| 2.15 Precisión de GPT-3 en función del tamaño de la ventana de contexto | 33 |
| 2.16 Ventana de contexto y pérdida de precisión en su interior | 34 |
| 2.17 Control de la temperatura en la generación de texto por un LLM | 35 |
| 2.18 Control de los parámetros top-k y top-p en la generación de texto por un LLM . . | 36 |
| 2.19 Técnica de few-shot prompting | 37 |
| 2.20 Chain of thoughts | 38 |
| 2.21 Esquema de los pasos en los que se divide la técnica Self Consistency of Chain of Thoughts | 39 |
| 2.22 Técnicas de <i>prompting engineering</i> | 40 |

| | |
|--|----|
| 2.23 Habilidades de los <i>Foundation Models</i> de OpenAI | 41 |
| 2.24 Esquema de funcionamiento de RAG | 42 |
| 2.25 Coste de entrenamiento en horas de GPU de diversos modelos de lenguaje | 44 |
| 2.26 Esquema de las diferentes aproximaciones a la generación de música con IA | 46 |
| 2.27 Esquema de funcionamiento en dos pasos de MuseCoco texto–música | 47 |
| 2.28 Esquema de funcionamiento de WavJourney | 48 |
| | |
| 3.1 Publicaciones por año de las referencias utilizadas | 51 |
| | |
| 4.1 «¡Hola Mundo!» en diferentes lenguajes de programación sonora | 56 |
| 4.2 Ejemplo de patch de Pure Data | 57 |
| 4.3 Melodía de 4 compases generada por GPT-4, «al estilo de Bach» | 58 |
| 4.4 Bucle infinito en Sonic Pi | 61 |
| | |
| 5.1 Respuesta de Bard y ChatGPT a un mismo prompt | 67 |
| 5.2 Conversación con ChatGPT para crear una estructura general de una obra | 69 |
| 5.3 Alucinación de ChatGPT utilizando una clase que no existe | 71 |
| 5.4 Código saturado en amplitud, generado por ChatGPT | 72 |
| 5.5 Error difícil de detectar en SuperCollider | 73 |
| 5.6 Código generado por ChatGPT con utilización de valor de frecuencia de 440 Hz . | 74 |
| 5.7 Desnivel semántico de ChatGPT entre el prompt y el código generado en Super- Collider | 76 |
| 5.8 Comparación de resultados de ChatGPT con zero-shot y few-shot | 78 |
| 5.9 Iteración de depuración de código con ChatGPT | 79 |
| 5.10 Ejemplo de prompt de sistema para generar código en SuperCollider | 80 |
| | |
| 6.1 Flujo de trabajo de <i>AI Muse</i> | 85 |
| 6.2 Ejemplo de archivo de configuración config.json de <i>AI Muse</i> | 87 |
| 6.3 Prompts de sistema de <i>AI Muse</i> | 88 |
| | |
| 7.1 Sonograma de una parte de la pieza electroacústica <i>AlgorAI</i> | 92 |
| 7.2 Gráfica circular con la distribución de las técnicas de síntesis sonora utilizadas en los materiales sonoros generados por GPT-4 en SuperCollider | 93 |
| 7.3 Ejemplo de código en Tidal Cycles escogido para la composición por su carácter irregular e impredecible | 94 |

Índice de tablas

Índice de acrónimos

- API** *application programming interface.* 44, 52, 53, 61, 63, 65, 66, 81–86, 89, 90, 96, 97, 99, 103, 133
- CNN** *red neuronal convolucional.* 25, 26
- COT** *chain of thoughts.* 37–39, 49, 67, 68, 76
- COT-SC** *self consistency of chain of thoughts.* 39
- DAW** *digital audio workstation.* 52, 90
- DL** *deep learning.* 16, 19, 23, 26, 27, 45, 46, 50
- IA** *inteligencia artificial.* 14–21, 43–45, 50, 52, 77, 81, 94–96, 99–103
- IAG** *inteligencia artificial generativa.* 14, 19, 28, 97, 103
- LLM** *large language model.* 6–8, 14, 16–18, 30–32, 34–45, 47–53, 55, 57, 59–63, 66, 68–71, 74–77, 79, 80, 83, 87–89, 91, 95–100, 102, 103
- LM** *language model.* 21–23, 28, 29, 40, 43
- LSTM** *long short-term memory.* 27
- ML** *machine learning.* 19, 21–23, 29, 30, 99
- RAG** *retrieval-augmented generation.* 40–43, 77, 80, 81, 84, 97, 98, 102, 123–126
- RNA** *red neuronal artificial.* 23–27
- RNN** *recurrent neural network.* 26, 27
- sCOT** *structured chain of thoughts.* 49, 76, 77

1. Introducción

ALL YOU NEED IS A GOOD INIT

— Mishkin y Matas (2016)

Este trabajo de fin de máster se centra en analizar de forma integral la interacción humano-máquina, con sistemas LLM, en el ámbito de la creación de música algorítmica mediante los lenguajes de programación musical de SuperCollider y Tidal Cycles. Se abordará, en concreto, cómo estos modelos de lenguaje, al ser capaces de generar código de programación a partir de texto en lenguaje natural, influyen en diversos aspectos del proceso creativo. Así mismo, se explorará la eficiencia y usabilidad de distintas interfaces de interacción entre el compositor y los LLM, así como el impacto de las técnicas de *prompting* en los resultados de composición. Además, se reflexionará sobre el valor artístico de las obras generadas y el efecto de estos sistemas de *inteligencia artificial* (IA) en el proceso creativo y la percepción subjetiva del artista humano creador, proporcionando así claves para un entendimiento profundo el papel de los modernos LLM en la innovación y evolución de la música algorítmica.

La aproximación metodológica será de carácter cualitativo, en consonancia con el enfoque exploratorio y descriptivo del estudio. De una forma práctica y como parte y producto de este estudio, se ha creado un software para la generación de código musical en vivo, *AI Muse*, que se presenta en el capítulo 6, y que queda a disposición de la comunidad científica como software libre; y una obra de arte sonoro, *AlgorAI*, compuesta a partir de los resultados del estudio, sobre la cual tratamos en el capítulo 7.

La intención final de este trabajo es aportar conocimiento sobre la interacción entre el ser humano y las herramientas modernas de *inteligencia artificial generativa* (IAG), desde un punto de vista artístico y creativo. Al mismo tiempo, se pretende ofrecer una visión crítica y reflexiva sobre el papel real que en este momento pueden desempeñar sistemas construidos sobre tecnologías de LLM, y apuntar hacia futuras líneas de investigación musical con IA.

1.1 Consideraciones

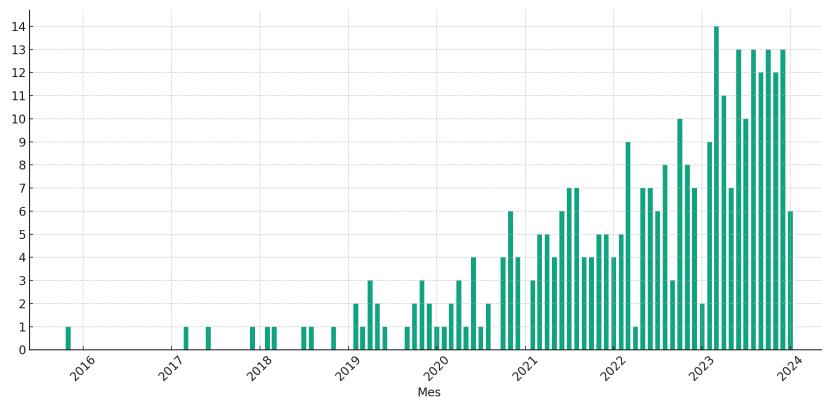
Muchos términos científicos utilizados se ha optado por escribirlos en inglés, ya que son ampliamente conocidos y usados en su forma original inglesa incluso en el ámbito científico hispano.

nohablante (como *machine learning* o *deep learning*), mientras otros se encuentran en español por haberse considerado la forma más común (como *inteligencia artificial* o *red neuronal artificial*). En todo caso, se ha ofrecido una traducción de todos los términos ingleses en su primer uso para facilitar la lectura. Los términos utilizados repetidamente a lo largo del texto se encuentran en forma de acrónimos, los cuales se han explicitado en su primer uso y en los lugares donde ha sido necesario para mantener la claridad del texto. El desglose de estos acrónimos puede consultarse en el glosario de la página 13, a donde apuntan todos ellos por medio de hipervínculos.

Los códigos QR enlazan a recursos complementarios al texto, los cuales pueden cambiar o desaparecer con el tiempo. Por ello, en el Anexo B se incluye un enlace al repositorio de GitHub donde se aloja el código fuente en \LaTeX de este trabajo, así como los materiales sonoros y de software generados en el marco de este trabajo.

El título de este trabajo hace un guiño a las ya cientos de publicaciones científicas que utilizan la expresión «All You Need» en el título (véase la Figura 1.1). Esta expresión se ha convertido en un ícono de los avances en IA, especialmente desde la publicación en 2017 del emblemático artículo *Attention Is All You Need* (Vaswani y cols., 2017). El lector curioso puede encontrar una lista actualizada de estas publicaciones en el repositorio de GitHub *Awesome "all you need" papers* (Nishi, 2024). En el mismo sentido, los epígrafes de los capítulos son algunos de estos títulos, escogidos sin más pretensión que significar un juego de palabras.

Figura 1.1 Número de publicaciones científicas del campo de la inteligencia artificial por mes, hasta el 21 de enero de 2024, que contienen la expresión «All You Need» en su título.



Fuente: Elaboración propia a partir del listado de Nishi (2024)

1.2 Justificación

La evolución de la IA, específicamente en el dominio del aprendizaje profundo o *deep learning* (DL), ha llevado al desarrollo de modelos de lenguaje a gran escala o *large language models* (LLM). Estos modelos, originalmente diseñados para emular el lenguaje humano, y conocidos por el gran público a través de aplicaciones tipo chatbot¹ han demostrado capacidades emergentes en áreas relacionadas con la creatividad y el razonamiento. Su éxito se extiende a sectores diversos como la literatura, el arte visual, la investigación, el marketing y la educación, afectando prácticamente a todas las disciplinas que involucran el uso del lenguaje o el razonamiento.

Los LLM han probado ser especialmente eficaces en la generación de código de programación, entre los que podemos incluir lenguajes relacionados con la creación musical o sonora, siempre que el modelo haya contado para su entrenamiento con suficiente material escrito en el lenguaje en cuestión.

Hay de señalar que existe una vertiente de desarrollo de modelos para la generación de audio, como Stable Audio (*Audio*, s.f.), MusicML (*MusicLM*, s.f.) de Google y AWS DeepComposer (*AWS DeepComposer*, s.f.) de Amazon. Dichos modelos generan audio a partir de texto natural, mostrando resultados muy prometedores —particularmente para el marketing y la creación de contenido— por la potencial reducción de costes de producción. Sin embargo, los archivos o flujos de audio generados son inalterables y escapan al control humano, lo cual limita su aplicación en composición musical donde, entendemos, la intervención humana es esencial². En contraste, los LLM generan texto, incluso texto musical, lo que permite la edición y el control por parte del compositor, y abre, así, amplias posibilidades de interacción humano-máquina en el proceso creativo.

Se han realizado muchos estudios sobre la capacidad de los LLM para generar algoritmos en lenguajes de programación enfocados en la resolución de problemas matemáticos y el desarrollo de software, campo en el que los LLM han mostrado una elevada efectividad. No obstante, existe una carencia en lo que respecta al estudio de la generación de código musical con finalidad artística y su alineación con las intenciones del creador sonoro o compositor. No es

¹Entre las aplicaciones de chatbot más conocidas están ChatGPT (*Introducing ChatGPT*, s.f.), Claude (*Introducing Claude*, s.f.), Bard (“Bard (Chatbot)”, 2024) o Microsoft Copilot (Mehdi, 2023), aunque la lista cada vez es más larga e incluye modelos de lenguaje de código abierto como LLaMa (Touvron y cols., 2023), Mistral (Jiang y cols., 2023) o Falcon (Almazrouei y cols., 2023), entre otros.

²En nuestro estudio se partirá del postulado de que el factor creador humano es esencial para que un producto sea considerado o no arte.

equiparable, en este sentido, desarrollar, por ejemplo, un código que muestre el conjunto de Mandelbrot³ con crear una obra musical inspirada en dicho fractal. Al contrario de lo que se podría intuir en un primer momento, el segundo caso requiere, con creces, de un nivel superior de comprensión, abstracción, conocimientos previos y conexión con la realidad que el primero, incluso si su código puede ser eventualmente más simple que el de la función matemática. Por ello, resulta fundamental realizar estudios específicos que exploren la capacidad de los LLM en la generación de código musical con finalidad artística así como los modos de potenciarla.

Un aspecto muy importante a investigar a la hora de estudiar la utilidad e influencia de sistemas de IA en la creación artística es el de la *interfaz* entre el artista y el sistema. La interfaz es el medio por el cual el artista interactúa con el sistema, y puede ser de muy diversa índole. En el caso de los LLM, la interfaz más común es la chatbot, donde el artista se comunica en forma de diálogo en lenguaje natural. Sin embargo, existen otras interfaces que pueden ser más adecuadas para la creación musical, como la interfaz gráfica, la interfaz de voz o la interfaz gestual, etc.

Aunque se dispone de numerosos lenguajes de programación enfocados en la creación musical relevantes para este estudio, la investigación se centrará en dos de ellos por limitaciones de espacio. El principal será *SuperCollider* (*SuperCollider*, 2024), un lenguaje consolidado para síntesis sonora y procesamiento de audio, que ofrece un entorno propicio para esta investigación por su riqueza expresiva y reconocimiento en la comunidad de música electrónica y experimental. El segundo, *Tidal Cycles* (*Tidal Cycles*, s.f.), se basa en la filosofía de la improvisación sonora en interpretaciones en vivo o *live coding*, lo cual permite la exploración de interfaces innovadoras. La combinación de las capacidades de codificación de los LLM con la versatilidad de estos lenguajes promete un panorama lleno de posibilidades creativas. Este estudio pretende investigar dicho panorama, buscando comprender cómo la creatividad emergente de los LLM se manifiesta en la composición musical algorítmica y cómo esta interacción puede ampliar las fronteras de la música digital. Los resultados, mutatis mutandis, serán eventualmente extrapolables a cualquier lenguaje estructurado orientado a la creación sonora o musical, como Overtone (*Overtone - Collaborative Programmable Music*, s.f.), Pure Data (*Pure Data — Pd Community Site*, s.f.) y Sonic Pi (*Sonic Pi - The Live Coding Music Synth for Everyone*, s.f.), entre otros.

Este trabajo busca aportar al mundo de la música y, en concreto, del arte sonoro, una nueva perspectiva sobre la creación musical, apoyada en la investigación en IA, para expandir las

³El *conjunto de Mandelbrot* es una de las gráficas fractales más conocidas popularmente, la cual es un subconjunto del plano complejo (“Conjunto de Mandelbrot”, 2024).

fronteras de la creatividad y explorar interfaces innovadoras entre el artista y la máquina. Aspiramos a que sea una invitación a la experimentación y la investigación en este campo en rápido desarrollo.

Asimismo, se espera que la comunidad científica de la IA encuentre en este estudio una retroalimentación valiosa desde la investigación musical, contribuyendo a mejorar los modelos de IA y sus aplicaciones en el campo artístico. Igualmente, este trabajo quiere incentivar la investigación y desarrollo de LLM *open source*, lo cual, eventualmente, fomentará la colaboración futura entre las comunidades científica y artística en la creación de tecnologías abiertas para la creación musical y sonora.

1.3 Objetivos de la investigación

Objetivo general: Analizar de forma integral la interacción con Modelos de Lenguaje a Gran Escala (LLM) en el ámbito de la creación de música algorítmica en SuperCollider y Tidal Cycles, centrándose en aspectos como la diversidad de interfaces, la adecuación de estos modelos a entradas en lenguaje natural y su influencia en las decisiones creativas durante el proceso compositivo y performativo.

Objetivos específicos:

- a) Examinar la habilidad de los LLM para generar código de programación musical basado en texto en lenguaje natural.
- b) Evaluar la eficiencia y usabilidad de distintas interfaces de interacción entre el compositor y los LLM en la creación sonora.
- c) Explorar el impacto de las diversas técnicas de *prompting* en los resultados producidos por sistemas de LLM.
- d) Reflexionar sobre el valor artístico de las composiciones y códigos musicales generados mediante la interacción con LLM.
- e) Considerar el impacto del uso de sistemas de IA en el proceso creativo y en la experiencia perceptual del artista.

2. Marco teórico y estado de la cuestión

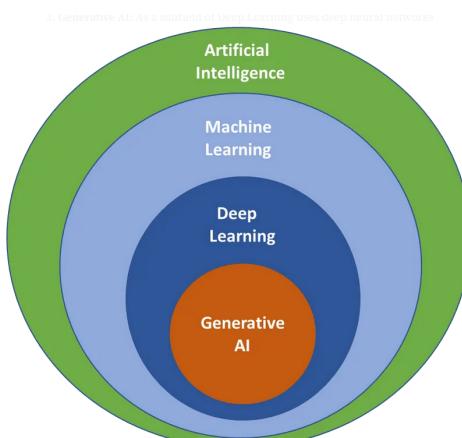
A BROAD DATASET IS ALL YOU NEED

— Michaelis, Bethge, y Ecker (2022)

2.1 Inteligencia artificial, machine learning, deep learning e inteligencia artificial generativa

Comencemos delimitando las diferentes áreas en las que se inscribe el estudio y desarrollo de sistemas de inteligencia artificial. A menudo se encuentran términos como inteligencia artificial, *machine learning* (ML), DL, e, incluso, IAG, usados de forma intercambiable. Sin embargo, cada uno tiene un significado específico, y consideramos esencial diferenciarlos para comprender el estado actual de la investigación. Estos conceptos se organizan jerárquicamente (Torres i Viñals, 2020), donde la IA es el término más general, que abarca el ML, y este último incluye al DL y la IAG (véase Figura 2.1).

Figura 2.1 Relación entre inteligencia artificial, machine learning, deep learning e inteligencia artificial generativa.



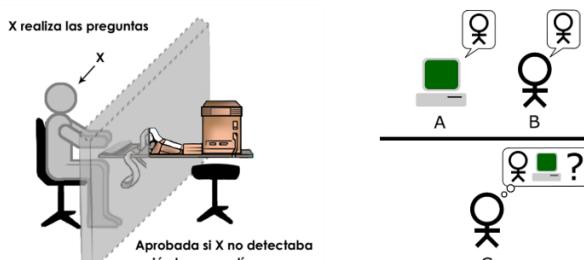
Fuente: Kainat (2023)

2.1.1 La inteligencia artificial y sus precursores

La IA, como concepto genérico, es el campo de estudio más antiguo entre todos, y no se limita únicamente al ámbito computacional. En su sentido más amplio, la IA ha sido abordada por la filosofía. Sin embargo, no es hasta el siglo XX cuando se empieza a considerar la posi-

bilidad matemática de que un sistema pueda ser inteligente. En 1950, Alan Turing publicó su artículo *Computing Machinery and Intelligence* (Turing, 1950), en el que propuso un test para determinar si una máquina puede pensar. Este test, conocido como *test de Turing* (véase Figura 2.2), tiene por objetivo determinar si una máquina puede exhibir un comportamiento inteligente indistinguible del de un ser humano. Su método consiste en la interacción de un humano-evaluador con una entidad artificial únicamente por medio de un terminal de texto como interfaz. La máquina pasará el test si el evaluador no puede discernir si está interactuando con una entidad artificial o humana. A pesar de sus limitaciones y su enfoque antropocéntrico sobre la inteligencia, este test sigue siendo una referencia común para evaluar sistemas modernos de IA. Estudios recientes, sin embargo, muestran que no es únicamente la inteligencia cognitiva aquello que un humano evalúa en el test, sino que existen otros factores como el estilo lingüístico y los rasgos emocionales (Jones y Bergen, 2023), o la creatividad y la originalidad (Noever y Ciolino, 2022).

Figura 2.2 Test de Turing. El humano-evaluador interactúa con un interlocutor desconocido a través de un terminal de texto. Si este no puede discernir si está interactuando con una entidad artificial o humana, la máquina pasa el test.



Fuente: *El Test de Turing* (s.f.)

No obstante, el concepto de IA trasciende a la mera imitación de lo humano. Si consideramos la racionalidad como un conjunto de estructuras lógicas que incluyen el pensamiento y el entendimiento humanos, la IA no necesita limitar su objetivo a superar el test de Turing. Las definiciones de IA a lo largo del tiempo han variado dependiendo de si se enfocan en imitar el pensamiento o acción humanos o el pensamiento y acción racional (Russell, 2021).

A Turing debemos también el concepto de *máquina universal*, que es la base de la computación moderna y de la inteligencia artificial computacional. Sobre este asunto debate sobradamente Roger Penrose en Penrose (2015). En esta misma línea, investigaciones en *procesamiento del lenguaje natural*, en las que destaca el concepto pionero de *entropía* de Claude E. Shannon

(Shannon, 1951) y que han acompañado el desarrollo de lenguajes de programación, son fundamentales para los sistemas actuales de IA. La entropía de Shannon mide cuán probable es cualquier combinación de letras en un idioma concreto. Sus estudios en esta área han resultado fundamentales para la teorización de los *modelos de lenguaje*, o *language model* (LM), que son la base de los sistemas de IA actuales.

2.1.2 *Machine learning*

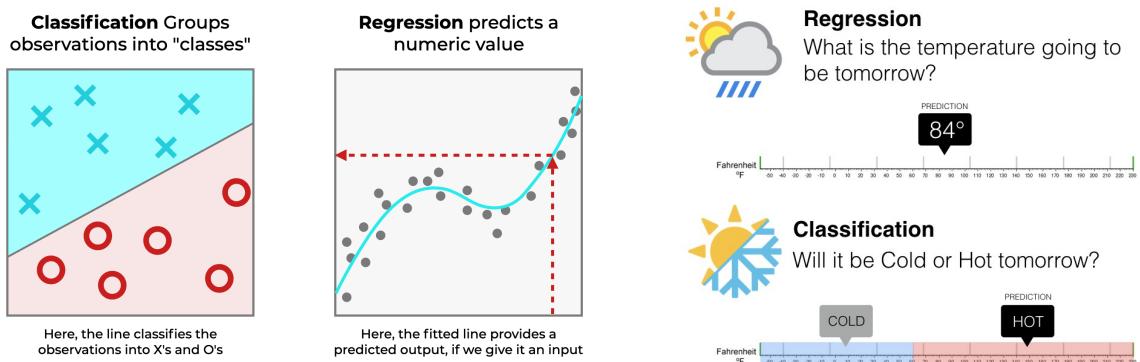
El *machine learning*, o *aprendizaje automático*, es una rama de la IA que investiga algoritmos y modelos matemáticos que habilitan a un sistema computacional a aprender a partir de datos sin ser explícitamente programado. Estos sistemas identifican patrones a partir de un conjunto de datos de entrenamiento, con poca o ninguna intervención humana, para después, en la fase de inferencia, predecir o clasificar datos a los que no tuvo acceso en la fase de entrenamiento (Gollapudi, 2016). En el ML, el sistema se autoconfigura basándose en los datos con los que se entrena. Las únicas intervenciones humanas en el proceso son el diseño de la arquitectura y la provisión de los datos de entrenamiento, aunque incluso estas tareas podrían delegarse a otro sistema de ML en determinadas circunstancias¹.

Aunque el ML ha sido una área de interés desde los inicios de la computación, es esencial reconocer que no todos los sistemas inteligentes utilizan ML. Como ejemplo, el software de ajedrez *Deep Blue* de IBM, que venció al campeón mundial Garry Kasparov en 1997, no se basaba en ML (Campbell, Hoane, y Hsu, 2002). En lugar de ello, Deep Blue utilizaba una vasta base de datos de jugadas de ajedrez y algoritmos de búsqueda heurísticos para decidir el mejor movimiento en cada situación. Otros sistemas no basados en ML, como los *sistemas expertos*, utilizan reglas predefinidas para tomar decisiones y se aplican ampliamente en áreas tan diversas como medicina, ingeniería o gestión empresarial.

Se utiliza el término *modelo* para referirse al sistema una vez que ha sido entrenado y posee capacidad predictiva. Dependiendo de su aplicación, un modelo puede ser empleado para predecir datos desconocidos o clasificarlos. Si produce un valor numérico, se habla de *regresión*; si el modelo categoriza datos, de *clasificación* (Zhang, Lipton, Li, y Smola, 2023). La clasificación es binaria cuando el modelo solo devuelve dos posibles respuestas, o multiclase cuando clasifica en más de dos categorías. La Figura 2.3 muestra de forma intuitiva la diferencia entre clasificación y regresión.

¹P. ej., muchos de los *datasets* utilizados en el entrenamiento de sistemas de IA actuales han sido sintetizados por medio de otros sistemas previos en producción (Z. Li, Zhu, Lu, y Yin, 2023), lo cual reduce considerablemente sus costes de entrenamiento.

Figura 2.3 Problema de clasificación versus regresión. Los modelos de machine learning aprenden de forma automática a categorizar datos nuevos (clasificación) o a predecir nuevos datos de una serie (regresión).



(a) Representación gráfica de un modelo de clasificación y otro de regresión. La clasificación consiste en categorizar un conjunto de datos en dos o más clases. La regresión permite predecir el siguiente dato a partir de una serie temporal o espacial.

(b) Diagrama intuitivo para comprender la regresión y la clasificación.

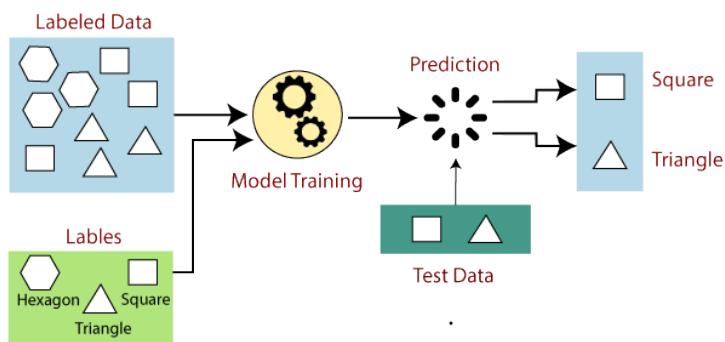
Fuente: *Regression vs. Classification* (2021)

El aprendizaje de sistemas de ML se produce a través de un proceso de *entrenamiento*. Según la naturaleza de los datos y el método de validación, existen tres tipos principales de aprendizaje automático (Torres i Viñals, 2020, p. 38):

1. *Aprendizaje supervisado*: Aquí, los datos se etiquetan con la respuesta esperada, como, p. ej., imágenes de animales etiquetadas como *gato* o *perro*. Tras el entrenamiento, se espera que el sistema clasifique imágenes no etiquetadas en la categoría correcta. Este procedimiento de aprendizaje e inferencia queda reflejado en la Figura 2.4. Una de las aplicaciones más comunes del aprendizaje supervisado es la clasificación de imágenes. Sin embargo, su debilidad consiste precisamente en la necesidad de etiquetado de los datos de entrenamiento, que puede ser un proceso costoso y laborioso si este solo se puede hacer manualmente.
2. *Aprendizaje no supervisado*: En este caso, los datos no están etiquetados. El sistema busca patrones y agrupa datos en categorías por sí mismo. Precisamente este es el tipo de aprendizaje utilizado en el entrenamiento de LM, donde los datos de entrenamiento son textos sin etiquetar (Radford y cols., 2019).
3. *Aprendizaje por refuerzo*: El sistema aprende interactuando con un entorno. No recibe etiquetas explícitas, sino recompensas por decisiones correctas y penalizaciones por errores. Este es el modo de aprendizaje más parecido al humano, ya que se basa en la ex-

periencia. Sin embargo, a pesar de ser uno de los métodos más atractivos de aprendizaje automático, es el más complejo de implementar y requiere un entorno de simulación adecuado, que no siempre es posible ni eficiente en términos de computación (Mohan, Zhang, y Lindauer, 2023).

Figura 2.4 Esquema de aprendizaje supervisado, usando datos de entrenamiento etiquetados. En la inferencia se espera que el modelo pueda clasificar datos no etiquetados en las categorías en las que fue entrenado.



Fuente: *Find ways to deal with the scarcity of labeled data* (s.f.)

2.1.3 Deep learning

Esta sección pretende ser una somera introducción a los conceptos de *y red neuronal artificial* (RNA), que constituyen la base de los LM, objeto de nuestro estudio, con la única finalidad de aportar un aparato teórico mínimo en el que contextualizarlo. En los últimos meses la bibliografía sobre este tema está creciendo exponencialmente². El DL es una rama del ML que utiliza RNA para aprender de forma automática. Las RNA son modelos matemáticos cuyos principios imitan el funcionamiento de las neuronas biológicas y hunden sus raíces en la intersección entre biología, matemáticas y ciencias de la computación ya en la primera mitad del siglo XX.

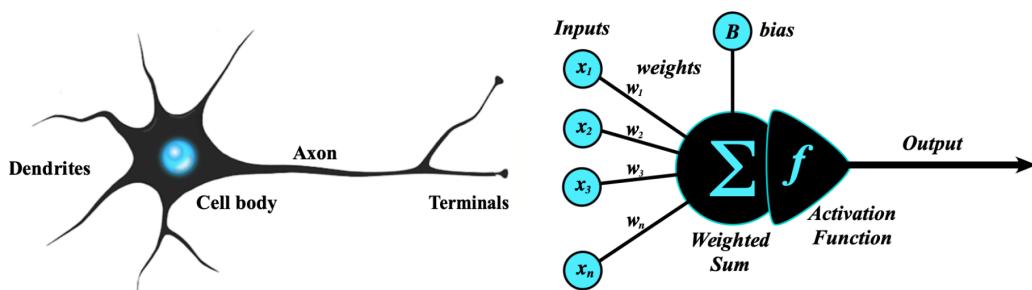
2.1.3.1 El concepto de *neurona artificial*

Para entender en qué consiste una RNA y, por extensión, un modelo de DL, antes es necesario comprender el concepto de neurona artificial. Una neurona artificial no es sino un modelo matemático simplificado de una neurona natural que encontramos en el sistema nervioso de los animales. Análogamente a su homóloga biológica, una neurona artificial recibe una serie de valores de entrada, los procesa y devuelve una salida. La entrada de una neurona artificial es la suma ponderada de las salidas de las neuronas de la capa anterior. Esta suma ponderada

²Además de las obras citadas en este trabajo, se puede consultar, a modo de introducción para no especialistas, *A Beginner's Guide to Neural Networks and Deep Learning* (s.f.).

se pasa a una función de activación, que determina la salida de la neurona. Sin necesidad de entrar en detalles matemáticos, la función de activación proporciona a la neurona y los sistemas de RNA una no linealidad, imprescindible para el comportamiento complejo de estos sistemas (Torres i Viñals, 2020; Zhang y cols., 2023). La Figura 2.5 muestra un esquema de una neurona artificial.

Figura 2.5 Neurona biológica y neurona artificial



Fuente: Deep Learning (DL) (s.f.)

El concepto de neurona artificial fue expuesto por primera vez por F. Rosenblatt en 1958 (Rothman, 2021) bajo el nombre de *perceptrón*. El perceptrón no es otra cosa que un modelo de clasificación binaria, es decir, que solo puede clasificar datos en dos categorías. Rosenblatt planteó su modelo para ser implementado en hardware, y, de hecho, se construyó un prototipo en 1959. Sin embargo, el perceptrón tenía ciertas limitaciones matemáticas que no permitían su aplicación a problemas más complejos. En 1969, Minsky y Papert publicaron el libro *Perceptrons: An introduction to computational geometry* (Minsky y Papert, 1969), en el que demostraban que el perceptrón no podía resolver problemas linealmente no separables, lo cual supuso un freno en la investigación en RNA³.

Actualmente sabemos que una red neuronal de varias capas con una función de activación no lineal puede aproximar arbitrariamente cualquier función matemática, lo que ha venido a llamarse *teorema de la aproximación universal*, propuesto en 1989 por Hornik, Stinchcombe, y White en su artículo *Multilayer Feedforward Networks Are Universal Approximators* (Hornik y

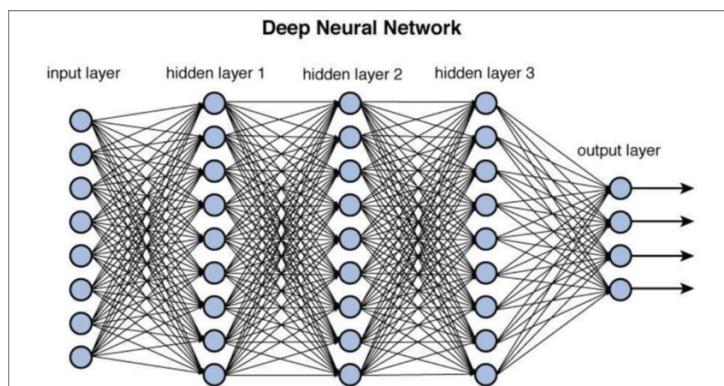
³Este freno en la investigación durante años, se ha denominado *invierno de la inteligencia artificial* ("Invierno IA", 2023), y no se retomaría hasta la década de los 80, cuando nuevos desarrollos en modelos de redes neuronales artificiales permitieron resolver problemas más complejos que un simple perceptrón.

cols., 1989). Este teorema tiene grandes implicaciones, no solo científicas, sino también filosóficas, en la medida en que podemos considerar que una RNA lo suficientemente compleja es capaz de imitar el razonamiento humano, susceptible de reducirse al de una función matemática por compleja que esta sea. En la literatura científico-filosófica del s. XX encontramos interesantes reflexiones sobre las implicaciones de que nuestro cerebro fuera un algoritmo matemático que genera pensamiento, como en la ya citada obra de Penrose (2015) o en Searle (1985).

2.1.3.2 Redes neuronales artificiales

Una RNA es un modelo matemático que se compone de varias capas de neuronas artificiales. La primera capa se denomina *capa de entrada* y recibe los datos de entrada. La última capa se denomina *capa de salida* y devuelve la predicción del modelo. Las capas intermedias se denominan *capas ocultas* y son las que permiten que el modelo pueda aproximar cualquier función matemática. La Figura 2.6 muestra un esquema de una RNA con una capa de entrada, dos capas ocultas y una capa de salida.

Figura 2.6 Estructura de capas de una posible red neuronal artificial, con una capa de entrada, tres capas ocultas y una capa de salida.



Fuente: Zrara (2020)

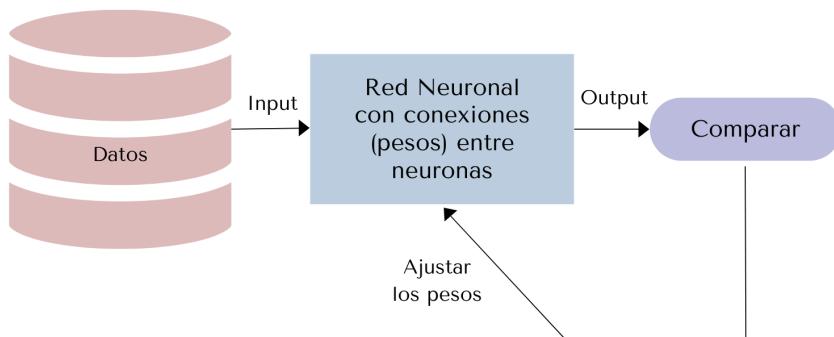
En función de los objetivos de la RNA, su arquitectura o estructura interna puede variar considerablemente. Una RNA puede constar de una sola capa, de una capa de entrada y otra de salida y de múltiples capas ocultas⁴. Cada arquitectura busca resolver un tipo de problema concreto. Una arquitectura especialmente compleja y que ha dado muy buenos resultados en la visión artificial es la *red neuronal convolucional* (CNN) (O'Shea y Nash, 2015), inspirada en el

⁴Precisamente de esta característica de *profundidad* que puede llegar a tener una RNA con varias capas ocultas, toma el calificativo de *profundo* o *deep* el aprendizaje *profundo* o *deep learning*.

funcionamiento y estructura de la corteza visual del cerebro humano. Las CNN se utilizan para el reconocimiento de imágenes y vídeos, y han sido la base de los avances en el campo de la visión artificial en los últimos años.

El entrenamiento de una RNA o modelo de DL implica ajustar sus parámetros internos, principalmente los *pesos* de las conexiones neuronales (véanse las variables w en la Figura 2.5), para que la salida generada se aproxime a la esperada, basándose en los datos de entrenamiento (Nur, Radzi, y Osman, 2014). Este proceso iterativo, ilustrado en la Figura 2.7, tiene como objetivo minimizar el margen de error entre la salida de la red y su valor esperado, hasta alcanzar un nivel aceptable. Una vez entrenada, la RNA puede realizar predicciones o clasificaciones. Inicialmente, un modelo no entrenado produce salidas aleatorias, pero a medida que se ajustan los pesos, la salida se vuelve más precisa. Hay una relación directa entre la cantidad de parámetros de un modelo y su capacidad predictiva, lo que implica también un aumento en el tiempo de entrenamiento y los recursos computacionales necesarios. Este aspecto ha determinado en parte los avances recientes que permiten a las RNA lograr resultados comparables a los del cerebro humano en tareas de generación de imágenes, video, audio o texto.

Figura 2.7 Diagrama de entrenamiento de una red neuronal artificial.



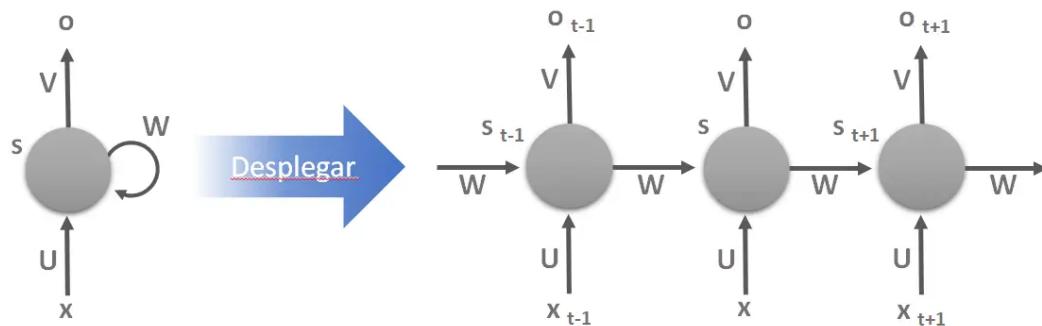
Fuente: Elaboración propia

2.1.3.3 Redes neuronales recurrentes

Hasta 2017, el estado del arte en el ámbito del procesamiento de lenguaje natural por medio de modelos de DL eran las *redes neuronales recurrentes* o *recurrent neural network* (RNN) (Schmidt, 2019). Estas se basaban en la idea de que la salida de una neurona se podía retroali-

mentar a la entrada, de forma que la salida de la neurona en el instante t se podía usar como entrada en el instante $t + 1$, lo cual permitía inputs de secuencias donde el elemento temporal es importante, como las temperaturas de un lugar durante un año o una oración lingüística. Esta arquitectura se ilustra en la Figura 2.8. Sin embargo, las RNN presentaban un problema conocido como *vanishing gradient* (Pascanu, Mikolov, y Bengio, 2013), que hacía que el modelo no pudiera retener contextos de secuencias largas, ya que su atención se desvanece en los tokens más alejados en el tiempo. Este problema se solucionó en parte con las redes neuronales de memoria a corto y largo plazo *long short-term memory* (LSTM) (Hochreiter, 1998), que permitían que la información fluyera a través de la red sin perderse. Sin embargo, las LSTM no podían trabajar con grandes cantidades de tokens de contexto, lo que limitaba su aplicación a tareas de procesamiento de lenguaje natural.

Figura 2.8 Arquitectura de una red neuronal recurrente.



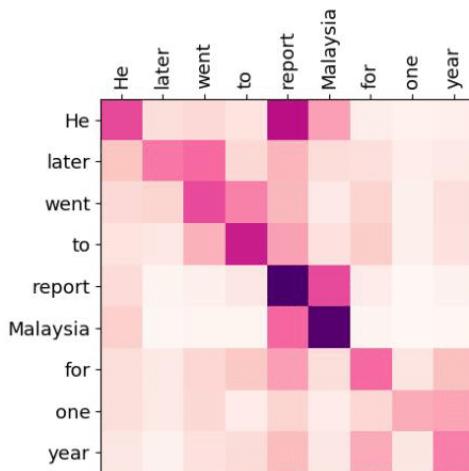
Fuente: Calvo (2018)

2.1.3.4 La arquitectura *transformer*

En junio de 2017, Vaswani y cols. publicaron su artículo *Attention Is All You Need* (Vaswani y cols., 2017) en el que proponían una nueva arquitectura de RNA, denominada *transformer*, que no se basaba en las RNN y que permitía trabajar con grandes cantidades de tokens de contexto. Esta arquitectura, que marcaría un antes y un después en los modelos de DL se basaba en el concepto de *atención*, que permite que el modelo pueda predecir nuevos tokens teniendo en cuenta todo el contexto de la cadena previa. En esta arquitectura, no importa la lejanía en la posición de los tokens de la secuencia de entrada, ya que el modelo asigna pesos a cada token en función de su relevancia respecto al resto de tokens para la predicción del siguiente. A diferencia de una RNN, un transformer puede procesar todos los tokens de una secuencia de entrada en paralelo, lo que lo hace mucho más eficiente computacionalmente tanto en

el entrenamiento como en la inferencia. La Figura 2.9 muestra una matriz de atención de un transformer entrenado con textos en inglés.

Figura 2.9 Matriz de atención de un transformer entrenado con textos en inglés. En ella queda representada la atención, o nivel de relación semántica de cada token dentro de la secuencia de entrada «He later went to report Malaysia for one year» con respecto a los demás.



Fuente: Du y Wang (2022)

Esta capacidad de generar eficientemente contenido original prestando atención a todos los tokens de un gran contexto, propia de los modelos entrenados con la arquitectura transformer, es lo que ha dado lugar a la explosiva evolución de la IAG. La IAG abarca tanto a la generación de texto como de música, imagen, vídeo y cualquier forma de información, por compleja que esta sea, susceptible de ser reducida a modelos estadísticos.

2.2 Modelos de lenguaje (LM)

Un LM es un modelo probabilístico que asigna una probabilidad a una secuencia de palabras (“Modelación del lenguaje”, 2024). Esencialmente, es una función matemática capaz de simular la forma en que se escribe en lenguaje natural⁵.

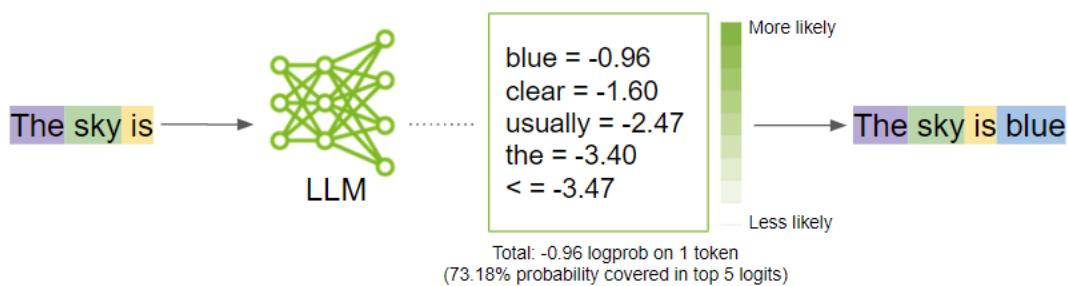
Desde una perspectiva técnica, un LM devuelve como salida la distribución de probabilidad del siguiente token, dada una secuencia de tokens como entrada (*Generation with LLMs*, s.f.). Un token es la unidad mínima de información que el modelo procesa y, generalmente, equivale a una palabra⁶. La Figura 2.10 ilustra un modelo de lenguaje que recibe una secuencia de palabras

⁵Una analogía común para entender el funcionamiento de un LM es la función predictiva de un teclado. Mientras escribimos en nuestro dispositivo móvil, el teclado nos sugiere palabras que probablemente seguirán a las ingresadas. Esta capacidad de predicción es el fundamento de los LM, incluyendo chatbots avanzados.

⁶Aunque un token suele ser una palabra, también puede ser un signo de puntuación, número, una desinencia, etc.

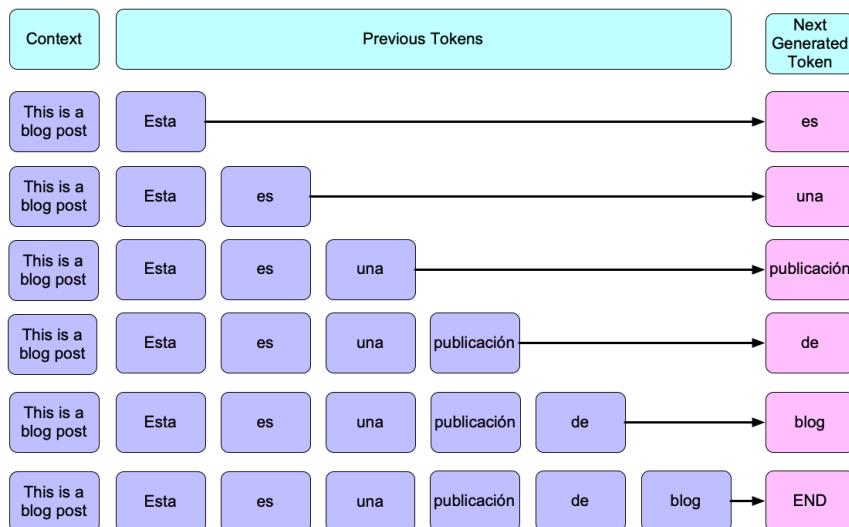
y devuelve la distribución de probabilidad del siguiente token después de haber sido entrenado con textos en inglés. Para generar cadenas de tokens, para formar, por ejemplo, oraciones completas, se vuelve a pasar al modelo el contexto inicial más el último token generado para producir el siguiente, y así sucesivamente hasta llegar al final del texto. Este modo de generación de texto se denomina *autorregresivo* (Malach, 2023). Un ejemplo de generación de una oración completa por un LM de forma autorregresiva se muestra en la Figura 2.11.

Figura 2.10 Inferencia de token de un LLM



Fuente: *How to Get Better Outputs from Your Large Language Model* (2023)

Figura 2.11 Generación de un texto completo por medio de la autorregresión por un LLM. Cada fila del diagrama representa una iteración en el tiempo con el modelo. Este recibe en cada paso el prompt inicial más la cadena de tokens actual y predice el siguiente token. Este proceso se repite hasta alcanzar el token «END».



Fuente: *An Understanding of Learning from Demonstrations for Neural Text Generation · The ICLR Blog Track* (s.f.)

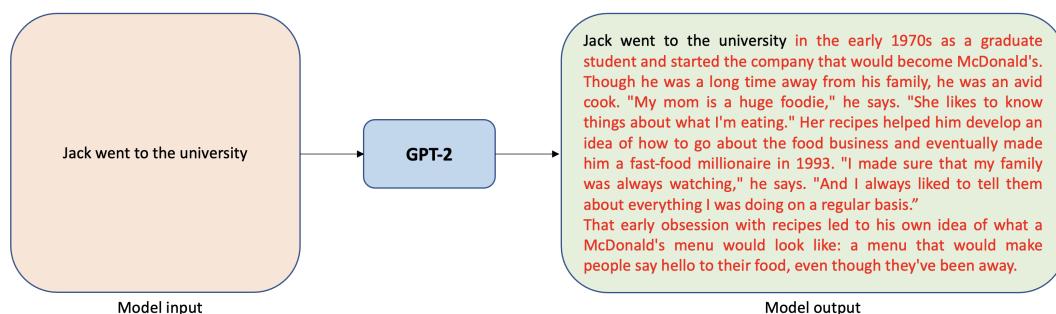
Los LM no se asocian exclusivamente a una arquitectura de ML. Pueden implementarse mediante diferentes tipos de redes neuronales, como las redes neuronales recurrentes o con-

volucionales. No obstante, el hito que ha propulsado avances significativos en ML ha sido la arquitectura *transformer* (Vaswani y cols., 2017), de la que se ha hablado más arriba.

2.2.1 Grandes modelos de lenguaje (LLM)

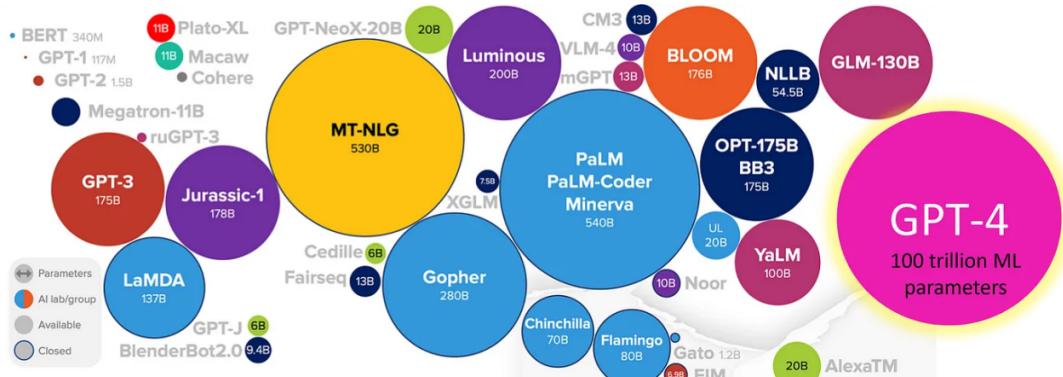
Un LLM posee un número de parámetros del orden del billón, lo cual es considerado «grande» o *large* desde el punto de vista computacional. El primer LLM fue *GPT-2*, creado y entrenado por OpenAI en 2019 (Radford y cols., 2019). *GPT-2* se entrenó con texto de Internet y alcanzó 1.5 billones de parámetros. Su capacidad para predecir la siguiente palabra en una secuencia sorprendió a la comunidad científica debido a la calidad de los textos generados. La Figura 2.12 muestra ejemplos de textos generados por este modelo.

Figura 2.12 Generación de textos por GPT-2



Fuente: *Run Text Generation with Bloom and GPT Models on Amazon SageMaker JumpStart | AWS Machine Learning Blog* (2022)

Los LLM emplean la arquitectura *transformer* o derivados. Se entranan con grandes cantidades de texto sin etiquetar, como libros, artículos de periódicos, páginas web, etc., realizándose este proceso en paralelo, lo que requiere una gran capacidad computacional. Sin embargo, una vez entrenados, estos modelos pueden ser utilizados para tareas de generación de texto, traducción automática, resumen de textos, etc. con una capacidad predictiva sorprendente. La Figura 2.13 muestra una comparativa de los tamaños de los LLM más conocidos.

Figura 2.13 Gráfico comparativo de tamaños de LLM

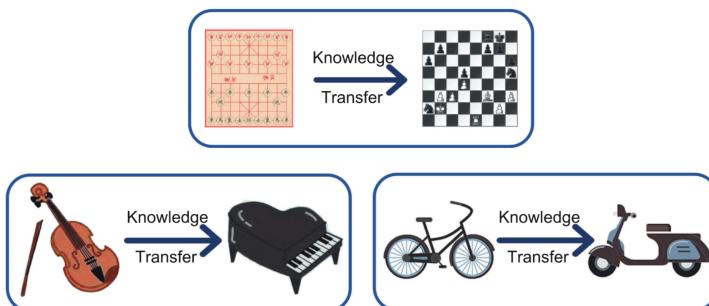
Fuente: *The Challenges Associated With Building Products Using Large Language Models (LLMs)*. (s.f.)

2.2.2 Modelos preentrenados y *fine-tuning*

Un LLM es entrenado desde cero con un gran corpus de textos de toda índole, normalmente tomados de internet: libros, blogs, artículos de periódicos, artículos científicos, *wikis*, etc. El modelo, tras ese proceso, queda *preentrenado* (Han y cols., 2021) y puede ser utilizado, de forma genérica, para tareas de generación de texto, traducción automática, resumen de textos, etc. Sin embargo, es posible *ajustar* (*fine-tuning*) el modelo para que se adapte a un dominio específico, como la música, la programación, la medicina, etc., o a una tarea específica: dialogar en forma de chatbot, ser amable, comportarse como un personaje concreto, etc. En este caso, el modelo se vuelve a entrenar con un corpus de textos específico del dominio, o con un conjunto de preguntas y respuestas con el estilo buscado, en el caso de ser un chatbot. Todo ello permite que el modelo se adapte a ese dominio, mejore su capacidad predictiva o se comporte de la forma deseada (Tian, Mitchell, Yao, Manning, y Finn, 2023). Este proceso de *fine-tuning* es más rápido y requiere menos datos que el entrenamiento desde cero, por lo que es una opción muy interesante para adaptar los LLM a dominios específicos.

El *fine-tuning* produce buenos resultados gracias a la capacidad de generalización de los *LLM* y de transformar el conocimiento adquirido (en el preentrenamiento) de un dominio a otro. Por ejemplo, un modelo entrenado con textos de biología puede ser ajustado para que genere textos de medicina, o viceversa. El mecanismo por el que un aprendizaje se transforma en otro se denomina *transfer learning* (Zhuang y cols., 2020) y es una de las características más interesantes de los LLM (véase Figura 2.14).

Figura 2.14 Ejemplos intuitivos de transfer learning en LLM. La capacidad de generalización de los LLM permite trasladar conocimientos y destrezas de un dominio a otro, especialmente si existe una relación o analogía entre ellos.



Fuente: Zhuang y cols. (2020)

2.2.3 Hiperparámetros y parámetros del modelo

Se denominan *hiperparámetros* a los parámetros que se fijan de forma previa al entrenamiento del modelo, delimitando su estructura y capacidades computacionales (*¿Qué es el ajuste de hiperparámetros?*, s.f.). Así, la arquitectura elegida, las dimensiones de cada una de sus partes, la tasa de aprendizaje, etc. Todo ello determinará las características del modelo entrenado, así como las necesidades de recursos computacionales necesarios tanto para la fase de entrenamiento como de inferencia.

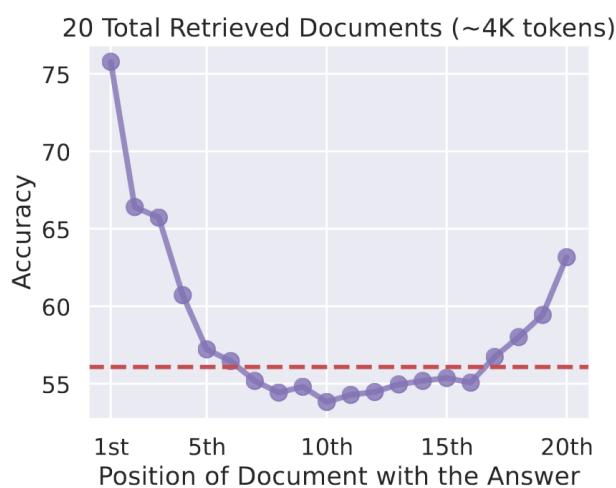
Cuanto mayor sea el modelo, más parámetros variables tendrá en su entrenamiento, por lo que mayor coste computacional demandará. Actualmente, los LLM tienen un número de parámetros absolutamente prohibitivo dentro de la computación personal, del orden del billón (Radford y cols., 2019), por lo que su entrenamiento se realiza en *clusters* de ordenadores de alto rendimiento, con cientos de GPUs. Sin embargo, una vez entrenados, son potencialmente utilizables en ordenadores personales⁷ para tareas de inferencia, como la generación de texto, traducción automática, etc., aunque lo más habitual es ofrecer sus servicios online a demanda.

Además del número de parámetros, el tamaño de la ventana de contexto es otro hiperparámetro a considerar en un LLM. Este dato hace referencia a la cantidad de tokens que un LLM puede recibir como input para realizar la inferencia del siguiente token. Cuanto mayor sea la ventana de contexto, más tokens tendrá en cuenta el modelo para generar el siguiente, y más coherentes serán los textos generados. Sin embargo, también será más lento en la inferencia (Gonzalo, Carabantes, y González Geraldo, 2023).

⁷En realidad, también para la fase de inferencia se necesita del uso de GPUs, pero podemos considerarlo un recurso asequible comparado con los requerimientos de la fase de entrenamiento.

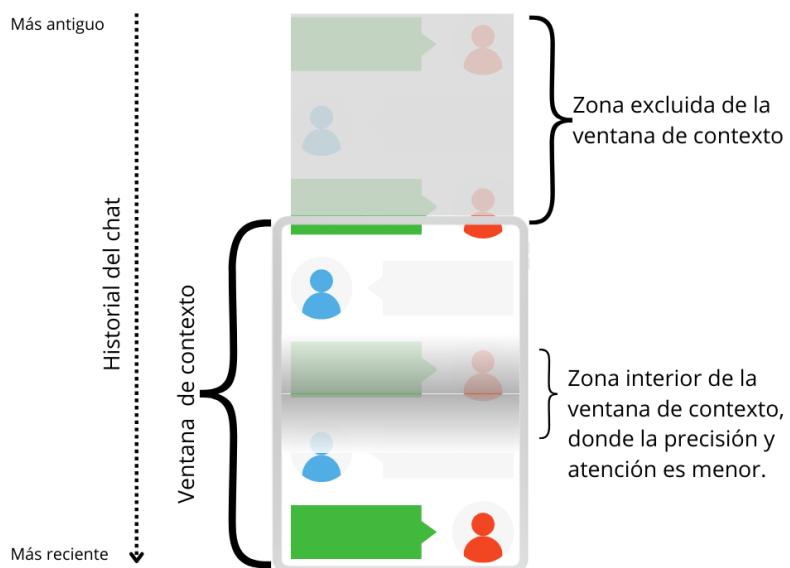
Por otra parte, independientemente del tamaño ventana de contexto, y especialmente en las ventanas grandes, se ha visto que los LLM no procesan por igual toda la información. Pueden existir lagunas importantes en el centro de la ventana, tal como demuestran recientes estudios (N. F. Liu y cols., 2023), lo cual puede llevar al modelo a alucinar. La Figura 2.15 muestra una gráfica con este fenómeno. Por tanto, en una conversación larga dentro de la creación de un proyecto de envergadura el LLM puede sufrir de pérdida absoluta de la parte inicial de la conversación (limitación de la ventana de contexto) y, por otra parte, de pérdida parcial de la parte central de la conversación (limitación de la precisión en el centro de la ventana de contexto). Véase la Figura 2.16, que ilustra gráficamente el problema combinado de conocimiento del contexto por parte de los LLM en conversaciones suficientemente largas.

Figura 2.15 *Precisión de GPT-3 en función del tamaño de la ventana de contexto. En la gráfica se aprecia cómo la precisión del modelo disminuye hacia el centro de la ventana de contexto. Esto puede ser una limitación en la creación de un proyecto, ya que la parte central de la conversación recordada por el modelo puede resultar minimizada en su importancia.*



Fuente: N. F. Liu y cols. (2023)

Figura 2.16 Ventana de contexto y pérdida de precisión en su interior. En un proyecto de medianas y grandes dimensiones, chatGPT puede perder la memoria de las directrices iniciales (limitación de la ventana de contexto) y perder precisión o alucinar en el centro de la ventana de contexto (limitación de la precisión en el centro de la ventana de contexto). Las partes sombreadas indican una pérdida total (fuera de la ventana de contexto) o parcial (hacia la mitad de la ventana de contexto) de la atención por parte del LLM.



Fuente: Elaboración propia

2.2.4 Temperatura, top-p y top-k

Consideremos la Figura 2.10. Un LLM predice su siguiente token devolviendo una distribución de probabilidad. Si el sistema tomara por salida siempre el token más probable, las respuestas del LLM serían absolutamente predecibles y repetitivas. Para evitarlo, el parámetro de *temperatura* permite controlar el peso que se dará a cada elemento de la distribución de probabilidad. Si la temperatura es baja, se dará más peso a los elementos más probables, y si es alta, se igualará la probabilidad de todos los elementos. En el primer caso, las respuestas serán más predecibles, y en el segundo, más variadas y creativas. Si la temperatura es demasiado alta, el modelo puede generar respuestas incoherentes. La Figura 2.17 ilustra este procedimiento.

Figura 2.17 Control de la temperatura en la generación de texto por un LLM. La temperatura controla el peso que se da a cada elemento de la distribución de probabilidad. Si la temperatura es baja, se dará más peso a los elementos más probables, y si es alta, se igualará la probabilidad de todos los elementos. En el primer caso, las respuestas serán más predecibles, y en el segundo, más variadas y creativas. Si la temperatura es demasiado alta, el modelo puede generar respuestas incoherentes.



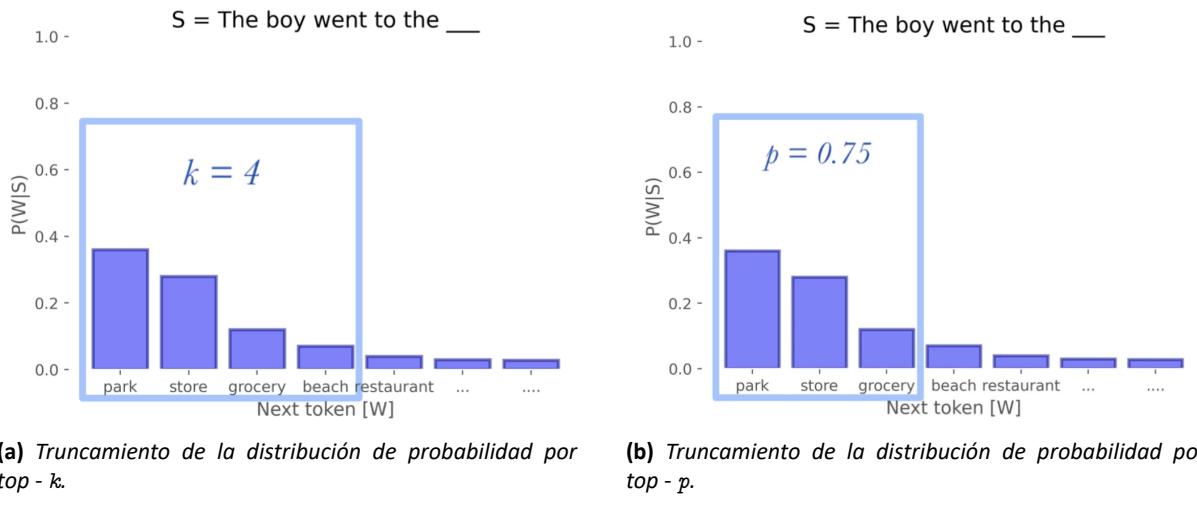
Fuente: *Top-k, Top-p, Temperature* (s.f.)

Por otra parte, los parámetros top-k y top-p permiten controlar cuántos elementos de la distribución de probabilidad se tendrán en cuenta en la generación del siguiente token. El parámetro top-k indica el número de elementos más probables, mientras que top-p indica el porcentaje de probabilidad que se tendrá en cuenta. En ambos casos, la distribución queda truncada a los elementos más probables. La Figura 2.18 ilustra el funcionamiento de estos parámetros. En todo caso, la elección de estos valores tiene un gran impacto en las respuestas del LLM (Chamand, Risser-Marroix, Kurtz, Joly, y Loménie, 2022; Holtzman, Buys, Du, Forbes, y Choi, 2020; C. Wang, Liu, y Awadallah, 2023; S. Wang, Ma, Xu, Wang, y Wu, 2022), y en la mayoría de los casos, se requiere un proceso de prueba y error para encontrar los valores óptimos.

2.3 Prompting engineering

El concepto de *prompting engineering* se refiere al estudio de diferentes métodos mediante los cuales se solicita a un modelo generativo que produzca una respuesta (*LLM Prompting Guide*, s.f.). Específicamente, en el contexto de un LLM, esto implica generar texto basado en un *prompt* o entrada de texto en lenguaje natural proporcionada por el usuario. Actualmente, el

Figura 2.18 Control de los parámetros *top-k* y *top-p* en la generación de texto por un LLM. (a) El parámetro *top-k* indica el número de elementos más probables, mientras que (b) *top-p* indica el porcentaje de probabilidad que se tendrá en cuenta. En ambos casos, la distribución queda truncada a los elementos más probables.



Fuente: Stokes (2023)

prompting representa un área de investigación dinámica que ha originado una amplia variedad de estudios y publicaciones⁸.

2.3.1 Técnicas más importantes de *prompting engineering*

En función del propósito de la interacción con sistemas basados en LLM, existen diferentes técnicas prompting (véase Figura 2.22). La forma más sencilla de *prompting* es la denominada *zero-shot*, en la que el usuario presenta al LLM un *prompt* y el LLM genera un texto de salida. En este caso, el LLM no ha sido entrenado para realizar una tarea específica, sino que ha sido entrenado con textos en lenguaje natural de propósito general. Sin embargo, el LLM puede generar un texto de salida que se corresponda con la tarea que el usuario quiere realizar. Por ejemplo, si el usuario presenta al LLM el prompt «¿Cuál es la capital de Francia», el LLM puede generar un texto de salida como «La capital de Francia es París». En este caso, el LLM no ha sido entrenado como experto en geografía, pero ha sido entrenado con textos en lenguaje natural de propósito general, y es capaz de generar un texto de salida que se corresponde con la petición del usuario. La limitación de esta técnica es la del propio entrenamiento del LLM.

Este no podrá generar nada en lo que no haya sido previamente entrenado.

⁸Sin embargo, el término *prompting engineering* es objeto de debate y no cuenta con un consenso universal dentro de la comunidad científica. A pesar de la controversia y de que algunos cuestionan su validez, existen numerosas investigaciones científicas que emplean esta terminología.

2.3.1.1 Few-shot prompting

Una primera técnica aplicable a un modelo de lenguaje es la de *few-shot*, en la que el modelo recibe como input una serie de pares «petición-respuesta» más una petición a la cual ha de responder de forma análoga al contexto. Se presenta muy útil para tareas que requieren de cierto automatismo, como la generación de respuestas con un formato concreto, ya que en el input se muestra al LLM qué tipo de respuesta se requiere (véase Figura 2.19). De hecho, esta es la técnica que subyace tras las interfaces tipo *chat* que se pueden encontrar en la mayoría de los LLM conversacionales. Si bien, fueron preentrenados para completar textos, pueden ser guiados por el *few-shot prompting* a generar respuestas en un formato y personalidad concreta, ya que el modelo tenderá a repetir el estilo de los ejemplos pasados por contexto.

Figura 2.19 Técnica de few-shot prompting. El LLM recibe como input una serie de pares petición-respuesta más una petición a la cual ha de responder de forma análoga al contexto. En este caso, el LLM, con papel de asistente, ha de devolver el input del usuario con las letras invertidas. Sin embargo, nunca se le pide explícitamente que invierta las letras, sino que lo ha de inferir a partir de los ejemplos.

| |
|------------------|
| Usuario: Casa |
| Asistente: Asac |
| Usuario: Ciclo |
| Asistente: Oleic |
| Usuario: Ropa |
| Asistente: Apor |

Fuente: Elaboración propia

2.3.1.2 Chain of thoughts

Una técnica muy interesante es la denominada *chain of thoughts* (COT), que consiste en presentar al LLM un problema y pedirle que explique su razonamiento antes de dar la respuesta. Esta técnica se ha mostrado muy efectiva en especial en tareas de resolución de problemas aritméticos, ya que obliga al LLM a razonar antes de dar una respuesta, lo que lleva a una respuesta correcta en la mayoría de los casos (Wei y cols., 2023). Esta técnica pone en evidencia el hecho de que los LLM no razonan necesariamente su respuesta, a pesar de que ésta pueda resultar conveniente y pasar por correcta (véase 2.4). Por otra parte, esta técnica ha mostrado que el *prompting* tiene un gran poder sobre el resultado de la inferencia en los modelos de lenguaje.

La Figura 2.20 muestra un ejemplo de COT aplicado a un problema aritmético. Obsérvese cómo el LLM llega a la respuesta correcta sólo cuando se le incita a dar una explicación de su razonamiento. En este caso no se le ha pedido esta explicación explícitamente, sino a través de la técnica de *few-shot prompting*. Las técnicas que se presentan a continuación pueden entenderse como derivaciones del principio de COT, en los cuales se fuerza un razonamiento explícito por parte del LLM antes de dar una respuesta.

Figura 2.20 Chain of thoughts. *En ambos ejemplos se plantea un problema aritmético a GPT-3. En (a), por medio de few-shot prompting, se le pide una respuesta directa al problema. En (b), aplicando, además, la técnica de chain of thoughts, se le pide que explique su razonamiento, lo cual lleva a una respuesta correcta. Se ha resaltado el texto correspondiente a los razonamientos.*

P: Roger tiene 5 pelotas de tenis. Compra 2 latas más de pelotas de tenis. Cada lata tiene 3 pelotas de tenis. ¿Cuántas pelotas de tenis tiene ahora?

R: La respuesta es 11.

P: La cafetería tenía 23 manzanas. Si usaron 20 para hacer el almuerzo y compraron 6 más, ¿cuántas manzanas tienen?

R: La respuesta es 29.

(a) Few-shot para forzar una resolución directa de un problema aritmético. La respuesta es incorrecta.

P: Roger tiene 5 pelotas de tenis. Compra 2 latas más de pelotas de tenis. Cada lata tiene 3 pelotas de tenis. ¿Cuántas pelotas de tenis tiene ahora?

R: Roger comenzó con 5 pelotas. 2 latas de 3 pelotas de tenis cada una hacen 6 pelotas de tenis. $5 + 6 = 11$. La respuesta es 11.

P: La cafetería tenía 23 manzanas. Si usaron 20 para hacer el almuerzo y compraron 6 más, ¿cuántas manzanas tienen?

R: Comenzaron con 23 manzanas. Usaron 20, por lo que quedaron $23 - 20 = 3$ manzanas. Compraron 6 más, por lo que ahora tienen $3 + 6 = 9$ manzanas. La respuesta es 9.

(b) Few-shot con chain of thoughts para provocar un razonamiento previo a la respuesta. La respuesta es correcta.

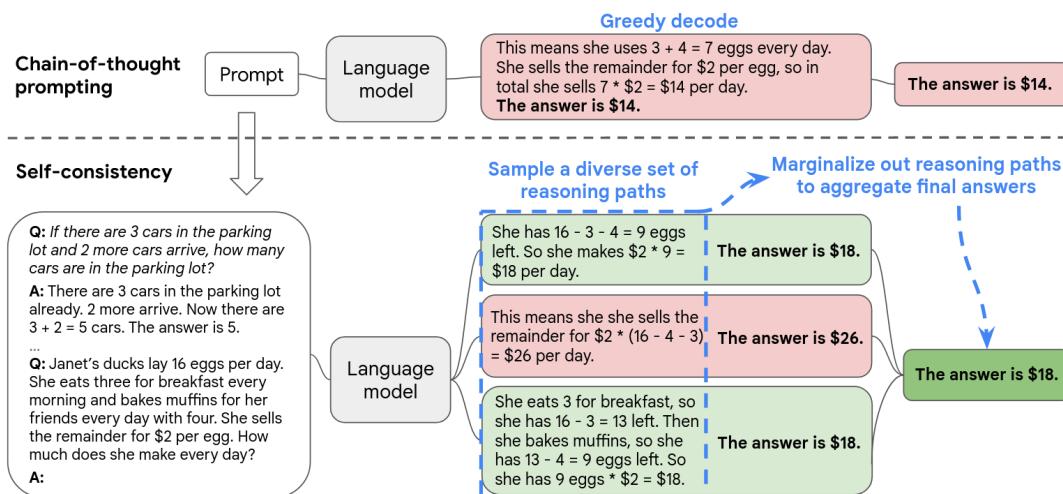
Fuente: Wei y cols. (2023). Traducción propia.

2.3.1.3 Self consistency of chain of thoughts

Esta técnica es una variante de COT que consiste en presentar al LLM un problema y pedirle que explique su razonamiento antes de dar la respuesta, pero en este caso se le pide realizar esta cadena de pensamientos en dos o más ocasiones. Tras ello se verifica qué respuesta es la más consistente. De este modo, no solo se provoca el razonamiento, sino y la posibilidad de llegar a un resultado desde diferentes puntos de vista. (X. Wang y cols., 2023) muestra cómo

esta técnica mejora a la de COT tanto en el terreno aritmético como en el del razonamiento por sentido común. La Figura 2.21 muestra un ejemplo de esta técnica aplicada a un problema aritmético.

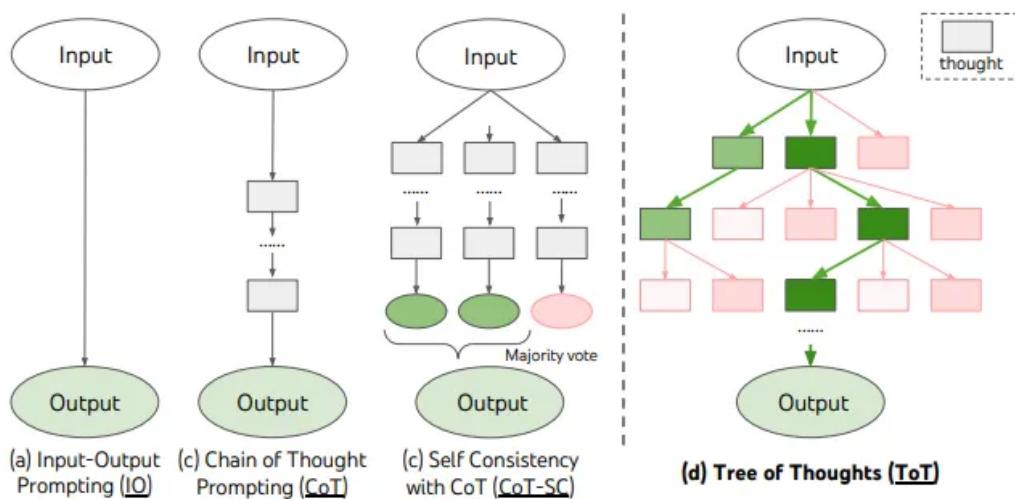
Figura 2.21 Esquema de los pasos en los que se divide la técnica *Self Consistency of Chain of Thoughts*



Fuente: X. Wang y cols. (2023)

2.3.1.4 Tree of thoughts

Esta técnica implementa un árbol de decisión. El LLM hace una búsqueda en profundidad y anchura en un árbol de pensamientos, pudiendo ir hacia atrás en la estructura si no encuentra una respuesta satisfactoria. La Figura 2.22 muestra un esquema de esta técnica comparada con la de COT y *self consistency of chain of thoughts* (COT-SC).

Figura 2.22 Técnicas de prompting engineering aplicadas a problemas aritméticos.

Fuente: Bhavsar (2023)

El buen desempeño de los LLM en tareas de código de programación lleva a estudios como el de J. Li, Li, Li, y Jin (2023), que propone un sistema de prompting que pide al LLM una reflexión explícita sobre la implementación a realizar, en una especie de pseudocódigo, que en un segundo paso será traducido al código final de programación. Este sistema de prompting se basa en la idea de que el LLM generará código de mayor calidad si se le permite *pensar* la respuesta antes de generarla. En este sentido, existen sobradas publicaciones que ponen de relieve la sensibilidad de los LLM a la forma en la que se le pide que procese la información en el prompt y la correlación en la calidad de sus resultados (*LLM Prompting Guide*, s.f.; Wei y cols., 2023; Zhou y cols., 2023).

2.3.2 Retrieval-augmented generation

Los LM están preentrenados con grandes cantidades de texto de propósito general, lo que los hace capaces y hábiles en múltiples campos sin necesidad de un entrenamiento específico (véase Figura 2.23). Es posible, sin embargo, rentrenarlos con datos más concretos y precisos para una tarea específica (por ejemplo, con datos médicos, ingenieriles, con datos privados de una empresa, etc.). A este proceso se le conoce como *fine-tuning* o ajuste fino. Sin embargo, el proceso de *fine-tuning* de un LLM requiere de grandes cantidades de datos y tiempo de entrenamiento, algo que en este momento no está al alcance del usuario medio. Por ello, se están desarrollando técnicas que permitan a los usuarios aprovechar los modelos preentrenados para tareas específicas sin necesidad de *fine-tuning*. Una de estas técnicas es *retrieval-augmented*

generation (RAG) (*What Is Retrieval-Augmented Generation?*, 2021), que podemos traducir por *Generación Mejorada por Recuperación*, que combina la recuperación de información con la generación de texto. Esta es una técnica que ofrece resultados muy notables en términos de eficacia y eficiencia frente al *fine-tuning*. RAG consiste en pasar como contexto al LLM junto a la petición del usuario, una serie de fragmentos de texto que se corresponden semánticamente con la petición, y que se obtienen de una base de datos de documentos relacionados con la tarea a realizar. Esta base de datos puede ser de cualquier tipo, desde una base de datos de texto en lenguaje natural, hasta una base de datos de código de programación, pasando por una base de datos de partituras musicales, o una base de datos de audio, y representan la base de conocimiento extra (*knowledge*) que el LLM manejará en sus interacciones con el usuario. Este sistema de *prompting* está en la base actual de muchos productos relacionados con LLM, como Assistant o GPTs de OpenAI. La Figura 2.24 muestra el esquema de funcionamiento de RAG. La base de datos del conocimiento del LLM es dividido en una primera fase en fragmentos más pequeños y analizados por el LLM para obtener una representación vectorial de cada uno de ellos. En una segunda fase, el LLM recibe el prompt del usuario y lo analiza para obtener una representación vectorial. En una tercera fase, el LLM busca en la base de datos de fragmentos aquellos que tienen similitud vectorial (y, por ende, semántica) con el prompt del usuario. En una cuarta fase, el LLM genera el texto de salida a partir del prompt del usuario y de los fragmentos de texto recuperados de la base de datos.

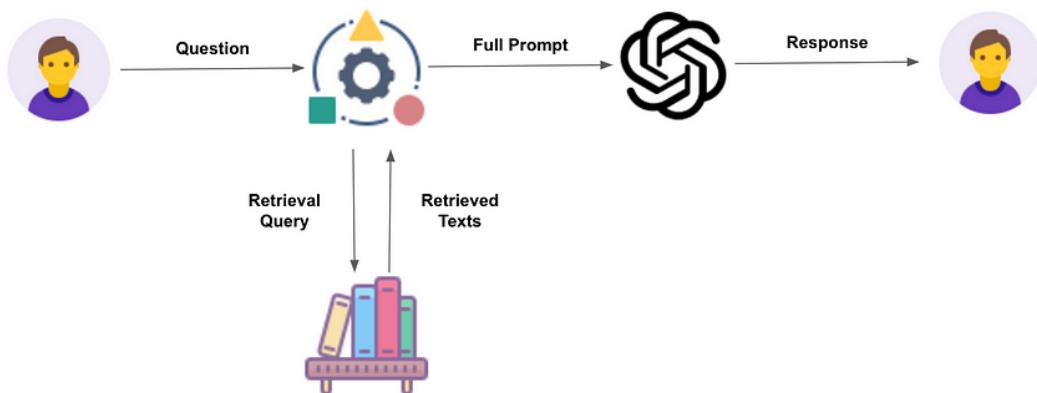
Figura 2.23 Habilidades de los Foundation Models de OpenAI.



Foundation Model Abilities

Fuente: *GPT-3 and the rise of foundation models* (s.f.)

Figura 2.24 Esquema de funcionamiento de RAG. El LLM recibe como input el prompt del usuario y, como contexto, una serie de fragmentos de texto de la base de datos de conocimiento, que tienen similitud vectorial con el prompt del usuario.



Fuente: *What is Retrieval Augmented Generation?* (s.f.)

2.3.2.1 *Prompting* versus *fine-tuning* versus *Retrieval-Augmented Generation*

Las diversas técnicas de prompting ofrecen una serie de ventajas frente al *fine-tuning*, como la posibilidad de utilizar modelos preentrenados de propósito general, tarea más eficiente que reentrenar modelos para una tarea específica, y la posibilidad de utilizar bases de datos de conocimiento de gran tamaño (técnica de RAG), que son más fáciles de obtener que los grandes conjuntos de datos necesarios para el *fine-tuning*. Además, la base de datos de conocimiento puede ser actualizada de forma independiente al modelo, lo que permite una mayor flexibilidad en el uso de los modelos. Por contra, el modelo no tiene conocimiento global de los contenidos de la base de datos, puesto que no ha sido reentrenado con ellos, por lo que no puede realizar inferencias globales sobre ella, y la calidad de los resultados depende en gran medida de cómo se ha dividido su contenido en fragmentos y de la calidad de la representación vectorial de cada uno de ellos. Se ha probado un mayor rendimiento, en tareas específicas, de técnicas combinadas de *fine-tuning* y RAG (Lewis y cols., 2021).

2.4 Limitaciones conocidas de los LLM

Arun Biji (s.f.) enumera una serie de limitaciones que actualmente muestran los LLM y que consideramos muy importantes a la hora de valorar su uso en el contexto de este trabajo. Estas limitaciones son:

2.4.1 Corte de conocimiento

Los LLM no aprenden nuevos datos tras su entrenamiento, por lo que existe una limitación en la información o el conocimiento disponible al no haber tenido acceso a información o acontecimientos ocurridos después de esa fecha límite. Esta limitación se minimiza con rentrenamientos tipo *fine-tuning* o con técnicas como RAG.

2.4.2 Alucinaciones

Se entiende por *alucinación* a un dato erróneo que el modelo genera con total apariencia de verosimilitud y corrección. Para entender por qué sucede esto, hay que comprender que la finalidad de un LM es *modelar* el lenguaje humano, con lo que, a priori, no se puede esperar que los datos que de en sus respuestas sean ciertos sino verosímiles. Los LLM se entrena con vastas cantidades de datos de diversa calidad, y la trazabilidad de los datos que devuelve en sus respuestas es muy difícil de determinar. Precisamente, la técnica RAG permite dibujar la trazabilidad de los datos de las respuestas, en tanto en cuanto estas se basan en los datos de la base de datos de conocimiento. Por otra parte, la constante mejora de los datos de entrenamiento de los LLM (Gunasekar y cols., 2023) hace que este problema de las alucinaciones se minimice aunque nunca se elimina completamente.

2.4.3 Ineficiencia y coste

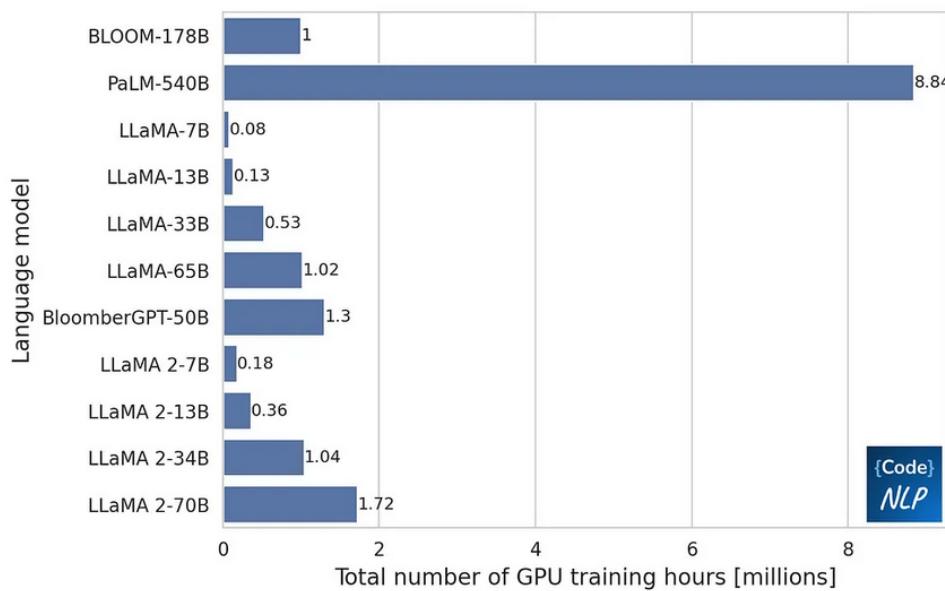
Una de las razones por las que los LLM, y toda la IA generativa en general, no se ha desarrollado hasta hace relativamente poco tiempo, es la alta demanda de recursos computacionales que requiere tanto su fase de entrenamiento como su fase de inferencia. La carrera actual en el desarrollo de la IA reside en ganar eficiencia en los modelos.

Los problemas de eficiencia y coste se pueden articular en torno a tres ejes (Arun Biji, s.f.):

- a) *Recursos de tiempo computacional:* El tiempo necesario para entrenar un LLM es muy alto, y depende del tamaño del modelo, del conjunto de datos de entrenamiento y de la capacidad de cómputo disponible. En general, se habla en términos de *años de GPU*, lo cual está fuera del alcance de la computación de usuario. La Figura 2.25 muestra el coste en millones de horas de GPU de diversos modelos de lenguaje de código abierto. En la fase de inferencia el coste computacional, aunque menor que en la fase de entrenamiento, sigue siendo alto, lo cual constituye un problema para el uso de LLM en local en dispositivos de usuario final.

- b) *Recursos de memoria*: Pero no es el tiempo de procesador el único origen de la ineficiencia. Estos modelos, de tamaños gigantescos, requieren de grandes cantidades de memoria RAM para su entrenamiento e inferencia. En LLM *open source* como Mistral o Llama, en versiones comparables en rendimiento a GPT-3.5, se necesita entre 16 y 100 GB en GPU para la inferencia.
- c) *Recursos económicos*: Finalmente, los costes computacionales se traducen en costes económicos, solo asumibles por grandes empresas. De ahí que la mayoría de LLM sean propiedad de grandes empresas como Google, Facebook o Microsoft, quienes los ponen a disposición de los usuarios a través de sus *application programming interface* (API).

Figura 2.25 Coste de entrenamiento en horas de GPU de diversos modelos de lenguaje.



Fuente: Ph.D (2023)

Es de esperar que la eficiencia de recursos mejore con el tiempo. De hecho, ya se están viendo significativos avances en este sentido, especialmente de la mano del *open source*, que está democratizando el acceso a los LLM y a la IA en general, con sus contribuciones a mejoras en la arquitectura de los modelos, su proceso de entrenamiento, la eficiencia en dispositivos de bajos recursos y la creación de conjuntos de datos de entrenamiento de alta calidad, entre otros. En estos momentos, la gran mayoría de modelos de código abierto que grandes empresas crean y entrena, y los propios usuarios rentrenan con un *fine-tuning*, se pueden descargar y utilizar de forma gratuita desde repositorios como Hugging Face.

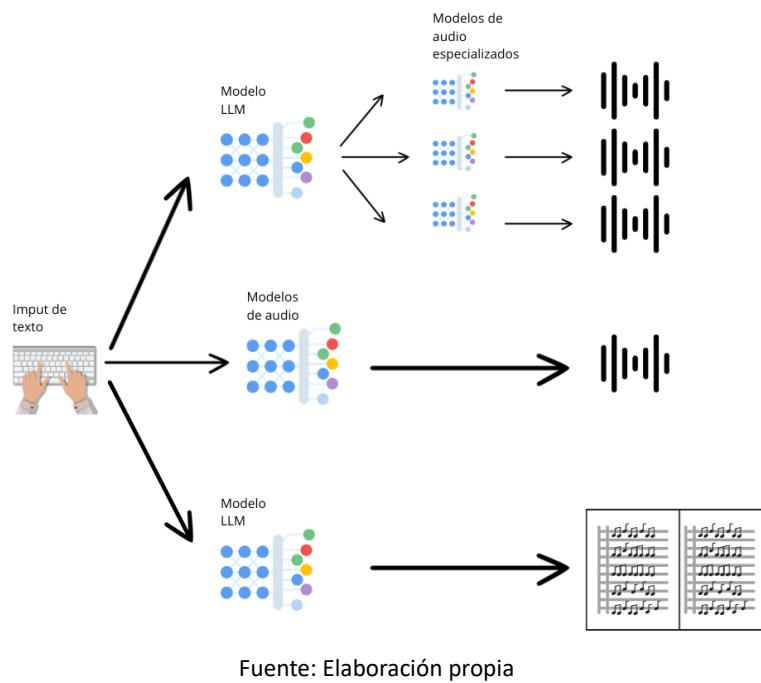
2.4.4 Modelos de *caja negra*, con sesgos y riesgos

Los LLM funcionan como modelos de *caja negra*, lo que significa que los detalles internos de su proceso de generación de respuestas son inaccesibles. Esta característica se debe a su naturaleza probabilística, donde la generación de texto se basa en la probabilidad de secuenciación de tokens y no en una serie de reglas predefinidas. Por consiguiente, es imposible discernir el procedimiento específico que emplea el modelo para elaborar sus respuestas. Esta limitación ha de ser tenida en consideración en el contexto de los LLM, que se distinguen de otros modelos de IA, como los de clasificación, en que no se entrenan mediante datos etiquetados, sino que desarrollan su aprendizaje de manera autónoma a partir de grandes volúmenes de datos no etiquetados.

La falta de transparencia en los LLM conlleva la especial dificultad de identificar cualquier sesgo inherente al modelo. La opacidad y el sesgo de los LLM no solo plantean dilemas éticos, sino que también constituyen un riesgo potencial de seguridad, consideración importante en el contexto de nuestro estudio.

2.5 Estado de la cuestión

En la actualidad, la generación de música o, en un sentido amplio, de sonido, constituye uno de los frentes de investigación activos y más prometedores en el campo de la IA. La generación música con modelos de DL ha sido abordado históricamente desde dos grandes perspectivas en función de la naturaleza de los datos generados por los modelos: la generación de música simbólica y la generación de música con audio. La primera de ellas consiste en la creación de elementos musicales como notas, acordes, melodías, etc. en un formato simbólico, ya sea MIDI, OSC, MusicXML, ABC, Lilypond, o cualquier formato musical reducible a símbolos textuales. La segunda, en la creación de un flujo de audio directamente reproducible sin necesidad de síntesis sonora o utilización de bibliotecas de *samples*. Encontramos investigaciones recientes en ambos campos, incluso en la combinación de ambos, donde se generan elementos simbólicos que son posteriormente convertidos a audio. La Figura 2.26 muestra un esquema de las diferentes aproximaciones a la generación de música con IA. Además de analizar el estado del arte en estos campos, se analizará también el uso de los LLM en la creación de código de programación, campo muy activo y que puede arrojar luz sobre el uso de estos modelos en la creación de código de lenguajes de programación musical.

Figura 2.26 Esquema de las diferentes aproximaciones a la generación de música con IA

2.5.1 Modelos de IA generativa aplicados a la música

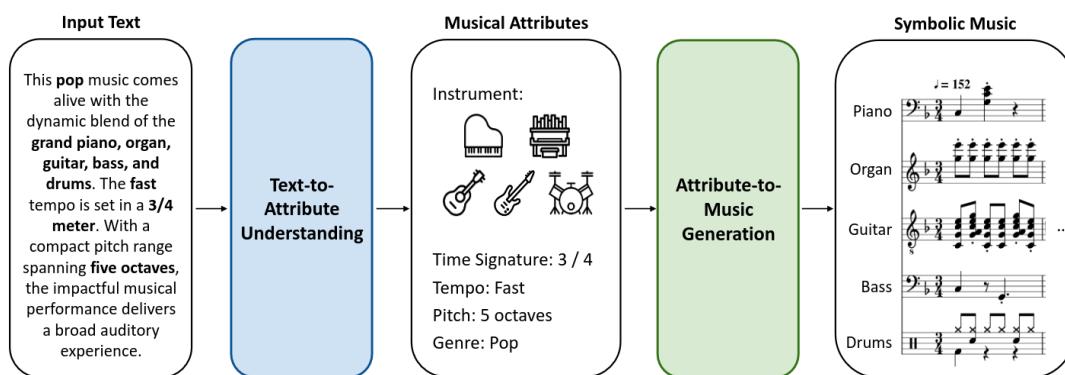
La primera aproximación, la de la composición informática de música a través de elementos simbólicos, se remonta a los inicios de la computación, con pioneros como Lejaren Hiller y Leonard Isaacson, que en 1957 crearon *Illiac Suite* (Ariza, 2011; Funk, 2018), la primera composición musical generada por ordenador. Desde entonces, la generación de música simbólica ha sido abordada desde diferentes perspectivas, como la generación de música aleatoria, la generación de música a partir de reglas, la generación de música a partir de modelos probabilísticos como las cadenas de Markov, música basada en matemática fractal, estocástica, etc. (Hernandez-Olivan, Hernandez-Olivan, y Beltran, 2022).

La generación de audio por medio de modelos de DL es un campo de investigación más reciente debido en parte a su gran demanda computacional, que ha estado fuera del alcance de investigadores y empresas hasta la última década. Actualmente, modelos como MuseNet (Department of Computer Science, SRM Institute of Science and Technology, Chennai, India. y cols., 2020), Jukebox (Dhariwal y cols., 2020), Stable Audio (*Stable Audio: Fast Timing-Conditioned Latent Audio Diffusion*, s.f.), o Suno (*Suno AI*, s.f.), entre otros, están consiguiendo diariamente inéditos avances en lo que se refiere a la generación de música en formato de audio de una alta calidad, y, al igual que está ocurriendo con modelos de generación de imagen o vídeo, se espera que en los próximos años se produzcan avances muy significativos en este campo.

2.5.2 LLM como asistentes para la creación de música simbólica

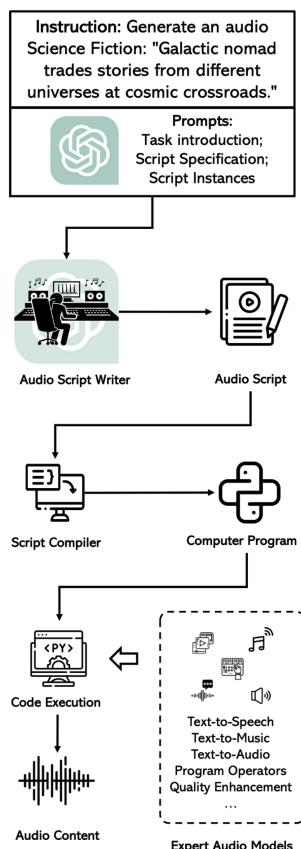
La aproximación a la generación de música a través de elementos simbólicos en lugar de puro audio adquiere una renovada fuerza por la aparición de los LLM, los cuales se basan en la arquitectura *transformer*, aparecida en 2017 (Vaswani y cols., 2017), por lo que es un tema de investigación muy reciente y al que se le está prestando menos atención. El grueso de las investigaciones candentes en este momento se están centrándolo en la generación de audio, campo más prometedor desde el punto de vista comercial. No obstante, existen recientes investigaciones en este campo, como MuseCoco (Lu y cols., 2023), que propone un procedimiento en dos etapas desde el input de texto en lenguaje natural del usuario hasta la generación de música simbólica en forma de partitura. La Figura 2.27 muestra el esquema de funcionamiento de este sistema.

Figura 2.27 Esquema de funcionamiento en dos pasos de MuseCoco en la generación de música simbólica desde texto



Fuente: Lu y cols. (2023)

Otro interesante trabajo sobre el uso de LLM para la generación musical es WavJourney (X. Liu y cols., 2023), donde el papel del LLM es el de agente compositivo capaz de conectarse con otros modelos de generación de audio (voz, instrumentos, ruidos, música, etc.) para *componer* a partir del input o prompt del usuario. En este caso, el LLM juega un papel de mediador y planificador de la composición, además de generador de prompts para los diversos modelos implicados en la generación de audio. La Figura 2.28 muestra el esquema de funcionamiento de este sistema. Una aproximación similar en cuanto al uso de los LLM en la generación de audio lo encontramos en Borsos y cols. (2023).

Figura 2.28 Esquema de funcionamiento de WavJourney en la generación de audio desde texto

Fuente: X. Liu y cols. (2023)

2.5.3 LLM en la creación de código de programación

La generación de código de programación por LLM, independientemente de su finalidad artística o técnica, sí que es un campo muy estudiado en los últimos años, y que ha dado lugar a la aparición de modelos como GitHub Copilot, que ha sido capaz de generar código de programación de considerable calidad a partir de inputs o prompts en lenguaje natural. GitHub Copilot es un servicio que integra un modelo GPT-4 junto a un conjunto de aplicaciones de integración en entornos de desarrollo como Visual Studio Code, Atom, etc. Debido a su amplia aplicación en el campo de la programación, es un modelo que ha recibido una gran atención mediática, y que ha sido objeto de numerosos estudios y análisis, por lo que podemos considerarlo como el estado del arte en la generación de código de programación por LLM. Es por ello que los principales estudios sobre la interacción de LLM con humanos en el campo de la programación se han centrado en este servicio. De todos estos artículos podemos extraer conclusiones de interés.

Se ha investigado con detalle el impacto que tiene el diseño de prompts adecuados a la tarea de generación de código. J. Li y cols. (2023) llegan a la conclusión de que un uso adecuado de la técnica de prompting COT puede mejorar significativamente la calidad y corrección del código generado. Esta variante es denominada *structured chain of thoughts* (sCOT), y consiste en solicitar la respuesta del LLM en dos pasos: primero, una descripción en pseudocódigo de la tarea a realizar, y segundo, la implementación en código final de la tarea. En el estudio se muestra que el uso de sCOT mejora el índice de corrección del código generado en un 13,79 % respecto al uso de COT en lenguajes como Python o C++.

Otro estudio relevante para nuestro trabajo es Chen, Lin, Schärli, y Zhou (2023), donde se analizan técnicas de *Self-Debugging*, donde, por medio de *few-shot*, se predispone al LLM para la autorrevisión del código generado, consiguiendo una mejora de hasta el 12 %.

Un aspecto importante, además de la corrección del código generado por medio de LLM, es el impacto en la productividad de su uso en tareas concretas de programación. (Peng, Kalliamvakou, Cihon, y Demirer, 2023) mostró que estas pueden llegar a realizarse un 55,8 % más rápido que sin su ayuda, aunque necesario señalar que en este estudio participaron investigadores de Microsoft, empresa que desarrolla GitHub Copilot, con lo cual los resultados pueden estar sesgados. En esta línea, investigadores Montreal y Toronto, observaron que herramientas como GitHub Copilot dan resultados muy comparables a los de programadores humanos, si bien su utilidad es mayor cuando lo utilizan programadores con experiencia, constituyendo cierto riesgo en manos de programadores noveles (Moradi Dakhel y cols., 2023). Otro interesante estudio sobre la fiabilidad de la generación de código de GitHub Copilot (Mastropaoletti y cols., 2023), halló que en casi la mitad de los casos, el código generado por dos descripciones en lenguaje natural semánticamente equivalentes, era diferente, y en más de una cuarta parte de los casos la corrección del código se vio comprometida, lo que pone en tela juicio la robustez del sistema.

3. Metodología

STRUCTURED PRUNING IS ALL YOU NEED

— Cai y cols. (2022)

3.1 Revisión bibliográfica

3.1.1 Objetivos de la revisión

La revisión bibliográfica en este trabajo tendrá como propósito contextualizar la investigación en un marco teórico, comprender el estado actual del DL y los LLM, e identificar el estado del arte en cuanto al uso de estos sistemas para la generación de música en código de programación.

3.1.2 Fuentes y búsqueda

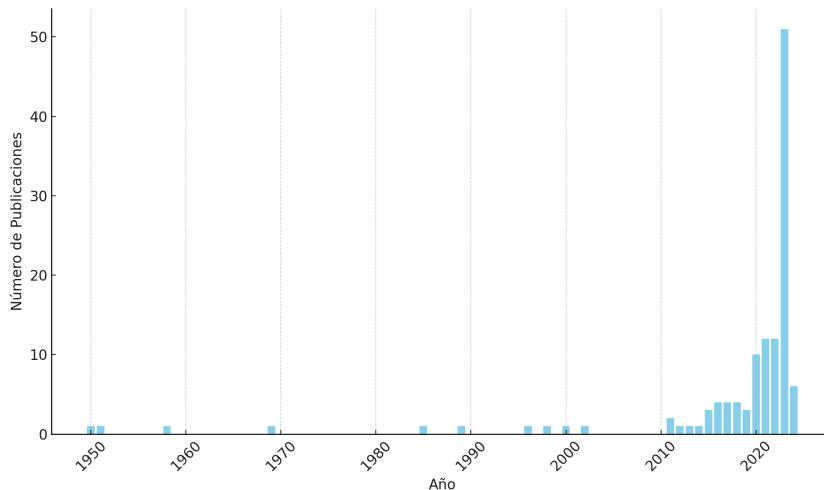
Para llevar a cabo esta revisión, se consultarán diversas bases de datos, revistas especializadas y conferencias de relevancia en el ámbito principalmente de la IA y el prompting. La publicación de artículos científicos en el área de la IA, y, en concreto, de LLM, es diaria, y debido a que la mayoría de estos trabajos han sido publicados durante 2022 y 2023, no cuentan con revisión por pares u otros mecanismos de verificación de las tesis en ellos contenidas. Por esta razón, sus resultados deben ser tomados con cautela y como líneas abiertas de investigación.

3.1.3 Selección y evaluación

Los criterios utilizados para seleccionar los trabajos incluidos en esta revisión se centraron en la relevancia y actualidad (véase la Figura 3.1), ya que la investigación en DL se mueve a gran velocidad. Se dieron prioridad a trabajos publicados en los últimos meses y aquellos que se centran específicamente en los LLM y la interacción mediante prompting y la generación de código de programación.

Aunque la mayor parte de las referencias utilizadas han sido publicados en los últimos años, se han incluido algunos trabajos de referencia publicados en años anteriores, especialmente para conceptos clave en el campo de la IA y el DL.

Figura 3.1 Publicaciones por año de las referencias utilizadas en este trabajo. Se observa un claro predominio de publicaciones relevantes especialmente en 2023.



Fuente: Elaboración propia

3.1.4 Resultados de la revisión

Los hallazgos más relevantes de esta revisión son discutidos en detalle en la sección correspondiente al estado de la cuestión. No obstante, es importante destacar que existen múltiples enfoques y técnicas para interactuar con los LLM, y que su aplicación en la composición musical aún se encuentra en una fase exploratoria, al no ser un objetivo primordial de los investigadores en el campo de la generación de código. Por otra parte, la mayoría de los trabajos publicados en el área de la música y los modelos generativos se centran en la generación de música como audio, y no en la interacción con los LLM como herramienta de composición.

3.2 Enfoque, alcance y diseño

3.2.1 Enfoque

El enfoque de esta investigación es cualitativo, aunque se apoya en estudios previos de carácter cuantitativo. Será el propio investigador quien lleve a cabo la práctica y la evaluación de los resultados, con un claro acento en la reflexión sobre los procesos creativos implicados.

3.2.2 Alcance

La investigación tiene un alcance exploratorio y descriptivo. Es exploratorio en la medida en que se busca comprender y descubrir posibles aplicaciones de los LLM en el contexto de la composición musical algorítmica, un campo que aún no ha sido ampliamente estudiado. Al

mismo tiempo, se adopta un enfoque descriptivo para detallar los resultados de la interacción con los sistemas de IA.

3.2.3 Diseño

El diseño de esta investigación es no estructurado, en tanto que se sigue un plan abierto de actuación, por su naturaleza exploratoria, en función de los resultados de fases anteriores.

El trabajo tendrá un carácter observacional, donde el papel del investigador será el de observador y participante en el proceso de interacción creativa con sistemas de IA.

3.3 Herramientas y recursos utilizados

Para llevar a cabo este estudio, se utilizarán ciertas herramientas informáticas:

1. Ordenador personal con sistema operativo Linux.
2. Visual Studio Code como editor de código.
3. Python como lenguaje de programación.
4. Diversos programas de síntesis de sonido, especialmente: SuperCollider, Tidal Cycles, Sonic Pi y Csound.
5. *Ardour* como *digital audio workstation* (DAW).
6. Servicio de *Advanced Data Analysis* de OpenAI para la elaboración de gráficas cuantitativas.
7. Servicio de ChatGPT Plus y uso de la API de OpenAI para la interacción con los LLM.
8. Servicios de GitHub y Google Drive para el almacenamiento de datos.

3.4 Desarrollo y aplicación

3.4.1 Discusión de los criterios de elección de los lenguajes de programación musical

En primer lugar se discutirán los criterios de elección de los lenguajes de programación musical utilizados en esta investigación: SuperCollider y Tidal Cycles. Se trata de una decisión importante, ya que el lenguaje de programación musical utilizado condicionará el tipo de interacción con los sistemas de LLM.

3.4.2 Elección de los sistemas de LLM a utilizar

Al mismo tiempo, y de forma relacionada con la elección de los lenguajes musicales, se discutirán las razones de la elección de los servicios de OpenAI para esta investigación.

3.4.3 Exploración de posibles interfaces de interacción con sistemas de LLM

De algún modo, esta será la guía de la investigación. A partir de las posibilidades ofrecidas principalmente por OpenAI, se explorarán las posibilidades de interacción en la creación de código de programación musical, así como la calidad de los resultados que se obtengan aplicando diversas técnicas conocidas de prompting. Se comenzará con la interfaz más inmediata, que es el uso de chatbots, y se irá avanzando hacia el uso de la API de OpenAI y la programación de scripts en Python.

3.4.4 Documentación de las interacciones

En cada fase de la investigación se guardarán los archivos de texto con las interacciones realizadas, los prompts utilizados, los resultados obtenidos y las reflexiones sobre el proceso.

3.4.5 Creación de una pieza de arte sonoro

Una vez exploradas diferentes interfaces de comunicación con sistemas de LLM, se procederá a la creación de una pieza de música algorítmica electroacústica que implique una interacción creativa con un sistema de LLM. La metodología concreta de composición se decidirá en función de los resultados obtenidos en las fases previas.

3.4.6 Análisis cualitativo y reflexión

En cada fase se procederá al análisis de los resultados obtenidos, dirigiendo así el proceso de investigación a pasos posteriores. Acabado todo el proceso experimental y de composición musical, se procederá a sintetizar los resultados y a realizar una reflexión final.

3.4.7 Reflexión de las limitaciones y prospectivas

Debido al carácter exploratorio de esta investigación, es necesario reflexionar sobre las limitaciones inherentes a la misma, así como discutir las posibles líneas de investigación futuras, especialmente hacia el diseño de investigaciones cuantitativas y de mayor alcance.

3.4.8 Disponibilidad y publicación de los scripts del trabajo

Todos los scripts utilizados en este trabajo, así como los prompts utilizados, códigos musicales obtenidos y el texto de esta memoria, quedarán almacenados y disponibles a la comunidad científica (véase anexo B).

4. Elección de lenguajes de programación musical para este trabajo

TRANSCRIPTION IS ALL YOU NEED

— Hung, Wichern, y Roux (2020)

Existen en la actualidad numerosos lenguajes de programación musicales, cada uno con sus propias características y orientados a diferentes propósitos. En este capítulo se describen los lenguajes de programación musicales más relevantes, comparando sus características, para finalmente elegir los lenguajes que se utilizarán en esta investigación a través de una serie de criterios.

4.1 Música en código de programación

Una característica común a los lenguajes de programación musicales es el hecho de que su representación simbólica se realiza por medio de código de programación, en texto plano, y por tanto, son lenguajes formales, con una sintaxis y una semántica definidas. Esto significa que son lenguajes que pueden ser generados por un modelo de lenguaje natural, como los LLM, incluso si no han sido entrenados específicamente en estos lenguajes. Esto es importante, ya que permite que los LLM puedan generar código de estos lenguajes, y por tanto, que puedan generar sonidos y música de un modo indirecto. Los LLM, en su preentrenamiento, han sido entrenados con todo tipo de código, incluyendo lenguajes con propósitos musicales o sonoros. Aunque no podemos esperar que los LLM tengan tanta destreza en Csound como en Python, sí podemos esperar que sean capaces de generar código de Csound, y que este código sea correcto, y que por tanto, pueda ser ejecutado para generar sonidos. Otra cosa es que el código generado sea interesante, o que el sonido generado sea interesante, ni siquiera que el código generado tenga sentido musical. No obstante, explorar esta capacidad de creación musical por medio de la generación de código es uno de los objetivos de esta investigación. La figura 4.1 muestra un *iHola, mundo!*¹ en diversos lenguajes de programación sonora, creados todos ellos por medio de GPT-4.

¹Un *iHola, mundo!*, o, más conocido como *Hello, World!*, es un programa informático mínimo de un lenguaje de programación, el cual lo único que hace es imprimir en pantalla el texto «*Hola, Mundo!*», con una finalidad pedagógica. Por extensión, un *iHola, Mundo!* de un lenguaje musical sería aquel que produce un sonido simple, como una onda sinusoidal.

Figura 4.1 Códigos de «¡Hola Mundo!» en diferentes lenguajes de programación sonora. (a) Csound, (b) SuperCollider, (c) Sonic Pi, (d) Overtone (Clojure), (e) FoxDot y (f) Chuck.

```
<CsInstruments>
instr 1
    a1 oscil 0.5, 440, 1
    out a1
endin
</CsInstruments>
<CsScore>
f1 0 1024 10 1
i1 0 2
</CsScore>
```

(a) Csound

```
(
SynthDef(\helloworld, {
    Out.ar(0, SinOsc.ar(440, 0, 0.2))
}).add;
)
Synth(\helloworld);
```

(b) SuperCollider

```
play 72
sleep 1
```

(c) Sonic Pi

```
(use 'overtone.live)
(definst hello-world [] (sin-osc 440))
(hello-world)
```

(d) Overtone (Clojure)

```
p1 >> pluck([0, 1, 2, 3])
```

(e) FoxDot

```
SinOsc s => dac;
440 => s.freq;
1::second => now;
```

(f) Chuck

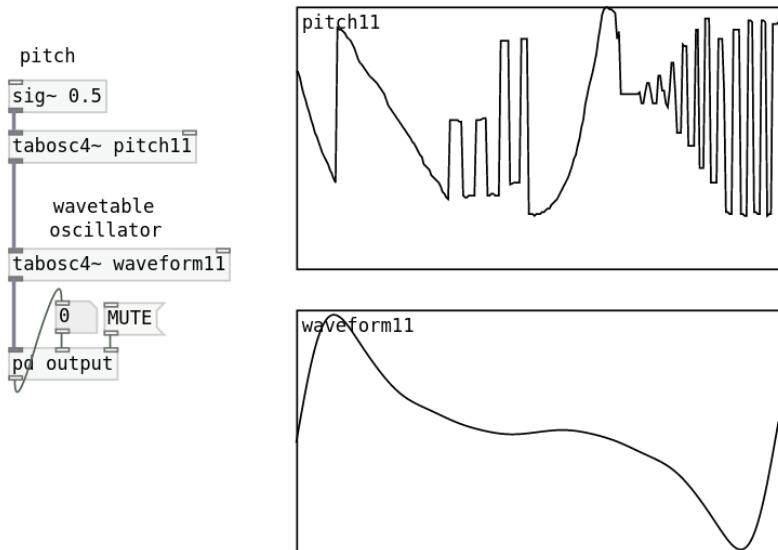
Fuente: Creado por el autor con GPT-4.

Excluimos lenguajes de programación de audio como Pure Data u Open Music, ya que estos lenguajes, si bien guardan sus archivos en formato de texto plano, estos códigos no están diseñados para ser escritos por el usuario, sino solo como un modo de representación interna de los grafos de objetos que se muestran en un canvas gráfico (Figura 4.2), y que luego son interpretados por un motor de audio.

Figura 4.2 Patch de Pure Data en su representación interna en código (a), y en su aspecto visual (b).

```
#N canvas 443 44 725 592 12;
#X declare -stdpath ./;
#X floatatom 127 60 6 0 0 0 - - - 0;
#N canvas 0 0 450 300 (subpatch) 0;
#X array table10 259 float 1;
#X coords 0 1.02 258 -1.02 258 130 1;
#X restore 279 92 graph;
#X text 459 533 updated for Pd version 0.34;
#X text 78 18 WAVETABLE OSCILLATORS;
#X text 25 123 wavetable;
...
...
```

(a) Representación interna en código de un patch de Pure Data



(b) Aspecto visual de la interfaz gráfica de un patch de Pure Data

Fuente: Elaboración propia

4.2 El problema de componer con notas musicales

Podemos incluir en nuestra lista sistemas de notación musical que se reducen a código, como Lilypond, la librería de Python *Music21*, *ABC notation* o el mismo formato *MusicXML*. Sin embargo, se ha podido comprobar desde un primer acercamiento a LLM como GPT-4, que aún no son capaces de generar un código mínimamente interesante desde el punto de vista artístico cuando hablamos de utilización de notas musicales, acordes, etc., a pesar de comprender perfectamente la sintaxis. La figura 4.3 muestra una partitura generada por GPT-4, cuando se le pidió crear una melodía de 4 compases al estilo de Bach. Como se puede apreciar, la melodía

es correcta desde el punto de vista de la sintaxis, pero no tiene ningún interés musical ni atisbo de aproximación estilística al la petición del usuario. Por tanto, se ha decidido excluir de esta investigación los lenguajes de notación musical.

Figura 4.3 *Melodía, en Lilypond de 4 compases generada por GPT-4, «al estilo de Bach». Se aprecia la correcta sintaxis pero la falta de interés musical y la lejanía conceptual a la petición del usuario.*

```
\score {
    \new Staff {
        \key c \major
        \time 4/4
        \tempo 4 = 100

        % Compás 1
        c'4 d' e' g' |

        % Compás 2
        f' e' d' c' |

        % Compás 3
        e'4 f' g' a' |

        % Compás 4
        g' f' e' d' |
    }
    \layout { }
    \midi { }
}
```

(a) Código Lilypond devuelto por GPT-4



(b) Renderizado a notación musical por Lilypond

Fuente: Elaboración propia

Este mismo problema lo encontramos en cualquier tipo de codificación que utilice notas musicales, como el formato MIDI. Componer música con notas musicales, con las cotas de complejidad que ha alcanzado la música a lo largo del tiempo y de la geografía, no es una tarea trivial, y va mucho más allá del conocimiento básico de los símbolos utilizados en las diferentes notaciones. La música es un arte, y como tal, requiere de un conocimiento profundo de la teoría musical, de la armonía, del contrapunto, de la historia de la música, de la cultura musical.

cal, de la psicología de la percepción musical, etc. Es por ello que no podemos esperar que un LLM sea capaz de generar música con notas musicales de un modo interesante, al menos por el momento.

4.3 Lenguajes orientados a la síntesis de sonido

La música experimental, la música electrónica, la música electroacústica, la música concreta, la música generativa, etc., son géneros musicales que se han desarrollado en el siglo XX, y que han utilizado la tecnología como medio de expresión. En este contexto, han surgido numerosos lenguajes de programación orientados a la síntesis de sonido, que han permitido a los compositores de música electrónica y experimental crear sonidos y música de un modo más flexible y potente que con los sintetizadores analógicos (Supper, 2012). Estos lenguajes de programación han sido utilizados también en la creación de instalaciones sonoras, en la creación de interfaces de usuario o en la composición musical, y su desarrollo ha ido parejo con el de la tecnología digital. Una completa introducción al campo de la programación informática aplicada a la síntesis sonora es la de Curtis Roads (1996).

El lenguaje sonoro en torno a estos géneros musicales es mucho más flexible, por lo general, que el lenguaje de la música notada. Especialmente las músicas autodenominadas experimentales, que no tienen un lenguaje musical definido y que se caracterizan por la búsqueda de nuevos sonidos y nuevas formas de expresión, son idóneas para una experimentación con elementos no humanos, como los LLM. Esta interacción eventualmente puede traducirse en la exploración de nuevos timbres y texturas sonoras.

Dentro de los lenguajes de programación orientados a la síntesis de sonido, podemos distinguir entre los diseñados para ser codificados en interpretaciones en vivo, y los diseñados para ser codificados en archivos de audio, a una composición más estática. Csound (Boulanger, 2000) fue construido para crear composiciones renderizadas en archivos de audio, si bien actualmente tiene características que le permiten ser ejecutado en tiempo real. SuperCollider (Wilson, Cottle, y Collins, 2011) tiene características de ambos mundos, lo que lo hace óptimo tanto para la composición de obras sonoras como para la codificación en vivo, en lo que se denomina *live coding*. Precisamente su ductilidad y potencia lo ha convertido en plataforma de audio sobre la que se han creado otros lenguajes orientados únicamente al tiempo real y *live coding*, como FoxDot (Kirkbride, 2023), Sonic Pi, Overtone (*Overtone - Collaborative Programmable Music*, s.f.) o Tidal Cycles (*Tidal Cycles*, s.f.). Actualmente proliferan lenguajes diseñados para

esta interacción en tiempo real con el usuario. Además de los expuestos, podemos destacar ChucK (Team, s.f.) o Strudel (*Strudel REPL*, s.f.), entre otros. Esto se debe, entre otros factores culturales, al aumento de la potencia de los ordenadores, que permite ejecutar en tiempo real programas de síntesis de sonido cada vez más complejos.

4.4 Criterios de selección de lenguajes de programación musical para esta investigación

Hubiera sido de mucho interés incluir en esta investigación todos los lenguajes de programación musicales mencionados en este capítulo, y muchos más. Sin embargo, esto hubiera resultado una tarea que excede los límites de recursos y tiempo. Por ello, se ha decidido acotar la investigación a únicamente dos lenguajes de programación musicales, como punto de partida de ulteriores trabajos. En la elección se han tenido en cuenta una serie de criterios, que se exponen a continuación.

4.4.1 Comunidad activa de usuarios y desarrolladores

Se ha considerado importante que los lenguajes de programación musicales seleccionados tengan una comunidad activa de usuarios, que permita al investigador tener un soporte en caso de dudas o problemas. Esto es especialmente importante en el caso de los lenguajes de programación musicales, ya que no se trata de lenguajes de programación generalistas, sino de lenguajes de programación con un propósito específico, y por tanto, con una comunidad de usuarios más reducida que la de lenguajes de programación generalistas como Python o Java.

4.4.2 Documentación extensa y accesible

Una extensa y accesible documentación es imprescindible, no solo para que el investigador tenga una inmediata y fácil referencia de los elementos del lenguaje, de las librerías disponibles, de los ejemplos de código, etc., sino para que el LLM pueda tener acceso a esta documentación, y por tanto, pueda generar código de este lenguaje. La primera fase del entrenamiento de los LLM es el denominado *preentrenamiento*, en el que se les expone a todo tipo de texto, incluyendo código de programación. Cuanta más información de calidad exista en la red sobre un lenguaje de programación, incluyendo piezas musicales y tutoriales, más probable es que el LLM haya tenido acceso a ella, y por tanto, más probable es que sea capaz de generar código de este lenguaje.

4.4.3 Capacidad de trabajo en tiempo real

Es importante que los lenguajes de programación musicales seleccionados sean capaces de trabajar en tiempo real, ya que esto permitirá al investigador interactuar con el LLM en tiempo real, y por tanto, tener una experiencia más inmersiva y más rica. El uso de API permite una interacción continua y automatizada con los modelos de lenguaje. Como se ha señalado más arriba, prácticamente todos los lenguajes modernos tienen de uno u otro modo esta capacidad de ser ejecutados y codificados en tiempo real, si bien algunos de ellos han sido creados con este objetivo en mente, como Sonic Pi o Tidal Cycles. El propio SuperCollider posee la librería *JITLib* (*Just in Time Library*), que contiene toda la funcionalidad necesaria para la codificación en tiempo real.

Explicado de una forma rápida y sencilla, la codificación en tiempo real consiste en la ejecución de código de programación en tiempo real, en el momento de la interpretación. Esto permite al usuario interactuar con el código de un modo más inmediato, y por tanto, tener una experiencia más inmersiva. En el caso de los lenguajes de programación musicales, esto se suele conseguir creando bucles infinitos, que se ejecutan continuamente y pueden ser modificados en vivo (o sustituidos) sin tener que reiniciar el programa, y permiten una intuitiva interacción con el usuario. La Figura 4.4 muestra un ejemplo de bucle infinito en Sonic Pi. Este bucle es sustituido por otro nuevo en vivo cada vez que el usuario hace una modificación en él. No es difícil imaginar el papel que puede tener un modelo de inteligencia artificial en este proceso, generando código de programación sonora en tiempo real y permitiendo una interacción continua con el usuario, e incluso sin la intervención del usuario, como se verá más adelante.

Figura 4.4 Bucle infinito en Sonic Pi. En este código se ejecutan cuatro notas (con la sentencia *play*), una por segundo (tiempo indicado a *sleep*) en un bucle infinito. El usuario puede modificar las notas, y el cambio se ejecuta en el siguiente ciclo del bucle.

```
live_loop :melody do
  play :E4
  sleep 1
  play :G4
  sleep 1
  play :B4
  sleep 1
  play :C5
  sleep 1
end
```

Fuente: Elaboración propia en *Sonic Pi*

4.4.4 Fácil gestión de errores

En el proceso de depuración de un código, incluido el de una pieza musical, es eminentemente iterativo. Se prueba el código, se detectan errores, se corrigen, se vuelve a probar, etc. Es importante que el lenguaje de programación musical permita una depuración rápida y eficiente, para que el investigador pueda centrarse en la creación musical, y no en la depuración del código. Esto es especialmente importante en el caso de los LLM, ya que estos no son capaces de detectar errores en el código, y por tanto, el investigador debe ser capaz de detectarlos y corregirlos de un modo rápido y eficiente.

Esta característica está muy unida a la anterior, ya que un buen lenguaje con capacidades de tiempo real ha de ser inmune a los errores. Si en una iteración el usuario comete un error, el programa no debe detenerse, sino que debe continuar ejecutándose, y el usuario debe ser capaz de corregir el error y continuar con la iteración. Los LLM, al igual que los músicos humanos, cometen errores, y en las interpretaciones en vivo estos errores deben ser ignorados en muchas ocasiones en pro de la fluidez de la sesión. En el caso de una composición renderizada en un archivo de audio, los errores deben ser corregidos antes de la renderización, y el proceso de depuración es más lento y minucioso. En este caso no es crítico que el lenguaje de programación sea inmune a los errores. Al contrario, es importante que devuelva mensajes de error claros y precisos, para que el investigador pueda corregirlos de un modo rápido y eficiente.

SuperCollider es un ejemplo que une ambos mundos. La librería JITLib ofrece una gran colección de clases para su uso en interpretaciones en vivo, donde el usuario ha de concentrarse en la creación musical, y no tanto en la depuración del código. Si lo que se desea es crear una composición separada del momento interpretativo, su consola de errores es muy precisa y permite una depuración rápida y eficiente.

Tidal Cycles, que está enfocado principalmente a la codificación en tiempo real (o *live coding*), permite una ejecución sonora continua por defecto, abstraído completamente de los errores. El artista ha de estar atento a la consola para detectar los errores, pero será muy raro que el software termine su ejecución por un error. Esta es una característica muy interesante para la interacción con LLM, asegurando que no existirán interrupciones indeseadas ante código erróneo o respuestas inesperadas del modelo.

4.4.5 Independencia del IDE

Esta característica es importante si se quiere utilizar un LLM por medio de la API. Si un lenguaje tiene un propia interfaz gráfica o IDE², no hay un modo sencillo de ejecutar dicho código por medio de scripts creados ad hoc u otros programas de terceros. Una investigación profunda en la interacción entre LLM y usuario humano requiere de un control total sobre el código que se le pasa al LLM, así como la posibilidad de automatizar la ejecución del código y convertirlo en sonido a través de scripts o software creado ad hoc. Por ello, se ha considerado importante que los lenguajes de programación musicales seleccionados no tengan un IDE propio, o que este IDE no sea imprescindible para la ejecución del código. Un ejemplo de utilización de IDE propio es Sonic Pi, que no permite por defecto la ejecución de código por medio de scripts externos. SuperCollider, aunque tiene un IDE propio, su intérprete de comandos, *sclang*, puede ser ejecutado por medio de scripts externos, y por tanto, es independiente de IDE. Esta característica de independencia del IDE utilizado es heredado por lenguajes que usan SuperCollider como motor de sonido, como FoxDot o Tidal Cycles.

4.4.6 Lenguajes dominados por el investigador.

Quizás sea este el más arbitrario de todos los criterios, pero un desnivel de conocimiento entre lenguajes podría sesgar los resultados de la investigación. En este caso, el investigador tiene amplia experiencia con SuperCollider³, Csound, Tidal Cycles y Sonic Pi. Es importante que el investigador domine los lenguajes de programación musicales seleccionados, para poder centrarse en la creación musical, y no tanto en la depuración del código. Esto es especialmente importante en el caso de los LLM, ya que estos no son capaces de detectar errores en el código, y por tanto, el investigador debe ser capaz de detectarlos y corregirlos de un modo rápido y eficiente.

4.4.7 SuperCollider y Tidal Cycles

Se ha decidido utilizar SuperCollider y Tidal Cycles como lenguajes de programación musicales para esta investigación. Ambos lenguajes cumplen sobradamente con los criterios expuestos. Se trata de lenguajes muy potentes y flexibles, suficientemente conocidos por el investigador y con una comunidad activa de usuarios y desarrolladores. Ambos soportan un trabajo fluido en tiempo real, aunque enfocaremos la creación de código para piezas musicales en SuperCollider principalmente, y el trabajo que implique codificación en tiempo real, en Tidal Cycles.

²Integrated Development Environment

³Véase el trabajo previo del propio autor (Guerra Parra, 2020).

Ambos lenguajes son independientes del IDE, y por tanto, pueden ser ejecutados por medio de programas de terceros.

Tabla 4.1: Algunos de los lenguajes de programación musicales más relevantes. Datos como el número de desarrolladores o la fecha de la última actualización se han obtenido de la página de GitHub de cada lenguaje. La documentación y el nivel de *Live* se han valorado de 1 a 3 asteriscos, siendo 3 asteriscos la máxima puntuación. Dichas valoraciones son del autor y no corresponden a ninguna métrica. Los datos mostrados corresponden a diciembre de 2023.

| Lenguaje | Lanzamiento | Actualización | Desarrolladores | Documentación | Live | Independiente de IDE |
|----------------------|-------------|---------------|-----------------|---------------|------|----------------------|
| <i>Csound</i> | 1985 | 2022 | 56 | *** | * | Sí |
| <i>SuperCollider</i> | 1996 | 2023 | 212 | *** | ** | Sí |
| <i>ChucK</i> | 2003 | 2023 | 44 | *** | ** | Sí |
| <i>Tidal Cycles</i> | 2009 | 2023 | 76 | *** | *** | Sí |
| <i>Overtone</i> | 2010 | 2023 | 68 | ** | ** | Sí |
| <i>Sonic Pi</i> | 2012 | 2023 | 532 | *** | *** | No |
| <i>FoxDot</i> | 2019 | 2021 | 35 | * | *** | Sí |
| <i>Strudel</i> | 2022 | 2023 | 29 | *** | *** | No |
| <i>GLICOL</i> | 2020 | 2023 | 3 | * | *** | No |
| <i>FAUST</i> | 2002 | 2023 | 106 | ** | ** | No |

Fuente: Elaboración propia

5. Análisis y evaluación de la práctica con diferentes interfaces en OpenAI

ATTENTION IS ALL YOU NEED

— Vaswani y cols. (2017)

La investigación presentada se detalla a continuación, siguiendo la secuencia cronológica del proceso exploratorio realizado con las herramientas proporcionadas por OpenAI. Inicialmente, se examinó ChatGPT, el chatbot desarrollado por OpenAI, para después avanzar hacia el *Playground* y, por último, hacia la API. Esta última herramienta ofrece la capacidad de integrar los modelos en entornos de programación personalizados, facilitando la automatización de solicitudes a los modelos y el procesamiento de resultados específicamente en el ámbito de la generación sonora y musical.

A lo largo de las distintas fases de este estudio, se investigó cómo la aplicación de variadas estrategias de *prompting* afecta la precisión y calidad de las respuestas obtenidas de los modelos. Paralelamente, se han evaluado diversas opciones de interfaces de usuario para optimizar la interacción con los modelos, buscando equilibrar usabilidad con eficacia en la generación de resultados.

5.1 ChatGPT, como primer banco de pruebas

La divulgación de las capacidades de los modelos de lenguaje al público general ha sido principalmente a través de chatbots. Aunque actualmente existen numerosos chatbots que incorporan modelos de lenguaje, ChatGPT de OpenAI fue uno de los primeros en ser presentado públicamente. Utilizando actualmente las versiones GPT-3.5 y GPT-4, este chatbot permite establecer conversaciones en lenguaje natural y, aunque hoy en día posee una amplia gama de funciones que lo posicionan como una herramienta útil en diversos sectores para potenciar la productividad, originalmente fue diseñado con el objetivo más simple de facilitar diálogos interactivos.

ChatGPT ha servido en este trabajo como plataforma experimental para investigar el potencial de estos modelos en la esfera de la creación sonora mediante lenguajes de programación, dando una idea clara de cómo avanzar en interfaces más complejas con el uso de la API.

5.1.1 GPT-4, modelo utilizado en todos los experimentos

En la sección 2.5.2 se ha expuesto el estado del arte de los modelos de lenguaje aplicados a la creación de código, habilidad esta que constituye una de las más notorias. Al mismo tiempo, se señaló que el modelo más avanzado en la actualidad es GPT-4, de OpenAI. No obstante, se realizaron pruebas con otros chatbots, como Bard, de Google, o el mismo GitHub Copilot, cuyos LLM han recibido un *fine-tuning* específico para la creación de código con todos los repositorios de GitHub. Sin embargo, en GPT-4 se ha encontrado un buen equilibrio entre precisión en respuestas y posibilidades de uso, especialmente gracias a la API y su extensa documentación. La Figura 5.1 muestra un ejemplo, realizado al azar, que pone de manifiesto la diferencia entre la corrección de código en SuperCollider de GPT-4 y Bard, en el que se puede apreciar cómo el código generado por este último no se puede ejecutar por sus numerosos errores sintácticos, mientras que el de GPT-4 es correcto. En cuanto al modelo exacto de GPT-4, si bien este ha ido cambiando a lo largo del tiempo en el que se han realizado el trabajo, la versión más utilizada, tanto por coste como por optimización, ha sido gpt-4-1106-preview, más conocida como *GPT-4 turbo*.

A priori se han descartado los modelos open source, ya que demandan, en el momento del trabajo, una gran cantidad de recursos que quedan fuera del alcance de esta investigación. Por otra parte, la aplicación de estos modelos a la generación de código está en pleno desarrollo y, por lo general, aún quedan lejos de las capacidades de GPT-4.

Un segundo LLM disponible en ChatGPT, GPT-3.5, si bien es capaz de generar código mejor que el de Bard, no lo hace con la tasa de corrección de GPT-4. En los experimentos realizados esto se pudo comprobar constantemente: una gran parte del código generado en SuperCollider por GPT-3.5 contenía demasiados errores de sintaxis y alucinaciones, lo cual lo convertía en ineficiente para la creación de código musical al lado de GPT-4.

5.2 Rendimiento de ChatGPT en tareas de generación de código sonoro

Por la propia interfaz de chatbot, ChatGPT solo permite una interacción tipo *diálogo*. Esto es, el usuario introduce un texto, y el modelo responde a este texto. Esto determina fuertemente

Figura 5.1 (a) Respuesta de Bard y (b) ChatGPT a un mismo prompt: «Haz un código sencillo en SuperCollider, listo para ejecutarse, que exemplifique una frecuencia modulada». El código de Bard cuenta con varios errores graves de sintaxis, mientras el de ChatGPT es correcto.

```
// Declaramos las variables
float fm = 440;
float fms = 100;
float amp = 0.5;

// Generamos la señal FM
SinOsc s = SinOsc(fm);
s.freq =
s.freq * (1 + fms * SinOsc(fms).cos);

// Asignamos la señal a un bus
Out(s);
```

(a) Bard (con Palm 2)

```
( 
SynthDef(\fmSimple, {
|out = 0, freq = 440,
modIndex = 10,
modFreqRatio = 1|
var modulator =
SinOsc.ar(freq * modFreqRatio)
* modIndex;
var carrier =
SinOsc.ar(freq + modulator);
Out.ar(out, carrier ! 2);
}).add;
)

// Tocar el SynthDef
(
Synth(\fmSimple, [
\freq, 440,
\modIndex, 20,
\modFreqRatio, 0.5
]);
)
```

(b) ChatGPT (con GPT-4)

Fuente: Elaboración propia

el tipo de tareas que con él podemos realizar a la hora de crear código musical. No resulta práctico para la creación de código musical en tiempo real, ya que es el usuario quien debe cortar y pegar constantemente texto desde el IDE al chat y viceversa, lo cual entorpece el flujo de trabajo. Sin embargo, se ha visto muy útil para las tareas que enumeramos a continuación:

- Crear esbozos generales de una obra.
- Crear bloques de código sencillos.
- Poner a prueba ciertas técnicas de prompting, como COT.

5.2.1 Creación de esbozos generales de una obra

Una de las tareas más útiles que se ha encontrado para ChatGPT en relación con la composición musical es la de crear esbozos generales de una obra. Esto es, estructuras generales, como la forma temporal, junto con la estructura del código con las clases y funciones que eventualmente se van a usar. Se trata de una planificación previa a la composición de los detalles. En este punto es tentador pensar que ChatGPT podría producir la obra completa, pero esta ilusión cae rápidamente al intentarlo, ya que lo más probable es encontrar multitud de errores en el código generado, los cuales provienen de una mala planificación general y no merece la

pena depurar. Sin embargo, si el objetivo de una conversación es simplemente la de crear una estructura general, el resultado puede ser satisfactorio.

Para llevar a cabo una tarea así, se ha encontrado conveniente hacer que nuestro prompt pida al LLM realizar una planificación previa de la obra de forma razonada, siguiendo técnicas de prompting como COT (véase sección 2.3). En este sentido, se ha visto muy útil que el prompt contenga una serie de indicaciones que guíen al modelo en la creación de la estructura general de la obra. Por otra parte, la mayor parte de las veces que se le ha pedido este tipo de trabajo, lo ha hecho correctamente solo tras iterar ciertas indicaciones en la conversación. Por ejemplo, en el prompt de la Figura 5.2 se puede ver cómo se le pide al modelo que cree una estructura general de una obra. Su primera respuesta, a pesar de ser correcta desde el punto de vista del lenguaje, no responde en absoluto a la dimensión temporal pedida, que es la esencia de la estructura. Pero simplemente devolviéndole una pista de lo que se busca, su segunda respuesta se adecua perfectamente a lo pedido originalmente.

Ya desde el inicio de los experimentos se ha visto que el *feedback* del usuario experto en la tarea pedida es fundamental para no frustrar la tarea inmediatamente. Sin embargo, llama la atención que ChatGPT no comprenda la esencia de la petición primera, en el ejemplo dado, la de crear una estructura temporal esquemática, y que baste una pista para que la respuesta sea correcta. Este hecho apunta a que la iteración es la clave para obtener buenos resultados. Los LLM *razonan* mejor en conversaciones, donde pueden ir recibiendo *feedback* del usuario, que en prompts aislados.

Figura 5.2 Conversación con ChatGPT para crear una estructura general de una obra. El prompt (a) pide al modelo que cree una estructura general de una obra de unos 5 minutos de duración. La respuesta ignora la dimensión temporal pedida, por lo que se le da una pista de lo que se busca. La respuesta al siguiente prompt (b) se adecua perfectamente a lo pedido originalmente.

```
// Definición de SynthDef
para la introducción
SynthDef(\granularSynth, {
    !out = 0, freq = 440, dur = 11
    // Código para síntesis granular
}).add;

// Patrón para la sección de introducción
Pbind(
    \instrument, \granularSynth,
    \dur, Pwhite(0.1, 0.3),
    \freq, Pexprand(400, 500)
).play;
```

(a) Respuesta a Prompt 1: «Crea la estructura en SuperCollider de una obra de música electroacústica. La duración ha de ser de unos 5 minutos. El desarrollo temporal ha de ser la de una modificación constante de la textura sonora. Ayúdate de las técnicas de síntesis sonora que sean necesarias en cada caso. El código ha de ser modular. Los detalles se trabajarán después. Interesa un adecuado uso de las clases y placeholders de SuperCollider. Haz primero una planificación razonada.» Nota: se han eliminado los comentarios previos al código de las respuestas.

```
// Definición de secciones como funciones
~introducción = {
    // Código para la introducción
    "Iniciando introducción".postln;
    // Duración de la introducción
    60.wait;
};

~desarrollo = {
    "Iniciando desarrollo".postln;
    120.wait;
};

~climax = {
    "Iniciando clímax".postln;
    60.wait;
};

~conclusion = {
    "Iniciando conclusión".postln;
    60.wait;
};

// Secuenciador temporal con Routine
Routine({
    ~introducción.value;
    ~desarrollo.value;
    ~climax.value;
    ~conclusion.value;
}).play;
```

(b) Respuesta a Prompt 2: «Utiliza algún tipo de rutina para organizar el aspecto temporal.»

Fuente: Elaboración propia

Una vez que ChatGPT nos ha provisto de un esqueleto para una obra musical, ¿por qué no continuar la conversación hasta tener todas sus partes completas? Los intentos de crear códigos completos desde cero, paso a paso, han sido casi siempre frustrados en algún punto de la conversación. Llegados a una cierta cantidad de peticiones y respuestas, puede resultar desconcertante el olvido del LLM de ciertos aspectos importantes del trabajo realizado en la misma conversación hasta el momento y el que queda por completar.

Esto se debe a varios factores: En primer lugar, la ventana de contexto definitivamente limita la memoria del LLM (el número de tokens que recibe como input para la siguiente respuesta) de forma que los primeros prompts, precisamente en los que se le daban las directrices generales para una obra, quedan fuera de su alcance. Su comportamiento en ese momento es impredecible, y tiende a enfocarse en las últimas tareas como objetivo principal. Y debido a que el

usuario desconoce cuál es el valor de esta ventana de contexto, puede arrastrar el proyecto en la conversación hacia un punto sin salida. En segundo lugar, independientemente de la ventana de contexto, y especialmente en las ventanas grandes, se ha visto que los LLM no procesan por igual toda la información. Pueden existir lagunas importantes en el centro de la ventana, tal como se vio en la sección 2.2.3, lo cual contribuye a que una conversación lo suficientemente larga pueda derivar a un bloqueo o a un caos, especialmente cuando el usuario, como en el caso de los chatbots, desconoce el tamaño de la ventana en la conversación.

5.2.2 Creación de bloques de código sencillos

Nos referimos con *bloque sencillo* a una pieza de código funcional, pero que no constituye un proyecto completo, como, por ejemplo, una función, un patrón, etc., con unas pocas líneas de código. Al no constituir un código de una obra completa, no se requiere del manejo de una gran información en las respuestas del LLM, por lo que se puede esperar una mayor precisión en las mismas, al tiempo que se evitan los problemas asociados con la ventana de contexto. Del mismo modo, se ha comprobado que el usuario, a priori, puede controlar mejor así la conversación, iterando de una forma más controlada y cómoda.

Esta forma de trabajo, sin embargo, requiere de una buena planificación por parte del usuario, en la medida en la que es este quien dará unidad a los diferentes elementos del trabajo sonoro y musical. En este sentido, es difícil pensar en un buen rendimiento de este flujo de trabajo sin conocimientos sólidos de las herramientas de programación utilizadas. Si bien existen técnicas de uso de LLM que implican la interacción de varios modelos o conversaciones paralelas, al modo de *agentes*, en este trabajo no se ha explorado esta posibilidad debido a su complejidad y a la falta de técnicas y plataformas maduras que permitan su uso¹.

En general, en nuestros experimentos se ha encontrado que GPT-4 se ha desenvuelto con notable corrección en sus respuestas en el sentido sintáctico, mostrando una alta capacidad de autocorrección cuando se le ha devuelto un código de error, requiriendo poca o nula intervención humana en la corrección de errores. No obstante, como ocurría en el caso de la creación de esbozos generales de una obra, se han encontrado casos de bloqueo en la conversación, en los que el LLM no ha sido capaz de avanzar en la tarea sin la intervención humana.

El lenguaje utilizado para comunicarse con GPT-4 siempre ha sido, deliberadamente, el lenguaje natural, orientado a la creación sonora y musical, con el fin de encontrar así los límites de esta

¹Con relación a técnicas de agentes aplicadas a la generación de código, véase Huang, Bu, Zhang, Luck, y Cui (2023). Un repositorio con extensa y actualizada bibliografía sobre el tema puede encontrarse en AGI-Edgerunners/LLM-Agents-Papers (2024).

herramienta. Incluso en el caso de conversaciones en las que se devuelve código correcto y funcional, se han encontrado diversos problemas a nivel artístico, que hay que tener muy en cuenta a la hora de utilizar esta herramienta con fines musicales y que pasamos a detallar.

5.2.2.1 Alucinaciones recurrentes

Las alucinaciones en los LLM (véase la sección 2.4.2) son una de las características más conocidas de estos modelos, al tiempo que una de las más difíciles de detectar sin una supervisión humana, y, por ende, uno de los mayores handicaps a la hora de delegar tareas de cualquier índole a sistemas de IA. En el mejor de los casos estas alucinaciones se producen en el uso de clases o métodos inexistentes en el lenguaje de programación en uso. Normalmente estos defectos son fáciles de detectar por el compilador o por el intérprete del lenguaje, que suele dar cuenta de dicho problema inmediatamente. Por ejemplo, en el caso de SuperCollider, un código como el de la Figura 5.3 es correcto desde el punto de vista sintáctico, pero no lo es desde el punto de vista de la semántica del lenguaje, ya que utiliza la clase KarplusStrong, que no existe en SuperCollider. En este caso, el intérprete de SuperCollider dio buena cuenta de este error.

Figura 5.3 Alucinación de ChatGPT utilizando una clase que no existe ante el prompt «Crea una textura sonora en supercollider inspirándote en el modelado físico de sonido.» ChatGPT utiliza como base el UGen KarplusStrong, el cual no existe en SuperCollider.

```
// Crear una función para el sonido modelado físicamente
(
SynthDef(\ksTexture, {
    arg freq=440, amp=0.5, gate=1, pan=0;
    var env, sig;
    env = EnvGen.kr(Env.adsr(0.01, 0.1, 0.6, 0.4), gate, doneAction: 2);
    // Crear el sonido utilizando KarplusStrong
    sig = KarplusStrong.ar(freq, 0.5, 0.1, 1, env);
    sig = Pan2.ar(sig * amp, pan);
    Out.ar(0, sig);
}).add;
)
```

Fuente: Elaboración propia

Encontramos errores aún más sutiles y difíciles de detectar, ya que requieren de un conocimiento profundo del lenguaje de programación en uso y el intérprete no da cuenta de ellos, pudiendo pasar desapercibidos. Esto puede constituir un problema de seguridad. Un ejemplo claro es cuando la amplitud de la salida de audio satura el sistema. Los lenguajes de programación sonora no cuentan a priori con limitaciones de volumen o frecuencia, por lo que un error de este tipo puede provocar daños en el equipo de sonido o en el oído del usuario (véase Figura

5.4). Similar a este error es, en SuperCollider, la acumulación de instancias de un Synth en el servidor de audio, lo cual puede provocar un bloqueo del sistema. En la Figura 5.5 se puede ver un ejemplo de este tipo de error. En este caso, el usuario ha tenido que intervenir para corregir el código generado por el modelo. Junto al fenómeno de la alucinación, este tipo de problemas pueden pasar desapercibidos especialmente si no existe un entendimiento profundo del lenguaje por parte del usuario.

Figura 5.4 Código saturado en amplitud, generado por ChatGPT. El código es correcto desde el punto de vista sintáctico, pero la amplitud de la señal de audio satura el sistema. Este tipo de errores no son detectados hasta que se ejecuta el código, y son potencialmente peligrosos para el equipo de sonido o el oído del usuario.

```

(
{
    var x, y, z, a, b, c, d, e, f, g, h, mix;

    x = SinOsc.ar((Decay.ar(Impulse.ar([4,4.005]), 1e3*MouseX.kr(0, 1).abs)*50), MouseY
        .kr(0, 1)).distort;
    y = BPF.ar(Saw.ar([40,40.001]), LFN Noise0.kr(128)+1*4e3+146, LFN Noise1.kr(1)+1*5e
        -2+0.01).tanh;
    z = DelayC.ar(LPF.ar(LFN Noise0.ar(8)**2 + x.tanh.round(0.05), 6e3), 1, LFN Noise0.ar
        (8!2).range(1e-4, 0.02));
    a = BHiPass.ar(LFN Noise1.ar(8)**3, y.midicps, y/2e3, 67-y);
    b = PitchShift.ar(Saw.ar(Demand.kr(Impulse.kr(4), 0, Drand([-12, -7, 0, 7, 12]+33)
        .midicps, inf)), Decay.kr(Impulse.kr(4), 3)), 7, 2);
    c = Impulse.ar(8)*LFNoise1.ar(2);
    d = Limiter.ar(GVerb.ar(BPF.ar(c, [400, 800], 1/9, 50).tanh, 100, 10), 0.9);
    mix = Mix.new([x, y, z, a, b, c, d]);
    Out.ar(0, Splay.ar(mix));
}.play;
)

```

Fuente: Elaboración propia

Figura 5.5 Error difícil de detectar en SuperCollider: El Synth \dual_osc no cuenta con mecanismo de autoliberación, por lo que se acumulan ilimitadas instancias en el servidor de audio llamadas desde el bucle loop hasta saturar el sistema. Este código fue creado por ChatGPT y posteriormente autocorregido tras pasarle la información del error.

```

(
    SynthDef(\dual_osc, {
        arg out=0, freq1=440, freq2=660, amp1=0.5, amp2=0.5;
        var sig;
        sig = SinOsc.ar([freq1, freq2], 0, [amp1, amp2]);
        Out.ar(out, sig);
    }).add;
)
// --- código intermedio omitido por brevedad ---
{
    loop {
        var freq1, freq2, amp1, amp2;
        freq1 = freqPattern1.next * 400 + 100;
        freq2 = freqPattern2.next * 400 + 100;
        amp1 = ampPattern1.next;
        amp2 = ampPattern2.next;
        Synth(\dual_osc, [\freq1, freq1, \freq2, freq2, \amp1, amp1, \amp2, amp2]);
        0.1.wait;
    }
}.fork;
)

```

Fuente: Elaboración propia

5.2.2.2 Tendencia a valores por defecto

Otro defecto observado con relación a los parámetros de los métodos y clases del código generado es la propensión a utilizar valores por defecto. Esto, aunque no compromete la seguridad, resta interés al código generado desde el punto de vista artístico. Los valores por defecto son aquellos que un elemento del código utiliza cuando el usuario no provee de un valor explícito. Por ejemplo, en el caso de SuperCollider, el método `SinOsc.ar` cuenta con un parámetro `mul` que, si no se provee de un valor explícito, toma el valor por defecto de 1; o `freq`, con el valor por defecto de 440 Hz. Por ejemplo, en la Figura 5.6 se puede ver cómo el modelo utiliza valores por defecto en el código generado, lo cual no es deseable desde el punto de vista artístico.

Figura 5.6 Código generado por ChatGPT con utilización de valor de frecuencia de 440 Hz. Este valor por defecto no es una excepción, sino que sistemáticamente se encuentran en los parámetros de los métodos y clases del código generados por ChatGPT. Esto, sin duda, resta interés al código generado desde el punto de vista artístico.

```

(
    SynthDef(\fmSound, {
        arg freq=440, amp=0.5, modIndex=5, attackTime=0.01, releaseTime=1;
        var carrier, modulator, env;

        // Envolvente
        env = EnvGen.kr(Env.perc(attackTime, releaseTime), doneAction:2);

        // Modulador
        modulator = SinOsc.ar(freq, 0, modIndex);

        // Portadora
        carrier = SinOsc.ar(freq + modulator) * amp;

        // Salida
        Out.ar(0, carrier * env);
    }).add;

    // Toca el sonido
    Synth(\fmSound, [
        \freq, 440, // Frecuencia en Hz
        \amp, 0.5, // Amplitud
        \modIndex, 5, // Índice de modulación
        \attackTime, 0.01, // Tiempo de ataque en segundos
        \releaseTime, 1 // Tiempo de liberación en segundos
    ]);
)

```

Fuente: Elaboración propia

5.2.2.3 Introducción de errores de sintaxis en código previamente correcto

Aunque infrecuente, no menos desconcertante es la introducción impredecible de errores de cualquier tipo en código previamente correcto. Si los errores son de sintaxis, el compilador o el intérprete del lenguaje darán cuenta de ellos, pero si se trata de errores de semántica, como puede ocurrir en casos de alucinaciones, pueden pasar desapercibidos. La introducción de nuevos errores en código previo es un problema que aparece según la conversación, o el tamaño del código, es comparable al tamaño de la ventana de contexto. De nuevo, el nivel de conocimientos de usuario es fundamental para detectar este tipo de errores.

5.2.2.4 Ocasionales errores de sintaxis que escapan a la autocorrección

Una característica interesante de los LLM es la de la autocorrección. Esto es, si el usuario le devuelve un el mensaje de error dado por el sistema, el modelo intenta interpretarlo corregir el problema en la siguiente respuesta. Sin embargo, factores como la introducción de alucinaciones, problemas con la ventana de contexto o la simple carencia de información de en-

trenamiento en el problema actual del código, pueden hacer que el modelo no sea capaz de autocorregirse, pudiendo llevar a un bloqueo en la conversación, especialmente si esta se extiende demasiado con relación a su ventana de contexto, lo cual, como vimos, puede producir lagunas de información.

5.2.2.5 Gran distancia entre el sonido producido y el sonido esperado

El poder de los LLM es del procesar el lenguaje natural y traducir descripciones en código funcional. Si el código requerido es el de una función, digamos, en Python, que ha de «devolver el sumatorio de las entradas», el nivel de comprensión requerido al LLM en dicha transcripción es trivial, en tanto en cuanto la propia descripción es prácticamente un pseudocódigo de dicha función. El desnivel semántico no es comparable cuando lo que pedimos al LLM es complejo artísticamente, como, por ejemplo, «crear un sonido rugoso que evolucione temporalmente de una textura granular a un sonido armónico con leves y agradables fluctuaciones de sus componentes parciales». En este caso, el LLM ha de comprender la descripción, traducirla a código y, además, generar un sonido que se adecue a la descripción. En este caso, el desnivel semántico es mucho mayor, ya que la descripción responde a criterios estéticos y psicoacústicos, por lo que el resultado puede ser muy distinto al esperado. En la Figura 5.7 se puede ver un ejemplo de este fenómeno. Este problema puede ser mitigado en parte con el uso de técnicas de prompting y una continua iteración en la conversación, pero en muchos casos el resultado final el resultado aceptado por el usuario puede estar lejos el esperado, lo cual constituye un problema del que hay que ser conscientes a la hora de usar esta herramienta para la creación musical.

Figura 5.7 Desnivel semántico de ChatGPT entre el prompt y el código generado en SuperCollider. El código aparentemente representa un conjunto de sonidos naturales de un bosque, pero el resultado es ruido blanco de alto nivel de amplitud, totalmente alejado del significado semántico de la descripción.

```

(
    // Definir los componentes del paisaje sonoro
    var arroyo, pajaros, viento, hojas;

    // Arroyo: Ruido blanco filtrado
    arroyo = { BPZ2.ar(WhiteNoise.ar(0.3), 1000) };

    // Pájaros: Tonos agudos y aleatorios
    pajaros = { SinOsc.ar([4000, 4500].choose, 0, 0.1) * EnvGen.ar(Env.perc(0.01, 0.1),
        Impulse.kr([5, 7].choose)) };

    // Viento: Ruido rosa
    viento = { PinkNoise.ar(0.2) };

    // Hojas y Ramas: Ruido blanco con envolventes
    hojas = { WhiteNoise.ar(0.1) * EnvGen.ar(Env.perc(0.005, 0.1), Impulse.kr(10)) };

    // Mezclar y reproducir todos los componentes
    { arroyo!2 + pajaros!2 + viento!2 + hojas!2 }.play;
)

```

Fuente: Elaboración propia

5.2.3 Influjo del uso de técnicas de prompting en ChatGPT

En la sección 2.3 se han descrito algunas de las técnicas más relevantes en cuanto al prompting de un LLM, especialmente las que han presentado buenos resultados en aritmética y generación de código de programación, como COT. De igual modo, en 2.5.2 se presentaban técnicas avanzadas como sCOT en la generación de código.

A continuación se analiza, de forma cualitativa y descriptiva, el impacto que han tenido diferentes técnicas de prompting en las respuestas obtenidas del modelo.

5.2.3.1 Zero-shot versus few-shot

Aunque ChatGPT está entrenado para establecer conversaciones tipo chat, es susceptible de recibir contextos junto a la petición del usuario con un conjunto de ejemplos de código, para que el modelo pueda aprender de ellos para elaborar su respuesta (véase sección 2.3.1.1). En este sentido, se ha encontrado que la presentación de ejemplos de código al modelo tiene un efecto positivo en cuanto estilo y formato del código, incluso en el uso de elementos de programación similares a dichos ejemplos, haciéndolo menos proclive a errores sintácticos. Sin embargo, no se ha hallado que el uso de ejemplos de código mejore sustancialmente la calidad del código generado en el sentido artístico.

La Figura 5.8 muestra el resultado de una misma petición a ChatGPT con *zero-shot* y *one-shot*. El resultado es claramente positivo en el segundo caso. Como veremos, este resultado es clave para el éxito en la utilización de técnicas más avanzadas, como RAG.

5.2.3.2 *Chain of Thoughts* (CoT) y *Structured Chain of Thoughts* (SCoT)

Esta técnica requiere de un nivel mayor de abstracción en el razonamiento del LLM, al exigirle una planificación previa a la implementación del código. Esto implica la elaboración de la respuesta en dos pasos, una para la planificación y razonamiento, y otra, para la implementación en sí. Dichos pasos pueden ser dados en una sola respuesta o en dos. En el caso de ChatGPT, o el de cualquier chat de IA, a cuyos parámetros de respuesta no podemos acceder, conviene hacerlo en dos respuestas consecutivas para asegurarnos de no alcanzar un límite de tokens en la única respuesta.

A diferencia de *few-shot*, esta técnica no mejora la respuesta del LLM de forma drástica, pero sí que se aprecian cambios positivos en cuanto a la complejidad de elementos que el código puede llegar a utilizar. Así, una petición compleja del usuario tiene más probabilidades de éxito si el LLM tiene la posibilidad de *razonar* previamente los pasos a dar que si lo implementa directamente. A cambio, puede introducir más errores en el código, que el usuario ha de corregir por sí mismo o llevar al LLM en una conversación iterada. Esta técnica es más apropiada que *few-shot* en el caso de buscar un código para una pieza en evolución temporal, donde se implican estructuras tipo rutinas, patrones y cambios de parámetros o sintetizadores a lo largo del tiempo, aunque, será útil en cualquier aplicación en la que se busque complejidad. En los experimentos realizados se han desecharido muchas conversaciones completas por llegar a momentos de bloqueo, en los que la complejidad del código era tal que su depuración era más costosa que la elaboración del mismo a partir de partes más pequeñas.

No se ha mostrado como relevante el uso de una técnica tipo sCOT, en la que el primer paso ha de realizarse en pseudocódigo en lugar de en lenguaje natural. La hipótesis que lanzamos para este resultado es que, a diferencia de las tareas de programación a las que esta técnica originalmente se dirige, el diseño de sintetizadores no requiere de tanta complejidad como una función en Python o C++ que se descompone en flujos como bucles y condicionales. En este sentido, el propio script en SuperCollider se representa a sí mismo y no tiene sentido planificar en un pseudocódigo.

Figura 5.8 Comparación de resultados de ChatGPT con (a) zero-shot y (b) few-shot.

Usuario: Crea una textura sonora de interés artístico usando el UGen FMGrain en SuperCollider.

```
{
    var dur, dens, fund, form, indx, pan;
    dur = MouseX.kr(0.01, 0.2, 1, 0.01);
    dens = MouseY.kr(20, 50, 1, 0.1);
    fund = 200;
    form = 800;
    indx = 2;
    pan = LFN noise2.kr(0.5).range(-1, 1);

    FMGrain.ar(
        numChannels: 2,
        trigger: Impulse.kr(dens),
        dur: dur,
        carfreq: fund,
        modfreq: form,
        index: indx,
        pan: pan
    )
}.play;
```

(a) Zero-shot en una petición de creación sonora en SuperCollider.



Usuario: Crea una textura sonora de interés artístico usando FMGrain en SuperCollider. Te paso un ejemplo de uso:

```
SynthDef(\fm_grain_test, {arg gate = 1,
    amp = 1;
    Out.ar(0,
        FMGrain.ar(Impulse.ar(20), 0.2,
        440, 200,
        LFN noise1.kr(1).range(1, 10)
        ,
        EnvGen.kr(
            Env([0, 1, 0], [1, 1],
                \sin, 1),
            gate,
            levelScale: amp,
            doneAction: 2
        )
    )
}).add;
s.sendMsg(\s_new, \fm_grain_test, a=s.
nextNodeID, 0, 1, \amp, 0.1);
```

```
SynthDef(\fm_grain_artistic, { arg gate =
1, amp = 1;
var sound, freq, dur, pan, rate, index;

// Parámetros modificables para la
// textura
freq = LFN noise1.kr(0.5).range(200,
1000);
dur = LFN noise2.kr(0.1).range(0.1, 0.5);
pan = LFN noise2.kr(0.2).range(-1, 1);
rate = LFN noise1.kr(0.3).range(5, 50);
index = LFN noise1.kr(0.4).range(100,
800);

// Creación de la textura sonora
sound = FMGrain.ar(
    Impulse.ar(rate), dur, freqindex,
    5, pan
);

// Aplicación del envolvente y salida
Out.ar(0,
    Pan2.ar(
        sound * EnvGen.kr(
            Env([0, 1, 0], [0.1, 0.1],
                \sin, 1),
            gate,
            levelScale: amp,
            doneAction: 2
        ),
        pan
    )
);
}).add;
s.sendMsg(\s_new, \fm_grain_artistic, a=s.
nextNodeID, 0, 1, \amp, 0.1);
```

(b) La misma petición con un ejemplo Few-shot. El resultado sonoro es más interesante desde el punto de vista artístico.



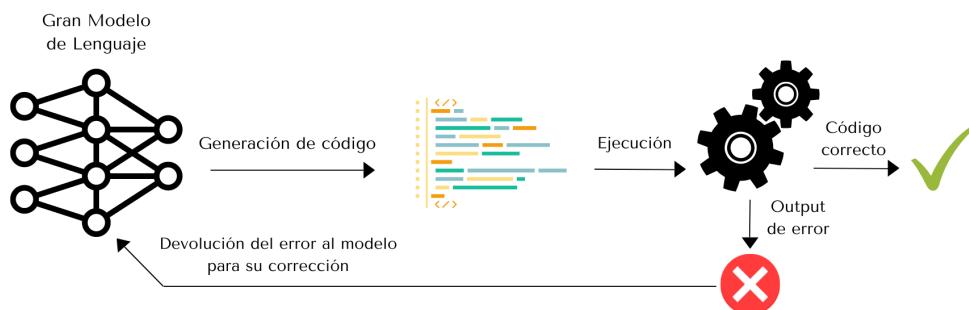
Fuente: Elaboración propia

5.2.3.3 Técnica de *self-debugging*

Una aproximación para intentar minimizar los errores sintácticos del código generado podría ser el ya expuesto *self-debugging*, en el que el LLM ha de dar un paso posterior a la implementación en código para revisar los posibles errores del código. En ninguno de los casos que se ha utilizado se han corregido errores algunos. La única forma de que el LLM corrija sus errores es que el usuario le devuelva el mensaje de error del compilador o del intérprete del lenguaje, en cuyo caso el LLM comprenderá el problema y tratará de resolverlo en la siguiente respuesta.

De hecho, este es un flujo habitual en el que se desenvuelve las conversaciones de creación sonora con ChatGPT. La inmensa mayoría de los bloques de código generados con éxito en este trabajo, han sido realizados de forma iterativa, limpiando el código a partir de los mensajes de error devueltos por el intérprete de SuperCollider. En este punto, el papel del usuario compositor y su nivel de conocimientos es fundamental para el éxito de la tarea. La Figura 5.9 muestra este proceso en bucle en el que el usuario devuelve al LLM los mensajes de error iterativamente.

Figura 5.9 Iteración de depuración de código con ChatGPT. El usuario devuelve al LLM los mensajes de error iterativamente.



Fuente: Elaboración propia

5.3 Mayor control de parámetros con el Playground de OpenAI

El Playground de OpenAI consiste en una interfaz web, similar a ChatGPT, en la cual, entre otras cosas, se pueden controlar importantes parámetros como la temperatura, la longitud de la respuesta, la ventana de contexto, top-*k* y top-*p*.

En la práctica, se ha visto que no conviene modificar la temperatura más allá de su valor por defecto 1. Incluso valores ligeramente altos producen efectos negativos en la generación de código, especialmente errores sintácticos y alucinaciones. Valores menores son posibles, aunque estos se tornan demasiado predecibles, y esto puede ser un problema si se busca variedad en las respuestas.

Una ventaja práctica que se ha encontrado en la interfaz de Playground es la posibilidad de escribir un prompt de sistema. Es decir, un prompt general que siempre antecede a los mensajes del usuario. Estos prompts de sistema tienen como finalidad contextualizar la conversación y el papel que el LLM tendrá dentro de ella. Estos prompts, redactados en lenguaje natural, son una suerte de instrucciones que el usuario puede escribir una sola vez, pero que se espera que el LLM respete en la conversación. Se ha notado, no obstante, que no siempre son respetadas estas instrucciones, y que prompts de sistema demasiado largos pueden caer en los defectos asociados a la ventana de contexto ya señalados en la sección 2.2.3. La Figura 5.10 muestra un ejemplo de prompt de sistema utilizado en los experimentos, en el que se le pide GPT-4 que genere código en SuperCollider. Más ejemplos de prompts de sistema utilizados en este trabajo se pueden encontrar en la Figura 6.3.

Figura 5.10 *Ejemplo de prompt de sistema para generar código en SuperCollider. Este prompt ha sido construido iterativamente según las necesidades de la conversación. Muchas instrucciones corresponden a defectos que el LLM ha ido mostrando previamente.*

Eres un experto en SuperCollider. Cada vez que se te pida, debes devolver una sentencia de SuperCollider (SC). Una sentencia puede ser una sola línea o un conjunto de líneas dentro de un bloque. A cada petición, responderás con una única sentencia que continúe el contexto proporcionado por la conversación. Si un usuario reporta un error en tu sentencia, debes corregirla. Importante: Utiliza Pbinddef (no Pbind) como base para la creación musical.

Evita usar el Synth default. En su lugar, crea tus propios SynthDef, asegurándote de: 1. Evitar valores predeterminados (como "freq=440") y 2. Que los SynthDef se autoliberen con "doneAction: 2". Crea patrones, ritmos, texturas, planos sonoros y espacializaciones experimentales. Utiliza siempre la sintaxis correcta de SC y no incluyas texto o comentarios, ya que no serán procesados.

Organiza tus respuestas de la siguiente manera: primero s.boot; luego, define SynthDefs; después, trabaja con Pbinds; y finalmente, termina con s.freeAll y s.quit. Evita bloques de código largos.

Fuente: Elaboración propia

5.4 Ampliando el Knowledge: retrieval-augmented generation

OpenAI provee de las herramientas necesarias para implementar sistemas de RAG (véase sección 2.3.2), bajo el nombre de *GPTs* y *Assistant*, lo cual fue explorado en el transcurso de la investigación. Básicamente, se ha de proveer al sistema de un prompt de sistema y de un conjunto de archivos sobre los que pueda realizar consultas.

Para ello, se elaboró una serie de archivos de texto plano a partir de la documentación oficial de las clases de SuperCollider y su guía práctica de uso de *patterns* (*SuperCollider 3.12.2 Help*, s.f.)² convertido a formato JSON para facilitar su lectura al sistema. Otro archivo incluido, esta vez en PDF, es el manual *A Gentle Introduction To SuperCollider* de Ruviaro (2015). Para el caso de los trabajos con el lenguaje de Tidal Cycles, se prepararon en archivos de texto los ejemplos de los cursos disponibles en la web oficial del lenguaje (*Tidal Cycles*, s.f.).

Por lo general, no se ha encontrado una gran diferencia de calidad en el código generado con RAG. En los sistemas dirigidos a Tidal Cycles se observó que los códigos generados estaban inspirados en ejemplos reales de código, con las transformaciones pertinentes, aunque probablemente se deba a que los experimentos realizados en Tidal Cycles fueron realizados en el contexto de *live coding*, como se verá, donde el sistema podía tomar al azar códigos para su contexto. Por otra parte, los archivos utilizados para Tidal Cycles contenían básicamente ejemplos de código con un comentario cada uno, mientras que los de SuperCollider contenían más texto, lo cual puede haber influido en el resultado.

5.5 Programando con la API de OpenAI

La comunicación por API con los servicios, en este caso, de OpenAI, constituye la forma más dúctil de interactuar con los modelos de IA. En este caso, se provee de una amplia y completa documentación y de librerías para su utilización dentro de diversos lenguajes de programación. En nuestro caso, el lenguaje utilizado es Python, por la sencillez de su uso y por ser un lenguaje robusto en el campo de la IA y de la investigación. El entorno de programación fue durante toda la investigación Visual Studio Code. Se crearon principalmente dos programas para la interacción con la API de OpenAI: uno para la generación automática de código en Tidal Cycles y otro para la generación interactiva de código tanto en Tidal Cycles como en SuperCollider.

5.5.1 Generación automática de código en Tidal Cycles

La API, además de interacciones del mismo tipo que ChatGPT o que la Playground, permite la interacción programada de peticiones y la gestión de las respuestas de forma automatizada. Uno de los primeros scripts que se crearon en este trabajo realizaba peticiones cíclicamente a la API. El código solicitado era el de un patrón de Tidal Cycles por medio de una petición de prompt de sistema. Una vez obtenido, el patrón es enviado al generador de sonido de Tidal en local y, tras un tiempo determinado, se vuelve a comenzar el ciclo enviando una nueva petición que

²Los archivos generados ad hoc para estos sistemas se encuentran en el repositorio del trabajo (véase el anexo B)

incluye tanto el prompt de sistema como los patrones que se han generado hasta el momento como contexto.

Una parte de estos patrones generados contienen errores en el código, normalmente por la utilización de objetos o clases inexistentes. En estos casos, como ya se ha mencionado, el error puede pasar desapercibido y sin consecuencias graves en la salida del sonido. Tidal no para la ejecución de los patrones previos aunque el último ejecutado contenga errores. El resultado sonoro del script es el de una evolución temporal de patrones, en forma de deriva, que solo puede ser controlado, en parte, por el usuario desde el prompt de sistema. El resultado es similar al de un intérprete de *live coding* escribiendo código musical en vivo.

Los resultados obtenidos con esta forma de interacción mostraron una notable diferencia tanto en lo cuantitativo como en lo cualitativo respecto a técnicas previas. Por una parte, el flujo continuo de peticiones a la API permite obtener una cantidad de código mayor que con las interfaces tipo chatbot, lo cual se muestra útil a la hora de buscar variedad y cierta serendipia en la búsqueda artística de sonidos y texturas. Por otra parte, la posibilidad de ejecutar directamente el código generado abre la puerta a la experimentación en tiempo real y abre ilimitadas posibilidades en el campo de la música en vivo.

Una decisión que resultó muy productiva es la de utilizar el lenguaje de Tidal en lugar del de SuperCollider. Además de la fácil y silenciosa gestión de errores de Tidal, por tratarse de un lenguaje expresamente diseñado para la creación musical en tiempo real, su código resulta potencialmente menos peligroso que el de SuperCollider, como también se señaló más arriba.

Las limitaciones de esta forma de interacción se encuentran en la forma de implementación de la interacción con el usuario, que, en este caso, se ha visto reducido absolutamente al prompt inicial. Esto provoca que no exista dominio temporal del flujo. Esta situación se puede solucionar, en parte, programando diversos prompts de sistema para ser lanzados en momentos determinados de la ejecución, aunque esto no fue explorado lo suficiente.

Precisamente, para posibilitar un control mayor por parte del usuario, se creó un segundo programa que permitiera una interacción mucho más directa y abierta, en términos de posibilidades, con la API. Debido a la importancia de este programa y su complejidad, se le dedicará un apartado propio (véase capítulo 6).

6. *AI Muse, herramienta en Python para la interacción en vivo con LLM en entornos de live coding*

ONE MODEL IS ALL YOU NEED

— Graham y cols. (2022)

En este capítulo pasamos a describir el programa *AI Muse*, creado durante esta investigación, al cual nos referimos en el capítulo anterior (véase sección 5.5.1), detallando su funcionamiento, diversas cuestiones de su implementación y los resultados de su utilización.

6.1 Descripción general de la herramienta

AI Muse es una herramienta de interacción con modelos de lenguaje en entornos de *live coding*. Está escrita en Python y se comunica con LLM de OpenAI a través de su API. Permite ejecutar código de SuperCollider y de Tidal Cycles. Además, tiene un archivo de configuración en formato JSON, así como un conjunto de comandos para modificar en vivo los parámetros de la configuración. Aunque está pensada para la utilización de modelos de OpenAI, esta librería permite eventualmente comunicarse con otros modelos de lenguaje incluso en local, detalle que no ha sido explorado en este trabajo. El paradigma de interacción es el del *live coding*.

6.2 ¿Por qué un entorno de live coding?

El *live coding* es el arte de programar música en tiempo real, constituyéndose así como un espacio de experimentación y creación artística que fusiona el campo de la programación con el de la interpretación musical y la improvisación. Por su naturaleza, esta práctica brinda una plataforma ideal para explorar las posibilidades que la interacción con modelos de lenguaje de forma automatizada puede ofrecer.

Dado su carácter improvisatorio, no se limita a estructuras musicales preestablecidas. Esta flexibilidad en la forma implica menores restricciones temporales en comparación con la composición musical tradicional, lo cual se alinea perfectamente con los hallazgos de fases previas de nuestro trabajo. En dichas etapas, se evidenció que los modelos de lenguaje contemporáneos son más eficientes con fragmentos de código de tamaño reducido. Este enfoque creativo, su-

mado a las ventajas de Tidal Cycles en términos de fiabilidad del código y tolerancia a fallos, ha propiciado el desarrollo de *AI Muse* como una herramienta de experimentación.

6.3 Integración de la API de OpenAI

AI Muse integra dos posibles interacciones que OpenAI permite con su API: utilización modelos de chatbot y de sistemas de RAG¹. La primera modalidad es la más sencilla, ya que se trata de realizar conversaciones con el modelo. En este caso, cada petición se limita a un prompt general de sistema unido al archivo actual con todo el código generado, de forma que el modelo pueda generar el siguiente bloque de código.

La diferencia de este programa respecto a scripts previos en este trabajo (véase sección 5.5.1) es que es el propio usuario quien decide cuándo hacer una petición a la API, en lugar de estar automatizado. La acción elegida para hacer esta petición es la de guardar el archivo. De esta forma, el programa puede ser utilizado en cualquier editor de textos, y no está limitado a un entorno de desarrollo concreto ni al uso de *plugins* o extensiones. En este caso, todo el programa ha sido escrito y testado en Visual Studio Code, pero eventualmente podría ser utilizado en cualquier otro editor de textos.

6.4 Lenguajes de programación musical disponibles

En un primer momento se implementó únicamente la posibilidad de ejecutar código de Tidal Cycles. Posteriormente, se añadió el soporte para SuperCollider, aprovechando las clases que este lenguaje ofrece para el *live coding*. En ambos casos se ofrece la posibilidad de que sea el propio programa el que se ocupe de llamar a los binarios de los propios lenguajes, o que este código sea ejecutado por el usuario por otros medios (uso de *plugins* o de entornos de desarrollo específicos).

6.5 Flujo de trabajo

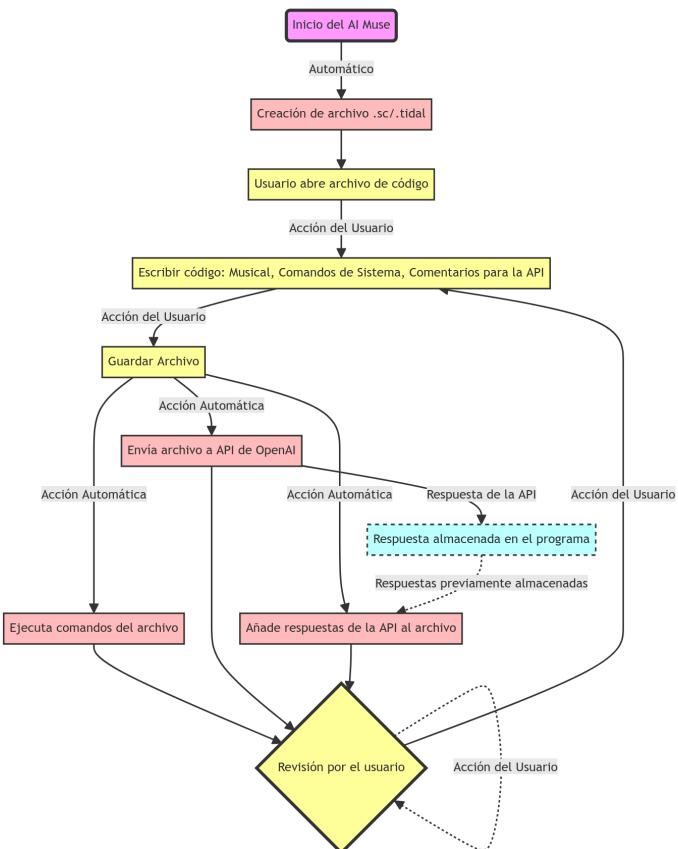
El flujo de trabajo típico de *AI Muse* es el siguiente (cfr. Figura 6.1):

1. El usuario ejecuta el programa. Automáticamente se creará un archivo en texto plano con la extensión adecuada (actualmente, .tidal o .sc, para Tidal Cycles y SuperCollider, respectivamente) para interactuar.

¹Los llamados *Assistants* por OpenAI.

2. El usuario abre un archivo de código en un editor de texto.
3. El usuario puede escribir código musical, un comando de sistema o un simple comentario-mensaje para el modelo.
4. Al guardar el archivo, *AI Muse* lee el archivo y lo envía a la API de OpenAI. Adicionalmente, ejecuta los comandos nuevos escritos en el archivo de código.
5. La API de OpenAI devuelve una respuesta con el siguiente bloque de código que es añadido automáticamente al archivo actual en la siguiente acción de guardar archivo.
6. El usuario ejecuta el nuevo bloque de código a voluntad, pudiendo modificarlo si lo cree oportuno.
7. El flujo continúa en el paso 3.

Figura 6.1 Flujo de trabajo de *AI Muse*.



Fuente: Elaboración propia

6.6 Archivo de configuración y comandos

La Figura 6.2 muestra una configuración concreta para el archivo config.json. Este archivo, en formato JSON, será cargado al ejecutar el programa y contiene toda la información relevante en cuanto al lenguaje musical, modelo, hiperparámetros y otras opciones referentes a la propia aplicación, como el tiempo que ha de pasar entre dos peticiones. La Tabla 6.1 muestra una breve descripción de estos parámetros, así como su traducción a comandos dentro de la propia aplicación.

Tabla 6.1: Descripción de la configuración del archivo config.json y de sus correspondientes comandos.

| Clave | Valores posibles | Efecto | Ejemplo de uso en comando |
|----------------------------|-------------------------|---|-----------------------------|
| mode_tidal_supercollider | tidal/sclang | Modo de operación para TidalCycles o SuperCollider | set mode tidal |
| create_log_file | true/false | Activa o desactiva la creación de archivos de registro | - |
| ghci_path | ./ruta/a/ghci | Ruta al ejecutable ghci | - |
| only_system_commands | true/false | Restringe a comandos del sistema | - |
| superollider_on | true/false | Enciende o apaga SuperCollider | - |
| sclang_path | ./ruta/a/sclang | Ruta al ejecutable sclang | - |
| boot_tidal_path | /ruta/a/BootTidal.hs | Ruta al archivo de arranque de Tidal | - |
| api_enabled | true/false | Activa o desactiva la API | set api_enabled true |
| bot_mode | assistant/chat | Establece el modo del bot | set bot mode chat |
| assistant_id | ID | ID del asistente | - |
| assistant_retrieval_folder | ./ruta/a/carpetas | Ruta a la carpeta de recuperación | - |
| api_key_file | apikey.txt | Archivo con la clave de la API | - |
| system_prompt_file | ./ruta/a/system_prompts | Archivo de prompt del sistema | - |
| model | gpt-4-1106-preview/otro | Modelo de lenguaje utilizado | set model gpt-4 |
| temperature | 0.0-2.0 | Temperatura para la generación de texto | set temperature 0.7 |
| max_tokens | Número entero | Número máximo de tokens por generación | set max tokens 256 |
| top_p | 0.0-1.0 | Ajusta la probabilidad de tomar caminos menos probables | set top_p 0.9 |
| frequency_penalty | Número | Penalización por frecuencia de tokens | set frequency penalty 0.5 |
| presence_penalty | Número | Penalización por presencia de tokens | set presence penalty 0.5 |
| wait_time_before_api | Segundos | Tiempo de espera antes de llamar a la API | set wait time before api 10 |
| wait_time_after_api | Segundos | Tiempo de espera después de llamar a la API | set wait time after api 10 |

Fuente: Elaboración propia

Figura 6.2 Ejemplo de archivo de configuración config.json de AI Muse. Los valores de cada clave pueden ser modificados directamente en el archivo o por medio de comandos. En este caso, el programa creará música en Tidal Cycles y utiliza un Assistant (RAG) previamente creado con una ID determinada. El modelo es GPT-4. El número de tokens por petición-respuesta es de 256.

```
{
    "mode_tidal_supercollider": "tidal",
    "create_log_file": false,
    "ghci_path": "ghci",
    "only_system_commands": false,
    "superollider_on": false,
    "sclang_path": "sclang",
    "boot_tidal_path": "/usr/share/haskell-tidal/BootTidal.hs",
    "api_enabled": false,
    "bot_mode": "assistant",
    "assistant_id": "asst_9Km6Fd3xywD4tGGY4yZzSEiw",
    "assistant_retrieval_folder": "./assistants/tidal_livecoding/retrieval_files/",
    "api_key_file": "apikey.txt",
    "system_prompt_file": "./assistants/tidal_livecoding/system_prompts/
        system_prompt_01",
    "model": "gpt-4-1106-preview",
    "temperature": 1,
    "max_tokens": 256,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0,
    "wait_time_before_api": 20,
    "wait_time_after_api": 20
}
```

Fuente: Elaboración propia

6.7 Resultados del uso de AI Muse

Como era de esperar, la automatización de los diálogos con el sistema del LLM permitió una interacción más fluida que en etapas previas, así como la concentración de la atención del usuario únicamente en cuestiones musicales. La música se produce ahora de forma sincrónica a la interacción usuario-LLM, lo cual se llega a percibir como una auténtica colaboración humano-máquina. Máxime cuando es posible comunicarse con comentarios en lenguaje natural durante el acto performativo.

Especial cuidado se puso en los prompts de sistema, ya que estos deben compendiar en el mínimo de tokens la finalidad de la conversación con el LLM, así como el formato esperado de sus respuestas para ser procesadas adecuadamente por el programa. En la Figura 6.3 se muestran sendos prompts de sistema para Tidal Cycles y para SuperCollider, los cuales fueron utilizados con éxito en la práctica.

Figura 6.3 Prompts de sistema de AI Muse para (a) SuperCollider y (b) Tidal Cycles. Estos prompts están diseñados para que el LLM responda únicamente con código musical breve a partir del contexto del archivo completo y sus comentarios.

Cada vez que se te escriba, debes devolver una sentencia de SuperCollider, o modificar una ya existente en el código.

Responderás con una única sentencia que continúe el contexto proporcionado por la conversación, a modo de sesión de live coding.

No incluyas texto o comentarios.

Técnicamente, tus patrones no han de ser complejos, para asegurar que no contienen bugs.

Artísticamente, se busca en tus patrones: imaginación, patrones atípicos, texturas sin explorar, etc. No se espera de ti: repetición de códigos típicos. Lo que se oye ha de ser poco percusivo, no ha de recordar a una batería. Es música experimental y ambiental. Juega con sonidos no reconocibles, transformados como solo tú sabes hacer...

Busca que su resultado sonoro sea complejo.

Ten en cuenta que el resultado sonoro sea audible psicoacústicamente, especialmente si modificas mucho el sonido.

El usuario te pasa el archivo actual.

Tus bloques o sentencias del código llevan un comentario con el nombre del modelo que lo ha creado para que los reconozcas (tú no tienes que ponerlo, los indica el usuario).

Si modificas un patrón existente, cambialo solo un poco, como haría un livecoder.

Utiliza sólo estructuras de live coding: Pbinddef, Ndef, y objetos modificables en caliente. Crea tus SynthDefs.

Cuestiones técnicas: Si creas o modificas un SynthDef, no olvides añadir condición de terminación.

Devuelve un único bloque o sentencia.

(a) *Prompt de sistema para SuperCollider*

Cada vez que se escriba, debes devolver una sentencia de patrones de Tidal Cycles o modificación de uno existente.

Responderás con una única sentencia que continúe el contexto proporcionado por la conversación, a modo de sesión de Live Coding.

No incluyas texto o comentarios.

Técnicamente, tus patrones no han de ser complejos, para asegurar que no contienen bugs.

Artísticamente, se busca en tus patrones: imaginación, patrones atípicos, texturas sin explorar, etc. No se espera de ti: repetición de códigos típicos. Lo que se oye ha de ser poco percusivo, no ha de recordar a una batería. Es música experimental y ambiental. Juega con sonidos estirados, llenos de eventos... transformados como solo tú sabes hacer...

Busca que su resultado sonoro sea complejo.

Ten en cuenta que el resultado sonoro sea audible psicoacústicamente, especialmente si modificas mucho el sonido.

El usuario te pasa el archivo actual.

Tú te ocuparás de algún patrón aún no creado (d1, d2, d3...), excepto si el usuario te pide que modifiques uno existente. Una vez existan 3 patrones tuyos, no crees más. Solo modifica los existentes.

Si modificas un patrón existente, cámbialo solo un poco, como haría un livecoder.

Cuestiones técnicas: no uses valores altos de room o sz, que pueden dar lugar a problemas de feedback.

Devuelve un único patrón. No uses “” para formatear código.

(b) *Prompt de sistema para Tidal Cycles*

Fuente: Elaboración propia

Se ha detectado una tenencia del LLM a imitar los patrones ya existentes en la conversación, si bien, por medio de comentarios es posible provocar más originalidad en sucesivas respuestas. Por esta razón, es interesante aplicar la técnica de prompting de *few-shot* para dirigir el inicio de la improvisación colaborativa.

No se ha hallado, sin embargo, una diferencia importante entre usar el modelo en modo `chat` o en modo `assistant`. No obstante, esto debería estudiarse con mayor detenimiento en función de unas cuidadas configuraciones de asistentes y de sus archivos de base de datos.

Este tipo de interacción se ha mostrado, en todo caso, muy prometedora de cara a la generación musical interactiva. Las aportaciones atómicas que se piden al LLM por medio de la API lo hacen muy adecuado como banco de pruebas tanto con los servicios de OpenAI, como eventualmente otros modelos, incluso open source.

7. *AlgorAI*: creación de una pieza de arte sonoro generado con ayuda de GPT-4

IMAGINATION IS ALL YOU NEED!

— Erker, Schaffer, y Spanakis (2023)



Como consecuencia de las pruebas y prácticas llevadas a cabo con la API y el Playground de OpenAI para este trabajo, se generaron decenas de archivos con código cuyo resultado sonoro fue juzgado de cierto interés, independientemente de su adecuación a los requerimientos pasados por el prompt. Estos bloques de código están escritos tanto en SuperCollider como en Tidal Cycles. En el primer caso responden a peticiones muy concretas de texturas, timbres o sucesiones sonoras, en el Playground; en el segundo, a sesiones de *live coding* compartidas, asistidas por la API en *AlgorAI*.

Ninguna de las construcciones de código conseguidas responden a un plan de una pieza completa, sino, más bien, a texturas sonoras continuas o con pocas variaciones en el tiempo. Por esta razón, el planteamiento de composición de una pieza a partir de estos materiales ha sido organizado absolutamente por el compositor-investigador. En este caso, se ha optado por organizar los sonidos elegidos en la DAW *Ardour*. Los procesados y efectos aplicados han sido mínimos, con el fin de mantener la esencia de los sonidos generados por GPT-4, limitándose a filtrados, cortes y compensación dinámica entre elementos.

7.1 Proceso y criterios de selección de material sonoro

La selección de los fragmentos de código a utilizar en la composición se realizó en un momento distinto de la propia generación, hasta dos meses más tarde en algunos casos, sin atender a las intenciones primeras a la hora de dialogar con la API ni a su grado de adecuación a las peticiones. Se buscó, más bien, la variedad de texturas y timbres, así como la posibilidad de combinarlos entre sí en un discurso sonoro coherente y equilibrado. La búsqueda se realizó

escuchando bloque por bloque y renderizando a archivos de audio una o varias versiones de los mismos, ya que en ocasiones el resultado sonoro dependía de factores externos al código como la posición o el movimiento del ratón en los ejes *x* e *y* para modular ciertos parámetros del código.

En el momento de la selección, se intentó diseñar una estructura base esquemática con la ayuda de GPT-4, pero siempre aparecían problemas asociados al entendimiento de los procesos en el tiempo por parte del LLM. Todas las soluciones propuestas por el LLM eran extremadamente simples en el mejor de los casos, y casi siempre lejos de lo que se pretendía, a saber, una composición temporal a partir de samples. Por esta razón, se optó por la intervención humana absoluta en el proceso de composición, relegando el papel de GPT-4 a la generación de material sonoro¹.

7.2 Descripción de la pieza

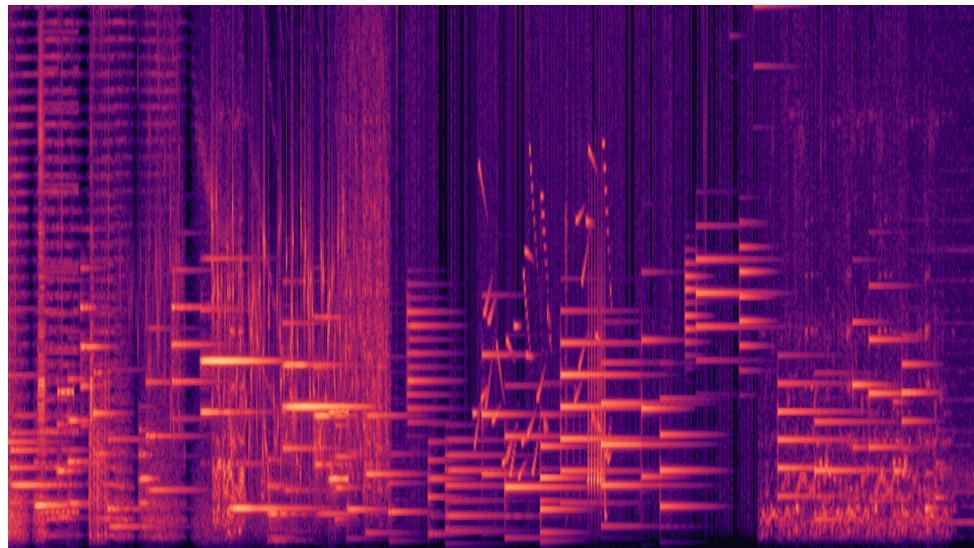
*AlgorAI*² es una obra de música electroacústica cuyos materiales provienen de código de programación. Su duración es de 5 minutos y está compuesta de un único movimiento, sin secciones claramente delimitadas. Su trama está compuesta por una sucesión polifónica de texturas sonoras de diversa índole, que se van superponiendo y combinando entre sí (véase una sección del sonograma de la Figura 7.1). Las decisiones de posición, duración y altura de cada sonido han sido tomadas en conjunto por el propio compositor, atendiendo a criterios subjetivos de equilibrio, coherencia e inteligibilidad desde el punto de vista psicoacústico. Su título, *AlgorAI*, es un acrónimo de *Algoritmo e Inteligencia Artificial*, y hace referencia a la naturaleza de la obra, cuya materia prima sonora ha sido compuesta a partir de algoritmos de IA, en concreto, por grandes modelos de lenguaje. Al mismo tiempo, hace un guiño al término *Algorave*, asociado a la práctica del *live coding*³. Los materiales sonoros son en su mayoría de origen íntegramente electrónico (aquellos producidos a través de SuperCollider), y en menor medida, de origen electroacústico (los producidos por Tidal Cycles), cuando la base de los mismos es un conjunto de samples de sonidos reales.

¹El anexo A contiene los materiales sonoros elegidos para esta composición.

²La pieza puede ser escuchada en los enlaces del código QR de la Figura 7.1

³Según Wikipedia, «Una *algorave* (de *algoritmo* y *rave*) es un evento en el que la gente baila con música generada a partir de algoritmos, a menudo utilizando técnicas de codificación en vivo» (“Algorave”, 2023).

Figura 7.1 Sonograma de una parte de la pieza electroacústica AlgorAI. Se puede apreciar parte de la trama polifónica creada por los diversos materiales sonoros que la componen.

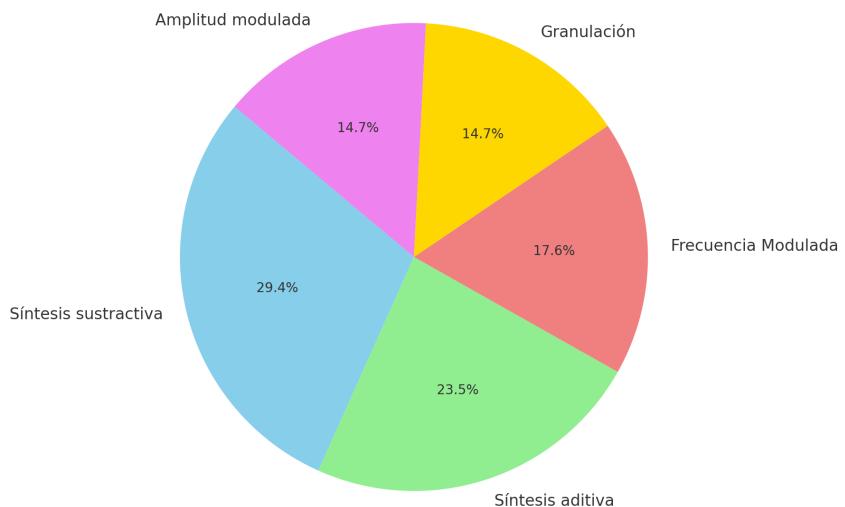


Fuente: Elaboración propia

7.3 Técnicas de síntesis sonora utilizadas en los materiales sonoros

Síntesis sustractiva, síntesis aditiva, frecuencia modulada, granulación y amplitud modulada son, en ese orden, las técnicas de síntesis sonora más utilizadas por GPT-4, a lo largo del trabajo, en lo que la utilización del lenguaje de SuperCollider se refiere, independientemente de la petición del usuario en los prompts. La Figura 7.2 muestra esta distribución de uso de las técnicas de síntesis sonora utilizadas en los fragmentos de código que han servido como base de la composición. Se aprecia una tendencia a la síntesis sustractiva, aquella que consiste en filtrar armónicos de un sonido complejo, generalmente ruido blanco o clics, para obtener sonidos complejos y ricos en matices.

Figura 7.2 Gráfica circular con la distribución de las técnicas de síntesis sonora utilizadas en los materiales sonoros utilizados en la composición de AlgorAI, generados por GPT-4 en SuperCollider. Se aprecia una clara tendencia a la utilización de la síntesis sustractiva.



Fuente: Elaboración propia

Una parte significativa de los materiales sonoros, a pesar de estar generados aparentemente por una técnica definida, en realidad deben su sonido más bien a combinaciones casuales y complejas, más fruto del azar que de la intención. Esto es común cuando los diversos parámetros de un sistema son llevados al extremo y producto sonoro es fruto de artefactos digitales como saturaciones, *aliasing*, etc. Este aspecto de serendipia e impredecibilidad en la utilización de modelos de lenguaje aplicados a lo sonoro no es desdeñable, ya que puede ser una fuente de inspiración para el compositor.

Es notoria la naturaleza distinta en los materiales generados en el lenguaje de Tidal Cycles respecto a SuperCollider, ya que este lenguaje se basa en la manipulación de samples y en su secuenciación en el tiempo en bucles. Los bucles más complejos que se aprecian en *AlgorAI* han sido producidos por GPT-4 en Tidal Cycles. En este caso, se han escogido siempre los bucles sonoros más impredecibles, aquellos que no se ajustan a una estructura rítmica regular. La Figura 7.3 muestra uno de los ejemplos escritos en Tidal Cycles escogidos para la composición por su carácter irregular.

Figura 7.3 Ejemplo de código en Tidal Cycles escogido para la composición por su carácter irregular e impredecible.



```
d1
$ degradeBy 0.1
$ slow 10
$ n (struct "[t(3,8)]*3 t(5,6) [t(6,7)]*4 t(3,4) t(2,3) t(7,8)"
$ slow 3
$ ((fast 3 sine * rand) * (1-(fast 5 tri)) * (1-(fast 45 tri * rand))) *20)
# s "[[rash, cpu] | cpu2 | numbers | [arp, <hi lo snare>] | [superpiano, [
# alphabet clap]?]]"
-- # s (choose["[rash, cpu]", "[cpu2, hc]", "numbers", "[arp, <hi lo snare>]",
# "[superpiano, [alphabet clap]?]]")
# speed (slow 7 $ sine*1+0.05)
# legato 10
# squiz (slow 20 $ sine*5)
# room 1
# size 0.8
# delay 0.8
# delaytime (choose [0.1, 0.5, 1])
# delayfb 0.8
# pan (range 0 1 rand)
# vowel (slow 2.7 "{u o i e a i u a, a e i o u o}")
```

Fuente: Elaboración propia

7.4 Interacción humano-LLM en el proceso de composición

Se puede plantear la interrogante acerca de si la implementación de materiales sonoros concebidos mediante IA ha ejercido influencia en la orientación y el desenlace de la obra compositiva. Desde la perspectiva subjetiva del autor de la composición, quien simultáneamente desempeña un papel de investigador, se concluye que la respuesta es negativa. La metodología empleada para la mixtura temporal de los diversos elementos sonoros, incluyendo su segmentación, extensión, secuencia, entre otros, es resultado de un proceso de decisión consciente y creativa por parte del compositor. Es preciso señalar que el uso de materiales alternativos habría conducido a la creación de una obra distinta, si bien manteniendo un carácter y estilo semejantes.

No obstante, no podemos subestimar el impacto que la incorporación de materiales sonoros originados por IA representa en el proceso creativo. En primera instancia, esta inclusión ha facilitado al compositor-investigador el descubrimiento de texturas y matices tímbricos inéditos, los cuales no habrían sido factibles de generar mediante métodos convencionales. Tradicionalmente, el artista sonoro se aventura en la búsqueda de sus materiales sonoros, ya sea cap-

turando sonidos del entorno real mediante dispositivos de grabación o experimentando en el ámbito electrónico o digital con herramientas como sintetizadores, secuenciadores y software especializado en la generación de sonido. La exploración a través de modelos de lenguaje programáticos ha posibilitado la identificación de materiales inaccesibles para el compositor por sí solo, no por una limitación en sus competencias técnicas, sino debido a que la diversidad de combinaciones posibles de elementos generativos es inabordable. Además, el ser humano tiende a replicar patrones y estructuras previamente exitosos o familiares, con lo que una herramienta de IA puede beneficiar al proceso de materiales sonoros originales.

Cabe destacar la reducción significativa en la carga de trabajo que ha representado el empleo de LLM para la generación de materiales sonoros, minimizando el tiempo requerido en comparación con métodos tradicionales de creación. Podría argumentarse que esto conlleva a una pérdida del control artístico sobre la obra; sin embargo, esta afirmación no es necesariamente acertada. La actitud del compositor hacia la interacción con el LLM a través de sus distintas interfaces se ha caracterizado por un proceso de selección y búsqueda constante. De los códigos generados, solo una fracción de los bloques sonoros ha sido seleccionada para la composición, tras un procedimiento de descarte basado en la audición y criterios estéticos subjetivos. Los fragmentos finalmente incorporados son el resultado de múltiples iteraciones de diálogo con el modelo. En el contexto de la música experimental, la incorporación de elementos externos a la intención original del compositor constituye una práctica habitual, y la utilización de LLM emerge como una herramienta a tener en cuenta en la exploración sonora.

8. Resultados y discusión

ALL YOU NEED IS A SECOND LOOK

— Cao y Zou (2020)

8.1 Resultados

A continuación, en aras de una mayor claridad, se enumeran y sintetizan los resultados clave obtenidos en este trabajo y que ya han sido expuestos de forma sincrónica a lo largo de las diferentes etapas del mismo:

1. *Utilidad de los sistemas de IA como herramientas de asistencia:* La investigación apunta a que las tecnologías de IA, aunque lejos de alcanzar en este momento la autonomía completa en generación musical, constituyen una asistencia valiosa al compositor, potenciando su creatividad y eficiencia.
2. *Fomento de la interacción creativa en tiempo real:* Las API y el entorno de *live coding* se muestran como plataformas efectivas para la exploración sonora dinámica, siempre que los lenguajes utilizados posean mecanismos eficientes para la gestión de errores y su ejecución en vivo.
3. *Potencial en la exploración de nuevas ideas y sonidos:* La capacidad de los LLM para inspirar y facilitar la experimentación sonora se destaca, especialmente entre los usuarios con conocimientos previos en el lenguaje de programación musical.
4. *Tendencia a la repetición de patrones en las respuestas:* Se observa una propensión de los LLM a generar, a priori, contenido de características similares, indicando una *personalidad* del modelo que pueden considerarse oportunidades en el proceso de generación sonora.
5. *Desnivel semántico entre la descripción en lenguaje natural del código generado y el sonido efectivo de dicho código:* Se trata, quizás, de la principal limitación actual de los LLM en la generación musical, ya que no han sido entrenados en la comprensión de la relación entre el código y el sonido resultante, la cual es una asociación psicoacústica muy compleja. La comprensión de este resultado es clave para comprender la necesidad de

la intervención humana en la composición y la limitación al papel de asistente creativo de los LLM en cualquiera de sus formas e interfaces.

6. *Importancia relativa del diseño de prompts:* Los prompts de sistema son fundamentales en la interacción en tiempo real como en el programa *AI Muse*, sin embargo, no se han comprobado mejoras significativas en la calidad del código a través de las técnicas clásicas de prompting. Técnicas más avanzadas, como RAG puede ofrecer mejoras cualitativas en las respuestas si posee un buen *dataset* de conocimiento a disposición del LLM.
7. *Limitaciones en la generación de estructuras temporales compositivas:* Los LLM muestran limitaciones en crear estructuras temporales o códigos largos y complejos, problema que solo puede salvarse a través de la intervención humana en la composición.
8. *Exploración y serendipia creativa:* La generación de sonidos y música por parte de los LLM facilita encuentros creativos inesperados, promoviendo la exploración de nuevos territorios sonoros a los que de otro modo no se accedería.
9. *Desafíos y limitaciones técnicas:* Se han identificado problemas en la gestión de proyectos de mediano a grande tamaño por parte de LLM, así como las limitaciones de la ventana de contexto y las alucinaciones recurrentes del modelo, que requieren supervisión y ajuste activo por parte del usuario creador.

8.2 Discusión crítica de los resultados

Queremos subrayar el hecho de que la capacidad de razonamiento actual de sistemas de IAG como los LLM no sustituye el proceso creativo humano. A juzgar por la velocidad en la que se desarrollan cuantitativa y cualitativamente estas tecnologías, es muy probable que en un futuro no muy lejano, la generación musical autónoma, sea una realidad. En todo caso, su papel a día de hoy puede ser el de asistente creativo, herramienta de exploración y fuente de inspiración.

La interacción en tiempo real por medio de las API se ha mostrado como el lugar más adecuado al potencial actual de los LLM. Por una parte, las limitaciones inherentes a estos sistemas en la creación sonora, especialmente la dificultad en la generación de estructuras temporales, se ven mitigadas a la hora de actuar en tiempo real, donde la dimensión temporal y su forma no tienen por qué ser decididas de antemano, y el papel del usuario-compositor es determinante en el devenir de la actividad performativa. Por otra parte, la interacción en tiempo real explota los puntos fuertes de los LLM, a saber, la generación de pequeñas piezas de código a partir de

contextos también pequeños. Subrayamos que es crítico que los lenguajes de programación musical elegidos tengan una buena abstracción de errores, para que el usuario pueda corregir y ajustar el código en tiempo real y en ningún caso se vea interrumpida la actividad creativa.

A pesar de que en otros campos de aplicación de los LLM, la técnica de RAG se manifiesta como un avance significativo, en el presente trabajo no se ha podido explorar en profundidad esta técnica. Sin embargo, se intuye que la utilización de un RAG con un *dataset* de conocimiento específico para la generación musical podría ofrecer mejoras cualitativas en las respuestas de los LLM.

La generación sonora por parte de los LLM, independientemente de la interfaz utilizada, se ha mostrado fértil para la exploración de nuevos sonidos y texturas, y para la búsqueda de ideas y soluciones creativas. Es de esperar que en función de las mejoras cualitativas de los modelos, estas herramientas se conviertan en asistentes donde poder hablar de auténtica *creatividad*.

9. Conclusiones

ALL YOU NEED IS BOUNDARY

— H. Wang y cols. (2019)

Este estudio ha centrado la intersección entre la IA y la generación algorítmica de música, destacando tanto las capacidades como las limitaciones de los LLM en la interacción con el proceso creativo musical humano, explorando desde la generación de código de programación musical hasta el impacto en la percepción en el propio proceso creativo del artista. A través de un enfoque metodológico cualitativo que combinó la experimentación práctica con el análisis teórico, se buscó comprender y evaluar la capacidad de los LLM en el ámbito de la composición musical.

A continuación, se presentan las conclusiones principales derivadas del análisis integral de la interacción entre el compositor y los LLM, planteado en el objetivo general, reflejando cómo cada objetivo específico (véase la sección 1.3) ha sido abordado y en qué medida se ha cumplido.

- a) *Examinar la habilidad de los LLM para generar código de programación musical basado en texto en lenguaje natural:* Este objetivo ha sido el motor transversal de toda la investigación, ya que toda interacción entre humano y LLM se ha examinado a través del lenguaje natural, propio de los ML. Los resultados (cfr. capítulo 8) dan amplia cuenta tanto de las capacidades como de las limitaciones halladas en la generación de código musical.
- b) *Evaluar la eficiencia y usabilidad de interfaces de interacción entre el compositor y los LLM en la creación sonora:* A lo largo del trabajo se han evaluado diversas interfaces de comunicación e interacción compositor-LLM, desde el diálogo con un chatbot hasta la utilización creativa de una API para administrar programáticamente las peticiones a un LLM y procesar sus respuestas en un entorno de codificación musical en vivo. El estudio de esta última interfaz ha dado como fruto la creación de un programa de *live coding* que permite la interacción en tiempo real con un LLM a través de la API de OpenAI (véase sección 5.5.1 y capítulo 6).
- c) *Explorar el impacto de las diversas técnicas de prompting en los resultados producidos por sistemas de LLM:* En cada fase de la investigación, se experimentó con diferentes

formas de prompting. Especial atención merecen los prompts de sistema utilizados en los programas de *live coding*, que constituyen el conjunto de instrucciones fundamentales sobre el que los LLM actúan en la creación musical.

- d) *Reflexionar sobre el valor artístico de las composiciones y códigos musicales generados mediante la interacción con LLM:* El trabajo destaca la capacidad de estos sistemas para producir texturas sonoras y composiciones que, aunque no siempre estructuradas como piezas completas, ofrecen una base rica para la creación musical. Se han estudiado las condiciones en las que la interacción con sistemas de LLM puede conducir a la creación de música con calidad artística. La composición realizada en el trabajo, *AlgorAI*, muestra un posible camino de integración de la generación de código musical por LLM en una obra musical completa.
- e) *Considerar el impacto del uso de sistemas de IA en el proceso creativo y en la experiencia perceptual del artista:* La búsqueda de interfaces y modos de comunicación con sistemas de IA en este trabajo ha sido un proceso de reflexión constante sobre el papel de los LLM en el proceso creativo, no solo desde un punto de vista del objeto final generado, sino también desde la perspectiva de la experiencia del artista en el proceso de creación. Así, por una parte, la creación de *AlgorAI*, como obra musical, y por otra, la escritura de *AI Muse*, han explorado la percepción del propio artista-investigador en todos los procesos implicados.

10. Limitaciones y prospectiva

IS BANG-BANG CONTROL ALL YOU NEED?

— Seyde y cols. (2021)

10.1 Limitaciones

Este trabajo se ha desarrollado bajo ciertas restricciones que son importantes considerar al momento de interpretar los resultados y conclusiones presentadas. A continuación, se detallan las principales limitaciones encontradas durante la investigación, bien sea por la naturaleza del enfoque metodológico, por las herramientas utilizadas o por las condiciones de desarrollo del proyecto:

1. *Naturaleza cualitativa y exploratoria:* El enfoque de este estudio es principalmente cualitativo y exploratorio, lo que implica que no se han buscado resultados experimentales cuantificables. Esta aproximación, a pesar de ofrecer un panorama global del tema y de dibujar futuras líneas de investigación, limita a priori la capacidad de generalizar los hallazgos más allá de los casos estudiados.
2. *Unidad del investigador y posibles sesgos:* La investigación fue llevada a cabo por un único individuo en interacción creativa, lo cual puede introducir sesgos en la recopilación y análisis de datos, así como en la interpretación de los resultados. Además, dada la familiaridad del investigador con la programación y la música, es posible que existan sesgos en la selección de prompts y en la interpretación de los resultados, influyendo en la dirección y conclusiones del trabajo.
3. *Obsolescencia de herramientas de IA utilizadas y analizadas:* Las herramientas y modelos de lenguaje empleados representan el estado del arte al comienzo de la realización del estudio. Sin embargo, el campo de la IA avanza rápidamente, lo que podría resultar en que estas herramientas se vuelvan obsoletas, y con ellas, los resultados, en un corto periodo de tiempo.
4. *Uso de modelos de código cerrado:* La investigación se apoyó en modelos de lenguaje de código cerrado, en concreto, con los de OpenAI, que limita la transparencia y comprensión completa de los procesos internos que guían las generaciones musicales.

5. *Limitaciones computacionales y de recursos del proyecto:* La potencia computacional a la que se ha tenido acceso es la personal del investigador, así como los recursos económicos para el uso de los servicios utilizados, lo cual ha influido en decisiones clave como los modelos elegidos o en la imposibilidad de explorar técnicas como la aplicación de *fine-tuning* en los modelos utilizados.

10.2 Prospectiva

Este estudio abre varias vías para investigaciones y desarrollos posteriores en el campo de la IA y la generación de música. Entre ellas:

1. *Aplicaciones educativas y de asistencia:* El potencial de estas herramientas para facilitar la educación en lenguajes musicales es considerable. El estudio y desarrollo de aplicaciones dirigidas al ámbito educativo musical en general, y del aprendizaje de lenguajes musicales de la complejidad de los aquí estudiados, en particular, se presenta como una dirección prometedora y necesaria.
2. *Metodología cuantitativa en futuros trabajos:* Sin perjuicio de eventuales estudios como el presente, aplicados a diversos ámbitos de la creación musical con IA, se sugiere la adopción de enfoques cuantitativos que utilicen métricas específicas para evaluar la calidad del código, la música, la originalidad, entre otros. Esto permitirá una evaluación más objetiva y comparativa de los resultados.
3. *Creación de datasets mejorados para fine-tuning y RAG:* Es preciso el desarrollo de una extensa colección de documentación y códigos de programación musical para potenciar el valor cualitativo de las respuestas de los LLM en el campo de la creación musical y sonora. Este corpus, debidamente comentado y etiquetado con descripciones de alto nivel, es esencial para abordar el notable desnivel semántico entre las intenciones del código y el resultado sonoro real. La implementación de técnicas de RAG utilizando esta documentación mejorada puede ser comparable al del *fine-tuning*.
4. *Diversidad de lenguajes de programación:* Ampliar la investigación para incluir más lenguajes de programación fortalecerá la versatilidad y aplicabilidad de los modelos de IA en la creación musical.

5. *Agentes autónomos*: Investigar sistemas que implementen agentes autónomos capaces de corregir errores de código de manera iterativa presenta un campo prometedor para mejorar la eficiencia y autonomía de los sistemas de generación de música.
6. *Manejo de piezas extensas por próximas arquitecturas y modelos de IA*: El constante y rápido avance de la IAG invita a explorar su potencial para gestionar composiciones musicales de gran envergadura y complejidad.
7. *Agentes en la planificación y manejo de código*: La utilización de agentes autónomos de IA para combinar la gestión de códigos pequeños con la planificación general aparece como un área de investigación con gran potencial.
8. *LLM locales y open source*: Una dirección futura de gran interés es la exploración de la viabilidad de reentrenar LLM de código abierto con una finalidad sonora en entornos locales. Para ello es imprescindible contar con gran capacidad computacional, que cada vez se presenta más accesible.
9. *Multimodalidad en el campo audiovisual*: La exploración de la multimodalidad abre nuevas posibilidades de interacción e interfaces posibles que han quedado fuera de este estudio tanto por limitaciones de tiempo como de recursos. La creación sonora puede estar condicionada por imágenes o vídeos, lo cual amplía las posibilidades creativas. Por otra parte, la integración de modelos de reconocimiento de audio con modelos de lenguaje eventualmente permitiría la generación de sonido y música condicionada por el resultado sonoro de iteraciones previas.
10. *Ulteriores y diversos trabajos sobre interacción con LLM en tiempo real para live coding*: Investigar diferentes aplicaciones de LLM en entornos de *live coding*, tal como se apunta con el programa informático creado en este estudio, *AI Muse*, se presenta como muy prometedor y adecuado a la naturaleza de los LLM y a las posibilidades inherentes a la utilización de API.

Referencias bibliográficas

TEXTBOOKS ARE ALL YOU NEED

— Gunasekar y cols. (2023)

- AGI-Edgerunners/LLM-Agents-Papers.* (2024, enero). AGI-Edgerunners. Descargado 2024-01-02, de <https://github.com/AGI-Edgerunners/LLM-Agents-Papers>
- Algorave. (2023, noviembre). *Wikipedia, la enciclopedia libre.* Descargado 2024-02-08, de <https://es.wikipedia.org/w/index.php?title=Algorave&oldid=155173649>
- Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., ... Penedo, G. (2023, noviembre). *The Falcon Series of Open Language Models* (n.º arXiv:2311.16867). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/2311.16867> doi: 10.48550/arXiv.2311.16867
- Ariza, C. (2011, septiembre). Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition. *MIT Press.* Descargado 2023-12-26, de <https://dspace.mit.edu/handle/1721.1/68626>
- Arun Biji, M. (s.f.). *RAG vs Finetuning vs Prompt Engineering: A pragmatic view on LLM implementation.* Descargado 2024-01-21, de <https://www.linkedin.com/pulse/rag-vs-finetuning-prompt-engineering-pragmatic-view-llm-mathew>
- Audio. (s.f.). Descargado 2023-12-26, de <https://stability.ai/stable-audio>
- AWS DeepComposer. (s.f.). Descargado 2024-02-05, de <https://aws.amazon.com/es/deepcomposer/>
- Bard (chatbot). (2024, enero). *Wikipedia.* Descargado 2024-01-28, de [https://en.wikipedia.org/w/index.php?title=Bard_\(chatbot\)&oldid=1199893524](https://en.wikipedia.org/w/index.php?title=Bard_(chatbot)&oldid=1199893524)
- A Beginner's Guide to Neural Networks and Deep Learning.* (s.f.). Descargado 2023-12-21, de <http://wiki.pathmind.com/neural-network>
- Bhavsar, K. (2023, noviembre). *Prompt Engineering for Arithmetic Reasoning Problems.* Descargado 2024-01-05, de <https://towardsdatascience.com/prompt-engineering-for-arithmetic-reasoning-problems-28c8bcd5bf0e>
- Borsos, Z., Marinier, R., Vincent, D., Kharitonov, E., Pietquin, O., Sharifi, M., ... Zeghidour, N. (2023, julio). *AudioLM: A Language Modeling Approach to Audio Generation* (n.º ar-

- Xiv:2209.03143). arXiv. Descargado 2024-01-04, de <http://arxiv.org/abs/2209.03143>
- Boulanger, R. (Ed.). (2000). *The Csound Book. Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. Massachusetts: The MIT Press.
- Calvo, D. (2018, diciembre). *Red Neuronal Recurrente - RNN*. Descargado 2024-01-22, de <https://www.diegocalvo.es/red-neuronal-recurrente/>
- Campbell, M., Hoane, A. J., y Hsu, F.-h. (2002, enero). Deep Blue. *Artificial Intelligence*, 134(1), 57–83. Descargado 2023-10-29, de <https://www.sciencedirect.com/science/article/pii/S0004370201001291> doi: 10.1016/S0004-3702(01)00129-1
- The Challenges Associated With Building Products Using Large Language Models (LLMs)*. (s.f.). Descargado 2023-10-31, de <https://www.linkedin.com/pulse/challenges-associated-building-products-using-large-language-das>
- Chamand, B., Risser-Maroux, O., Kurtz, C., Joly, P., y Loménie, N. (2022, octubre). *Fine-tune your Classifier: Finding Correlations With Temperature* (n.º arXiv:2210.09715). arXiv. Descargado 2023-11-07, de <http://arxiv.org/abs/2210.09715> doi: 10.48550/arXiv.2210.09715
- Chen, X., Lin, M., Schärli, N., y Zhou, D. (2023, octubre). *Teaching Large Language Models to Self-Debug* (n.º arXiv:2304.05128). arXiv. Descargado 2023-10-23, de <http://arxiv.org/abs/2304.05128> doi: 10.48550/arXiv.2304.05128
- Conjunto de Mandelbrot. (2024, enero). *Wikipedia, la enciclopedia libre*. Descargado 2024-01-28, de https://es.wikipedia.org/w/index.php?title=Conjunto_de_Mandelbrot&oldid=157536731
- Deep Learning (DL)*. (s.f.). Descargado 2023-12-21, de <http://mriquestions.com/what-is-a-neural-network.html>
- Department of Computer Science, SRM Institute of Science and Technology, Chennai, India., Pal*, A., Saha, S., Department of Computer Science, SRM Institute of Science and Technology, Chennai, India., Anita, y Department of Computer Science, SRM Institute of Science and Technology, Chennai, India. (2020, abril). Musenet : Music Generation using Abstractive and Generative Methods. *International Journal of Innovative Technology and Exploring Engineering*, 9(6), 784–788. Descargado 2023-12-26, de <https://www.ijitee.org/portfolio-item/F3580049620/> ([TLDR] The goal is to generate novel music, which will be non-repetitive and enjoyable, and almost anyone will be able to use deep learning

- to generate concordant music based on their needs.) doi: 10.35940/ijitee.F3580.049620
- Dhariwal, P., Jun, H., Payne, C., Kim, J. W., Radford, A., y Sutskever, I. (2020, abril). *Jukebox: A Generative Model for Music* (n.º arXiv:2005.00341). arXiv. Descargado 2023-12-26, de <http://arxiv.org/abs/2005.00341> doi: 10.48550/arXiv.2005.00341
- Du, S., y Wang, H. (2022, marzo). Addressing Syntax-Based Semantic Complementation: Incorporating Entity and Soft Dependency Constraints into Metonymy Resolution. *Future Internet*, 14(3), 85. Descargado 2023-12-25, de <https://www.mdpi.com/1999-5903/14/3/85> doi: 10.3390/fi14030085
- El Test de Turing.* (s.f.). Descargado 2024-01-27, de https://edea.juntadeandalucia.es/bancoreursos/file/d1bb5f09-682f-4e93-a799-e90b1c3364ed/2/COM_3ESO_REA_01_V02.zip/411_el_test_de_turing.html
- Find ways to deal with the scarcity of labeled data.* (s.f.). Descargado 2023-12-21, de <https://www.linkedin.com/pulse/find-ways-deal-scarcity-labeled-data-amit-asawa>
- Funk, T. (2018). A Musical Suite Composed by an Electronic Brain: Reexamining the Illiac Suite and the Legacy of Lejaren A. Hiller Jr. *Leonardo Music Journal*, 28, 19–24. Descargado 2023-12-28, de https://direct.mit.edu/lmj/article-abstract/doi/10.1162/lmj_a_01037/69560
- Generation with LLMs.* (s.f.). Descargado 2023-10-31, de https://huggingface.co/docs/transformers/main/en/llm_tutorial
- Gollapudi, S. (2016). *Practical machine learning*. Packt Publishing. Descargado de <https://books.google.es/books?id=WmsdDAAAQBAJ>
- Gonzalo, J., Carabantes, D., y González Geraldo, J. L. (2023, junio). *Asomándonos a la ventana contextual de la Inteligencia Artificial: decálogo de ayuda para la identificación del uso de ChatGPT en textos académicos* [Billet]. Descargado 2023-11-08, de <https://cuedespyd.hypotheses.org/13299>
- GPT-3 and the rise of foundation models.* (s.f.). Descargado 2023-12-10, de <https://www.linkedin.com/pulse/gpt-3-rise-foundation-models-joseph-boland>
- Guerra Parra, C. A. (2020). *Synthi GME Modular Emulator. Una propuesta de emulación digital del sintetizador EMS Synthi 100 del Gabinete de Música Electroacústica de Cuenca* (Tesis de Master, Universidad de Barcelona). Descargado 2023-12-09, de https://github.com/mesjetiu/TFM_Arte_Sonoro_MEMORIA

- Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., ... Li, Y. (2023, junio). *Textbooks Are All You Need* (n.º arXiv:2306.11644). arXiv. Descargado 2023-06-29, de <http://arxiv.org/abs/2306.11644> (Comment: 26 pages Comment: 26 pages) doi: 10.48550/arXiv.2306.11644
- Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., ... Zhu, J. (2021, agosto). *Pre-Trained Models: Past, Present and Future* (n.º arXiv:2106.07139). arXiv. Descargado 2024-01-25, de <http://arxiv.org/abs/2106.07139> doi: 10.48550/arXiv.2106.07139
- Hernandez-Olivan, C., Hernandez-Olivan, J., y Beltran, J. R. (2022, noviembre). *A Survey on Artificial Intelligence for Music Generation: Agents, Domains and Perspectives* (n.º arXiv:2210.13944). arXiv. Descargado 2023-09-27, de <http://arxiv.org/abs/2210.13944> (Comment: Under review) doi: 10.48550/arXiv.2210.13944
- Hochreiter, S. (1998, abril). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6, 107–116. doi: 10.1142/S0218488598000094
- Holtzman, A., Buys, J., Du, L., Forbes, M., y Choi, Y. (2020, febrero). *The Curious Case of Neural Text Degeneration* (n.º arXiv:1904.09751). arXiv. Descargado 2023-11-07, de <http://arxiv.org/abs/1904.09751> (Comment: Published in ICLR 2020) doi: 10.48550/arXiv.1904.09751
- Hornik, K., Stinchcombe, M., y White, H. (1989, enero). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359–366. Descargado 2023-12-21, de <https://www.sciencedirect.com/science/article/pii/0893608089900208> doi: 10.1016/0893-6080(89)90020-8
- How to Get Better Outputs from Your Large Language Model.* (2023, junio). Descargado 2024-01-25, de <https://developer.nvidia.com/blog/how-to-get-better-outputs-from-your-large-language-model/>
- Huang, D., Bu, Q., Zhang, J. M., Luck, M., y Cui, H. (2023, diciembre). *AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation* (n.º arXiv:2312.13010). arXiv. Descargado 2024-01-02, de <http://arxiv.org/abs/2312.13010> (Comment: 21 pages, 12 figures) doi: 10.48550/arXiv.2312.13010
- Introducing ChatGPT.* (s.f.). Descargado 2024-01-28, de <https://openai.com/blog/chatgpt>

- Introducing Claude.* (s.f.). Descargado 2024-01-28, de <https://www.anthropic.com/news/introducing-claude>
- Invierno IA. (2023, septiembre). *Wikipedia, la encyclopédie libre.* Descargado 2024-01-28, de https://es.wikipedia.org/w/index.php?title=Invierno_IA&oldid=154109150
- Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., ... Sayed, W. E. (2023, octubre). *Mistral 7B* (n.º arXiv:2310.06825). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/2310.06825> (Comment: Models and code are available at <https://mistral.ai/news/announcing-mistral-7b/>) doi: 10.48550/arXiv.2310.06825
- Jones, C., y Bergen, B. (2023, octubre). *Does GPT-4 Pass the Turing Test?* (n.º arXiv:2310.20216). arXiv. Descargado 2024-01-27, de <http://arxiv.org/abs/2310.20216> (Comment: 25 pages, 21 figures) doi: 10.48550/arXiv.2310.20216
- Kainat. (2023, agosto). *Introduction to Generative AI.* Descargado 2024-01-23, de <https://medium.com/@kitkat73275/introduction-to-generative-ai-833c9c467dfa>
- Kirkbride, R. (2023, diciembre). *Qirky/FoxDot.* Descargado 2023-12-09, de <https://github.com/Qirky/FoxDot>
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... Kiela, D. (2021, abril). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* (n.º arXiv:2005.11401). arXiv. Descargado 2023-11-07, de <http://arxiv.org/abs/2005.11401> (Comment: Accepted at NeurIPS 2020) doi: 10.48550/arXiv.2005.11401
- Li, J., Li, G., Li, Y., y Jin, Z. (2023, septiembre). *Structured Chain-of-Thought Prompting for Code Generation* (n.º arXiv:2305.06599). arXiv. Descargado 2023-10-23, de <http://arxiv.org/abs/2305.06599> (Comment: arXiv admin note: text overlap with arXiv:2303.17780)
- Li, Z., Zhu, H., Lu, Z., y Yin, M. (2023, octubre). *Synthetic Data Generation with Large Language Models for Text Classification: Potential and Limitations* (n.º arXiv:2310.07849). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/2310.07849> (Comment: EMNLP 2023)
- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., y Liang, P. (2023, noviembre). *Lost in the Middle: How Language Models Use Long Contexts* (n.º arXiv:2307.03172). arXiv. Descargado 2023-12-12, de <http://arxiv.org/abs/2307.03172> (Comment: 18 pages, 16 figures. Accepted for publication in Transactions of the Association for Compu-

- tational Linguistics (TACL), 2023)
- Liu, X., Zhu, Z., Liu, H., Yuan, Y., Cui, M., Huang, Q., ... Wang, W. (2023, noviembre). *WavJourney: Compositional Audio Creation with Large Language Models* (n.º arXiv:2307.14335). arXiv. Descargado 2024-01-04, de <http://arxiv.org/abs/2307.14335> (Comment: GitHub: <https://github.com/Audio-AGI/WavJourney>)
- LLM prompting guide.* (s.f.). Descargado 2023-10-30, de <https://huggingface.co/docs/transformers/main/en/tasks/prompting>
- Lu, P., Xu, X., Kang, C., Yu, B., Xing, C., Tan, X., y Bian, J. (2023, mayo). *MuseCoco: Generating Symbolic Music from Text* (n.º arXiv:2306.00110). arXiv. Descargado 2024-01-04, de <http://arxiv.org/abs/2306.00110>
- Malach, E. (2023, septiembre). *Auto-Regressive Next-Token Predictors are Universal Learners* (n.º arXiv:2309.06979). arXiv. Descargado 2023-09-27, de <http://arxiv.org/abs/2309.06979>
- Mastropaoletto, A., Pasarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., y Bavota, G. (2023, febrero). *On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot* (n.º arXiv:2302.00438). arXiv. Descargado 2023-12-26, de <http://arxiv.org/abs/2302.00438>
- Mehdi, Y. (2023, septiembre). *Announcing Microsoft Copilot, your everyday AI companion.* Descargado 2024-01-28, de <https://blogs.microsoft.com/blog/2023/09/21/announcing-microsoft-copilot-your-everyday-ai-companion/>
- Minsky, M., y Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. MIT Press. Descargado de <https://books.google.es/books?id=0w10AQAAIAAJ>
- Modelación del lenguaje. (2024, enero). *Wikipedia, la enciclopedia libre*. Descargado 2024-01-24, de https://es.wikipedia.org/w/index.php?title=Modelaci%C3%B3n_del_lenguaje&oldid=157157282
- Mohan, A., Zhang, A., y Lindauer, M. (2023, agosto). *Structure in Reinforcement Learning: A Survey and Open Problems* (n.º arXiv:2306.16021). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/2306.16021> doi: 10.48550/arXiv.2306.16021
- Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., y Jiang, Z. M. J. (2023, septiembre). GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software*, 203, 111734. Descargado 2023-12-26, de <https://www.sciencedirect.com/science/article/pii/S0164121223001292> doi: 10.1016/

j.jss.2023.111734

- MusicLM.* (s.f.). Descargado 2024-02-05, de <https://google-research.github.io/seanet/musiclm/examples/>
- Nishi, K. (2024, enero). *KentoNishi/awesome-all-you-need-papers*. Descargado 2024-01-27, de <https://github.com/KentoNishi/awesome-all-you-need-papers>
- Noever, D., y Ciolino, M. (2022, diciembre). *The Turing Deception* (n.º arXiv:2212.06721). arXiv. Descargado 2024-01-27, de <http://arxiv.org/abs/2212.06721> doi: 10.48550/arXiv.2212.06721
- Nur, A., Radzi, N., y Osman, A. (2014, septiembre). Artificial neural network weight optimization: A review. *TELKOMNIKA*, 12. doi: 10.11591/telkomnika.v12i9.6264
- O’Shea, K., y Nash, R. (2015, diciembre). *An Introduction to Convolutional Neural Networks* (n.º arXiv:1511.08458). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/1511.08458> (Comment: 10 pages, 5 figures) doi: 10.48550/arXiv.1511.08458
- Overtone - Collaborative Programmable Music.* (s.f.). Descargado 2023-12-09, de <https://overtone.github.io/>
- Pascanu, R., Mikolov, T., y Bengio, Y. (2013, febrero). *On the difficulty of training Recurrent Neural Networks* (n.º arXiv:1211.5063). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/1211.5063> (Comment: Improved description of the exploding gradient problem and description and analysis of the vanishing gradient problem)
- Peng, S., Kalliamvakou, E., Cihon, P., y Demirer, M. (2023, febrero). *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot* (n.º arXiv:2302.06590). arXiv. Descargado 2023-12-27, de <http://arxiv.org/abs/2302.06590>
- Penrose, R. (2015). *La nueva mente del emperador*. Penguin Random House Grupo Editorial España.
- Ph.D, M. M. (2023, noviembre). *The training time of the foundation models (from scratch)*. Descargado 2024-01-21, de <https://medium.com/codenlp/the-training-time-of-the-foundation-models-from-scratch-59bbce90cc87>
- Pure Data — Pd Community Site.* (s.f.). Descargado 2024-01-28, de <https://puredata.info/>
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., y Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners.. Descargado 2024-01-27, de <https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask>

-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe ([TLDR])

It is demonstrated that language models begin to learn these tasks without any explicit supervision when trained on a new dataset of millions of webpages called WebText, suggesting a promising path towards building language processing systems which learn to perform tasks from their naturally occurring demonstrations.)

Regression vs Classification, Explained - Sharp Sight. (2021, abril). Descargado 2024-01-26, de <https://www.sharpsightlabs.com/blog/regression-vs-classification/>

Roads, C., Strawn, J., Abbott, C., Gordon, J., y Philip, G. (1996). *The Computer Music Tutorial*. Cambridge: The MIT Press.

Rothman, D. (2021). *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*. Birmingham: Packt Publishing.

Run text generation with Bloom and GPT models on Amazon SageMaker JumpStart / AWS Machine Learning Blog. (2022, noviembre). Descargado 2023-10-31, de <https://aws.amazon.com/blogs/machine-learning/run-text-generation-with-gpt-and-bloom-models-on-amazon-sagemaker-jumpstart/>

Russell, S. J. (2021). *Artificial intelligence : A modern approach* (4th Edition.; Global edition. ed.). Harlow: Pearson.

Ruviaro, B. (2015). *A Gentle Introduction to SuperCollider*. Descargado 2024-01-30, de https://github.com/brunoruviaro/A_Gentle_Introduction_To_SuperCollider/blob/master/A_Gentle_Introduction_To_SuperCollider_BOOK_9x7_Landscape.tex

Schmidt, R. M. (2019, noviembre). *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview* (n.º arXiv:1912.05911). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/1912.05911> doi: 10.48550/arXiv.1912.05911

Searle, J. R. (1985). *Mentes, cerebros y ciencia*. Cátedra.

Shannon, C. E. (1951, enero). Prediction and entropy of printed english. *Bell System Technical Journal*, 30, 50–64. Descargado de <http://languageelog.ldc.upenn.edu/myl/Shannon1950.pdf>

Sonic Pi - The Live Coding Music Synth for Everyone. (s.f.). Descargado 2024-01-28, de <https://sonic-pi.net/>

- Stable Audio: Fast Timing-Conditioned Latent Audio Diffusion.* (s.f.). Descargado 2023-09-27, de <https://stability.ai/research/stable-audio-efficient-timing-latent-diffusion>
- Stokes, J. (2023, septiembre). *A guide to language model sampling in AllenNLP*. Descargado 2024-01-30, de <https://blog.allenai.org/a-guide-to-language-model-sampling-in-allennlp-3b1239274bc3>
- Strudel REPL*. (s.f.). Descargado 2023-12-09, de <https://strudel.cc/>
- Suno AI*. (s.f.). Descargado 2023-12-26, de <https://www.suno.ai/>
- SuperCollider*. (2024, enero). SuperCollider. Descargado 2024-01-28, de <https://github.com/supercollider/supercollider>
- SuperCollider 3.12.2 Help*. (s.f.). Descargado 2024-01-30, de <https://doc.sccode.org/Help.html>
- Supper, M. (2012). *Música electrónica y música con ordenador. Historia, estética, métodos, sistemas*. Madrid: Alianza Música.
- Team, C. (s.f.). *ChucK: A Strongly-Timed Music Programming Language*. Descargado 2023-12-09, de <https://ccrma.stanford.edu/software/chuck/>
- ¿Qué es el ajuste de hiperparámetros?* (s.f.). Descargado 2024-01-26, de <https://aws.amazon.com/es/what-is/hyperparameter-tuning/>
- Tian, K., Mitchell, E., Yao, H., Manning, C. D., y Finn, C. (2023, noviembre). *Fine-tuning Language Models for Factuality* (n.º arXiv:2311.08401). arXiv. Descargado 2024-01-25, de <http://arxiv.org/abs/2311.08401> doi: 10.48550/arXiv.2311.08401
- Tidal Cycles*. (s.f.). Descargado 2023-12-09, de <https://tidalcycles.org/>
- Top-k, Top-p, Temperature*. (s.f.). Descargado 2024-01-30, de <https://zhuanlan.zhihu.com/p/631786282>
- Torres i Viñals, J. (2020). *Python Deep Learning. Introducción práctica con Keras y TensorFlow 2* (1ª ed.). Marcombo.
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., ... Lample, G. (2023, febrero). *LLaMA: Open and Efficient Foundation Language Models* (n.º arXiv:2302.13971). arXiv. Descargado 2024-01-28, de <http://arxiv.org/abs/2302.13971> doi: 10.48550/arXiv.2302.13971
- Turing, A. M. (1950). Computing machinery and intelligence. *Mind; a quarterly review of psychology and philosophy*, 49, 433–460.

- An Understanding of Learning from Demonstrations for Neural Text Generation · The ICLR Blog Track.* (s.f.). Descargado 2024-01-26, de <https://iclr-blog-track.github.io/2022/03/25/text-gen-via-lfd/>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017, diciembre). *Attention Is All You Need* (n.º arXiv:1706.03762). arXiv. Descargado 2023-05-23, de <http://arxiv.org/abs/1706.03762> (Comment: 15 pages, 5 figures) doi: 10.48550/arXiv.1706.03762
- Wang, C., Liu, S. X., y Awadallah, A. H. (2023, agosto). *Cost-Effective Hyperparameter Optimization for Large Language Model Generation Inference* (n.º arXiv:2303.04673). arXiv. Descargado 2023-11-07, de <http://arxiv.org/abs/2303.04673>
- Wang, S., Ma, C., Xu, Y., Wang, J., y Wu, W. (2022, diciembre). A Hyperparameter Optimization Algorithm for the LSTM Temperature Prediction Model in Data Center. *Scientific Programming*, 2022, e6519909. Descargado 2023-11-07, de <https://www.hindawi.com/journals/sp/2022/6519909/> doi: 10.1155/2022/6519909
- Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., ... Zhou, D. (2023, marzo). *Self-Consistency Improves Chain of Thought Reasoning in Language Models* (n.º arXiv:2203.11171). arXiv. Descargado 2024-01-20, de <http://arxiv.org/abs/2203.11171> (Comment: Published at ICLR 2023. V2: added PaLM results; V3: added UL2 results; V4: camera ready version at ICLR 2023) doi: 10.48550/arXiv.2203.11171
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., ... Zhou, D. (2023, enero). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models* (n.º arXiv:2201.11903). arXiv. Descargado 2023-10-23, de <http://arxiv.org/abs/2201.11903>
- What is Retrieval Augmented Generation?* (s.f.). Descargado 2024-01-05, de <https://www.linkedin.com/pulse/what-retrieval-augmented-generation-grow-right>
- What is retrieval-augmented generation?* (2021, febrero). Descargado 2023-11-07, de <https://research.ibm.com/blog/retrieval-augmented-generation-RAG>
- Wilson, S., Cottle, D., y Collins, N. (2011). *The SuperCollider Book*. The MIT Press.
- Zhang, A., Lipton, Z. C., Li, M., y Smola, A. J. (2023). *Dive into deep learning*. Cambridge University Press. (<https://D2L.ai>)
- Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., ... Chi, E. (2023, abril). *Least-to-Most Prompting Enables Complex Reasoning in Large Language Models* (n.º arXiv:2205.10625). arXiv. Descargado 2023-06-25, de <http://arxiv.org/abs/2205.10625> (Comment:

ICLR 2023) doi: 10.48550/arXiv.2205.10625

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., ... He, Q. (2020, junio). *A Comprehensive Survey on Transfer Learning* (n.º arXiv:1911.02685). arXiv. Descargado 2024-01-25, de <http://arxiv.org/abs/1911.02685> (Comment: 31 pages, 7 figures) doi: 10.48550/arXiv.1911.02685

Zrara, L. (2020). *PORTFOLIO OPTIMIZATION USING DEEP LEARNING FOR THE MOROCCAN MARKET*. Descargado 2024-01-27, de <http://www.aui.ma/sse-capstone-repository/pdf/fall-2020/PORTFOLIO%20OPTIMIZATION%20USING%20DEEP%20LEARNING%20FOR%20THE%20MOROCCAN%20MARKET.pdf>

Anexo A. Materiales sonoros generados por GPT-4 utilizados en la composición *AlgorAI*

Se presentan aquí los códigos de todos los materiales sonoros que se han utilizado en la composición de la obra de arte sonoro *AlgorAI* (véase capítulo 7). Junto a cada código se da la posibilidad de escucharlo en un MP3 alojado en la nube. Los ejemplos creados en SuperCollider (sección A.1) se generaron a través de conversaciones en modo chatbot y utilizando *Assistants* creados ad hoc en el trabajo (véase capítulo 5). Todos los códigos presentados en Tidal Cycles (sección A.2) fueron creados a través de la API de OpenAI con el software creado en este trabajo *AI Muse* (véase capítulo 6).

A.1 Códigos de materiales sonoros escritos en SuperCollider

```
(  
{  
    var n=6, freqs = LFN noise1.kr(n.linrand).range(500, 12000);  
    Splay.ar(  
        RLPF.ar(  
            Dust.ar(freqs * 0.004) * 50,  
            freqs,  
            LFN noise1.kr(0.1, 0.5).range(0.05, 0.15)  
        )  
    )  
}.play;  
)
```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

(
SynthDef(\acidBold, {
    arg out, freq = 1000, gate = 1, pan = 1, cut = 4000, rez = 0.8, amp = 1, lfoFreq =
        0.25;
    var lfo, source;
    lfo = SinOsc.kr(lfoFreq).range(800, 2000); // LFO para modulación de frecuencia
    source = Pulse.ar(freq + lfo, 0.05);
    Out.ar(out,
        Pan2.ar(
            RLPF.ar(
                source,
                cut * LFN Noise1.kr(1).range(0.8, 1.2), // Modulación de cut
                rez + LFN Noise1.kr(0.5).range(-0.1, 0.1)), // Modulación de resonancia
                pan) * EnvGen.kr(Env.perc(0.01, 0.3), gate, amp, doneAction: Done.freeSelf);
    )
}).add;
)

(
Pbind(\instrument, \acidBold, \dur, Pseq([0.25, 0.5, 0.75], inf), \root, -12,
    \degree, Pseq([0, 3, 6, 9, 12, 15, 18, 21], inf), // Grados modificados
    \pan, Pfunc({1.0.rand2}),
    \cut, Pxrand([1000, 500, 2000, 300], inf),
    \rez, Pfunc({0.7.rand +0.3}), \amp, 0.3).play;
)

(
Pdef(\buckyballBold, Pbind(\instrument, \acidBold, \dur, Pseq([0.06, 0.07, 0.02, 0.01], inf),
    \root, [-24, -17],
    \degree, Pseq([2, 3, 20, 7, 1, 11, [5, 17], 30], inf)+22, \pan, Pfunc({[1.0.rand2, 1.0.
        rand2]}),
    \cut, Pxrand([1000, 500, 2000, 300], inf), \rez, Pfunc({0.7.rand +0.3}), \amp, [0.15,
        0.22])).play;
)

(
Pdef(\buckyballBold, Pbind(\instrument, \acidBold, \dur, Pseq([0.25, 0.5, 0.75], inf),
    \root, [-24, -17],
    \degree, Pseq([0b, 3b, 5b, 7b, 9b, 11b, 5b, 0b], inf), \pan, Pfunc({1.0.rand2}),
    \cut, Pxrand([1000, 500, 2000, 300], inf), \rez, Pfunc({0.7.rand +0.3}), \amp, 0.3)).play;
)

(
Pdef(\randomRhythm, Pbind(
    \instrument, \acidBold,
    \dur, Pwhite(0.03, 0.05, inf), // Duraciones aleatorias entre 0.1 y 1.0 segundos
    \degree, Pseq([0, 10, 4, 2, 7, 9, 11]-24, inf),
    \pan, Pwhite(-1, 1, inf), // Pan aleatorio
    \cut, Prand([500, 1000, 1500, 2000], inf),
    \rez, Pwhite(0.3, 0.8, inf), // Resonancia aleatoria
    \amp, Pwhite(0.1, 0.5, inf) // Amplitud aleatoria
)).play;
)
)

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

// 1. Definimos el SynthDef con modulación de frecuencia.
(
SynthDef(\fmModulator, { |out = 0, freq = 440, modDepth = 100, modFreq = 5, pan = 0, dur =
1|
// Generador de modulación
var modulator = SinOsc.ar(modFreq) * modDepth;

// Oscilador principal con frecuencia modulada
var signal = SinOsc.ar(freq *1 + modulator) * 0.5;

// Envoltura para dar principio y fin al sonido
var env = EnvGen.kr(Env.perc(0.01, dur*3 - 0.01), doneAction: 2);

// Multiplicamos la señal por la envoltura
signal = signal * env;

// Aplicamos paneo
signal = Pan2.ar(signal, pan);

// Aplicamos reverb
signal = FreeVerb.ar(signal, 0.2, 0.6, 0.5); // Puedes ajustar estos parámetros para
cambiar la reverb

// Enviamos la señal al canal de salida
Out.ar(out, signal);
}).add;
)

// 2. Usamos Pbind para secuenciar eventos de nuestro SynthDef con paneo.
(
Pbind(
\instrument, \fmModulator, // Usamos el SynthDef definido anteriormente
\freq, Pwhite(0.0, 1).linexp(0, 1, 100, 600), // Frecuencias aleatorias entre 300 y
600 Hz
\modDepth, Pwhite(50, 150), // Profundidad de modulación variable
\modFreq, Pwhite(30, 200.0), // Frecuencia de modulación variable
\pan, Pwhite(-1.0, 1), // Paneo aleatorio
\dur, Pwhite(0.5, 9), // Duración de cada nota
\legato, 3
).play;
)

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

(
{
    var x, y, z, a, b, c, d, e, f, g, h, mix;

    // Tomando la idea de osciladores sinusoidales y distorsión
    x = SinOsc.ar((Decay.ar(Impulse.ar([4,4.005])), 1e3*MouseX.kr(0, 1).abs)*50), MouseY.kr(0, 1)).distort;

    // Incorporando la idea de modulación de frecuencia y filtrado
    y = BPF.ar(Saw.ar([40,40.001]), LFNNoise0.kr(128)+1*4e3+146, LFNNoise1.kr(1)+1*5e-2+0.01).tanh;

    // Utilizando un delay con modulación
    z = DelayC.ar(LPF.ar(LFNNoise0.ar(8)**2 + x.tanh.round(0.05), 6e3), 1, LFNNoise0.ar(8!2).range(1e-4, 0.02));

    // Juega con la modulación de un filtro
    a = BHiPass.ar(LFNNoise1.ar(8)**3, y.midicps, y/2e3, 67-y);

    // Incorporando elementos de pitch shifting
    b = PitchShift.ar(Saw.ar(Demand.kr(Impulse.kr(4), 0, Drand([-12, -7, 0, 7, 12]+33).midicps, inf)), Decay.kr(Impulse.kr(4), 3)), 7, 2);

    // Creando una secuencia de ritmo
    c = Impulse.ar(8)*LFNoise1.ar(2);

    // Añadiendo un poco de reverb y limitador para controlar la dinámica
    d = Limiter.ar(GVerb.ar(BPF.ar(c, [400, 800], 1/9, 50).tanh, 100, 10), 0.9);

    // Mezclando las señales juntas
    mix = Mix.new([x, y, z, a, b, c, d]);

    // Aplicando un poco de panoramización y envío al output
    Out.ar(0, Splay.ar(mix));
}.play;
)

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

(
SynthDef(\customNTubeSound, {
    var source, filteredNoise, output, pannedOutput;

    // Generamos un sonido que se asemeje a la respiración utilizando PinkNoise, SinOsc y
    // la técnica NTube.ar
    source = PinkNoise.ar * SinOsc.ar(0.1);
    filteredNoise = NTube.ar(
        source,
        [0.97, 1.0, 1.0, 1.0, 1.0, 0.97], // Loss factors
        [0.5, MouseY.kr(-1.0, 1.0), 0.2, -0.4], // Junction reflection coefficients
        [0.01, 0.02, 0.01, 0.005, 0.05] * MouseX.kr(0.001, 1.0, 'exponential') // Delay
        lengths
    ) * 0.1;

    // Panoramización
    pannedOutput = Pan2.ar(filteredNoise, 0); //SinOsc.kr(0.1).range(-1, 1));

    // Aplicamos un limitador
    output = Limiter.ar(pannedOutput, 0.99, 0.01);
    output = output.min(1.0).max(-1.0);

    Out.ar([0, 1], output);
}).add;
)

// Reproducimos el sonido
x = Synth(\customNTubeSound);

(
SynthDef(\customAmbientSound, {
    var source, nTubeProcessed, modulatedFilter, output, stereoOutput;

    // Generamos un noise basado en PinkNoise pero modulado con una SinOsc de baja
    // frecuencia para darle movimiento
    source = PinkNoise.ar(0.5) * SinOsc.kr(0.1).range(0.5, 1.5);

    // Utilizamos NTube para darle un carácter tubular y resonante al noise
    nTubeProcessed = NTube.ar(
        source,
        [0.95, 0.98, 1.0, 1.0, 0.98, 0.95], // Loss factors
        LFNNoise1.kr(0.1).range(0.1, 0.9), // Junction reflection coefficients variando con
        el tiempo
        LFNNoise2.kr(0.05).range(0.01, 0.05) // Delay lengths variando aleatoriamente
    ) * 0.3;

    // Añadimos un filtro paso banda que se mueva con el tiempo
    modulatedFilter = BPF.ar(
        nTubeProcessed,
        freq: LFNNoise2.kr(0.2).range(300, 2000),
        rq: 0.5
    );

    // Aplicamos un delay estéreo con modulación tipo ping-pong
    stereoOutput = CombC.ar(modulatedFilter, 2.0, LFNNoise2.kr(0.1).range(0.05, 0.5), 0.5);
    stereoOutput = DelayN.ar(stereoOutput, 0.2, SinOsc.kr(0.3).range(0.01, 0.1));

    Out.ar([0, 1], stereoOutput);
}).add;
)

// Reproducimos el sonido
y = Synth(\customAmbientSound);

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

(
SynthDef(\fmSynthNoReverb, {
    arg freq=440, mDepth=100, mFreq=5, dur=1, amp=0.5, pan=0;
    var modulator, carrier, env, signal;

    env = EnvGen.kr(Env.perc(0.01, dur, 1, -1), doneAction: 2);

    modulator = SinOsc.ar(mFreq) * mDepth;
    carrier = SinOsc.ar(freq + modulator);
    signal = carrier + (SinOsc.ar(freq * 2 + modulator * 0.5) * 0.5);
    signal = signal * env * amp;

    // Aplicar panorámica
    signal = Pan2.ar(signal, pan);

    // Enviar a un bus de audio en lugar de la salida principal
    Out.ar(\bus.kr(0), signal);
}).add;
)

(
SynthDef(\reverbSynth, {
    arg bus, mix=0.2, room=0.8;
    var inSignal;

    inSignal = In.ar(bus, 2);
    inSignal = FreeVerb.ar(inSignal, mix: mix, room: room);

    // Devolver el sonido procesado al bus principal
    Out.ar(0, inSignal);
}).add;
)

// Configurar el bus
~reverbBus = Bus.audio(s, 2);

// Iniciar el Synth de reverberación
~reverbSynth = Synth(\reverbSynth, ['bus': ~reverbBus]);

// Pbind
(
Pbind(
    \instrument, \fmSynthNoReverb,
    \freq, Pwhite(300, 600, inf),
    \mDepth, Pwhite(50, 500, inf),
    \mFreq, Pseq([Pwhite(2, 10, 1), Pwhite(100, 1000, 1)], inf),
    \dur, Pwhite(0.2, 1, inf),
    \amp, Pwhite(0.3, 0.7, inf),
    \pan, Pwhite(-1, 1, inf), // Valores entre -1 (izquierda) y 1 (derecha)
    \bus, ~reverbBus // Enviar al bus de reverberación
).play;
)

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

(
// Inicialización de la orquestación
s.waitForBoot {
    var introDur = 10, desarrolloDur = 20, interactivoDur = 15, climaxDur = 15, cierreDur =
        5;
    var totalDur = introDur + desarrolloDur + interactivoDur + climaxDur + cierreDur;
    var timeline;

    timeline = Routine {
        // Introducción con gabor e imp
        Synth(\gabor, [freq: 440, pan: 0, width: 0.3, sustain: introDur]);
        Synth(\imp, [freq: 440, speed: 1, begin: 0, end: 1, pan: 0, accelerate: 0.5, offset
            : 0, sustain: introDur]);
        (introDur).wait;

        // Desarrollo con psin y pmsin
        Synth(\psin, [freq: 440, speed: 1, begin: 0, end: 1, pan: 0, accelerate: 0.5,
            offset: 0, modfreq: 40, sustain: desarrolloDur]);
        Synth(\pmsin, [freq: 440, speed: 1, begin: 0, end: 1, pan: 0, accelerate: 0.5,
            offset: 0, sustain: desarrolloDur]);
        (desarrolloDur).wait;

        // Interacción en vivo con in e in1
        Synth(\in, [soundin: 0, pan: 0, inputFadeTime: 0.03, sustain: interactivoDur]);
        Synth(\in1, [soundin: 0, pan: 0, inputFadeTime: 0.03, sustain: interactivoDur]);
        (interactivoDur).wait;

        // Climax con una mezcla de synths
        // Aquí puedes añadir más Synths según tu preferencia
        (climaxDur).wait;

        // Cierre con gabor y dirac
        Synth(\gabor, [freq: 440, pan: 0, width: 0.3, sustain: cierreDur]);
        Synth(\dirac, [pan: 0, sustain: cierreDur]);
        (cierreDur).wait;
    };

    timeline.play;
};

)

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

(
SynthDef(\paulstretchPoly, { |out = 0, bufnum, envBufnum, pan = 0, stretch = 8, window =
0.25, amp = 1, rate = 1|
// Los detalles internos de la SynthDef se mantienen iguales que en la versión
monofónica
var trigPeriod, sig, chain, trig, pos, fftSize;
fftSize = 2**floor(log2(window*SampleRate.ir));
trigPeriod = fftSize/SampleRate.ir;
trig = Impulse.ar(1/trigPeriod);
pos = Demand.ar(trig, 0, demandUGens: Dseries(0, trigPeriod/stretch));
sig = [GrainBuf.ar(1, trig, trigPeriod, bufnum, rate, pos, envbufnum: envBufnum),
GrainBuf.ar(1, trig, trigPeriod, bufnum, rate, pos + (trigPeriod/(2*stretch)),
envbufnum: envBufnum)]*amp;
sig = sig.collect({ |item, i|
chain = FFT(LocalBuf(fftSize), item, hop: 1.0, wintype: -1);
chain = PV_Diffuser(chain, 1 - trig);
item = IFFT(chain, wintype: -1);
});
sig = sig*PlayBuf.ar(1, envBufnum, 1/(trigPeriod), loop:1);
sig[1] = DelayC.ar(sig[1], trigPeriod/2, trigPeriod/2);
Out.ar(out, Pan2.ar(Mix.new(sig), pan));
}).add;

s.waitForBoot({
var numSynths = 8; // Número de instancias de Synth que quieres crear
var envBuf, envSignal, buffer, synths;

// Carga del buffer con el archivo de sonido
buffer = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav");

// Creación y carga del buffer de envolvente
envBuf = Buffer.alloc(s, s.sampleRate, 1);
envSignal = Signal.newClear(s.sampleRate).waveFill({|x| (1 - x.pow(2)).pow(1.25)},
-1.0, 1.0);
envBuf.loadCollection(envSignal);
s.sync();

// Creación de múltiples instancias de Synth con diferentes parámetros
synths = Array.fill(numSynths, { |i|
var delayTime = i.rand; // Tiempo de espera antes de iniciar la instancia
var rateValue = [0.125, 0.25, 0.5, -0.125, -0.25, -0.5].choose; // Velocidades de
reproducción, incluidas negativas
var stretchValue = (8.rand + 0.5).round(0.01); // Valores de estiramiento
var panValue = [-1, 1].choose; // Posición estéreo
var windowHeight = [0.1, 0.2, 0.3, 0.4].choose; // Valor aleatorio para el tamaño de
la ventana

// Se utiliza 'fork' para permitir tiempos de inicio escalonados
fork({
delayTime.wait; // Espera antes de empezar
Synth(\paulstretchPoly, [
\bufnum, buffer.bufnum,
\envBufnum, envBuf.bufnum,
\pan, panValue,
\stretch, stretchValue,
\rate, rateValue,
\amp, 0.3.rand + 0.2, // Volumen aleatorio dentro de un rango
>window, windowHeight // Tamaño de ventana aleatorio
]);
});
});
});
});
}
)

```

Código generado por GPT-4 en interacción iterativa en modo chatbot.



```

// Definimos un bus de audio para la reverberación
~reverbBus = Bus.audio(s, 2);
// SynthDef de reverberación
(
SynthDef(\reverbEffect, { |outBus = 0|
    var sig = In.ar(~reverbBus, 2);
    sig = JPverb.ar(sig[0] + sig[1], t60: 3, size: 0.9, damp: 0.2, earlyDiff: 0.7);
    ReplaceOut.ar(outBus, sig);
}).add;
// SynthDefs para gránulos con correcciones de FM y adiciones para variación
SynthDef(\grainSynth, { |outBus, freq = 440, dur = 0.1, pan = 0, amp = 0.1, type = 0|
    var sig, env, mod;
    env = EnvGen.ar(Env.perc(0.01, dur, amp), doneAction: 2);
    // Tipo de grano: varía desde FM a pulse a noise
    mod = Select.ar(type,
        [
            SinOsc.ar(freq + SinOsc.ar(Rand(1.0, 100.0)).range(-200, 200)) * env, // FM
            Pulse.ar(freq, 0.5) * env, // Pulse
            WhiteNoise.ar() * env * LFOise1.kr(5).range(0,1) // Noise
        ]
    );
    Out.ar(outBus, Pan2.ar(mod, pan));
}).add;
)
// Creamos la instancia del efecto de reverberación
~reverb = Synth(\reverbEffect, [\outBus, 0]);
// Función para variar la densidad temporal de los granos y tipo de grano
(
~density = { |minTime = 0.05, maxTime = 0.2, typeMin = 0, typeMax = 2|
    Pbind(
        \instrument, \grainSynth,
        \freq, Pexprand(200.0, 2000.0, inf),
        \dur, Pwhite(minTime, maxTime, inf),
        \pan, Pwhite(-1, 1, inf),
        \amp, Pwhite(0.01, 0.1, inf),
        \type, Pwhite(typeMin, typeMax, inf),
        \outBus, ~reverbBus
    ).play;
};
)
// Iniciar con densidad inicial y tipos de grano
~grainPattern = ~density.value(0.05, 0.1, 0, 0);
// Modificar la densidad y el tipo de grano cada 10 segundos
(
~changeDensity = {
    inf.do {
        ~grainPattern.stop;
        ~grainPattern = ~density.value(
            0.05.rand2(0.2), // minTime
            (0.2.rand2(0.5)).abs, // maxTime
            0, // typeMin
            2 // typeMax
        );
        10.wait;
    };
}.fork;
)
// Parar el cambio de densidad y el patrón de grano
(
~stopPatterns = {
    ~changeDensity.stop;
    ~grainPattern.stop;
};
)
~changeDensity.stop;
~grainPattern.stop;
~reverb.free;
// Ejecutar ~changeDensity para iniciar el cambio automático en la densidad y tipo de grano
~changeDensity;
~stopPatterns;

```



```

(
s.boot;

// Definir un sintetizador más suave para el sonido del viento y las hojas
SynthDef(\softFoliage, {
    arg out=0, amp=0.1, pan=0;
    var sound, filterFreq;

    // Crear un ruido rosado que tiene una calidad más suave que el ruido blanco
    sound = PinkNoise.ar(amp);

    // Modular la frecuencia del filtro para simular el movimiento del viento en las hojas
    filterFreq = LFOise1.kr(Rand(0.1, 0.5)).exprange(300, 2000);
    sound = LPF.ar(sound, filterFreq);

    // Utilizar una envolvente suave para modelar el sonido con más naturalidad
    sound = sound * EnvGen.kr(Env.linen(5, 10, 5, 0.1), doneAction: 2);

    Out.ar(out, Pan2.ar(sound, pan));
}).add;

// Definir el synth para los pájaros como se hizo inicialmente
SynthDef(\birds, {
    arg out=0, amp=0.05, pan=0;
    var z;
    z = SinOsc.ar(2200 + (SinOsc.kr([5, 6]) * 100)).abs - 0.97;
    z = HPF.ar(z * amp, 1000);
    z = z * EnvGen.ar(Env.perc(0.1, Rand(2, 3)), doneAction: 2);
    Out.ar(out, Pan2.ar(z, pan));
}).add;

// Inicializar el patrón para generar el sonido del viento
Pbind(
    \instrument, \softFoliage,
    \dur, Pwhite(8, 12), // Duración más larga para un sonido continuo
    \amp, 0.2,
    \pan, Pwhite(-1, 1)
).play;

// Inicializar el patrón para los pájaros
Pbind(
    \instrument, \birds,
    \dur, Pexprand(0.1, 1),
    \amp, 0.05,
    \pan, Pwhite(-1, 1)
).play;
)

```

Código generado por GPT-4 en interacción iterativa utilizando la técnica de RAG en un Assistant de OpenAI.



```

(
s.waitForBoot {
    var numChannels = 2; // Estéreo

    s.options.numOutputBusChannels = numChannels;
    s.options.numInputBusChannels = numChannels;
    s.boot;

    SynthDef("grainTexture", {
        arg out=0, dur=2, rate=1, pan=0;
        var env, grain, freq;

        freq = LFOise1.kr(0.5).exprange(200, 1200);

        grain = GrainSin.ar(
            numChannels: numChannels,
            trigger: Impulse.kr(10),
            dur: LFOise2.kr(0.5).range(dur * 0.5, dur),
            pan: LFOise2.kr(0.1).range(-1, 1),
            freq: freq,
            maxGrains: 50
        ) * 0.1;

        env = EnvGen.kr(Env.perc(0.01, 1), doneAction: 2);

        Out.ar(out, grain * env);
    }).add;

    Routine({
        inf.do {
            Synth("grainTexture", [
                \dur, rrand(0.1, 0.5),
                \rate, rrand(0.5, 2),
                \pan, rrand(-1, 1)
            ]);

            (rrand(0.1, 0.5)).wait;
        }
    }).play;
};

)

```

Código generado por GPT-4 en interacción iterativa utilizando la técnica de RAG en un Assistant de OpenAI.



```

(
SynthDef.new(\birds, {
    var env, freq, sig, modFreq, pan;
    modFreq = LFOise1.kr(rrand(0.5, 2)).range(5, 15);
    env = Env.perc(0.01, 0.1, 0.1).kr(doneAction: 2);
    freq = LFDNoise3.kr(modFreq).exprange(4000, 8000);
    sig = SinOsc.ar(freq) * env;
    pan = LFOise2.kr(0.5).range(-1, 1);
    Out.ar(0, Pan2.ar(sig, pan));
}).add;

SynthDef.new(\wind, {
    var sig, filt, env, ampMod;
    ampMod = LFOise1.kr(0.2).range(0.2, 0.7);
    env = Env.linen(2, 10, 2, 1).kr(doneAction: 2);
    sig = WhiteNoise.ar(ampMod);
    filt = BPF.ar(sig, LFOise1.kr(rrand(0.05, 0.1)).range(150, 250), 0.1) * 0.25;
    Out.ar(0, Pan2.ar(filt * env, LFOise2.kr(rrand(0.3, 0.5)).range(-1, 1)));
}).add;

SynthDef.new(\leaves, {
    var sig, mod, env;
    env = Env.linen(0.1, 8, 0.1, 0.3).kr(doneAction: 2);
    sig = Dust2.ar(LFOise2.kr(1).range(50, 400));
    mod = LFOise1.kr(4).range(0, 0.5);
    Out.ar(0, Pan2.ar(sig * mod * env, LFOise1.kr(rrand(0.4, 0.6)).range(-1, 1)));
}).add;

SynthDef.new(\insects, {
    var sig, env;
    env = Env.linen(2, 12, 2, 0.1).kr(doneAction: 2);
    sig = Mix.fill(2, {
        LFPulse.ar(rrand(40, 120), 0, LFOise2.kr(1).range(0.1, 0.5)) * 0.05
    });
    Out.ar(0, Pan2.ar(sig * env, LFOise1.kr(0.5).range(-1, 1)));
}).add;

Pbind(
    \instrument, \birds,
    \dur, Pfunc({ rrand(0.1, 0.8) }),
    \amp, Pfunc({ rrand(0.3, 0.6) })
).play;

Pbind(
    \instrument, \wind,
    \dur, 1,
    \amp, Pfunc({ rrand(0.2, 0.4) })
).play;

Pbind(
    \instrument, \leaves,
    \dur, Pwrand([0.1, 0.2, 0.5], [0.5, 0.3, 0.2], inf),
    \amp, Pfunc({ rrand(0.1, 0.4) })
).play;

Pbind(
    \instrument, \insects,
    \dur, Pwrand([0.3, 0.7, 1], [0.5, 0.3, 0.2], inf),
    \amp, Pfunc({ rrand(0.05, 0.2) })
).play;
)

```

Código generado por GPT-4 en interacción iterativa utilizando la técnica de RAG en un *Assistant* de OpenAI.



A.2 Códigos de materiales sonoros escritos en Tidal Cycles

```
d1 $ slow 4
$ s "cpu2*8 snare hi*4"
# speed "-4 2 -1 3"
# cut 1
# delay 0.7
# delaytime 1
# delayfeedback 0.8
# lpf 500
# hpf 100
# room 0.2
# sz 0.2 -- varía ete de alguna forma más clara
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d2 $ degradeBy 0.3
$ s "gab*12" #
up (irand 8)
# speed (slow 6 $ sine)
# room 0.1
# sz 0.1
# delay 0.2
# delayt 0.5
# delayfb 0.8
# lpf 800
# hpf 200
# gain 0.8
# speed ( 2 + irand 4)-- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d3 $ sometimesBy 0.25 (striate 16)
$ sound "arpv*24"
# n (run 24)
# speed (choose [0.6, 0.9, 1.1, 1.3, 1.6])
# sustain (slow 2 $ rand)
# room 0.1
# sz 0.1
# gain 0.7
# pan (fast 2 "0 0.8 0.2 1")-- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d3 $ sometimesBy 0.3 (striate 16)
$ sound "cpu*24"
# n (run 24)
# speed (choose [0.6, 0.9, 1.1, 1.3, 1.6])
# sustain (slow 3 $ sine)
# room 0.1
# sz 0.1
# gain 0.65
# pan (slow 3 $ saw) -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d2 $ degradeBy 0.35
$ s "gab*30"
# up (irand 9)
# speed (slow 7 $ sine)
# room 0.1
# sz 0.1
# delay 0.25
# delayt 0.55
# delayfb 0.75
# lpf 850
# hpf 220
# gain 0.6
# speed (5 + irand 10) --
# pan (slow 3 $ saw) -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d4 $ degradeBy 0.35
$ s "gab*14"
# up (irand 10)
# speed (slow 7 $ sine)
# room 0.1 # sz 0.1
# delay 0.3
# delayt 0.6
# delayfb 0.7
# lpf 850
# hpf 220
# gain 0.6
# speed (3 + irand 2) -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d2 $ degradeBy 0.5
$ s "cpu2*30"
# n (run 8 + irand 32)
# up (slow 2 $ sine)
# room 0.2
# sz 0.2
# orbit 1 -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d3 $ slow 8
$ n "[0 2 4 5 7]*16"
# s "arp"
# speed (slow 16 $ sine * 0.6 * (slow 0.3 $ sine))
# room 0.1
# sz 0.1
# orbit 2
# lpf (slow 2 $ range 400 8000 $ rand)
# resonance 0.3 -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d2 $ degradeBy 0.4
$ s "cpu3:1*8 foley*4"
# n (iter 4 $ slow 4 $ run 12 + (irand 4 - 2))
# pan (slow 1 $ sine)
# room 0.3
# speed (slow 2 $ sine)
# sz 0.2
# orbit 1 -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d1
$ degradeBy 0.1
$ slow 5
$ n (struct "[t(3,8)]*3 t(5,6) [t(6,7)]*4 t(3,4) t(2,3) t(7,8)"
$ slow 3
$ ((fast 3 sine * rand) * (1-(fast 5 tri)) * (1-(fast 45 tri * rand))) *20
-- # s "[[rash, cpu] | cpu2 | numbers | [arp, <hi lo snare>] | [superpiano, [
    alphabet clap]?]]"
# s (choose["[rash, cpu]" , "[cpu2, hc]" , "numbers" , " [arp, <hi lo snare>]" ,
    "[superpiano, [alphabet clap]?]]")
# speed (slow 7 $ sine*1+0.05)
# legato 10
# squiz (slow 20 $ sine*5)
# room 0.7
# size 0.8
# delay 0.8
# delaytime (choose [0.1, 0.5, 1])
# delayfb 0.8
# pan (range 0 1 rand)
# vowel (slow 2 "u o i e a")
# gain 0.7
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software AI Muse.



```
d2 $ stack [
  s "cpu2*16",
  n (run 8),
  speed (range 0.7 1.3 $ slow 2 sine),
  room 0.5,
  size 0.2,
  shape (slow 4 $ saw)
] # pan (slow 3 $ sine) -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software AI Muse.



```
d4 $ s "gabba*16"
# n (irand 32)
# speed (0.5 - (slow 6 sine * 0.2))
# room 0.2
# size 0.1
# lpf (slow 3 $ sine * 500 + 800)
# hpf (fast 7 $ sine * 300 + 400)
# pan (slow 5 $ range (-1) 1 $ sine)
# orbit 2 -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software AI Muse.



```
d2 $ degradeBy 0.3
$ s "cpu*16"
# speed (slow 8 $ range 0.9 1.1 $ sine)
# room 0.3
# size 0.1
# shape (0.8)
# pan (slow 4 $ range (-1) 1 $ sine)
# delay 0.7
# delaytime (slow 3 $ sine * 0.3 + 0.5)
# delayfb 0.75
# lpf (slow 5 $ range 300 1200 $ saw)
# hpf (slow 2 $ range 400 600 $ sine) -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d6 $ linger 0.3
$ s "gab*6"
# n (slow 12 $ sine * 7)
# speed (slow 7 $ range 0.2 1.8 $ rand)
# room 0.2
# size 0.1
# shape 0.6
# pan (slow 9 $ range (-1) 1 $ sine)
# lpf (slow 15 $ range 250 750 $ sine)
# hpf 600 # orbit 0 -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d2 $ stack [
  s "[cpu:1*8 ~ cpu:1*16]",
  speed (slow 6 $ range 0.5 1.5 $ rand),
  room 0.3,
  size 0.1,
  shape (slow 7 $ sine)
] -- # pan (slow 3.5 $ sine) -- gpt-4-1106-preview -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d2 $ degradeBy 0.2
$ slow 2 $ s "cpu*24"
# speed (slow 8 $ range 0.85 1.15 $ sine)
# room 0.25
# size 0.12
# shape 0.85
# pan (slow 4 $ range (-1) 1 $ sine)
# delay 0.6
# delaytime (slow 3 $ sine * 0.25 + 0.45)
# delayfb 0.65
# lpf (slow 5 $ range 280 1100 $ saw)
# hpf (slow 2 $ range 380 580 $ sine) -- gpt-4-1106-preview -- gpt-4-1106-preview
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



```
d1 $ slow (sine*20+7) -- 21 segundos
$ degradeBy 0.2
$ note (run 30)
# sound "[cpu2|cpu|snare|clap|hi]"
# vowel "o e a u i a u o i e a"
# room 0.2
# size (slow 6 $ sine*(0.2) + 0.8)
# pan "[0|0.5|1]"
# squiz (slow 6 $ sine*2+1)
# speed "<0.05 0.1 3 0.05 0.06> <0.5 0.2 0.1 1 2>"
# gain 0.7
```

Código en Tidal Cycles generado en una sesión de *live coding* con el software *AI Muse*.



Anexo B. Repositorio con los materiales de la investigación



URL del Repositorio:

https://github.com/mesjetiu/LLM_SuperCollider

Contenido del Repositorio:

Este repositorio contiene todos los materiales utilizados y generados a lo largo de esta investigación. Esto incluye:

- Archivos de L^AT_EX correspondientes a este PDF.
- Scripts de Python utilizados para la interacción con la API de OpenAI.
- Archivos de SuperCollider y Tidal Cycles con los prompts obtenidos en el trabajo.
- Otros materiales y recursos utilizados.

Versión de Entrega Final:

La versión correspondiente a la entrega final de este trabajo a UNIR está identificada con la etiqueta `Entrega_final` en los commits del repositorio. Y el PDF correspondiente se encuentra en el directorio raíz bajo el nombre `TFM_AI_Muse_UNIR_v1.0.0.pdf`

Versión actualizada de este PDF:

Eventuales actualizaciones de este documento, especialmente debidas a modificaciones en los enlaces externos, seguirán el esquema de versionado semántico para reflejar la naturaleza y el alcance de los cambios. Las versiones actualizadas se nombrarán siguiendo el patrón `TFM_AI_Muse_UNIR vX.Y.Z.pdf`, donde X, Y, y Z representan el número de versión mayor, menor y de corrección, respectivamente. Por ejemplo, una actualización menor que corrige enlaces rotos podría nombrarse `TFM_AI_Muse_UNIR_v1.0.1.pdf`. Encuentre estas versiones en el directorio raíz del repositorio.