UNIVERSIDADE FEDERAL FLUMINENSE – UFF

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA DE PRODUÇÃO

GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

INTELIGÊNCIA ARTIFICIAL NO PROCESSO CRIATIVO MUSICAL: UM ESTUDO DE CASO DE GERAÇÃO DE MÚSICA COM REDES NEURAIS

JOÃO PEDRO SANTOS MURAD

ORIENTADOR DO TRABALHO

JOSÉ KIMIO ANDO

NITERÓI

2025

JOÃO PEDRO SANTOS MURAD

INTELIGÊNCIA ARTIFICIAL NO PROCESSO CRIATIVO MUSICAL: UM ESTUDO DE CASO DE GERAÇÃO DE MÚSICA COM REDES NEURAIS

Projeto Final apresentado ao curso de Graduação em Engenharia de Produção da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Engenheiro de Produção.

ORIENTADOR: JOSÉ KIMIO ANDO, D. Sc.

Niterói

2025

JOÃO PEDRO SANTOS MURAD

INTELIGÊNCIA ARTIFICIAL NO PROCESSO CRIATIVO MUSICAL: UM ESTUDO DE CASO DE GERAÇÃO DE MÚSICA COM REDES NEURAIS

Projeto Final apresentado ao curso de Graduação em Engenharia de Produção da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Engenheiro de Produção.

Aprovado em ___ de _____ de 2025.

BANCA EXAMINADORA

_____

Prof. Dr. José Kimio Ando (Orientador) – UFF

_____

Prof. Dr. Valdecy Pereira – UFF

_____

Prof. Dr. Diogo Ferreira – UFF

Niterói

2025

AGRADECIMENTOS

Aos meus pais, Karina Santos e Marco Paulo Murad, e às minhas avós, Sonia Santos e Glória Murad, pelo apoio incondicional em todas as etapas da minha vida. Vocês foram o alicerce fundamental, especialmente nos últimos anos, onde diversos desafios surgiram. Agradeço por sempre me mostrarem que eu era capaz de superar qualquer barreira. Esta conquista não seria possível sem vocês.

Às minhas irmãs, Mariana Santos e Maria Paula Murad, por serem a motivação diária para ser uma pessoa melhor, mais esforçada e determinada.

Aos meus tios, Patrícia Santos, Maria Helena Murad, Rodrigo Santos e Priscila Santos e ao meu padrasto, Ronald Marcelo, pelos exemplos de vida e por suas contribuições fundamentais em minha formação. Agradeço por serem referências tão importantes.

A todos os meus amigos, que estiveram presentes nos momentos mais difíceis desta jornada. Um agradecimento especial a Arthur Yaman, Gabriel Marinho, Luisa Campos, João Pedro Queiroz e Vitor Cardoso, que acompanharam de perto toda a trajetória e nunca me deixaram desistir.

Ao meu mestre e amigo Luan Dias, por me mostrar o que é esforço de verdade e por sempre acreditar em mim dentro e fora dos tatames.

Por fim, agradeço imensamente ao meu orientador, Professor José Kimio Ando. Sua proposta de tema foi o ponto de partida para este trabalho, e sua orientação sábia ao longo deste ano foi essencial. Obrigado por toda paciência, dedicação e por me desafiar intelectualmente, permitindo que eu sempre pudesse avançar. Sua presença como um grande norte será lembrada para muito além desta graduação.

A todos vocês, meu mais sincero agradecimento.

RESUMO

Este trabalho apresenta um estudo de caso de aplicação da Inteligência Artificial (IA) no processo criativo musical. Inicialmente são feitas considerações sobre a indústria da música e apresentados alguns aspectos teóricos sobre música, inteligência artificial e redes neurais. São utilizadas as arquiteturas de rede neural Long Short-Term Memory (LSTM) e Transformer na tarefa de geração de música simbólica para piano. É definida uma configuração para cada uma das redes. Todas são treinadas com os *datasets* Chopin (estilisticamente focado) e MAESTRO (amplo e variado), na geração de música. Os quatro resultados gerados são avaliados quantitativa e qualitativamente (utilizando um painel de cinco avaliadores). A partir destas avaliações são implementados ajustes nas arquiteturas e feitas novas avaliações. Os resultados quantitativos mostram que o Transformer foi computacionalmente mais rápido nos treinamentos e que o LSTM apresentou gargalos de RAM no pré-processamento. Na análise qualitativa, o LSTM apresentou limitações severas com o *dataset* MAESTRO, enquanto o Transformer produziu a composição musical mais coerente e bem avaliada pelos avaliadores. Conclui-se que, para a geração de música com mérito artístico utilizando inteligência artificial, a arquitetura Transformer é superior para a tarefa, mas que o treinamento com um *dataset* rico, como o MAESTRO, é fundamental.

Palavras-chave: Inteligência Artificial, Geração de Música, Deep Learning, LSTM, Transformer.

ABSTRACT

This work presents a case study of the application of Artificial Intelligence (AI) in the musical creative process. Initially, considerations are made about the music industry, and some theoretical aspects of music, artificial intelligence, and neural networks are presented. The Long Short-Term Memory (LSTM) and Transformer neural network architectures are used in the task of generating symbolic music for piano. A configuration is defined for each network. All are trained with the Chopin (stylistically focused) and MAESTRO (broad and varied) datasets in music generation. The four generated results are evaluated quantitatively and qualitatively (using a panel of five evaluators). Based on these evaluations, adjustments are implemented in the architecture and new evaluations are performed. The quantitative results show that the Transformer was computationally faster in training and that the LSTM presented RAM bottlenecks in preprocessing. In the qualitative analysis, the LSTM presented severe limitations with the MAESTRO dataset, while the Transformer produced the most coherent musical composition and was well-evaluated by the evaluators. In conclusion, for generating music with artistic merit using artificial intelligence, the Transformer architecture is superior for the task, but training with a rich dataset, such as MAESTRO, is essential.

Key words: Artificial Intelligence, Music Generation, Deep Learning, LSTM, Transformer.

LISTA DE FIGURAS

Sumário

1 INTRODUÇÃO

1.1 INDÚSTRIA DA MÚSICA NA ERA DIGITAL

A indústria musical, em sua essência, opera na produção e distribuição de um produto cultural de alta demanda: a música. No século XXI, a forma como este produto é criado, consumido e monetizado passou por uma transformação radical, impulsionada pela digitalização. Em 2024, as receitas globais de músicas gravadas continuaram sua trajetória ascendente com um crescimento de 4.8%, um movimento liderado de forma esmagadora pelo formato de streaming. Este modelo de consumo, que já representa 69% das receitas totais da indústria, alcançou a marca de 752 milhões de assinantes pagos em todo o mundo, consolidando a transição da posse física para o acesso digital como o novo padrão de mercado (IFPI, 2024).

Essa mudança não apenas democratizou o acesso à música para os consumidores, mas também reconfigurou as demandas da cadeia de produção. A lógica das playlists algorítmicas, a necessidade de trilhas sonoras para uma quantidade crescente de conteúdo em vídeo (de filmes a mídias sociais) e a competição pela atenção do ouvinte criaram uma necessidade incessante por um volume cada vez maior de novo material musical. Nesse contexto de produção em alta escala, a otimização e a inovação nos processos criativos se tornam não apenas uma vantagem competitiva, mas uma necessidade estratégica, abrindo um vasto campo de atuação para a Engenharia de Produção aplicada à indústria criativa. É precisamente nesta intersecção que a Inteligência Artificial (IA) emerge como a força motriz da próxima grande transformação do setor.

1.2 A INTELIGÊNCIA ARTIFICIAL COMO FERRAMENTA CRIATIVA

Diante da crescente demanda por conteúdo na era digital, a indústria musical começa a voltar-se para a Inteligência Artificial (IA) não apenas como uma ferramenta de análise ou distribuição, mas como uma força ativa no próprio processo de produção. Esta transição representa a próxima fronteira na evolução tecnológica do setor: a ascensão da IA como uma ferramenta criativa. O conceito de utilizar algoritmos para compor música não é novo, tendo suas raízes nos primeiros experimentos computacionais da década de 1950, liderados por pioneiros como Alan Turing, que

exploraram as capacidades do computador Manchester Mark II para gerar melodias simples (RANWALA, 2020).

Essa jornada evoluiu significativamente nas décadas seguintes. Nos anos 1980 e 1990, o foco deslocou-se de algoritmos baseados em regras simples para modelos generativos mais complexos, que buscavam não apenas criar sequências de notas, mas imbuí-las de uma "inteligência musical" (RANWALA, 2020). A partir dos anos 2000, com o advento do *Deep Learning* e o aumento exponencial do poder computacional, a IA na música deu um salto qualitativo. Ferramentas influentes como o Magenta, um projeto de pesquisa do Google Brain, e o Jukebox, da OpenAI, demonstraram a capacidade das redes neurais de analisar vastos corpora de música e gerar novas obras com um nível de complexidade e coerência estilística antes inatingível (BRIOT; HADJERES; PACHET, 2020).

Este avanço tecnológico culmina no que pode ser descrito como a segunda grande democratização da indústria musical. Se o *streaming* democratizou o acesso ao consumo, a IA agora promete democratizar o acesso à criação, fornecendo ferramentas que podem auxiliar desde o músico amador até o produtor profissional. É neste ponto de inflexão que a investigação sobre a eficácia das diferentes abordagens de *deep learning* se torna crucial. A escolha da arquitetura de rede neural não é apenas uma decisão técnica, mas uma aposta em uma filosofia de como a criatividade pode ser modelada e gerada computacionalmente.

1.3 BIFURCAÇÃO TECNOLÓGICA: RECORRÊNCIA VS. ATENÇÃO NA GERAÇÃO MUSICAL

A capacidade das redes neurais de aprender com dados sequenciais abriu uma bifurcação metodológica no campo da geração musical, com duas abordagens principais dominando a pesquisa recente: a recorrente e a baseada em atenção. A primeira, personificada pela arquitetura *Long Short-Term Memory* (LSTM), processa a informação de forma inerentemente sequencial. O modelo lê os dados musicais passo a passo, mantendo um estado de memória interno que é atualizado a cada nova nota, de forma análoga a um músico que lê uma partitura da esquerda para a direita, acumulando

contexto ao longo do tempo. Esta abordagem foi, por muitos anos, o padrão de fato para a modelagem de sequências.

Em contraste, a segunda abordagem, representada pela arquitetura Transformer, abandonou completamente a necessidade de processamento sequencial. Introduzida por Vaswani *et al.* (2017) , ela emprega um mecanismo de autoatenção (*self-attention*) que permite ao modelo ponderar a importância de todas as notas em uma sequência simultaneamente, independentemente de sua posição. Isso resolve um dos principais gargalos das redes recorrentes: a dificuldade de manter o contexto em sequências muito longas, permitindo, em teoria, uma captura mais eficaz de estruturas musicais de longo alcance.

Diante dessas duas tecnologias de ponta, que representam filosofias fundamentalmente diferentes sobre como modelar dados temporais, surge a questão central para este estudo: qual arquitetura se mostra mais eficaz e eficiente para a tarefa de composição de música simbólica? Quais são os *trade-offs* entre a abordagem sequencial e metódica do LSTM e a abordagem paralela e holística do Transformer? Este trabalho se propõe a investigar essa questão através de um estudo experimental direto, formulando a seguinte hipótese principal: a arquitetura Transformer, devido ao seu mecanismo de autoatenção, será superior à LSTM na geração de música com coerência estrutural de longo prazo, especialmente quando treinada em um conjunto de dados grande e diversificado.

1.4 OBJETIVOS E DELIMITAÇÃO DA PESQUISA

Para investigar a questão de pesquisa formulada, este trabalho estabelece um objetivo principal claro, suportado por uma série de objetivos secundários que guiam a metodologia experimental.

O objetivo principal é realizar um estudo experimental e comparativo da eficácia de duas arquiteturas de redes neurais de *deep learning* — a *Long Short-Term Memory* (LSTM) e o Transformer — na tarefa de geração de música simbólica para piano solo.

Para alcançar este fim, os seguintes objetivos secundários foram definidos:

a) Analisar e comparar o desempenho computacional de cada arquitetura, avaliando métricas como tempo de treinamento e o uso de recursos de hardware (RAM de sistema e GPU).

b) Avaliar a qualidade musical das composições geradas através de um painel de avaliação qualitativa, utilizando critérios de coerência melódica, riqueza harmônica e estrutura rítmica.

c) Investigar o impacto de diferentes conjuntos de dados — um estilisticamente focado (Chopin) e outro amplo e variado (MAESTRO) — no resultado final de cada modelo.

d) Implementar e analisar o efeito de uma rodada de ajustes nos modelos, avaliando se o aumento da complexidade da rede e do tempo de treinamento resulta em uma melhora significativa na qualidade musical.

A delimitação do projeto é fundamental para contextualizar os resultados. Este estudo concentra-se exclusivamente na geração de música simbólica, utilizando o formato MIDI como principal meio de representação, e não na geração de áudio bruto. O domínio musical é delimitado a composições para piano solo, refletido na escolha dos *datasets*. As arquiteturas de *deep learning* investigadas são especificamente a LSTM, implementada com a API Keras sobre TensorFlow , e o Transformer, implementado em PyTorch.

1.5 IMPORTÂNCIA DO ESTUDO

A relevância deste estudo reside na intersecção de três domínios atuais: a transformação da indústria musical, o avanço exponencial da inteligência artificial e a necessidade de aplicar uma análise sistemática, própria da Engenharia de Produção, a processos criativos cada vez mais mediados por tecnologia.

Considerando a música como um produto no centro de uma indústria em franca expansão digital, o desenvolvimento de processos de criação torna-se um campo de estudo pertinente e estratégico. A eficiência de uma "linha de produção" de conteúdo musical não se mede apenas em termos de velocidade ou custo, mas também na qualidade, coerência e valor artístico do produto final. Neste contexto, as diferentes

arquiteturas de *deep learning*, como a LSTM e o Transformer, podem ser vistas como duas "tecnologias de processo" concorrentes. Compreender os custos computacionais, os benefícios qualitativos e as limitações de cada uma não é apenas um exercício acadêmico, mas uma investigação fundamental para o futuro do desenvolvimento de ferramentas criativas.

Este trabalho se justifica, portanto, pela necessidade de uma análise comparativa e pragmática que possa auxiliar na tomada de decisões futuras. Para desenvolvedores, os resultados oferecem *insights* sobre os *trade-offs* entre complexidade arquitetural e demanda de recursos. Para músicos e produtores, a pesquisa ajuda a desmistificar o "estado da arte" da composição algorítmica, contextualizando o que é realisticamente alcançável com as ferramentas atuais. Por fim, para a Engenharia de Produção, este estudo serve como um caso prático de aplicação de metodologia experimental e análise quali-quantitativa a um processo produtivo não-tradicional e eminentemente criativo, contribuindo para a expansão do escopo de atuação da área.

1.6 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em cinco capítulos para guiar o leitor de forma lógica, desde a contextualização teórica até a apresentação e discussão dos resultados experimentais.

O Capítulo 1 introduz o tema, estabelecendo a relevância da geração musical por Inteligência Artificial no contexto da indústria musical digital. Apresenta-se a bifurcação tecnológica entre as abordagens recorrentes e baseadas em atenção, formulando a questão de pesquisa, a hipótese e os objetivos que norteiam o estudo.

O Capítulo 2 apresenta a revisão da literatura, construindo a base teórica para o trabalho. Inicia-se com os conceitos fundamentais de música, seguidos por um panorama histórico da composição algorítmica. Aprofunda-se, então, nos fundamentos técnicos das redes neurais, detalhando a estrutura e o funcionamento das arquiteturas LSTM e Transformer, e finaliza com uma visão geral do ecossistema de *software* Python utilizado para a implementação.

O Capítulo 3 detalha a metodologia empregada na pesquisa. Descreve-se o tipo de pesquisa, os conjuntos de dados musicais selecionados (Chopin e MAESTRO) e os dois *pipelines* de pré-processamento para criar as representações simbólica e parametrizada. Em seguida, são especificadas as arquiteturas de cada modelo, as configurações de treinamento e os critérios de avaliação quantitativos e qualitativos.

O Capítulo 4 é dedicado à apresentação e análise dos resultados. Este capítulo está focado em duas fases: a primeira analisa os resultados da rodada experimental inicial (linha de base), comparando o desempenho computacional e a qualidade musical dos quatro cenários. A segunda fase detalha as estratégias de otimização implementadas e analisa os resultados da nova rodada de experimentos, comparando a performance dos modelos otimizados com a linha de base.

Finalmente, o Capítulo 5 apresenta as conclusões do estudo, sintetizando os principais resultados e respondendo aos objetivos de pesquisa. Discutem-se também as limitações do trabalho e são propostas recomendações para futuras investigações na área.

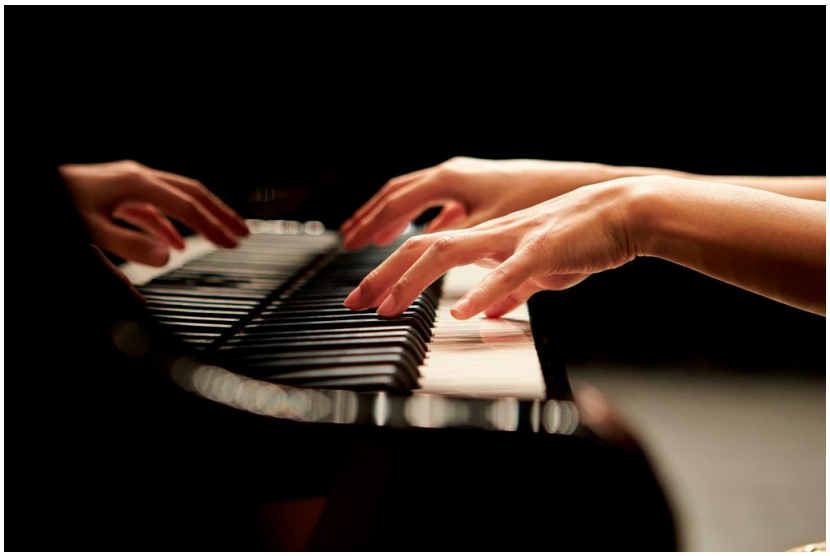
2 REVISÃO DA LITERATURA

2.1 MÚSICA E SEUS CONCEITOS BÁSICOS

De acordo com Nietzsche (1889), música é tanto a excitação quanto a liberação geral dos sentimentos em um estado mais pleno, uma arte onde, sobretudo, os sentidos são menos primais, mais organizados e moldados, de maneira que não se faz mais necessário o uso do corpo de forma tão intensa quanto na antiguidade.

Segundo Laitz (2012), a música ocidental, escrita nos períodos Barroco, Clássico e Romântico, flutua ao redor do conceito de tonalidade, uma única nota, chamada tônica. A partir da mesma, atribuindo conceitos de intervalo, acidentes, descansos, ritmos e andamentos, temos a construção do que hoje é conhecido como música moderna.

Figura 1 - Pianista Tocando

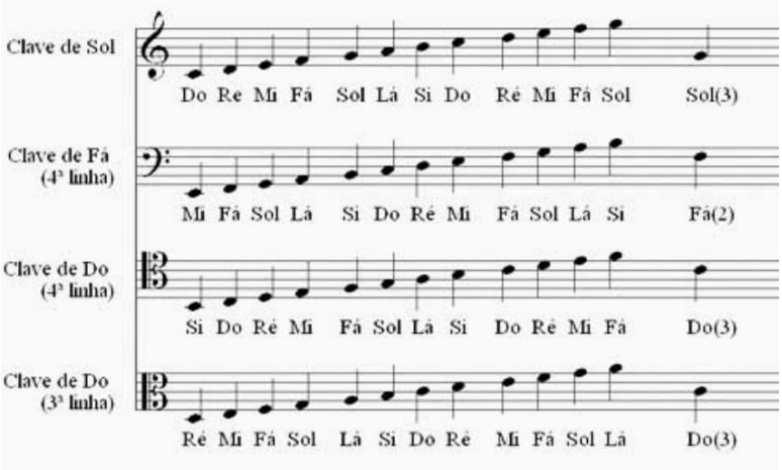


Fonte: (Fritz Dobbert Pianos, 2024)

Uma nota, em suma, é um som emitido de um corpo vibrante (podendo ser a corda de um violino, o soprar de uma flauta ou até mesmo o barulho do vento) e a velocidade das vibrações regulares deste corpo, chamada frequência, determina o conceito de nota (LAITZ, 2012). É importante ressaltar também, as notações modernas para as notas, pois tal ponto será abordado ao longo deste trabalho de maneira frequente. São elas: C (Dó), D (Ré), E (Mi), F (Fá), G (Sol), A (Lá) e B (Si). Assim como

seus respectivos acidentes (notas posicionadas entre as 7 notas principais): C# ou Db (Dó Sustenido ou Ré Bemol), D# ou Eb (Ré Sustenido ou Mi Bemol), F# ou Gb (Fá Sustenido ou Sol Bemol), G# ou Ab (Sol Sustenido ou Lá Bemol) e A# ou Bb (Lá Sustenido ou Si Bemol). A Figura 2 mostra a notação das notas, sem os acidentes, em uma partitura.

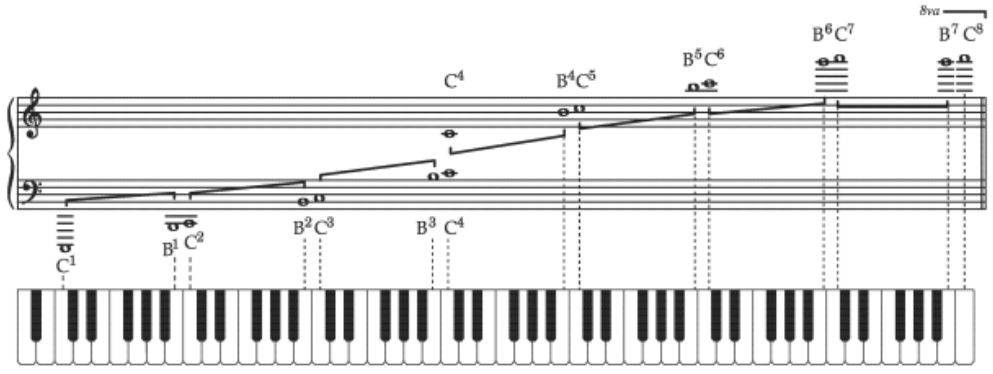
Figura 2 - Notas musicais em uma partitura



Fonte: (Wikipedia – Notação Musical, 2025).

É possível visualizar melhor o funcionamento de uma partitura a partir da Figura 3, onde é ilustrada a localização de algumas notas nas teclas de um piano.

Figura 3 - Localização de algumas notas no Piano



Fonte: (Laitz, 2012)

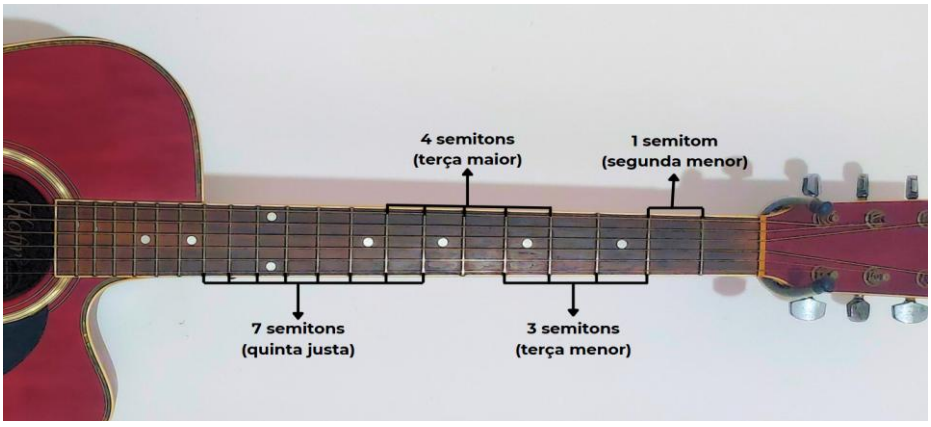
2.1.1 INTERVALOS

Tendo em mente o conceito de nota, o “espaço” entre duas notas é chamado de intervalo. O conceito de espaço é a distância relativa de uma nota para outra. Na Figura 4 é fácil entender este conceito observando o braço de um violão) (BRIOT; HADJERES; PACHET, 2020). Os intervalos são as bases para formação dos acordes, que serão abordados um pouco mais a frente. Alguns exemplos são: terças maiores (distância de 4 semitons), terças menores (3 semitons), quinta justa (7 semitons), etc.

Figura 4 - Violão

Fonte: (Acervo Digital do Violão Brasileiro, 2014)

Figura 5 - Intervalos no Braço do Violão



Fonte: (Autoria Própria)

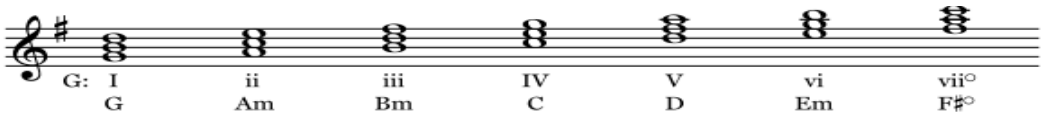
2.1.2 ACORDES

Um acorde é a representação da junção (ou interação) de pelo menos 3 notas (uma tríade) (BRIOT; HADJERES; PACHET, 2020). A característica de um acorde é dada pelos intervalos que o formam, ou seja, considerando que este acorde contenha um intervalo de terça menor em sua formação, ele será um acorde menor. Sua representação pode ser feita das seguintes maneiras:

a) Enumerando as exatas notas que compõem o acorde, permitindo especificar precisamente qual a “posição” daquela nota, em relação às outras notas do acorde, permitindo indicar oitavas na partitura;

b) Usando símbolos combinados que representam características do acorde (se ele tem intervalos de terças menores, maiores, sétimas, nonas, décimas terceiras, etc). Exemplo: C#m (dó sustenido menor), C (dó maior), E7M (mi com sétima menor). É possível visualizar na Figura 6 os acordes (Tríades) em uma partitura.

Figura 6 - Acordes em uma partitura



Fonte:(Clendinning e Marvin, 2016)

2.1.3 RITMO

Para Briot, Hadjeres e Pachet (2020) ritmo é peça fundamental para que algo seja entendido como música. Pulsos e batidas que são acentuadas nos momentos desejados, fato indispensável para o movimento. No geral, são ciclos de pulsações que, contêm uma sequência de notas e acordes.

Batidas são aglutinadas e organizadas em “compassos”, com informações do número de batidas e duração, construindo assim a assinatura de tempo ou métrica de uma música (conhecido popularmente como andamento). Ela é representada no formato de fração, o numerador sendo a quantidade de batidas no compasso e o denominador sendo a representação da nota que será acentuada. Com isso em mente, temos as formas métricas mais utilizadas na música moderna, são os compassos de 2/4, 3/4 e 4/4.

O compasso 4/4 é o mais utilizado mundialmente pela maioria das obras e gêneros, pela sua coesão sonora ser agradável aos ouvidos e de fácil entendimento, o compasso 3/4 é o compasso de valsa, presente também em diversas obras do período romântico da sociedade. Existem também os chamados compassos compostos, que em suma, são a soma de dois compassos (5/8, 7/8 e por aí vai), geralmente usados em Jazz e algumas vertentes de Rock e Metal.

Um conceito também importante de ser entendido é o conceito de duração de cada nota e sua representação em uma partitura. A Figura 7 ilustra o entendimento deste conceito.

Figura 7 - Divisões Rítmicas

Nomes da figuras das notas musicais	Figuras das notas musicais	Representação do valor das notas musicais.	Valor relativo das notas musicais
Semibreve			1
Mínima			2
Semínima			4
Colcheia			8
Semicolcheia			16
Fusa			32
Semifusa			64

Fonte: (Károlyi, 1990)

2.2 MÚSICA E INTELIGÊNCIA ARTIFICIAL

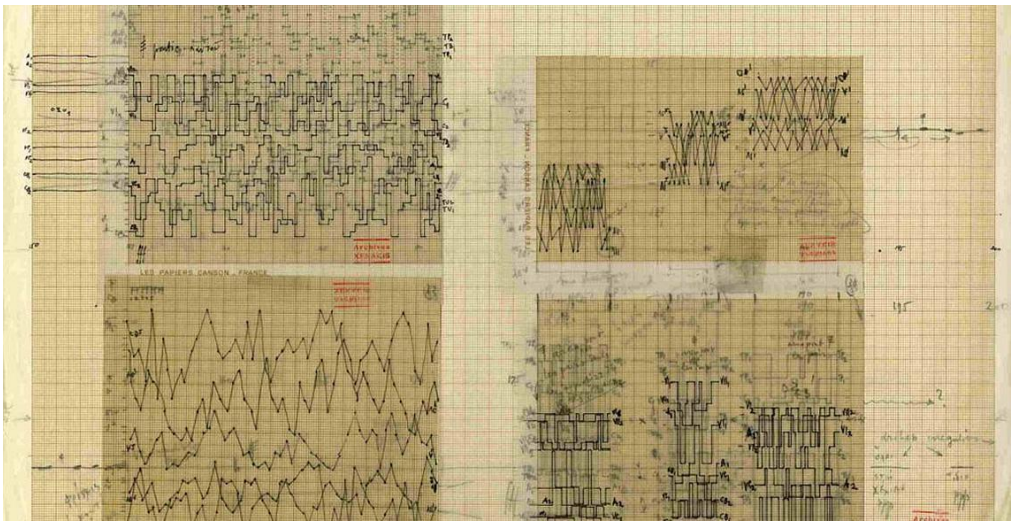
2.2.1 HISTÓRICO DA COMPOSIÇÃO ALGORÍTMICA

A ideia de utilizar regras e sistemas para automatizar a criação musical não é um fenômeno recente exclusivo da era digital. As raízes da composição baseada em algoritmo são datadas do século XVIII, com os “jogos de dados musicais”, um sistema que permitia a criação de valsas e outras peças através do lançamento de dados para selecionar compassos. Um desses jogos é atribuído a Wolfgang Amadeus Mozart, ilustrando uma atenção já antiga pela sistematização do processo criativo (HEDGES, 2011).

Contudo, foi com o advento dos computadores, em meados do século XX, que composições algorítmicas emergiram como um campo de pesquisa mais robusto. O crédito para o início desta nova era vai para Lejaren Hiller e Leonard Isaacson que, em 1957, utilizaram o computador ILLIAC I para gerar a *Illiac Suite for String Quartet*. Sendo esta, a primeira obra musical significativa composta com o auxílio direto de um computador, que, com princípios da teoria da informação e técnicas de Monte Carlo para tomar decisões composicionais, desde a geração das notas individuais, até a organização da estrutura formal da peça. (HILLER; ISAACSON, 1959)

Nas décadas seguintes, outros compositores e teóricos expandiram o campo de estudo. Iannis Xenakis foi pioneiro no uso de processos estocásticos, empregando cálculos de probabilidade complexos para organizar eventos sonoros e certas texturas musicais inovadoras e diferentes, denominadas músicas estocásticas (XENAKIS, 1992). A Figura 8 ilustra um trabalho de Xenakis.

Figura 8 - Manuscrito da obra "Metastasis"



Fonte: (XENAKIS, 1992)

Um ponto de mudança ocorreu a partir dos anos 1980, com o trabalho de David Cope com o “EMI – Experimentos em Inteligência Musical”. Diferentemente das abordagens anteriores, que geravam música a partir de regras matemáticas ou aleatórias, o sistema de Cope foi projetado para que fosse possível analisar obras de algum compositor específico e gerar novas peças que imitassem aquele estilo. O EMI

funcionava com base em um sistema de regras e conhecimento musical explícito, representando um passo fundamento em direção à simulação da criatividade (COPE, 1991).

A transição para o século XXI marcou a transformação mais recente no campo, com o crescimento das técnicas de aprendizado de máquina (*Machine Learning*) e, mais especificamente, do *deep learning*. Arquiteturas de redes neurais, como as LSTMs e os *Transformers*, aprendem padrões, estilos e estruturas musicai, ao serem treinadas com conjuntos de dados musicais pré-existentes. Projetos como o *Magenta*, do *Google*, e o *Jukebox*, da *OpenAI*, são exemplos desta nova fase, na qual os modelos não apenas imitam um estilo, mas são capazes de gerar obras complexas com um nível de coerência antes inatingível (BRIOT; HADJERES; PACHET, 2020).

2.2.2 Representação Musical para Modelos de Redes

Uma etapa fundamental do processo de geração de música por redes neurais é a representação dos dados musicais. Um computador não é capaz de “compreender” a música em sua forma conceitual/auditiva; ele opera sobre dados numéricos estruturados. Com isso, a informação musical precisa ser traduzida para um formato que a máquina consegue processar. Entende-se que essa tradução ou representação não é apenas uma decisão técnica, mas também metodológica, que vai definir as capacidades e limitações do modelo que será utilizado. De acordo com Briot, Hadjeres e Pachet (2020), as abordagens de representação podem ser categorizadas nas seguintes três grandes vertentes: representação simbólica, representação do áudio bruto e representação parametrizada.

Uma das tecnologias mais importantes que viabiliza a representação simbólica da música é o padrão MIDI (*Musical Instrument Digital Interface*). É crucial entender que MIDI não é um formato de áudio; ele não contém sons gravados. Em vez disso, o MIDI é um protocolo de comunicação, uma linguagem digital que descreve eventos musicais. Um arquivo MIDI armazena um conjunto de instruções sobre como uma peça deve ser tocada, incluindo qual nota pressionar (altura), quando pressionar (início), quando soltar (fim) e com que força (velocidade), entre outros parâmetros de controle. Por ser um

formato leve e estruturado, que separa as notas de sua sonoridade final, o MIDI se tornou o padrão de fato para a pesquisa em geração musical simbólica (ROADS, 1996).

A representação simbólica é a abordagem mais tradicional e direta. Nela, a música é tratada de forma similar a uma linguagem, composta por uma sequência de símbolos discretos. Esses símbolos podem ser representações textuais de notas (“C4”), acordes (“C#m”), ou extraídos diretamente de eventos em arquivos no formato MIDI, que indicam o início e o fim de uma nota. Pensando no modelo LSTM, a peça musical é convertida em uma longa sequência de “palavras” que representam as notas e os acordes. Neste modelo podemos destacar a simplicidade e a eficiência computacional do método. Entretanto, sua maior desvantagem é a possível perda de informação: ao focar em um detalhe como a sequência de notas, detalhes como ritmo, duração precisa das notas e dinâmica (intensidade) são frequentemente descartados ou excessivamente simplificados.

Outra abordagem é a representação do áudio bruto. O Jukebox (OpenAI) e o WaveNet (Google), operam diretamente sobre a forma de onda do som (um arquivo WAV). Neste tipo de representação, temos mais alta fidelidade nas nuances da música: timbre exato dos instrumentos, acústica do ambiente, dinâmica da performance e os efeitos da produção envolvidos, beirando o potencial do realismo. Como contraponto, temos uma exigência computacional mais elevada, pois apenas um único segundo de um áudio, pode contar milhares de pontos de dados. Tal limitação torna o treinamento desses modelos um processo lento e caro, exigindo uma estrutural computacional maior e mais robusta, sendo um dos grandes desafios do campo a manutenção da coerência estrutural em áudios mais longos.

Como meio termo entre a interpretação simbólica e a complexidade de trabalhar com o áudio bruto, existe a representação parametrizada. Esta abordagem vai tratar a música como uma sequência de eventos, mas cada evento é descrito por um conjunto de parâmetros musicais explícitos. O modelo Transformer emprega esta metodologia. Ao invés de um único símbolo, cada nota é representada por um vetor, que carrega características como altura, duração e intervalo de tempo desde a nota anterior. Este método se mostra eficiente pois mantém uma informação rítmica e temporal essencial,

que acaba sendo perdida no método utilizado pelo LSTM, sem ter grandes limitações computacionais.

2.3 REDES NEURAIS

2.3.1 DEFINIÇÃO E HISTÓRIA

As Redes Neurais Artificiais (RNAs) são modelos computacionais que formam a base do *Deep Learning* (Aprendizagem Profunda), um dos mais proeminentes subcampos da Inteligência Artificial (IA). Inspiradas na estrutura e no funcionamento do cérebro humano, essas redes consistem em unidades de processamento interconectadas – os neurônios artificiais – dispostas em múltiplas camadas. A principal característica do *Deep Learning* é justamente a profundidade dessas camadas, que permite ao modelo aprender representações de dados em níveis de abstração progressivamente mais altos, possibilitando o reconhecimento de padrões de extrema complexidade (GOODFELLOW; BENGIO; COURVILLE, 2016).

A ascensão do *Deep Learning* nas últimas décadas não se deve a uma única inovação teórica, mas sim a uma confluência de fatores tecnológicos. Dois pilares foram fundamentais para viabilizar a aplicação prática e o sucesso das redes neurais profundas: o aumento exponencial do poder computacional, especialmente com o advento das Unidades de Processamento Gráfico (GPUs), que se mostraram excepcionalmente eficientes para os cálculos matriciais massivos exigidos pelas redes; e a disponibilidade de grandes volumes de dados (o chamado *big data*), essenciais para o treinamento eficaz desses modelos. Sem essa infraestrutura, redes com milhões de parâmetros seriam teoricamente interessantes, mas praticamente intratáveis.

Essa evolução representa uma mudança de paradigma fundamental: em vez de serem programadas com regras lógicas explícitas, como nos sistemas especialistas do passado, as redes neurais modernas as "aprendem" diretamente dos dados. A distinção entre uma rede neural básica e uma rede de *Deep Learning* reside, portanto, na sua profundidade. Por convenção, uma rede neural que possui mais de uma camada oculta (totalizando mais de três camadas, incluindo entrada e saída) já é considerada um algoritmo de *Deep Learning*, capaz de aprender hierarquias de características que modelos mais rasos não conseguem capturar.

2.3.2 ESTRUTURA DE REDES NEURAIS

As Redes Neurais Artificiais constituem um conjunto de algoritmos de aprendizado de máquina inspirados na estrutura do cérebro humano. Compostas por múltiplas camadas de neurônios artificiais interconectados, projetadas para modelar e aprender relações complexas em dados.

A estrutura básica de uma rede neural compreende os seguintes componentes essenciais:

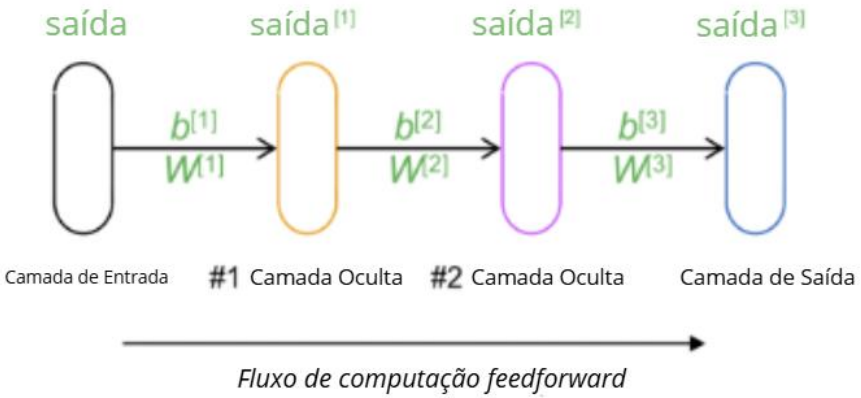
a) Camada de Entrada (Input Layer): Porta de entrada da rede, onde os dados brutos são inseridos para processamento. Cada neurônio nesta camada corresponde a uma característica do conjunto de dados, como a altura de uma nota musical ou o valor de um pixel em uma imagem.

b) Camadas Ocultas (Hidden Layers): Localizadas entre a entrada e a saída, são onde a maior parte da extração de características e do aprendizado ocorre. O termo "profundo" (Deep) em Deep Learning refere-se à presença de múltiplas camadas ocultas. Por convenção, uma rede com mais de uma camada oculta é considerada uma rede neural profunda.

c) Camada de Saída (Output Layer): É a camada final, responsável por produzir o resultado do modelo. A sua configuração depende da tarefa, como classificar um gênero musical ou, neste trabalho, prever a próxima nota em uma sequência.

Dentro dessa estrutura, os neurônios operam com base em dois componentes chave: peso (*weights)* e vieses (*biases).* As entradas recebidas por um neurônio são multiplicadas por pesos, que denotam a força de cada conexão. Um viés é então adicionado a essa soma ponderada. O resultado é passado por uma função de ativação, que introduz não-linearidades no modelo, capacitando a rede a aprender padrões complexos que vão além de simples relações lineares. O processo pelo qual os dados percorrem a rede, da camada de entrada até a saída é conhecido como propagação direta *(forward propagation).* É nesta etapa que a rede gera sua previsão inicial, antes de iniciar o processo de aprendizado para corrigir seu erros. É possível visualizar de maneira mais ilustrativo o exemplo de uma Rede Neural na Figura 9.

Figura 9 - Exemplo de uma Rede Neural *Feedforward* (abstrato) e computação *pipeline*.



Fonte: Briot, Hadjeres e Pachet (2020, p. 79)

2.3.3 FUNCIONAMENTO DAS REDES NEURAIS

Após a definição da estrutura de uma rede neural, seu "funcionamento" refere-se primordialmente ao processo pelo qual ela aprende a partir dos dados. Este processo não é um evento único, mas um ciclo iterativo de previsão e correção, conhecido como treinamento. O objetivo é ajustar os parâmetros internos da rede (pesos e vieses) para que ela possa mapear com precisão as entradas às saídas desejadas. Este ciclo é universal para a maioria das redes neurais supervisionadas e pode ser dividido em cinco etapas fundamentais (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.3.3.1 Preparação dos Conjuntos de Dados

Antes que qualquer processo de aprendizado possa ocorrer, os dados brutos devem ser submetidos a uma rigorosa etapa de preparação. No desenvolvimento de um modelo de *deep learning*, esta fase é frequentemente a mais trabalhosa e uma das mais críticas para o sucesso do projeto, especialmente em um domínio de alta complexidade como a música. A preparação envolve não apenas a escolha de uma representação digital adequada para a informação musical, mas também a divisão estratégica do dataset em conjuntos de dados distintos que garantam um treinamento robusto e uma avaliação honesta do modelo.

Conforme discutido anteriormente, a tradução da música para um formato numérico é um pré-requisito fundamental. A escolha da representação — seja ela

simbólica, como o piano-roll, ou parametrizada — está intrinsecamente ligada à arquitetura da rede, definindo a estrutura das camadas de entrada e saída. Uma estratégia comum para representações temporais é a discretização do tempo através de um passo de tempo (time step) fixo, frequentemente definido como a menor duração rítmica presente no dataset (por exemplo, uma semicolcheia). Isso garante que a duração de todas as notas seja um múltiplo inteiro desse passo, simplificando a tarefa de modelagem sequencial para a rede (BRIOT; HADJERES; PACHET, 2020).

Um outro aspecto a ser considerado na preparação dos dados é o tamanho do *dataset* a ser utilizado. Especialmente em *deep learning*, onde o desempenho do modelo é altamente dependente da quantidade de dados, o *overfitting* é um risco constante quando o *dataset* é limitado. Para mitigar isso, aplicam-se técnicas de aumento de dados, que geram novas amostras de treinamento a partir das existentes, aplicando transformações realistas. No domínio musical, a técnica mais eficaz e difundida é a transposição, que consiste em alterar a tonalidade de uma peça. Por exemplo, uma sonata em Dó maior pode ser transposta para todas as outras onze tonalidades, multiplicando por doze a quantidade de dados de treinamento. Essa técnica não apenas aumenta o volume de dados, mas também ensina ao modelo que padrões melódicos e harmônicos são independentes da tonalidade, melhorando drasticamente sua capacidade de generalização (BRIOT; HADJERES; PACHET, 2020; CHOLLET, 2021).

2.3.3.2 Divisão Estratégica dos Dados: Treino, Validação e Teste

O princípio central do aprendizado de máquina é a generalização: a capacidade de um modelo performar bem em dados novos e nunca vistos. Para desenvolver e medir essa capacidade, o conjunto de dados total é particionado em três subconjuntos independentes, cada um com um propósito distinto:

O primeiro conjunto é o Conjunto de Treinamento (*Training Set*): Este é o "livro-texto" do modelo, compreendendo a maior parte dos dados disponíveis. É exclusivamente a partir deste conjunto que a rede neural aprende, ajustando seus pesos e vieses através do ciclo de treinamento. A totalidade do "conhecimento musical" do modelo — seu entendimento de harmonia, melodia e ritmo — é derivada dos padrões presentes neste conjunto.

O segundo conjunto é o Conjunto de Validação (*Validation Set*). Se o conjunto de treinamento é o material de estudo, o conjunto de validação funciona como um "teste simulado" periódico. Trata-se de uma amostra de dados que o modelo não utiliza para aprender, mas que serve para monitorar seu desempenho durante o treinamento. A performance na validação é um indicador crucial da capacidade de generalização do modelo e é a principal ferramenta para diagnosticar os seguintes dois problemas comuns:

a) *Underfitting* (Subajuste): Ocorre quando o modelo é simples demais para capturar a complexidade dos dados. Nesse caso, o erro será alto tanto no treinamento quanto na validação.

b) *Overfitting* (Sobreajuste): Ocorre quando o modelo se torna complexo demais e começa a "memorizar" os dados de treinamento, incluindo seus ruídos e particularidades, em vez de aprender os padrões subjacentes. O sintoma clássico do *overfitting* é observado quando o erro no conjunto de treinamento continua a diminuir, enquanto o erro no conjunto de validação começa a estagnar ou aumentar. Monitorar a perda de validação permite aplicar técnicas como a parada antecipada (*early stopping*), onde o treinamento é interrompido no momento em que a performance de generalização para de melhorar.

O terceiro conjunto é o Conjunto de Teste (*Test Set*): Este conjunto é o "exame final". Ele deve ser mantido "intocado" durante todo o processo de treinamento e ajuste de hiperparâmetros. É utilizado apenas uma vez, ao final, para fornecer uma avaliação final e imparcial da performance do modelo. O resultado no conjunto de teste é a estimativa mais confiável de como o modelo se comportará no mundo real, com dados completamente novos.

A forma como os dados são divididos é de extrema importância. Para muitos problemas, uma divisão aleatória simples é suficiente. No entanto, para dados com estrutura inerente, como a música, essa abordagem é falha. Dividir aleatoriamente as notas de uma peça musical poderia fazer com que o início da melodia ficasse no treino e o final no teste, uma forma de "vazamento de dados" que não mede a real capacidade de composição do modelo. A prática recomendada é, portanto, a divisão por grupo, separando os dados por peças musicais ou por compositores, para garantir que o modelo

seja testado em material genuinamente novo (GOODFELLOW; BENGIO; COURVILLE, 2016).

Para *datasets* menores ou para obter uma estimativa de performance ainda mais robusta, pode-se empregar a validação cruzada (*k-fold cross-validation*). Nesta técnica, o conjunto de treinamento é dividido em k partes (ou "*folds*"). O treinamento é executado k vezes; a cada vez, uma parte diferente é usada como conjunto de validação e as k-1 partes restantes são usadas para o treinamento. A métrica de performance final é a média dos resultados obtidos nas k rodadas. Isso garante que cada amostra de dados seja usada tanto para treinar quanto para validar o modelo, resultando em uma avaliação de desempenho mais estável e confiável, ao custo de um maior tempo computacional (CHOLLET, 2021).

2.3.3.3 Aplicação da Propagação Direta

A propagação direta, também conhecida como *forward pass* ou *feedforward propagation*, é o processo fundamental pelo qual uma rede neural processa uma entrada para gerar uma saída ou predição. É a primeira etapa do ciclo de treinamento e a única etapa utilizada durante a fase de inferência (quando o modelo treinado é usado para fazer previsões em novos dados). De forma conceitual, a propagação direta é a "linha de montagem" da rede em pleno funcionamento, um fluxo unidirecional de informação que transforma dados brutos em um resultado significativo (GOODFELLOW; BENGIO; COURVILLE, 2016).

O processo ocorre através de uma sequência de cálculos em cascata, camada por camada, com as seguintes etapas:

a) Entrada na Rede: O ciclo se inicia quando um vetor de características, representando uma amostra de dados (por exemplo, os pixels de uma imagem ou os atributos de uma nota musical), é apresentado à camada de entrada.

b) Cálculo nas Camadas Ocultas: A partir da camada de entrada, a informação flui para a primeira camada oculta. Cada neurônio nesta camada calcula sua saída realizando a soma ponderada de todas as saídas da camada anterior, adicionando seu viés e aplicando sua função de ativação. Matematicamente, para uma camada inteira, isso é eficientemente implementado como uma multiplicação de matrizes entre a matriz

de pesos da camada e o vetor de saídas da camada anterior, seguida pela adição do vetor de viés e a aplicação da função de ativação.

c) Propagação Sequencial: O vetor de saídas de uma camada oculta torna-se o vetor de entradas para a camada seguinte. Este processo se repete sucessivamente para todas as camadas ocultas da rede. A cada etapa, a rede aprende a extrair características cada vez mais abstratas e complexas dos dados.

d) Geração na Camada de Saída: Finalmente, a última camada oculta passa seus resultados para a camada de saída. A configuração desta camada, especialmente sua função de ativação (por exemplo, Softmax para classificação multiclasse ou uma função linear para regressão), transforma as representações internas da rede na predição final, no formato desejado para a tarefa em questão.

Este fluxo computacional, do início ao fim, gera a predição inicial da rede. O resultado desta propagação é então utilizado para calcular o erro do modelo, que servirá de base para o processo de aprendizado através da retropropagação.

No contexto da geração musical, a propagação direta é o mecanismo pelo qual o modelo "compõe" ativamente. Sua função é responder à pergunta: "dada a sequência de notas até agora, qual deve ser a próxima nota?". Em modelos simbólicos autorregressivos, como os LSTMs e Transformers deste trabalho, o processo ocorre de forma iterativa.

A rede recebe como entrada uma sequência de eventos musicais, codificada numericamente. Através da propagação direta, ela processa essa sequência e, em sua camada de saída, gera um vetor de probabilidades (*logits* passados por uma função Softmax). Este vetor atribui uma probabilidade a cada possível evento musical no vocabulário do modelo (por exemplo, a cada uma das 128 notas MIDI). A nota com a maior probabilidade é a predição do modelo. Para continuar a composição, esta nota recém-gerada é adicionada ao final da sequência de entrada, e uma nova propagação direta é executada para prever a próxima nota. Este ciclo de "prever e anexar" é o que permite ao modelo construir uma peça musical coesa, um evento de cada vez, onde cada nova nota é gerada com base em todo o contexto musical que a precedeu (BRIOT; HADJERES; PACHET, 2020).

2.3.3.4 Cálculo da Função de Perda Total

Após a rede neural realizar a propagação direta e gerar uma predição, é necessário um mecanismo para quantificar a qualidade dessa predição. A função de perda (ou função de custo) é a ferramenta matemática que cumpre esse papel. Ela funciona como uma medida de erro, calculando a discrepância ou a "distância" entre a saída prevista pelo modelo ($\hat{y}$) e o valor real esperado (y), que é o alvo do conjunto de treinamento. O resultado da função de perda é um valor escalar único que serve como um guia para o processo de otimização: quanto maior o valor, pior a performance do modelo; quanto menor, melhor. O objetivo fundamental de todo o processo de treinamento é, portanto, encontrar um conjunto de pesos e vieses que minimize o valor desta função (CHOLLET, 2021).

O processo de cálculo ocorre a cada iteração do treinamento sobre um lote (*batch*) de dados. Para cada amostra no lote, a perda é calculada individualmente e, ao final, essas perdas são agregadas (geralmente através de uma média) para se obter a perda total do lote. Esse valor final pode ser visualizado como um ponto em uma "paisagem de perda" (*loss landscape*), uma superfície multidimensional onde cada eixo representa um parâmetro do modelo e a altitude representa o erro. O objetivo do treinamento é "navegar" por essa paisagem para encontrar o vale mais profundo, que corresponde ao erro mínimo.

A escolha da função de perda é uma decisão de *design* crucial, pois ela define o objetivo que a rede irá otimizar. As mais comuns são:

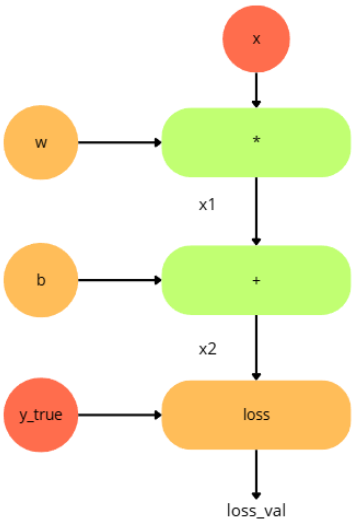
a) Erro Quadrático Médio (MSE - *Mean Squared Error*): Padrão para problemas de regressão, onde o objetivo é prever um valor contínuo. Sua fórmula, $MSE = (1/n) * \Sigma(y_i - \hat{y}_i)^2$, calcula a média dos quadrados das diferenças entre os valores reais (y_i) e os previstos ($\hat{y}_i$). Ao elevar o erro ao quadrado, o MSE penaliza desproporcionalmente os erros maiores, tornando o modelo mais sensível a outliers.

b) Entropia Cruzada (*Cross-Entropy*): É a função de perda padrão para problemas de classificação. Derivada da teoria da informação, ela mede a divergência entre duas distribuições de probabilidade: a distribuição prevista pelo modelo e a distribuição verdadeira (onde a classe correta tem probabilidade 1 e as outras 0). Minimizar a entropia cruzada é equivalente a maximizar a probabilidade que o modelo atribui à

classe correta. Ela se manifesta como Entropia Cruzada Binária para classificação de duas classes e Entropia Cruzada Categórica para problemas com múltiplas classes (GOODFELLOW; BENGIO; COURVILLE, 2016).

Toda a sequência de operações realizadas durante a propagação direta, desde as entradas brutas até o cálculo final da perda, pode ser representada como um grafo computacional. Este grafo mapeia cada operação matemática (multiplicação, adição, função de ativação) como um nó, deixando explícita a maneira como os parâmetros da rede influenciam o valor final da perda. Conforme ilustrado na Figura 10, essa estrutura é fundamental, pois é a partir dela que frameworks como o TensorFlow conseguem aplicar a regra da cadeia de forma automática e eficiente para calcular os gradientes durante a retropropagação (CHOLLET, 2021).

Figura 10 - Um exemplo básico de um grafo computacional.



Fonte: Adaptado de Chollet (2021).

A escolha da função de perda no domínio musical traduz o objetivo artístico em um alvo matemático. Ela define o que o modelo considerará uma "boa" ou "má" nota, harmonia ou ritmo.

Para a vasta maioria dos modelos de geração musical simbólica, como os LSTMs e Transformers, que são objeto deste estudo, a tarefa é prever o próximo evento musical a partir de um vocabulário finito de possibilidades (por exemplo, as 128 notas MIDI). Este é, fundamentalmente, um problema de classificação multiclasse.

Consequentemente, a Entropia Cruzada Categórica é a função de perda mais utilizada. Durante o treinamento, se a próxima nota correta em uma peça de Bach é um Dó central (MIDI 60), a função de perda penalizará o modelo proporcionalmente ao quão baixa for a probabilidade que ele atribuiu a essa nota. Ao minimizar essa perda, a rede aprende as complexas relações contextuais que tornam o Dó central a continuação mais provável naquela situação musical específica (BRIOT; HADJERES; PACHET, 2020).

Embora menos comum para a geração de sequências de notas, o Erro Quadrático Médio (MSE) encontra sua aplicação quando o objetivo é prever atributos musicais contínuos. Por exemplo, um modelo projetado para adicionar expressividade a uma melodia poderia ser treinado para prever a velocidade (intensidade) de cada nota, um valor contínuo entre 0 e 127. Nesse caso, o MSE seria uma escolha adequada para medir o erro entre a velocidade prevista e a velocidade real de uma performance humana.

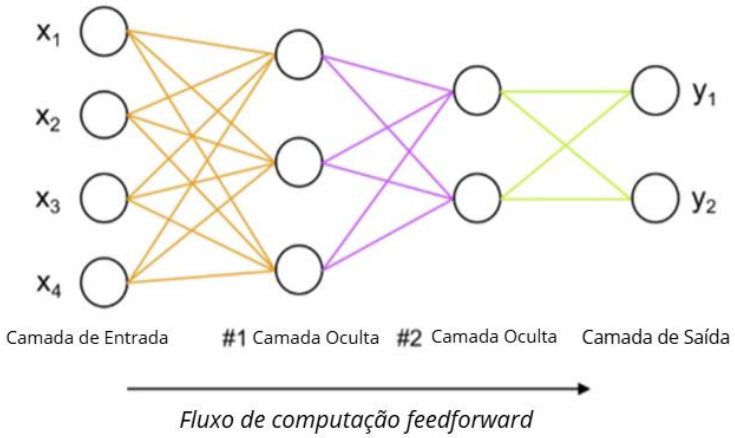
Além disso, pesquisadores frequentemente projetam funções de perda customizadas para embutir conhecimento musical no processo de treinamento. Uma perda customizada poderia, por exemplo, adicionar um termo de penalidade se a rede gerar intervalos harmônicos dissonantes que são estilisticamente inadequados para o dataset de treinamento. De forma análoga, em abordagens que utilizam Aprendizado por Reforço (RL), uma função de recompensa é definida para guiar o modelo, premiando-o por gerar música que segue certas regras teóricas (como resolver tensões harmônicas) e penalizando-o por comportamentos indesejados, como a repetição excessiva de notas (JAQUES *et al.*, 2017).

2.3.3.5 Aplicação de *Backpropagation* para Cálculo de Gradientes

Após a propagação direta ter gerado uma predição e a função de perda ter quantificado o erro dessa predição, a rede neural precisa de um mecanismo para aprender com seus erros e ajustar seus parâmetros internos. A retropropagação (*backpropagation*) é o algoritmo que torna esse aprendizado possível, sendo considerado o motor do *deep learning* moderno. Popularizado por Rumelhart, Hinton e Williams (1986), seu objetivo é calcular de forma eficiente o gradiente da função de perda em relação a cada peso e viés da rede.

De forma conceitual, a retropropagação opera sobre a estrutura da rede, como a representada na Figura 11, mas no sentido inverso: do final para o começo.

Figura 11 - Diagrama esquemático de uma rede neural *feedforward*



Fonte: Briot, Hadjeres e Pachet (2020, p. 68)

O processo é uma aplicação engenhosa da regra da cadeia (*chain rule*) do cálculo diferencial. Como a rede neural é, em essência, uma grande função matemática composta por muitas operações aninhadas (multiplicações, somas, ativações), a regra da cadeia permite decompor o cálculo do gradiente total em uma série de cálculos locais. O algoritmo inicia calculando o gradiente do erro em relação à saída da última camada. Em seguida, ele propaga esse erro para trás, camada por camada, calculando em cada uma o quanto seus pesos e vieses contribuíram para o erro da camada seguinte. Pode-se fazer uma analogia com a gestão de um grande projeto: após a entrega final (a predição), o gerente (o algoritmo) analisa o resultado e rastreia para trás, avaliando a contribuição de cada equipe e de cada indivíduo (os neurônios e pesos) para os sucessos e falhas do projeto, para saber exatamente onde os ajustes devem ser feitos.

O resultado final deste processo é o gradiente, um vetor multidimensional que aponta, para cada parâmetro da rede, a direção de maior crescimento da função de perda. Para minimizar o erro, o algoritmo de otimização (que veremos na próxima seção) simplesmente "dará um passo" na direção oposta a este gradiente. Visualmente, se imaginarmos a função de perda como uma paisagem montanhosa (conforme ilustra a

Figura 12), o gradiente em qualquer ponto nos diz qual é a direção da subida mais íngreme. Para encontrar o “vale” (o erro mínimo), basta seguir na direção oposta.

Figura 12 - Gráfico da função de perda

Fonte: Chollet (2021, p.53)

A importância da retropropagação reside em sua eficiência. Antes de sua popularização, o cálculo de gradientes em redes multicamadas era um grande gargalo computacional. O *backpropagation* reduziu drasticamente essa complexidade, tornando o treinamento de redes neurais profundas uma realidade prática. Hoje, *frameworks* modernos como TensorFlow e PyTorch abstraem completamente essa complexidade através da diferenciação automática (*automatic differentiation*). Eles constroem um grafo computacional de todas as operações durante a propagação direta e, em seguida, percorrem esse grafo no sentido inverso para calcular os gradientes, eliminando a necessidade de implementação manual do algoritmo (CHOLLET, 2021).

No contexto da geração musical, a retropropagação é o processo que transforma um "ouvinte" passivo (a rede durante a propagação direta) em um "aluno" ativo. É através deste algoritmo que a rede neural aprende as complexas e muitas vezes implícitas "regras" da música, como a teoria harmônica, as estruturas rítmicas e os contornos melódicos característicos de um estilo. Ao ajustar os pesos com base no erro retropropagado, a rede refina seu "gosto" musical, aprendendo, por exemplo, que após

um acorde de dominante em uma peça de Bach, a resolução em um acorde de tônica é altamente provável.

Para arquiteturas especializadas em sequências, como as Redes Neurais Recorrentes (RNNs) e as LSTMs, uma variante do algoritmo chamada *Backpropagation Through Time* (BPTT) é utilizada. Como a saída de uma RNN em um determinado passo de tempo depende não apenas da entrada atual, mas também do estado oculto dos passos anteriores, o erro também precisa ser propagado "para trás no tempo". O BPTT, portanto, "desdobra" a rede ao longo da sequência temporal como se fosse uma única rede *feedforward* muito profunda, e então aplica o algoritmo de retropropagação padrão. Isso permite que o modelo aprenda as dependências temporais que são a essência da música. É através do BPTT que um modelo LSTM aprende a conectar uma frase melódica em sua introdução com uma variação dessa mesma frase minutos depois. No entanto, o BPTT em RNNs simples sofre com o problema dos gradientes que desaparecem ou explodem ao serem propagados por muitos passos de tempo, uma limitação que a arquitetura com portões das LSTMs foi projetada para mitigar (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.3.3.6 Atualização dos pesos e vieses da rede

Após a fase de propagação direta ter produzido uma predição, a função de perda ter quantificado o erro, e a retropropagação ter calculado a contribuição de cada parâmetro para esse erro (os gradientes), chegamos à etapa final e mais crucial do ciclo de treinamento: a atualização dos pesos e vieses. Este processo é o "motor" do aprendizado em redes neurais. Se a retropropagação é o diagnóstico que aponta o que está errado e quem é o responsável, a atualização de pesos é o tratamento que corrige o problema. É o mecanismo pelo qual a rede utiliza a informação do erro para se autocorrigir, ajustando seus parâmetros internos de modo a minimizar a função de perda em iterações futuras e, consequentemente, melhorar seu desempenho na tarefa designada (BRIOT; HADJERES; PACHET, 2020).

Essa atualização é orquestrada por algoritmos de otimização, que são responsáveis por ditar exatamente como os parâmetros devem ser modificados com base nos gradientes calculados.

O otimizador básico é a Descida de Gradiente (*Gradient Descent*). A sua operação pode ser compreendida através da analogia da paisagem de perda (*loss landscape*): uma representação abstrata e multidimensional onde cada ponto no "terreno" corresponde a uma configuração específica dos pesos da rede e sua "altitude" corresponde ao valor do erro (perda). O objetivo do treinamento é encontrar o ponto mais baixo desta paisagem, o chamado mínimo global.

O gradiente, calculado pela retropropagação, aponta em cada ponto para a direção da subida mais íngreme. Portanto, para "descer a montanha" e minimizar o erro, o algoritmo simplesmente ajusta os parâmetros "dando um passo" na direção oposta ao gradiente. Esse processo é visualizado na Figura 13 e formalizado pela equação (1) de atualização (BRIOT; HADJERES; PACHET, 2020; CHOLLET, 2021).

$$\theta_{novo} = \theta_{antigo} - \alpha \cdot \frac{\partial J[\theta(h)]}{\partial \theta} \quad (1)$$

Onde tem-se

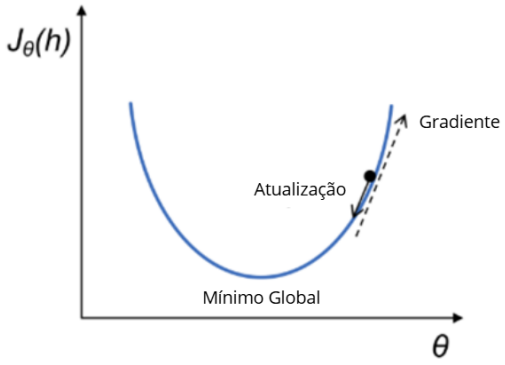
θ_i (Parâmetros) - Refere-se aos parâmetros ajustáveis do modelo, que incluem os pesos (W) e os vieses (b) da rede neural. Inicialmente, esses parâmetros são frequentemente preenchidos com valores aleatórios. A otimização busca encontrar os valores de θ_i que resultam no melhor desempenho do modelo.

J_θ(h) (Função de Custo ou Perda): É uma função que quantifica a discrepância entre as previsões do modelo e os valores reais esperados. O objetivo do processo de treinamento é minimizar essa função, de forma que o modelo faça previsões cada vez mais precisas.

∂J_θ(h)/∂θ_i (Gradiente da Função de Custo): Representa a derivada parcial da função de custo em relação a um parâmetro específico. O gradiente indica a direção de maior crescimento da função de custo. Para minimizar a função, os parâmetros são ajustados na direção oposta ao gradiente. O cálculo desses gradientes é feito eficientemente pelo algoritmo de Retropropagação.

α (Taxa de Aprendizado) -: É um hiperparâmetro crucial que controla o tamanho do passo dado durante cada atualização dos parâmetros. A escolha adequada de α é vital para a convergência e eficiência do treinamento.

Figura 13 - Gradient Descent



Fonte: Briot, Hadjeres e Pachet (2020, p. 56)

Na prática, a paisagem de perda de redes profundas é extremamente complexa, cheia de vales (mínimos locais), platôs e pontos de sela, o que torna a navegação um desafio. Para tornar o processo mais eficiente e robusto, diversas variantes do algoritmo foram desenvolvidas.

Uma das variantes mais utilizadas é a Descida de Gradiente Estocástico (SGD). Em vez de calcular o gradiente sobre todo o conjunto de dados (o que seria computacionalmente dispendioso), o SGD o faz para pequenos lotes aleatórios (mini-batches). Essa abordagem introduz ruído no cálculo do gradiente, mas acelera drasticamente o treinamento e, paradoxalmente, a natureza ruidosa das atualizações pode ajudar o modelo a escapar de mínimos locais rasos (GOODFELLOW; BENGIO; COURVILLE, 2016).

Uma limitação do SGD é o uso de uma única taxa de aprendizado para todos os parâmetros. Otimizadores adaptativos resolvem isso calculando taxas de aprendizado individuais para cada parâmetro. O mais proeminente é o Adam (*Adaptive Moment Estimation*), que combina as seguintes duas ideias poderosas:

a) Momento (Momentum): Acumula uma média móvel dos gradientes passados, funcionando como uma "inércia" ou "momentum" que acelera a descida em direções consistentes e amortece as oscilações.

b) Taxas de Aprendizado Adaptativas (como no RMSProp): Ajusta a taxa de aprendizado para cada parâmetro, diminuindo-a para parâmetros que recebem atualizações grandes e frequentes e aumentando-a para aqueles com atualizações pequenas ou raras.

Ao integrar ambas as técnicas, o Adam se mostra extremamente eficaz e robusto para uma vasta gama de problemas e arquiteturas, tornando-se a escolha padrão na maioria das aplicações de *deep learning* (KINGMA; BA, 2014).

A escolha do otimizador e a configuração da taxa de aprendizado são decisões críticas que impactam profundamente a capacidade de um modelo aprender a compor música. Para modelos musicais, que frequentemente possuem milhões de parâmetros e são treinados em sequências longas e complexas, os otimizadores adaptativos como o Adam (ou suas variantes, como AdamW e Adamax) são quase universais.

A taxa de aprendizado (α) atua como um "termômetro" da criatividade e precisão do modelo durante o aprendizado. Uma taxa muito alta pode fazer com que o modelo nunca convirja, com a perda oscilando erraticamente, resultando em uma música caótica. Por outro lado, uma taxa muito baixa pode fazer com que o treinamento seja excessivamente lento e que o modelo fique "preso" em um mínimo local, aprendendo apenas padrões musicais muito simples e repetitivos. Para otimizar esse balanço, é comum o uso de agendadores de taxa de aprendizado (*learning rate schedulers*), que ajustam o valor de α dinamicamente durante o treinamento. Uma estratégia comum, especialmente no treinamento de Transformers, é usar um "aquecimento" (*warm-up*), onde a taxa de aprendizado começa baixa, aumenta linearmente por alguns milhares de passos e depois decai gradualmente. Isso ajuda a estabilizar o treinamento nas fases iniciais e a refinar o modelo nas fases finais (SOUZA; AVILA, 2018).

Conforme explica Chollet (2021), essa etapa de atualização dos pesos e vieses não é um evento isolado, mas sim parte de um *loop* de treinamento iterativo. Os dados de treinamento são processados em passes, onde cada "passo" envolve a propagação direta, o cálculo da perda, a retropropagação para obter os gradientes e, finalmente, a atualização dos parâmetros. Este ciclo se repete por um número definido de "épocas" (um passe completo por todo o conjunto de dados de treinamento) ou até que o

desempenho da rede em um conjunto de validação atinja um nível satisfatório ou pare de melhorar.

O objetivo final deste processo contínuo de ajuste é permitir que a rede neural aprenda os padrões e as estruturas complexas inerentes aos dados. No contexto da geração musical, isso capacita o modelo a induzir automaticamente estilos musicais a partir de *corpora* arbitrários, produzindo então amostras que são coerentes, estruturalmente plausíveis e até mesmo próximas de estilos composicionais humanos. A profundidade das camadas, conforme mencionado, é o que permite esse aprendizado de características cada vez mais complexas e a melhoria da precisão ao longo do tempo (CHEN; HUANG; GOU, 2024).

2.3.4 HIPERPARÂMETROS DE REDES NEURAIS

Se os parâmetros de uma rede neural — seus pesos e vieses — são os conhecimentos aprendidos autonomamente a partir dos dados, os hiperparâmetros são as decisões estratégicas e configurações externas definidas pelo pesquisador antes do início do processo de treinamento. Eles não são aprendidos pela rede, mas governam a maneira como o aprendizado ocorre. Em uma analogia, se o treinamento é o ato de cozinhar um prato, os parâmetros são os sabores que se desenvolvem durante o cozimento, enquanto os hiperparâmetros são a receita, a temperatura do forno e o tempo de preparo definidos pelo cozinheiro (GOODFELLOW; BENGIO; COURVILLE, 2016).

A escolha e o ajuste fino (*tuning*) dos hiperparâmetros representam uma das tarefas mais críticas e empíricas da engenharia de *deep learning*. Uma configuração inadequada pode impedir que um modelo, mesmo com uma arquitetura poderosa, aprenda de forma eficaz, enquanto uma configuração bem ajustada pode levar a resultados de ponta. Dada a sua relevância, os hiperparâmetros podem ser categorizados de acordo com a fase do processo em que atuam: arquitetura, treinamento e geração.

2.3.4.1 Hiperparâmetros arquiteturais

Os hiperparâmetros arquiteturais são as decisões de mais alto nível no *design* de uma rede neural. Eles definem o "esqueleto" ou o *blueprint* do modelo, estabelecendo sua complexidade, capacidade de representação e os limites fundamentais do que ele é

capaz de aprender. Diferentemente de outras configurações que podem ser ajustadas durante o treinamento, as escolhas arquiteturais são, em geral, fixadas antes que a primeira amostra de dados seja processada.

Os principais hiperparâmetros são:

a) Profundidade da Rede (Número de Camadas);

b) Largura da Rede (Número de Neurônios por Camada); e

c) Função de Ativação.

2.3.4.1.1 Profundidade da Rede (Número de Camadas)

A profundidade, definida pelo número de camadas ocultas, é talvez a característica mais emblemática do *Deep Learning*. A intuição fundamental é que múltiplas camadas permitem o aprendizado hierárquico de características. As camadas iniciais da rede, mais próximas da entrada, aprendem a reconhecer padrões simples e locais. As camadas subsequentes, por sua vez, recebem como entrada as características aprendidas pelas camadas anteriores e as combinam para formar conceitos progressivamente mais complexos e abstratos (GOODFELLOW; BENGIO; COURVILLE, 2016).

No domínio da música, essa hierarquia é particularmente intuitiva. Uma primeira camada oculta pode aprender a reconhecer intervalos melódicos simples (terças, quintas) ou durações rítmicas básicas. A camada seguinte pode combinar esses intervalos para identificar arpejos ou motivos melódicos curtos. Camadas ainda mais profundas poderiam aprender a reconhecer progressões de acordes inteiras, estruturas de frases e, em modelos muito profundos, até mesmo relações temáticas de longo prazo que definem a estrutura de uma peça musical. A profundidade do modelo Music Transformer, por exemplo, é um fator chave em sua celebrada capacidade de gerar música com coerência em longos períodos (HUANG *et al.*, 2019a). Contudo, o aumento da profundidade também acarreta desafios, como um maior custo computacional e o risco acentuado do desaparecimento de gradiente, que, embora mitigado por arquiteturas como LSTMs e Transformers, ainda pode ser um fator limitante.

2.3.4.1.2 Largura da Rede (Número de Neurônios por Camada)

A largura de uma camada, definida pelo número de neurônios (ou unidades) que ela contém, determina a capacidade de representação daquela camada. Cada neurônio pode ser visto como um detector de um padrão específico; portanto, uma camada mais larga tem mais "espaço" para aprender um conjunto mais rico e diversificado de características em um determinado nível de abstração. Se a profundidade permite a hierarquia, a largura permite a riqueza de detalhes em cada nível hierárquico.

A largura de uma camada em um modelo musical impacta diretamente a complexidade dos padrões que ele pode internalizar. Uma camada oculta muito "estreita" (com poucos neurônios) pode não ter a capacidade necessária para representar a riqueza harmônica e melódica de um compositor como Chopin, resultando em música excessivamente simplista ou genérica. Em contrapartida, uma camada excessivamente "larga" aumenta o número de parâmetros (pesos e vieses) do modelo, o que não apenas eleva o custo computacional, mas também aumenta drasticamente o risco de *overfitting*. Um modelo muito largo pode começar a "memorizar" frases e sequências específicas do conjunto de treinamento, em vez de aprender as regras estilísticas subjacentes, perdendo assim a capacidade de gerar composições novas e originais.

2.3.4.1.3 Função de Ativação

Conforme detalhado na Seção 2.3.2, a função de ativação é o componente não-linear aplicado à saída de cada neurônio. A escolha da função de ativação para as camadas ocultas é um hiperparâmetro arquitetural crucial que influencia tanto a capacidade expressiva da rede quanto a estabilidade do seu treinamento.

A escolha da função de ativação está frequentemente atrelada à própria arquitetura do modelo. Em LSTMs, as funções Sigmoide e Tangente Hiperbólica (Tanh) são componentes intrínsecos e fixos de seus mecanismos de portões (gates), sendo fundamentais para a regulação do fluxo de informação. Já na arquitetura Transformer, as subcamadas *feed-forward* que existem dentro de cada bloco do modelo tipicamente utilizam a função ReLU (Unidade Linear Retificada) ou suas variantes (como GeLU), devido à sua eficiência computacional e por mitigarem o problema do desaparecimento de gradiente, permitindo o treinamento de modelos muito mais profundos (VASWANI *et al.*, 2017).

2.3.4.2 Hiperparâmetros de treinamento

De acordo com Chen, Huang e Gou (2024), estes hiperparâmetros controlam o processo de aprendizagem da rede. Se os hiperparâmetros arquiteturais definem o "corpo" do modelo, os hiperparâmetros de treinamento são os que governam sua "educação". Eles controlam diretamente o algoritmo de otimização e o processo de aprendizado, ditando a velocidade, a estabilidade e a eficácia com que o modelo ajusta seus pesos e vieses para minimizar a função de perda. Um ajuste inadequado nesta fase pode fazer com que até a arquitetura mais poderosa falhe em aprender de forma eficaz (DOMINGOS, 2012).

Vamos abordar a seguir os seguintes hiperparâmetros de treinamento:

a) Taxa de Aprendizado (*Learning Rate*);

b) Tamanho do Lote (*Batch Size*);

c) Número de Épocas (*Epochs*); e

d) Técnicas de Regularização.

2.3.4.2.1 Taxa de Aprendizado (*Learning Rate*)

A taxa de aprendizado (α) é, indiscutivelmente, o hiperparâmetro de treinamento mais influente. Ela determina o tamanho do passo que o algoritmo de otimização (como a Descida de Gradiente) dá na direção oposta ao gradiente a cada atualização de pesos. A escolha deste valor representa um balanço crítico: se a taxa for muito alta, o otimizador pode dar passos tão largos que "salta" sobre o ponto de erro mínimo na paisagem de perda, levando a oscilações ou à divergência do treinamento. Se for muito baixa, o treinamento pode se tornar excessivamente lento, com o risco de ficar "preso" em um mínimo local de baixa qualidade (GOODFELLOW; BENGI; COURVILLE, 2016).

No treinamento de modelos musicais, uma taxa de aprendizado dinâmica, controlada por agendadores (schedulers), é frequentemente uma necessidade. Uma estratégia comum, especialmente para Transformers, é o "aquecimento" (*warm-up*), onde a taxa começa baixa, aumenta gradualmente nos primeiros milhares de passos para estabilizar o início do aprendizado, e depois decai (VASWANI *et al.*, 2017). Isso permite que o modelo primeiro aprenda as estruturas musicais mais gerais e, em seguida, refine os detalhes mais sutis do estilo com passos de ajuste menores.

2.3.4.2.2 Tamanho do Lote (*Batch Size*)

O tamanho do lote define o número de exemplos de treinamento que são processados antes que os pesos do modelo sejam atualizados. Essa escolha impacta o balanço entre a precisão do gradiente e a eficiência computacional. Lotes maiores fornecem uma estimativa mais precisa do gradiente da função de perda, levando a uma convergência mais estável. No entanto, exigem significativamente mais memória (VRAM da GPU) e podem levar o modelo a convergir para mínimos "agudos", que às vezes generalizam pior. Lotes menores, por outro lado, introduzem ruído no processo de atualização, o que pode atuar como uma forma de regularização e ajudar o modelo a encontrar mínimos "planos", associados a uma melhor generalização (CHOLLET, 2021).

Dada a natureza sequencial e o comprimento dos dados musicais, o tamanho do lote é frequentemente limitado por restrições de memória. É impraticável carregar centenas de longas peças de piano na memória da GPU simultaneamente, um desafio prático descrito em diversos trabalhos de implementação (SOUZA; AVILA, 2018). Portanto, tamanhos de lote modestos (como 16, 32 ou 64 trechos musicais curtos) são comuns. A escolha do tamanho do lote está intrinsecamente ligada à taxa de aprendizado, e ajustar um geralmente requer o reajuste do outro para manter a estabilidade do treinamento.

2.3.4.2.3 Número de Épocas (Epochs)

Uma época representa um passe completo do algoritmo de aprendizado por todo o conjunto de dados de treinamento. O número de épocas, portanto, determina por quanto tempo o modelo irá "estudar" os dados. O desafio aqui é encontrar o ponto ótimo de treinamento. Poucas épocas resultarão em um modelo subajustado (underfit), que não aprendeu os padrões essenciais. Muitas épocas, por outro lado, levarão ao sobreajuste (overfit), onde o modelo começa a memorizar o conjunto de treinamento em vez de aprender regras generalizáveis (GOODFELLOW; BENGIO; COURVILLE, 2016).

Este hiperparâmetro é quase sempre ajustado em conjunto com a técnica de parada antecipada (early stopping). O modelo é treinado por um número potencialmente grande de épocas, mas sua performance no conjunto de validação é monitorada a cada passo. O treinamento é interrompido automaticamente quando a performance na validação para de melhorar (CHOLLET, 2021), garantindo que o modelo salvo seja

aquele com a melhor capacidade de generalização para compor novas músicas no estilo aprendido, em vez de plagiar trechos que ele já viu.

2.3.4.2.4 Técnicas de Regularização

A regularização refere-se a qualquer técnica que impõe restrições ao modelo para prevenir o *overfitting*. A mais comum é o *Dropout*, que, durante o treinamento, desativa aleatoriamente uma fração dos neurônios (e suas conexões) em cada etapa. Isso força a rede a aprender representações mais robustas e distribuídas, pois ela não pode depender de um pequeno subconjunto de neurônios para tomar decisões (SRIVASTAVA *et al.*, 2014). É análogo a treinar um time onde, a cada jogada, alguns jogadores são aleatoriamente retirados, forçando os restantes a aprenderem a colaborar de maneiras mais flexíveis.

O *dropout* é particularmente vital quando se treina em um *dataset* de nicho ou de um único compositor (como os corais de Bach). Sem regularização, o modelo poderia simplesmente memorizar as melodias e harmonias das poucas centenas de peças disponíveis. Ao aplicar o *dropout*, o modelo é forçado a aprender as regras estilísticas subjacentes da música barroca, em vez de apenas as instâncias específicas, capacitando-o a gerar corais novos, mas que soam autenticamente como Bach (HADJERES; PACHET; NIELSEN, 2017a).

2.3.5 Principais tipos de rede neural

A partir dos fundamentos de estrutura e treinamento de redes neurais, o campo do *deep learning* se ramificou em uma série de arquiteturas especializadas, cada uma projetada para otimizar o processamento de tipos específicos de dados. Enquanto as Redes Neurais Convolucionais (CNNs) se tornaram a ferramenta padrão para dados com estrutura de grade, como imagens, o desafio de modelar dados sequenciais — como texto, séries temporais e, crucialmente, a música — exigiu o desenvolvimento de arquiteturas distintas.

Neste contexto, duas famílias de modelos emergiram como pilares para a geração musical: as Redes Neurais Recorrentes, particularmente sua variante mais robusta, a Long Short-Term Memory (LSTM), e a arquitetura Transformer. A escolha entre estas duas abordagens representa uma das principais bifurcações metodológicas

na literatura recente, refletindo diferentes filosofias sobre como capturar e processar as dependências temporais que definem a estrutura musical (BOULANGER-LEWANDOWSKI *et al.*, 2012; HUANG *et al.*, 2018a; VASWANI *et* al., 2017). Esta seção irá detalhar o funcionamento, as vantagens e as aplicações de cada uma dessas arquiteturas no domínio da composição algorítmica.

2.3.5.1 LSTM (Long Short-Term Memory)

As Redes Neurais de Memória de Longo Prazo (LSTM), uma forma avançada de Rede Neural Recorrente (RNN), foram por muito tempo a arquitetura de ponta para a maioria das tarefas envolvendo dados sequenciais. Propostas por Hochreiter e Schmidhuber (1997), elas foram especificamente projetadas para resolver a principal limitação das RNNs simples: a incapacidade de aprender dependências de longo prazo devido ao problema do desaparecimento de gradiente.

A inovação central da LSTM é sua unidade recorrente, a célula de memória (*cell state*), que atua como uma "esteira transportadora" de informação, permitindo que o contexto relevante flua através da sequência com pouca degradação. O fluxo de informação para dentro e para fora desta célula é inteligentemente regulado por três estruturas chamadas portões (*gates*).

Os seguintes portões são, em si, pequenas redes neurais (compostas por uma camada linear e uma função de ativação sigmoide) que aprendem a tomar decisões sobre a informação.

a) Portão de Esquecimento (*Forget Gate*): É o primeiro portão. Ele "olha" para a informação do estado anterior e a entrada atual, e sua saída (um valor entre 0 e 1) decide qual porcentagem da informação antiga deve ser esquecida ou mantida na célula de memória.

b) Portão de Entrada (*Input Gate*): Este portão decide quais novas informações da entrada atual são relevantes para serem armazenadas. Ele opera em duas partes: uma camada sigmoide decide quais valores atualizar, e uma camada Tanh cria um vetor de novos valores candidatos, que são então adicionados à célula de memória.

c) Portão de Saída (*Output Gate*): Finalmente, este portão determina qual parte da informação da célula de memória será usada para gerar a saída daquele passo de

tempo. A célula é passada por uma função Tanh e o resultado é filtrado pela saída da camada sigmoide do portão de saída.

Essa arquitetura de portões permite que a LSTM preserve o contexto por centenas de passos de tempo, tornando-a capaz de aprender as complexas dependências que caracterizam a música (OLAH, 2015; GOODFELLOW; BENGIO; COURVILLE, 2016).

A partir da arquitetura LSTM original, diversas variantes foram propostas para otimizar ou modificar seu comportamento. As GRUs (Gated Recurrent Units) são uma versão simplificada da LSTM que combina os portões de esquecimento e entrada em um único "portão de atualização". Elas possuem menos parâmetros e são computacionalmente mais eficientes, apresentando performance similar à LSTM em muitas tarefas (CHOLLET, 2021).

As LSTMs Bidirecionais (BiLSTMs) são uma variação utilizada em muitos problemas sequenciais, em que o contexto de eventos futuros é tão importante quanto o de eventos passados. Uma BiLSTM processa a sequência em duas direções simultaneamente (do início para o fim e do fim para o início) com duas LSTMs separadas, e concatena suas saídas. Isso permite que a previsão em um determinado ponto da sequência seja informada por todo o contexto, tanto anterior quanto posterior (SCHUSTER; PALIWAL, 1997).

A capacidade das LSTMs de modelar sequências temporais tornou-as a ferramenta dominante na geração musical por quase duas décadas. Elas aprendem a distribuição de probabilidade de uma sequência, prevendo o próximo evento musical com base em todos os eventos anteriores (GRAVES, 2013). Suas aplicações são vastas e demonstram sua flexibilidade:

a) Música Polifônica e Harmonia: O sistema RNN-RBM foi um marco ao combinar uma LSTM para modelar a sequência temporal de notas com uma Máquina de Boltzmann Restrita (RBM) para modelar a harmonia (as notas tocadas simultaneamente), tornando-se uma referência na área (BOULANGER-LEWANDOWSKI *et al.*, 2012). O DeepBach também utilizou LSTMs para gerar corais polifônicos no estilo de J.S. Bach, demonstrando a capacidade da arquitetura de aprender regras complexas de contraponto (HADJERES; PACHET; NIELSEN, 2017a).

b) Melodia e Performance Expressiva: O Performance RNN, do projeto Magenta do Google, utilizou uma arquitetura LSTM para gerar performances de piano com timing e dinâmicas expressivas, capturando nuances que vão além da simples escolha de notas (SIMON; OORE, 2017). Outros sistemas, como o Anticipation-RNN, exploraram o uso de LSTMs para a geração musical interativa, onde o modelo pode compor em torno de restrições impostas pelo usuário (HADJERES; NIELSEN, 2017).

Apesar de seu sucesso, LSTMs podem apresentar dificuldades em manter a coerência estrutural em escalas de tempo muito longas (vários minutos). Devido à sua natureza sequencial, a informação ainda pode se degradar com o tempo, e o processo de geração não é facilmente paralelizável. Em alguns casos, podem gerar sequências com padrões repetitivos, uma limitação que técnicas como o Aprendizado por Reforço buscaram mitigar (JAQUES *et al.*, 2016).

2.3.5.2 Transformers

A arquitetura Transformer, introduzida por Vaswani *et al.* (2017) no artigo seminal "*Attention Is All You Need*", representa uma mudança de paradigma no processamento de dados sequenciais. Diferente das RNNs e LSTMs, que processam a informação em ordem cronológica, passo a passo, o Transformer abandonou completamente a recorrência em favor de um mecanismo de autoatenção (*self-attention*). Essa inovação permitiu que o modelo processasse todos os elementos de uma sequência simultaneamente (em paralelo), resolvendo o gargalo computacional das arquiteturas recorrentes e, mais importante, possibilitando a captura de dependências entre elementos distantes na sequência de forma muito mais eficaz.

2.3.5.2.1 O Mecanismo Interno: Autoatenção, Q, K, V e Múltiplas Cabeças

O coração do Transformer é o mecanismo de autoatenção. Sua função é ponderar a importância de todos os outros elementos em uma sequência ao processar um único elemento. Para fazer isso, para cada elemento de entrada (por exemplo, a representação de uma nota musical), a rede cria os seguintes três vetores distintos:

a) Query (Q - Consulta): Representa o elemento atual, que está "buscando" informações relevantes.

b) Key (K - Chave): Representa uma "etiqueta" ou um "rótulo" de um elemento na sequência. A consulta (Q) é comparada com todas as chaves (K) para encontrar correspondências.

c) Value (V - Valor): Representa o conteúdo real do elemento.

O processo funciona de forma análoga a uma busca em um banco de dados: a consulta (Q) de um elemento calcula um "score de atenção" com a chave (K) de todos os outros elementos da sequência. Esses scores são normalizados (usando uma função Softmax) e usados como pesos para criar uma soma ponderada dos vetores de valor (V). O resultado é uma nova representação para o elemento atual que é enriquecida pelo contexto de todos os outros elementos da sequência, com mais peso dado àqueles que são mais "relevantes". A expressão (2) para essa operação é a conhecida *Scaled Dot-Product Attention*

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad ((2)$$

Para aumentar o poder de representação, o Transformer não faz isso apenas uma vez, mas em paralelo múltiplas vezes através da Atenção Multi-cabeças (*Multi-Head Attention*). Cada "cabeça" aprende a focar em diferentes tipos de relações na sequência (por exemplo, uma cabeça pode focar em relações rítmicas, outra em relações harmônicas), e suas saídas são combinadas para formar a representação final (VASWANI *et al.*, 2017).

2.3.5.2.2 Codificadores, Decodificadores e Posição

Como o modelo processa todos os elementos de uma vez, ele não tem uma noção inerente da ordem da sequência. Para resolver isso, a informação de posição é injetada artificialmente através de uma codificação posicional (*positional encoding*), que é um vetor somado à representação de cada elemento, informando ao modelo sua localização na sequência.

A arquitetura completa geralmente consiste em uma pilha de Codificadores (*Encoders*) e Decodificadores (*Decoders*). O codificador processa a sequência de entrada para construir uma representação rica em contexto. O decodificador, por sua vez, utiliza essa representação para gerar a sequência de saída, um elemento de cada

vez. Para tarefas de geração de música, que são autorregressivas, apenas a arquitetura do decodificador é frequentemente utilizada (GOODFELLOW; BENGIO; COURVILLE, 2016; CHOLLET, 2021).

A capacidade do Transformer de modelar relações de longo alcance o tornou extremamente poderoso para a geração musical, superando muitas das limitações das LSTMs em manter a coerência estrutural.

a) Coerência de Longo Prazo: O Music Transformer foi um dos primeiros modelos a demonstrar essa capacidade de forma impressionante. Ao utilizar um mecanismo de atenção relativa, ele conseguiu gerar peças de piano de vários minutos que mantinham uma estrutura coerente, com temas e motivos que se repetiam e se desenvolviam, algo extremamente desafiador para modelos recorrentes (HUANG *et al.*, 2019a).

b) Geração Condicionada por Texto (Text-to-Music): A flexibilidade do mecanismo de atenção tornou os Transformers a base para modelos de geração musical condicionados por texto. Sistemas como o MusicLM e o Noise2Music podem receber um *prompt* como "uma melodia de rock dos anos 80 com um solo de guitarra" e gerar um áudio que corresponde a essa descrição. Eles aprendem a "prestar atenção" às palavras no texto para guiar a geração do áudio (AGOSTINELLI *et al.*, 2023; HUANG *et al.*, 2023a).

c) Geração de Áudio Bruto: O modelo Jukebox da OpenAI utiliza uma variante do Transformer com atenção esparsa (para lidar com as sequências extremamente longas do áudio bruto) para gerar música com vocais e múltiplos instrumentos. Ele aprende a estrutura hierárquica da música, desde as amostras de áudio individuais até a estrutura global da canção (DHARIWAL *et al.*, 2020).

A principal desvantagem dos Transformers é seu custo computacional. O cálculo dos *scores* de atenção cresce quadraticamente com o comprimento da sequência, tornando o processamento de sequências muito longas (como minutos de áudio em alta resolução) extremamente demandante em termos de memória e poder de processamento. Diversas variantes (como o Transformer Esparso ou o Performer) foram propostas para mitigar esse problema.

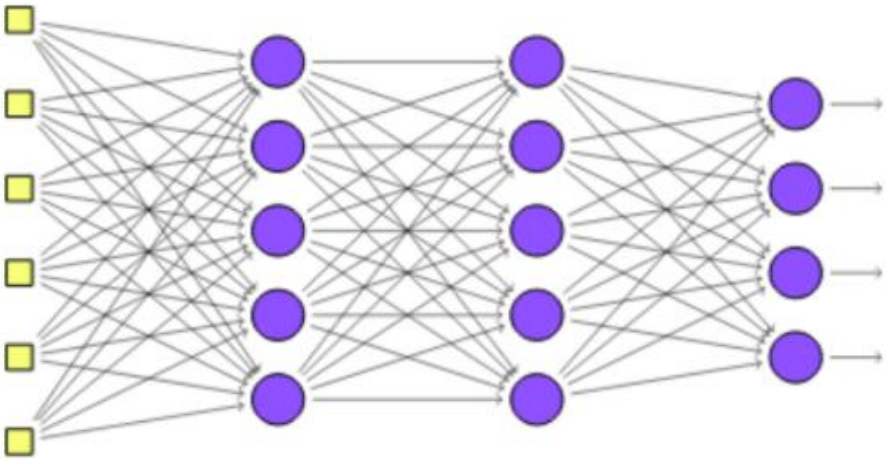
2.3.6 Panorama da Geração Musical com Redes Neurais: Aplicações e Desafios

A aplicação de redes neurais na geração musical é uma área de pesquisa que expandiu as fronteiras da inteligência artificial para além de tarefas analíticas, adentrando o domínio da criação artística. O impulso para essa expansão reside na capacidade das técnicas de deep learning de aprenderem automaticamente estilos musicais a partir de vastos corpora e, em seguida, gerar novas amostras a partir da distribuição aprendida. Essa abordagem, contudo, não visa substituir a criatividade humana, mas sim aumentá-la, funcionando como uma ferramenta de "inteligência aumentada" para músicos e compositores (BRIOT, 2019).

As redes neurais são hoje empregadas para gerar uma vasta gama de conteúdos musicais, que variam em complexidade e formato. Destacamos os seguintes empregos:

a) Melodias Monofônicas: A geração de uma única linha melódica foi um dos primeiros desafios abordados. Desde sistemas pioneiros que utilizavam redes *feedforward* simples, a área evoluiu para modelos recorrentes sofisticados. O *Anticipation*-RNN, por exemplo, permite a geração de melodias com restrições posicionais impostas pelo usuário, tornando o processo interativo (HADJERES; NIELSEN, 2017). A Figura 14 ilustra a estrutura de uma rede recorrente, fundamental para essa tarefa, que processa a informação de forma sequencial.

Figura 14 - Esquema de uma rede neural *feedforward*



Fonte: Souza e Avila (2018)

b) Polifonia: A geração de múltiplas vozes simultâneas (polifonia) é uma tarefa consideravelmente mais complexa, pois exige que o modelo compreenda não apenas a progressão temporal de cada voz (a dimensão horizontal), mas também as relações harmônicas entre as notas tocadas ao mesmo tempo (a dimensão vertical). O sistema RNN-RBM foi um marco nesse campo, combinando uma LSTM para a sequência temporal com uma Máquina de Boltzmann Restrita (RBM) para a harmonia, tornando-se uma arquitetura de referência (BOULANGER-LEWANDOWSKI *et al.*, 2012). Outras abordagens, como o Bi-Axial LSTM, modelam a música com LSTMs que operam tanto na dimensão do tempo quanto na da nota, mostrando-se eficazes para estruturas complexas (BRIOT; HADJERES; PACHET, 2020).

c) Acompanhamentos e Progressões de Acordes: Além de criar peças do zero, as redes neurais podem ser treinadas para gerar acompanhamentos para melodias existentes. O sistema MidiNet, por exemplo, utiliza uma Rede Generativa Adversarial (GAN) com camadas convolucionais para gerar melodias condicionadas a uma sequência de acordes pré-definida, ou vice-versa (YANG; CHOU; YANG, 2017).

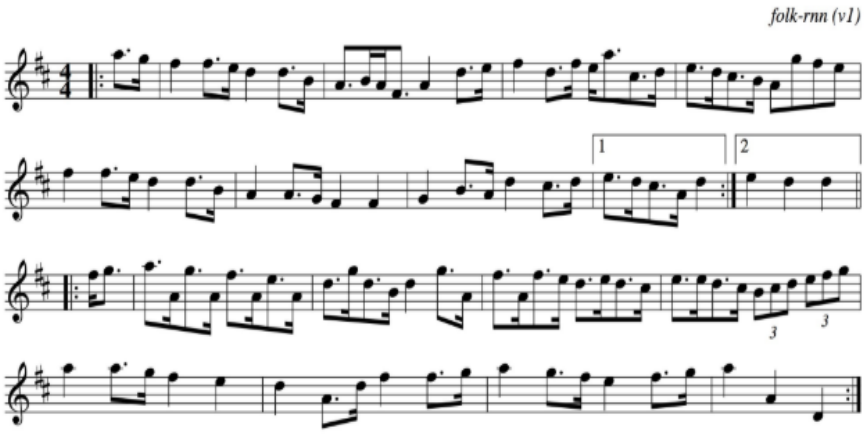
2.3.7 Abordagens de Representação e Técnicas Avançadas

Além da geração de conteúdo musical direto, como melodias e polifonias, a evolução das redes neurais permitiu o desenvolvimento de técnicas mais sofisticadas que abordam a música de maneiras mais abstratas e controláveis. Essas abordagens se aprofundam tanto na forma como a música é representada digitalmente — um contínuo entre o simbólico e o áudio bruto — quanto em tarefas de alto nível, como a manipulação e a especialização de estilos composicionais. A seguir, são detalhadas algumas dessas aplicações avançadas que definem a fronteira da pesquisa na área.

a) Geração Simbólica vs. Áudio Bruto: Conforme discutido, a geração pode ocorrer no domínio simbólico (MIDI, piano-roll), que é computacionalmente mais simples, ou diretamente no domínio do áudio bruto. O WaveNet da DeepMind foi pioneiro na geração de formas de onda de áudio, enquanto o Jukebox da OpenAI expandiu essa capacidade, gerando áudio com vocais a partir de entradas como gênero e artista (DHARIWAL *et al.*, 2020; OORD *et al.*, 2016).

b) Transferência de Estilo e Composição Especializada: A IA também é utilizada para transferir o estilo de um compositor para outra peça ou para gerar novas obras em um estilo específico. O MusicVAE, do projeto Magenta, utiliza um autoencoder variacional para aprender um "espaço latente" de estilos musicais, permitindo a interpolação entre eles e a manipulação de atributos como melodia e ritmo (ROBERTS *et al.*, 2018). De forma mais focada, sistemas como o DeepBach se especializaram na geração de corais no estilo de J.S. Bach, com alta fidelidade estilística, como pode ser visto no excerto da Figura 15 (HADJERES; PACHET; NIELSEN, 2017a).

Figura 15 - Exemplo de Música Gerada pelo DeepBach (excerto).



Fonte: (BRIOT, 2019)

Apesar dos avanços notáveis, a geração musical por redes neurais enfrenta desafios significativos que definem as fronteiras da pesquisa atual:

a) Estrutura e Coerência de Longo Prazo: Manter uma estrutura musical coerente por vários minutos continua a ser um grande desafio. Modelos recorrentes podem gerar sequências que, embora localmente plausíveis, carecem de um tema ou direção global. Arquiteturas hierárquicas, como no MusicVAE, e baseadas em atenção, como no Music Transformer, são as principais abordagens para resolver essa limitação.

b) Originalidade vs. Plágio: Modelos de deep learning são excelentes em imitar os padrões presentes nos dados de treinamento. No entanto, isso cria um risco de plágio inadvertido, onde o modelo pode reproduzir trechos quase idênticos do seu dataset.

Além disso, há um debate filosófico e técnico sobre como incentivar a verdadeira originalidade e criatividade, em vez de apenas uma recombinação estilística.

c) Interatividade e Controle: Para que a IA seja uma ferramenta verdadeiramente útil para músicos, ela precisa ser interativa e controlável. A capacidade de guiar o modelo, impor restrições, regenerar trechos específicos e colaborar com a IA em tempo real é uma área de pesquisa ativa e crucial para a adoção dessas tecnologias por artistas.

d) Qualidade Artística e Acessibilidade: Finalmente, a barreira entre resultados tecnicamente impressionantes e obras artisticamente comoventes ainda existe. Muitos dos sistemas exigem um conhecimento técnico aprofundado para serem utilizados, e a qualidade dos resultados ainda pode ser percebida como "simples" ou "repetitiva" por ouvintes treinados. A busca por modelos que capturem a expressividade e a profundidade da composição humana continua a ser o objetivo final do campo.

2.4 PYTHON APLICADO A REDES NEURAIS

Considera-se que é importante falar sobre a linguagem de programação Python, pois ela consolidou-se como a língua franca no campo da Inteligência Artificial (IA), especialmente em *machine learning* e *deep learning*. Este domínio não se deve a uma performance de execução superior à de linguagens compiladas como C++, mas sim a uma combinação de fatores que criaram um ecossistema ideal para a pesquisa e o desenvolvimento. A filosofia de *design* do Python, que enfatiza a clareza e a legibilidade da sintaxe, facilita a prototipagem rápida de ideias complexas. Adicionalmente, sua vasta gama de bibliotecas especializadas e otimizadas, mantidas por uma comunidade global ativa de desenvolvedores e pesquisadores, oferece ferramentas robustas para praticamente qualquer desafio (GOODFELLOW; BENGIO; COURVILLE, 2016).

Python resolve eficientemente o chamado "problema das duas linguagens": a necessidade de prototipar em uma linguagem de alto nível e fácil de usar, mas capaz de reimplementar em uma linguagem de baixo nível e de alta performance para produção. Através de suas robustas APIs de integração, Python atua como uma "linguagem-cola" de alta produtividade, permitindo que pesquisadores orquestrem operações

computacionalmente intensivas, que são executadas em *backends* eficientes escritos em C++, Fortran ou CUDA, sem sacrificar a simplicidade e a flexibilidade do ambiente de desenvolvimento (HARRIS *et al.*, 2020).

No contexto das redes neurais, o Python oferece um ambiente poderoso para todo o *pipeline* de desenvolvimento. A clareza do código é particularmente benéfica na área de *deep learning*, onde as arquiteturas podem ser intrincadas, facilitando a compreensão, a depuração e a colaboração em projetos de pesquisa.

2.4.1 Principais Frameworks de *Deep Learning*

Os *frameworks* de *deep learning* são o coração do desenvolvimento de IA em Python. Eles abstraem as complexidades da computação numérica, como a diferenciação automática para o *backpropagation* e a execução paralela em GPUs, permitindo que os desenvolvedores se concentrem na arquitetura do modelo. Para este trabalho, dois *frameworks* são de particular relevância: Tensorflow e Pytorch.

2.4.1.1 TensorFlow e Keras

Desenvolvido pela equipe do *Google Brain*, o TensorFlow é um dos *frameworks* de código aberto mais utilizados para *deep learning* em larga escala (ABADI *et al.*, 2016). Originalmente baseado em um paradigma de grafos computacionais estáticos (*define-and-run*), onde toda a computação era declarada antecipadamente e depois executada, ele oferecia grandes vantagens para otimização e implantação em ambientes distribuídos e de produção. Com o lançamento do TensorFlow 2, o framework adotou a execução ávida (*eager execution*) como padrão, tornando-se mais flexível e intuitivo. O ecossistema TensorFlow é vasto, incluindo ferramentas como o TensorBoard para a visualização do treinamento e da arquitetura do modelo, e o TensorFlow Lite para a implantação em dispositivos móveis e embarcados.

Integrado ao TensorFlow como sua API oficial de alto nível, o Keras foi projetado por François Chollet com foco na experiência do desenvolvedor, seguindo princípios de simplicidade, modularidade e facilidade de extensão (CHOLLET, 2021). Ele abstrai grande parte da complexidade do TensorFlow, permitindo a construção de modelos sofisticados com uma sintaxe clara e intuitiva. No presente trabalho, a implementação do

modelo LSTM foi realizada utilizando a API Keras com o TensorFlow como *backend*, aproveitando sua simplicidade para a construção rápida de uma arquitetura sequencial.

2.4.1.2 PyTorch

Desenvolvido pelo laboratório de pesquisa em IA do Facebook (Meta), o PyTorch emergiu como o principal concorrente do TensorFlow, tornando-se o favorito na comunidade de pesquisa acadêmica (PASZKE *et al.*, 2019). Sua principal característica distintiva é o uso nativo de grafos computacionais dinâmicos (*define-by-run*). Isso significa que a estrutura da computação é definida em tempo de execução, permitindo o uso de estruturas de controle padrão do Python (como laços for e condicionais if) diretamente no design do modelo. Essa abordagem "pythônica" torna o processo de depuração muito mais simples e direto, pois os valores podem ser inspecionados a qualquer momento. Sua flexibilidade é ideal para a pesquisa e o desenvolvimento de arquiteturas complexas e não convencionais.

No presente trabalho, a implementação do modelo Transformer foi realizada utilizando PyTorch, cuja natureza dinâmica facilitou a manipulação das complexas operações de atenção e das representações de dados musicais.

2.4.2 Bibliotecas de Suporte Essenciais

Os frameworks de *deep learning* são apoiados por um robusto ecossistema de bibliotecas que lidam com tarefas fundamentais de computação numérica, manipulação de dados e, crucialmente, com as particularidades dos dados musicais.

a) NumPy (Numerical Python): É a biblioteca pilar da computação científica em Python. Ela fornece o objeto fundamental ndarray, uma estrutura de dados para arrays multidimensionais que é significativamente mais eficiente em termos de memória e velocidade do que as listas padrão do Python. A performance do NumPy deriva de suas operações serem implementadas em C e de sua capacidade de realizar operações vetorizadas, que aplicam uma única instrução a um conjunto de dados inteiro (HARRIS *et al.*, 2020). Todos os frameworks de *deep learning* são projetados para interagir de forma transparente com arrays NumPy.

b) Bibliotecas Musicais (music21 e pretty-midi): Para a geração musical simbólica, o pré-processamento dos dados MIDI é uma etapa crítica. A principal diferença entre as

seguintes duas bibliotecas utilizadas neste trabalho reside em suas respectivas filosofias:

i. music21: Desenvolvida no MIT, é uma caixa de ferramentas "pesada" e orientada à musicologia computacional. Ela permite a análise de estruturas musicais complexas (tonalidade, harmonia) e a manipulação de objetos musicais de alto nível, como Partitura, Nota e Acorde. Foi a ferramenta utilizada no projeto LSTM para analisar os arquivos MIDI e extrair as sequências de notas e acordes como símbolos textuais (CUTHBERT; ARIZA, 2010).

ii. pretty-midi: É uma biblioteca mais leve e direta, focada na extração e manipulação eficiente de eventos MIDI. Sua API é ideal para converter rapidamente um arquivo MIDI na representação parametrizada ([pitch, duration, delta-time]). Foi a ferramenta utilizada no projeto Transformer por sua eficiência e simplicidade para esta tarefa específica.

c) Anaconda: A distribuição Anaconda é uma plataforma de gerenciamento de ambientes e pacotes que se tornou padrão na ciência de dados. Sua principal vantagem é o gerenciador conda, que é agnóstico à linguagem e capaz de gerenciar dependências complexas que vão além dos pacotes Python, incluindo bibliotecas C/C++ e drivers CUDA. Ao criar ambientes virtuais isolados, o Anaconda garante que cada projeto tenha seu próprio conjunto de dependências, evitando conflitos e garantindo a reprodutibilidade da pesquisa (SILVA, 2023).

3 METODOLOGIA

Este capítulo detalha o conjunto de procedimentos e a abordagem metodológica empregada para investigar a geração de música através de redes neurais. O objetivo é fornecer uma descrição transparente e sistemática do experimento, permitindo a compreensão e a potencial reprodutibilidade dos resultados. A estrutura aborda a natureza da pesquisa, os dados utilizados, as arquiteturas dos modelos, as configurações de treinamento e os critérios de avaliação empregados na comparação entre os modelos LSTM e Transformer.

3.1 DETERMINAÇÃO DA ESTRATÉGIA DE PESQUISA

A presente pesquisa caracteriza-se como um estudo de natureza aplicada, com uma abordagem experimental e comparativa. Conforme Gil (2008). A pesquisa aplicada visa gerar conhecimentos para aplicação prática, dirigidos à solução de problemas específicos. Neste caso, o problema é a geração de música coerente e estilisticamente plausível. A abordagem é experimental, pois envolve a implementação, o treinamento e a observação do comportamento de modelos de *deep learning* em um ambiente controlado.

Este trabalho se configura como um estudo de caso focado na comparação de duas arquiteturas de redes neurais: a LSTM (Long Short-Term Memory), baseada em recorrência, e o Transformer, baseado em atenção. Para esta análise, são utilizados dois conjuntos de dados distintos, o corpus Chopin (estilisticamente focado) e o MAESTRO (amplo e variado). É fundamental ressaltar que as análises e conclusões apresentadas são, portanto, específicas e limitadas a estes cenários experimentais definidos.

Adicionalmente, o estudo emprega uma abordagem quali-quantitativa. A análise quantitativa se manifestará na avaliação de métricas objetivas de performance, como a eficiência computacional (tempo de treinamento, uso de recursos). A análise qualitativa, por sua vez, é indispensável para avaliar a dimensão artística e subjetiva do resultado. Esta será conduzida através da avaliação crítica e auditiva das peças musicais geradas, com base em critérios de musicalidade, coerência e complexidade.

3.2 OBJETIVOS DA PESQUISA (REITERAÇÃO E ESPECIFICAÇÃO)

Embora os objetivos gerais deste trabalho tenham sido apresentados na introdução, esta seção os reitera e especifica sob uma perspectiva metodológica. O propósito é delinear claramente os alvos do experimento, conectando a pergunta de pesquisa com os procedimentos que serão executados. Os objetivos específicos desta fase experimental são:

a) Analisar e comparar o desempenho computacional de cada arquitetura, avaliando métricas como tempo de treinamento e o uso de recursos de hardware (RAM de sistema e GPU).

b) Avaliar a qualidade musical das composições geradas através de um painel de avaliação qualitativa, utilizando critérios de coerência melódica, riqueza harmônica e estrutura rítmica.

c) Investigar o impacto de diferentes conjuntos de dados — um estilisticamente focado (Chopin) e outro amplo e variado (MAESTRO) — no resultado final de cada modelo.

d) Implementar e analisar o efeito de uma rodada de ajustes nos modelos, avaliando se o aumento da complexidade da rede e do tempo de treinamento resulta em uma melhora significativa na qualidade musical.

3.3 COLETA E REPRESENTAÇÃO DE DADOS MUSICAIS

A qualidade, a natureza e a formatação dos dados de treinamento são fatores determinantes para o sucesso de qualquer modelo de aprendizado de máquina. Esta seção descreve os conjuntos de dados (corpora) selecionados e, crucialmente, os dois processos distintos de pré-processamento e representação musical empregados. Cada dataset foi processado de duas maneiras diferentes para gerar quatro conjuntos de dados de treinamento, permitindo que cada arquitetura (LSTM e Transformer) fosse treinada e avaliada em ambos os cenários musicais.

3.3.1 SELEÇÃO E PREPARAÇÃO DO *DATASET* MUSICAL

Para realizar uma comparação abrangente, são selecionados os seguintes dois conjuntos de dados com características distintas:

a) *Dataset* 1 (Estilisticamente Focado): Obras de Frédéric Chopin. Este *dataset* é composto por um conjunto de obras para piano de Frédéric Chopin em formato MIDI. A escolha de um *dataset* de compositor único foi deliberada, visando testar a capacidade dos modelos de aprender e gerar música dentro de um idioma estilístico altamente consistente e reconhecível. A música de Chopin, conhecida por sua riqueza melódica e complexidade harmônica, oferece um material denso para o aprendizado de padrões.

b) *Dataset* 2 (Amplo e Variado): MAESTRO *Dataset*. Este *dataset* é o MAESTRO (MIDI and Audio Edited for Synchronous Tracks and Organization) *dataset* v3.0.0 (HAWTHORNE *et al.*, 2019). É um conjunto de dados em larga escala, contendo mais de 170 horas de performances de piano virtuoso. Sua principal vantagem é a diversidade de compositores e a alta qualidade dos dados, que capturam nuances expressivas de tempo e dinâmica. Este *dataset* foi escolhido para testar a capacidade dos modelos de aprender a partir de um conjunto de dados mais vasto e estilisticamente variado

Para ambos os conjuntos de dados, é aplicada a divisão padrão em conjuntos de treinamento, validação e teste, conforme discutido no referencial teórico. Esta separação é crucial para garantir que os modelos sejam treinados de forma robusta, ajustados com base em dados não vistos e, finalmente, avaliados de maneira imparcial em sua capacidade de generalização.

3.3.2 Processamento e Representação dos Dados

Cada um dos dois *corpora* (Chopin e MAESTRO) é processado através de dois *pipelines* de representação distintos, um para cada arquitetura de modelo.

a) *Pipeline* de Representação 1 (Simbólica, para o LSTM) Para preparar os dados para o modelo LSTM, é adotada uma representação puramente simbólica, tratando a música como uma sequência de "palavras". Utilizando a biblioteca music21 (CUTHBERT; ARIZA, 2010), cada arquivo MIDI de ambos os *corpora* foi percorrido e convertido em uma lista de eventos tokenizados como strings:

    i. Notas individuais: Representadas pelo seu nome e oitava (ex: 'C#4').

ii. Acordes: Representados por um conjunto de suas classes de altura em ordem normal, separadas por pontos (ex: '0.4.7' para um acorde de Dó Maior). Ao final deste processo, obtivemos dois *datasets* prontos para o LSTM: Chopin-Simbólico e MAESTRO-Simbólico. Esta abordagem testa a capacidade do modelo de aprender relações sequenciais a partir de uma representação que abstrai as informações de ritmo e duração.

b) *Pipeline* de Representação 2 (Parametrizada, para o Transformer). Para o modelo Transformer, é utilizada uma representação parametrizada, que preserva mais informações estruturais da música. Com o auxílio da biblioteca pretty-midi, cada evento de nota em ambos os corpora foi convertido em uma tupla de três atributos numéricos:

i. Pitch (Altura): O número da nota MIDI (0-127).

ii. Duration (Duração): A duração da nota em segundos.

iii. Delta-time (Tempo Relativo): O tempo, em segundos, desde o início do evento anterior. Os valores contínuos de duração e tempo são então quantizados em um vocabulário finito de valores discretos. Ao final deste processo, obtemos mais dois *datasets* para o Transformer: Chopin-Parametrizado e MAESTRO-Parametrizado. Esta abordagem testa a capacidade do modelo de aprender relações complexas entre altura, ritmo e tempo.

3.4 ARQUITETURAS DAS REDES NEURAIS

Com os conjuntos de dados devidamente preparados, esta seção descreve os *blueprints* arquitetônicos dos dois modelos de redes neurais implementados. É fundamental ressaltar que a estrutura de cada modelo (LSTM e Transformer) permanece fixa em todos os experimentos. Essa constância garante que as diferenças observadas nos resultados se devam às capacidades inerentes de cada arquitetura e à sua interação com os diferentes conjuntos de dados (Chopin vs. MAESTRO), e não às variações na configuração dos modelos.

3.4.1 Arquitetura do Modelo LSTM

Para a abordagem baseada em recorrência, foi implementado um modelo LSTM utilizando a API de alto nível Keras com o TensorFlow como *backend*. A estrutura do código foi adaptada e teve como base a implementação proposta por Kapoor (2020) em seu notebook público, sendo ajustada para os propósitos deste estudo. A arquitetura escolhida é uma rede sequencial empilhada (*stacked* LSTM), uma técnica padrão para aumentar a capacidade do modelo de aprender hierarquias de padrões temporais. A estrutura detalhada do modelo é a seguinte:

a) Primeira Camada LSTM: Uma camada LSTM com 512 unidades. O argumento return_sequences=True é utilizado para que a saída desta camada seja a sequência completa de estados ocultos, e não apenas o estado final. Isso é essencial para que a camada LSTM subsequente possa processar a sequência temporal completa. Teoricamente, esta primeira camada aprende a reconhecer padrões locais e de curto prazo na música.

b) Primeira Camada de Regularização: Uma camada *Dropout* com uma taxa de 0.1 é inserida para mitigar o overfitting, forçando a rede a aprender representações mais robustas ao desativar aleatoriamente 10% das unidades durante o treinamento.

c) Segunda Camada LSTM: Uma segunda camada LSTM com 256 unidades. Esta camada recebe a sequência de saídas da anterior e aprende a combinar os padrões de curto prazo em estruturas de maior complexidade. Como esta é a última camada recorrente na pilha, ela retorna apenas a saída final da sequência.

d) Camadas Densas e de Saída: A saída da segunda camada LSTM é processada por uma camada totalmente conectada (Dense) com 256 neurônios, seguida por outra camada de *Dropout* de 0.1. Finalmente, uma camada Dense de saída com ativação softmax produz a distribuição de probabilidade sobre todo o vocabulário musical. O número de neurônios nesta camada final é dinâmico, correspondendo ao tamanho do vocabulário do *dataset* específico em uso (seja o Chopin-Simbólico ou o MAESTRO-Simbólico).

Essa arquitetura empilhada, baseada no trabalho seminal de Hochreiter e Schmidhuber (1997), foi escolhida por seu balanço entre capacidade de representação

e complexidade computacional, sendo um modelo clássico para a geração de sequências simbólicas.

3.4.2 Arquitetura do Modelo Transformer

Para a abordagem baseada em atenção, foi implementado um modelo Transformer em PyTorch. A estrutura do código foi baseada na implementação de Badoa (2020), que explora a geração musical com Transformers, sendo adaptada para este experimento. A arquitetura é do tipo decodificador-apenas (*decoder-only*), similar à utilizada em modelos de linguagem como o GPT, pois a tarefa de geração musical é puramente autorregressiva. A estrutura detalhada do modelo é a seguinte:

a) Camada de *Embedding*: Uma camada *Embedding* que converte os *tokens* inteiros de entrada (representando *pitch*, *duration* e *delta-time*) em vetores densos e contínuos de alta dimensão (d_model=1024). A concatenação dos *embeddings* dos três atributos resulta em uma representação rica para cada evento musical.

b) Codificação Posicional (*Positional Encoding*): Como o mecanismo de atenção, por si só, não processa a informação em ordem, vetores posicionais sinusoidais são somados aos *embeddings* para injetar a informação sobre a posição de cada nota na sequência.

c) Pilha de Camadas Transformer: O coração do modelo é uma pilha de 6 camadas de decodificador de Transformer (nlayers=6), conforme proposto por Vaswani *et al.* (2017). Cada camada é composta pelos seguintes dois sub-módulos principais:

    i. Atenção Multi-cabeças Mascarada (*Masked Multi-Head Attention*): Um mecanismo de autoatenção com 16 cabeças de atenção independentes (nhead=16). A utilização de múltiplas cabeças permite que o modelo aprenda simultaneamente diferentes tipos de relações musicais — como rítmicas, melódicas e harmônicas — em diferentes subespaços de representação. Uma máscara causal é aplicada para garantir que a predição de uma nota só possa depender das notas que a precederam, evitando que o modelo "veja o futuro".

    ii. Rede *Feed-Forward*: Uma pequena rede neural totalmente conectada, com duas camadas, que é aplicada a cada posição da sequência de forma independente para processamento adicional.

b) Camada de Saída Linear: Uma camada linear (Linear) final que projeta a saída da pilha de Transformers de volta para a dimensão do vocabulário para cada um dos três atributos musicais, gerando os *logits* que serão convertidos em probabilidades para a seleção da próxima nota.

Esta arquitetura foi escolhida por sua capacidade, demonstrada na literatura, de capturar dependências de longo alcance de forma superior às LSTMs, um fator crucial para gerar música com estrutura e coerência em escalas de tempo maiores.

3.5 CONFIGURAÇÃO EXPERIMENTAL E TREINAMENTO

Com as arquiteturas dos modelos e os conjuntos de dados definidos, esta seção detalha os procedimentos e hiperparâmetros para o treinamento de cada rede. O objetivo é descrever um processo experimental claro e reprodutível. A metodologia comparativa é estruturada em quatro cenários de treinamento distintos:

a) Cenário 1: Modelo LSTM treinado com o *dataset* Chopin-Simbólico.

b) Cenário 2: Modelo LSTM treinado com o *dataset* MAESTRO-Simbólico.

c) Cenário 3: Modelo Transformer treinado com o *dataset* Chopin-Parametrizado.

d) Cenário 4: Modelo Transformer treinado com o *dataset* MAESTRO-Parametrizado.

Para cada cenário, o respectivo modelo foi treinado do zero (from scratch), com os pesos inicializados aleatoriamente, seguindo as configurações detalhadas abaixo.

3.5.1 TREINAMENTO DO MODELO LSTM

O treinamento do modelo LSTM, implementado em Keras/TensorFlow, é configurado com os seguintes hiperparâmetros:

a) Função de Perda (Loss Function): É utilizada a *categorical_crossentropy* (Entropia Cruzada Categórica). Esta é uma escolha padrão para problemas de classificação multiclasse, como a predição da próxima nota a partir de um vocabulário fixo, pois ela é eficaz em medir a divergência entre a distribuição de probabilidade prevista pelo modelo e a distribuição real (CHOLLET, 2021).

b) Otimizador: É empregado o otimizador Adamax, uma variante do otimizador Adam baseada na norma do infinito. Ele é conhecido por sua robustez e bom desempenho em uma variedade de modelos, incluindo arquiteturas recorrentes (KINGMA; BA, 2014).

c) Taxa de Aprendizado (*Learning Rate*): A taxa de aprendizado para o otimizador Adamax é fixada em 0,01.

d) Tamanho do Lote (*Batch Size*): O treinamento é realizado com um tamanho de lote de 256. Este valor representa um balanço entre a estabilidade da estimativa do gradiente e as limitações de memória computacional.

e) Número de Épocas (*Epochs*): Cada modelo LSTM é treinado por 50 épocas, ou seja, 50 passagens completas sobre o respectivo conjunto de dados de treinamento.

3.5.2 Treinamento do Modelo Transformer

O treinamento do modelo Transformer, implementado em PyTorch, segue uma configuração distinta, alinhada às práticas comuns para esta arquitetura:

a) Função de Perda (*Loss Function*): Utilizou-se uma implementação customizada de Entropia Cruzada Categórica, adaptada para lidar com a predição simultânea dos três atributos da representação parametrizada (*pitch*, *duration* e *delta-time*).

b) Otimizador: Foi utilizado o otimizador Adam (*Adaptive Moment Estimation*), que é o padrão de fato para o treinamento de modelos Transformer devido à sua eficiência em lidar com um grande número de parâmetros e sua capacidade de adaptar a taxa de aprendizado para cada um deles (VASWANI *et al.*, 2017).

c) Taxa de Aprendizado (*Learning Rate*): A taxa de aprendizado para o otimizador Adam foi fixada em 0,0001, um valor conservador, mas comum para o ajuste fino de arquiteturas complexas.

d) Tamanho do Lote (*Batch Size*): O treinamento foi realizado com um tamanho de lote de 32. Um lote menor é necessário devido ao alto consumo de memória da arquitetura Transformer, especialmente ao processar sequências longas.

e) Número de Épocas (*Epochs*): Cada modelo Transformer é treinado por 2 épocas. Embora o número de épocas seja menor, a complexidade e o número de parâmetros

do Transformer permitem que ele aprenda representações ricas em menos passagens sobre os dados em comparação com a LSTM.

Adicionalmente, o *pipeline* do Transformer inclui a possibilidade de treinamento de um Discriminador em um esquema de Rede Generativa Adversarial (GAN), como uma etapa de refinamento posterior ao treinamento inicial supervisionado.

3.6 ESTRATÉGIAS DE GERAÇÃO DE MÚSICA

Uma vez que os modelos tenham sido treinados, a etapa seguinte consiste em utilizá-los para a tarefa de composição, ou seja, a geração de novas sequências musicais. Esta fase de inferência, embora computacionalmente menos intensiva que o treinamento, envolve decisões estratégicas sobre como extrair a música do modelo treinado. Para ambos os modelos, LSTM e Transformer, foi empregada uma abordagem autorregressiva de geração.

3.6.1 GERAÇÃO AUTORREGRESSIVA

A geração autorregressiva é um processo iterativo no qual o modelo constrói a sequência musical um evento de cada vez. O processo se inicia com uma sequência semente (*seed sequence*), que é um pequeno trecho de música (real ou gerado aleatoriamente) fornecido como a entrada inicial para o modelo. A partir desta semente, o ciclo de geração ocorre da seguinte forma:

a) Predição: O modelo realiza uma propagação direta (*forward pass*) com a sequência atual para gerar uma distribuição de probabilidade sobre todo o vocabulário, indicando a probabilidade de cada evento musical ser o próximo.

b) Amostragem (*Sampling*): Em vez de sempre escolher o evento mais provável (uma abordagem determinística chamada de *greedy search*), um evento é amostrado a partir dessa distribuição. Este passo é crucial para introduzir variabilidade e criatividade nas composições.

c) Anexação: O evento amostrado é então anexado ao final da sequência.

d) Repetição: A nova sequência, agora um passo mais longa, torna-se a entrada para a próxima iteração.

Este ciclo de "prever, amostrar e anexar" se repete até que a composição atinja um comprimento pré-definido, permitindo que o modelo gere peças musicais de durações arbitrárias (BRIOT; HADJERES; PACHET, 2020).

3.6.2 Técnicas de Amostragem e Controle Criativo

A qualidade e a natureza da música gerada são fortemente influenciadas pela forma como a amostragem é realizada a partir da distribuição de probabilidade do modelo. Para este trabalho, em vez de explorar a variabilidade através de diferentes temperaturas, optou-se por uma abordagem de amostragem determinística, a fim de avaliar o conhecimento central aprendido por cada modelo, sem a influência da aleatoriedade.

A técnica empregada foi a busca gulosa (*greedy search*). Nesta abordagem, a cada passo da geração autorregressiva, o modelo sempre seleciona o evento musical (nota ou acorde) que possui a maior probabilidade na distribuição de saída da camada *softmax*. Este método foi implementado utilizando a função np.argmax no *pipeline* do LSTM e torch.argmax no *pipeline* do Transformer.

Esta escolha metodológica é funcionalmente equivalente a definir o hiperparâmetro de temperatura em um valor baixíssimo, próximo de zero. Ao fazer isso, a aleatoriedade é completamente eliminada do processo de geração. A principal vantagem desta abordagem é a reprodutibilidade e a interpretabilidade: dada uma mesma sequência semente, o modelo sempre produzirá a mesma composição. Isso permite uma análise direta do que a rede neural considera a continuação mais "correta" ou "provável" em cada ponto, oferecendo uma visão clara dos padrões que foram mais fortemente aprendidos durante o treinamento (GOODFELLOW; BENGIO; COURVILLE, 2016).

3.6.3 Refinamento Experimental com Variação de Épocas

Reconhecendo que o ponto ótimo de treinamento pode variar significativamente entre arquiteturas e conjuntos de dados, a metodologia inclui uma fase de refinamento experimental. Após a análise inicial dos resultados obtidos com os parâmetros definidos na Seção 3.5, são conduzidos testes adicionais para avaliar o impacto do número de épocas na qualidade final da música gerada.

Para isso, os modelos são retreinados por um número alternativo de épocas. O objetivo é identificar se um treinamento mais curto ou mais longo resulta em modelos com melhor capacidade de generalização, evitando os problemas de subajuste (*underfitting*) ou sobreajuste (*overfitting*). As composições geradas a partir desses modelos retreinados serão então submetidas aos mesmos critérios de avaliação, permitindo uma análise mais aprofundada sobre a relação entre a duração do treinamento e a qualidade artística do produto final.

3.7 CRITÉRIOS DE AVALIAÇÃO COMPARATIVA

Para avaliar e comparar de forma abrangente o desempenho dos quatro cenários experimentais (LSTM em Chopin, LSTM em MAESTRO, Transformer em Chopin, Transformer em MAESTRO), foi adotada uma metodologia de avaliação mista, combinando métricas quantitativas, focadas na eficiência computacional, e métricas qualitativas, focadas na análise musical do conteúdo gerado.

3.7.1 MÉTRICAS QUANTITATIVAS

A avaliação quantitativa visa medir os aspectos objetivos e computacionais de cada modelo. Os seguintes critérios são analisados e comparados:

a) Eficiência de Treinamento: É registrado e comparado o tempo total de treinamento para cada um dos quatro cenários. A análise considerará o tempo médio por época e o custo computacional geral para atingir um estado treinado, fornecendo uma medida direta da eficiência de cada arquitetura em cada tipo de dado.

b) Recursos Computacionais: São documentados os recursos de *hardware* necessários para o treinamento de cada modelo.

c) Complexidade do Modelo: É comparado o número de parâmetros treináveis de cada arquitetura. Este valor serve como um *proxy* para a complexidade do modelo e sua capacidade teórica de representação.

3.7.2 MÉTRICAS QUALITATIVAS (ANÁLISE MUSICAL)

Dada a natureza artística do objeto de estudo, considera-se que uma avaliação puramente quantitativa é insuficiente. Portanto, uma análise qualitativa baseada na escuta crítica será o método adotado para julgar a qualidade musical das composições

geradas. Este processo será conduzido por um painel de avaliadores e guiado por uma matriz de pontuação estruturada.

Para conduzir a avaliação, foi formado um painel heterogêneo de cinco avaliadores, selecionados com base em suas diferentes experiências e relações com a música, a fim de capturar uma gama diversificada de percepções. O perfil dos avaliadores é o seguinte:

a) Avaliador A: Vocalista com cinco anos de experiência em uma banda de rock e cinco anos de estudo de canto em escola de música;
b) Avaliador B: Baterista com cinco anos de experiência em uma banda de rock;
c) Avaliador C: Produtor musical com seis anos de experiência profissional;
d) Avaliador D: Músico amador com mais de vinte anos de prática instrumental; e
e) Avaliador E: Estudante de artes com formação pelo Conservatório de Música Villa-Lobos.

A cada avaliador serão apresentados os áudios gerados de forma cega (sem identificação da origem) para que atribuam suas notas. Os avaliadores foram instruídos a atribuir uma nota de 1 (muito fraco) a 5 (excelente) para cada peça, com base nos seguintes três critérios musicais fundamentais:

a) Critério 1: Coerência Melódica: Avalia a qualidade da linha melódica principal, observando aspectos como fluidez, desenvolvimento temático e a presença de frases musicais reconhecíveis;
b) Critério 2: Riqueza Harmônica: Avalia a complexidade e a adequação das progressões de acordes e da harmonia subjacente, analisando se há um centro tonal claro e se as escolhas são estilisticamente apropriadas; e
c) Critério 3: Estrutura Rítmica: Avalia a consistência e a complexidade do ritmo, observando a presença de um pulso claro, a variedade de padrões e a coerência geral do fluxo temporal.

A "Nota Final" de cada composição será calculada como o produto das três notas atribuídas, de forma análoga ao cálculo de priorização da matriz GUT. Este método de

pontuação ponderada permite uma comparação final que leva em conta a performance do modelo em todas as três dimensões musicais simultaneamente.

4 APLICAÇÃO DA METODOLOGIA

4.1 APRESENTAÇÃO DOS RESULTADOS EXPERIMENTAIS

Este capítulo apresenta e analisa os resultados obtidos a partir dos procedimentos metodológicos descritos no capítulo anterior. A investigação sobre a eficácia das arquiteturas LSTM e Transformer na geração de música simbólica foi conduzida através de quatro cenários experimentais distintos, cruzando cada modelo com os dois conjuntos de dados selecionados (Chopin e MAESTRO). Os códigos-fonte completos para cada um desses quatro experimentos da rodada de linha de base encontram-se detalhados nos Adendos A a D.

A análise está estruturada em duas etapas. Primeiramente, é realizada uma análise quantitativa, focada no desempenho computacional, nos recursos de hardware demandados e na eficiência de treinamento de cada cenário. Esta é seguida por uma análise qualitativa, que busca avaliar a coerência e a complexidade musical das composições geradas, servindo como base para a comparação da capacidade artística de cada abordagem.

São também apresentadas as dificuldades encontradas na implementação computacional deste trabalho e as soluções adotadas, pois considera-se que, apesar de não apresentarem contribuições à metodologia, são itens que devem ser conhecidos por impactarem no desenvolvimento de trabalhos semelhantes.

4.2 ANÁLISE QUALITATIVA E DE DESEMPENHO COMPUTACIONAL

A primeira dimensão da análise comparativa foca nos aspectos mensuráveis e objetivos do processo de treinamento. A demanda por recursos computacionais e a eficiência temporal são fatores pragmáticos essenciais na avaliação da viabilidade de modelos de *deep learning*.

4.2.1 TEMPOS DE EXECUÇÃO E EFICIÊNCIA DOS MODELOS

A análise dos tempos totais de execução, desde o carregamento dos dados até a finalização do treinamento, revela diferenças drásticas no custo computacional entre as arquiteturas e os *datasets*.

O impacto do tamanho do *dataset* foi notável em ambos os modelos. O modelo LSTM, por exemplo, concluiu seu treinamento com o dataset de Chopin em 10 minutos e 50 segundos. Ao ser submetido ao *dataset* MAESTRO, substancialmente maior, seu tempo de execução aumentou drasticamente para 4 hora, 41 minutos e 53 segundos, um crescimento de mais de 2400%.

Por outro lado, a comparação entre as arquiteturas evidencia a notável eficiência temporal do modelo Transformer. Com o *dataset* de Chopin, o treinamento foi concluído em apenas 2 minutos e 38 segundos. Mesmo no cenário mais custoso, com o *dataset* MAESTRO, o Transformer finalizou o processo em 1 hora e 23 minutos. Este resultado demonstra que, embora seja uma arquitetura mais complexa, seu design baseado em atenção permite um processamento paralelo superior ao da natureza sequencial do LSTM, resultando em uma eficiência temporal significativamente maior em todos os cenários testados.

4.2.2 Análise do Uso de Recursos de Hardware

Além do tempo, o consumo de recursos de hardware como memória RAM do sistema, memória da GPU e espaço em disco foi monitorado ao longo da execução de cada experimento. Os gráficos da Figura 16, Figura 18, Figura 20 e Figura 22 ilustram o perfil de consumo de cada cenário, fornecendo insights sobre a demanda computacional de cada abordagem.

4.2.2.1 Cenário rede LSTM com *dataset* Chopin

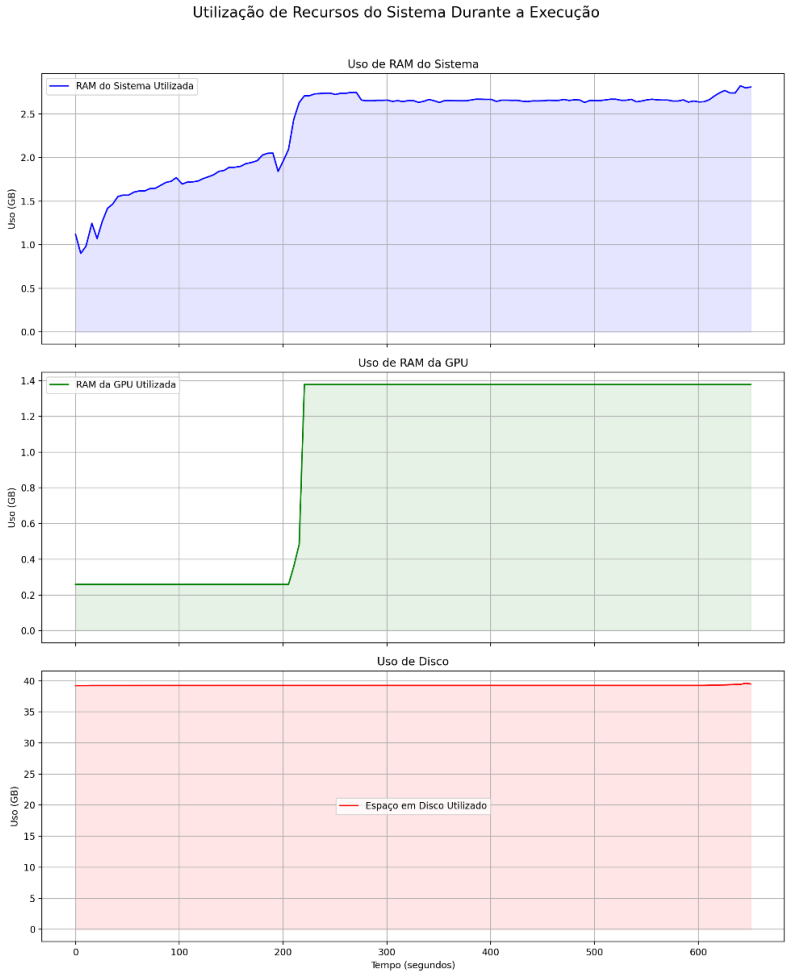
A análise do perfil de recursos para o cenário LSTM com Chopin (Figura 16) revela um comportamento de consumo progressivo e estável.

No gráfico de "Uso de RAM do Sistema", observa-se um aumento gradual durante os primeiros 220 segundos, correspondente ao carregamento e pré-processamento do *dataset*. Após essa fase, o consumo estabiliza-se em aproximadamente 2.5 GB.

O "Uso de RAM da GPU" exibe um padrão claro: o consumo permanece baixo até o início do treinamento (cerca de 220s), quando ocorre um salto abrupto para um platô de 1.3 GB, que se mantém constante até o final da execução.

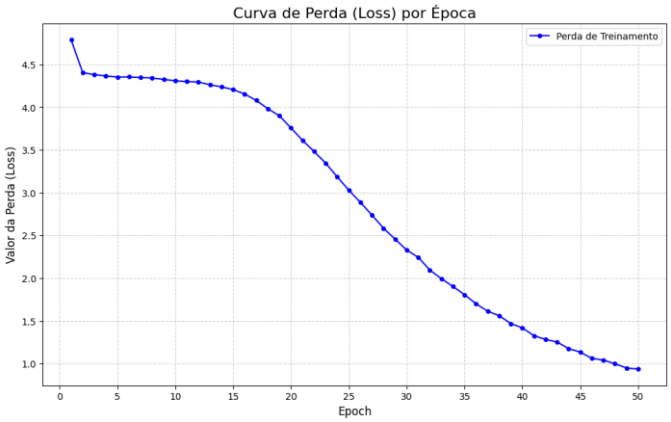
O "Uso de Disco" permaneceu estável durante todo o processo.

Figura 16 - Utilização de Recursos do Sistema (LSTM com *dataset* Chopin)



Fonte: (Autoria Própria)

Figura 17 – Curva de Perda por Epoch (LSTM com *dataset* Chopin)



Fonte: (Autoria Própria)

O gráfico da Figura 17 exibe uma trajetória de aprendizado clara ao longo das 50 épocas. A perda (*loss*) inicia em aproximadamente 4.79 e, após um platô inicial (épocas 1-13), entra em uma fase de descida íngreme e constante, caindo de 4.26 para 1.41. Este é o período principal de aprendizado do modelo. É notável que, ao final das 50 épocas, a curva continua em trajetória descendente, terminando em 0.9363. Isso sugere que o modelo estava aprendendo ativamente e ainda não havia atingido a convergência completa, o que justifica tentar melhorar o resultado estendendo o tempo de treinamento.

4.2.2.2 Cenário rede LSTM com *dataset* MAESTRO

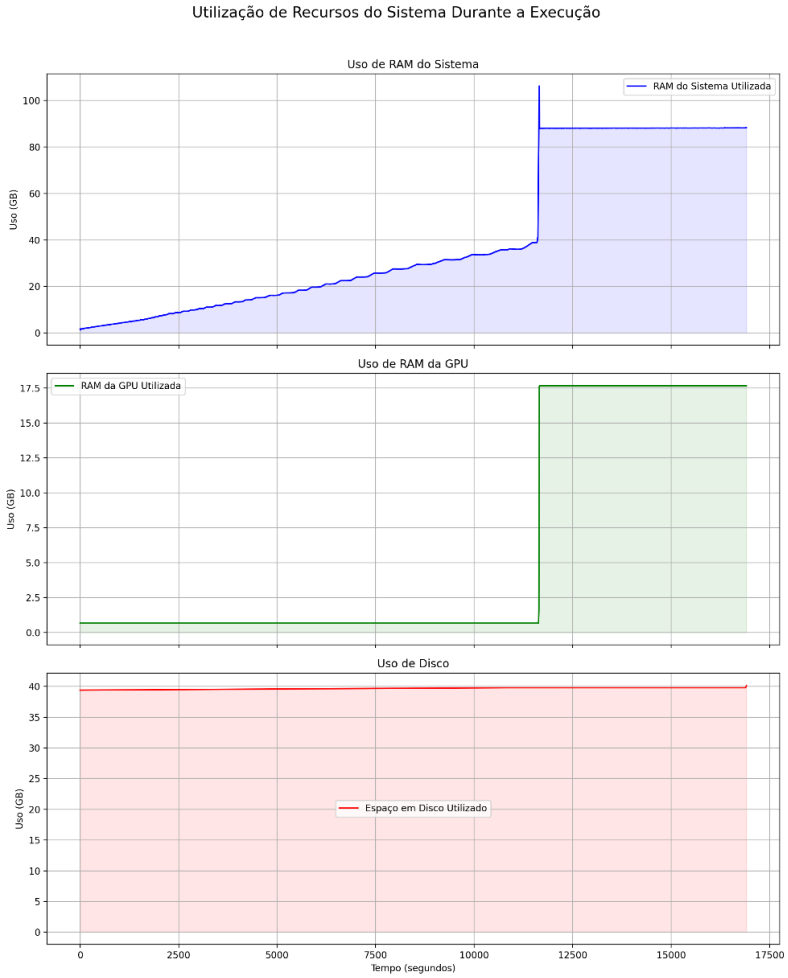
O perfil de recursos para o cenário LSTM com MAESTRO (Figura 18) demonstra uma demanda extrema por memória de sistema.

O gráfico de "Uso de RAM do Sistema" mostra uma escalada contínua por um longo período de mais de 10.000 segundos (quase 3 horas), correspondente à fase de pré-processamento e preparação dos dados. O consumo atinge um patamar de aproximadamente 40 GB, com um pico abrupto que ultrapassa os 100 GB no momento da transição para o treinamento. Este pico extremo sugere que a conversão final dos dados para o formato de tensor antes de enviá-los para a GPU é uma operação de altíssimo custo de memória.

O "Uso de RAM da GPU", por sua vez, mostra um salto para um nível de utilização muito elevado, próximo a 17.5 GB, indicando que o tamanho do vocabulário e do modelo final também exigiu uma quantidade massiva de VRAM. Este perfil de consumo posiciona o *pipeline* do LSTM, especialmente sua fase de pré-processamento com music21, como o principal gargalo de recursos de todo o estudo.

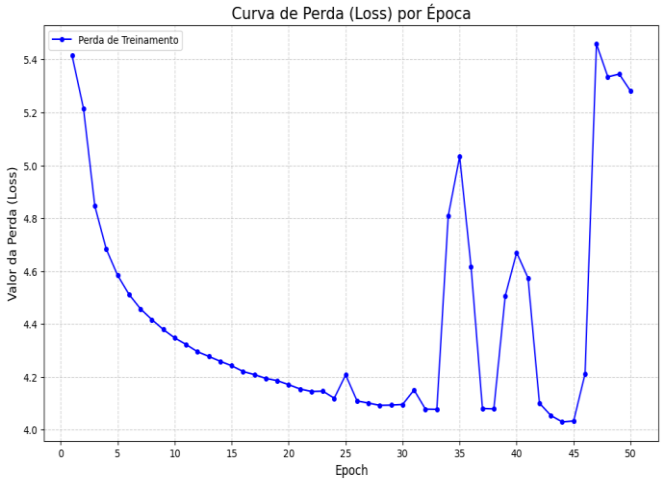
O "Uso de Disco" permaneceu estável durante toda a execução.

Figura 18 - Utilização de Recursos do Sistema (LSTM com *dataset* Maestro)



Fonte: (Autoria Própria)

Figura 19 – Curva de Perda por Epoch (LSTM com *dataset* Maestro)



Fonte: (Autoria Própria)

O cenário LSTM + MAESTRO apresentou um comportamento de treinamento significativamente mais instável. O gráfico da Figura 19 apresenta uma fase inicial de aprendizado (épocas 1-24), onde a perda diminui de 5.41 para 4.11. No entanto, após este ponto, a curva de perda torna-se altamente errática.

Observam-se picos acentuados e repentinos no valor da perda, notadamente nas épocas 34, 35 e 47, onde o erro "explode" e retorna a níveis próximos aos do início do treinamento. Esta instabilidade sugere que o modelo, embora capaz de aprender padrões iniciais, falhou em convergir de forma estável no *dataset* MAESTRO, que é muito maior e mais complexo. A combinação de uma taxa de aprendizado relativamente alta (0.01) com a complexidade dos dados provavelmente fez com que o otimizador "saltasse" para fora de boas regiões na curva de perda, desta forma pode-se tentar melhorar o resultado usando outro otimizador e outra taxa de aprendizado. Este gráfico de treinamento instável é um forte indicador que justifica o resultado qualitativo ruim (o "colapso de modelo") e corrobora a estratégia de usar um otimizador e uma taxa de aprendizado mais conservadores na segunda rodada.

4.2.2.3 Cenário rede Transformer com *dataset* Chopin

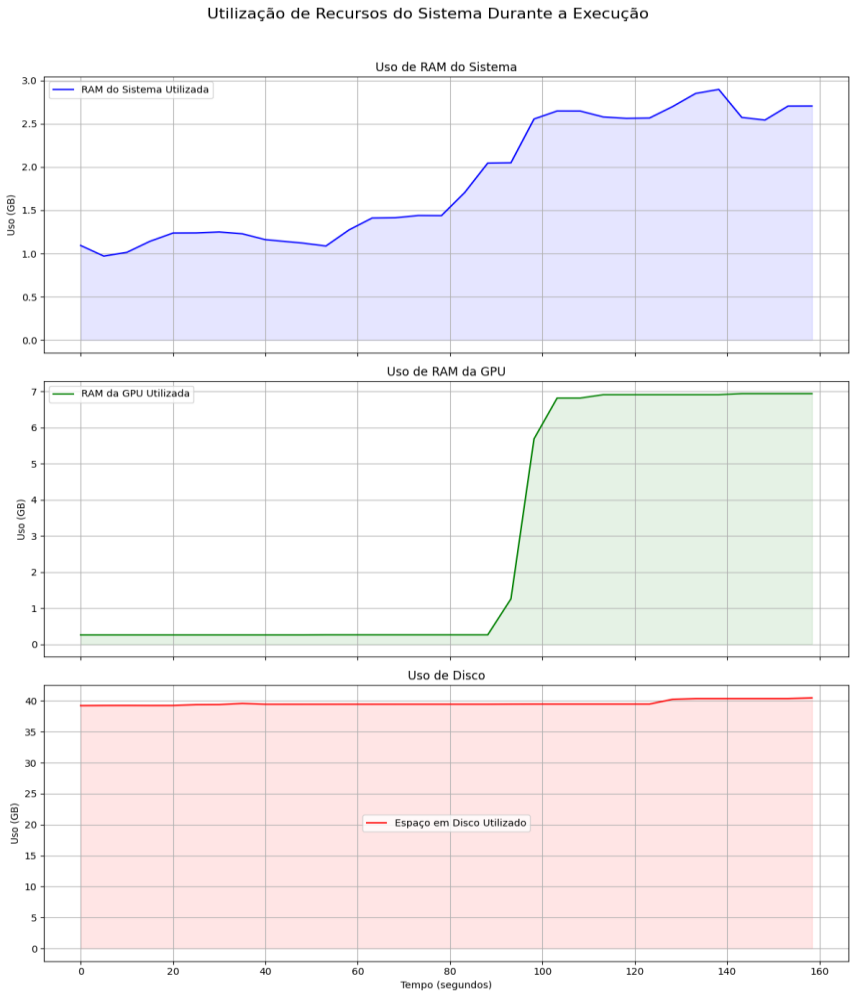
A análise do cenário Transformer com Chopin (Figura 20) revela um perfil de consumo contrastante.

A execução é notavelmente rápida (menos de 3 minutos), com um uso de RAM do sistema contido, atingindo um pico de 3 GB.

A principal diferença reside no "Uso de RAM da GPU", que exibe um salto vertiginoso para quase 7 GB ao iniciar o treinamento. Este uso intensivo de VRAM é uma característica intrínseca da arquitetura Transformer, que precisa manter em memória seus múltiplos e complexos blocos de atenção.

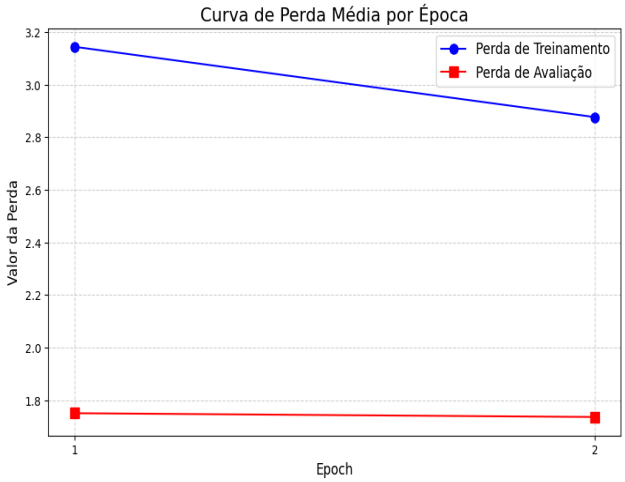
O "Uso de Disco" permaneceu estável, apresentando apenas um leve aumento no final, consistente com o salvamento do arquivo de música gerado.

Figura 20 - Utilização de Recursos do Sistema (Transformers com *dataset* Chopin)



Fonte: (Autoria Própria)

Figura 21 – Curva de Perda e Avaliação por Epoch (Transformers com *dataset* Chopin)



Fonte: (Autoria Própria)

O gráfico para o cenário Transformer + Chopin (Figura 21) ilustra um modelo em estágio inicial de aprendizado, com apenas 2 épocas. A curva de perda de treinamento (linha azul) mostra uma clara tendência de queda, indicando que o aprendizado estava ocorrendo. Notavelmente, a perda de avaliação (linha vermelha) permanece estável e significativamente mais baixa que a perda de treinamento. Este fenômeno é comum em arquiteturas com *dropout*, como o Transformer, pois a regularização é aplicada durante o treino (aumentando a perda) mas não durante a avaliação (GOODFELLOW; BENGIO; COURVILLE, 2016). O estado de *underfitting* (subajuste) evidente neste gráfico justifica o resultado qualitativo atonal e textural observado e a necessidade de estender o treinamento na rodada de otimização para obter um melhor resultado.

4.2.2.4 Cenário rede Transformer com *dataset* MAESTRO

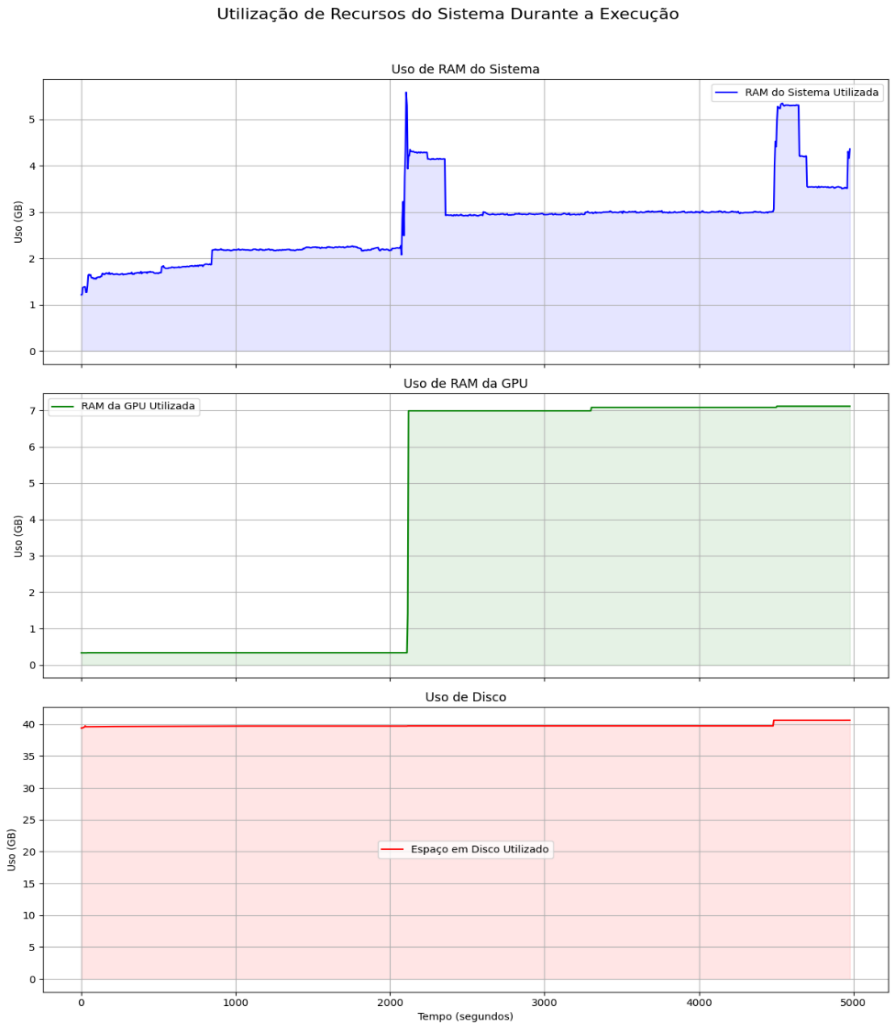
Finalmente, o cenário Transformer com MAESTRO (Figura 22), apresenta um perfil de consumo que reforça sua alta demanda.

O "Uso de RAM do Sistema" exibe um comportamento com múltiplos picos: um salto inicial acentuado para mais de 5 GB por volta dos 2000 segundos, correspondente à fase de carregamento e processamento do *dataset* MAESTRO, seguido por um platô de treinamento em torno de 3 GB. Observam-se novos picos de alta utilização de RAM próximo ao final da execução, possivelmente relacionados a processos de salvamento do modelo e geração da amostra musical.

A RAM da GPU, de forma consistente com a arquitetura, salta para seu nível máximo de utilização, próximo a 7 GB, logo no início do treinamento e permanece nesse patamar até o fim, confirmando que a VRAM é o principal gargalo para esta arquitetura.

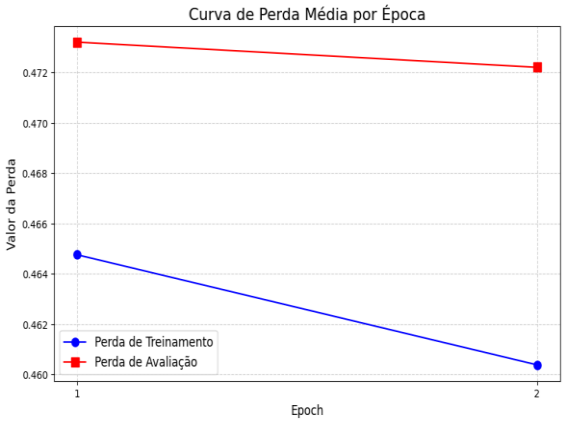
O "Uso de Disco" apresentou um leve aumento no final, consistente com o salvamento dos arquivos gerados.

Figura 22 - Utilização de Recursos do Sistema (Transformers com *dataset* Maestro)



Fonte: (Autoria Própria)

Figura 23 – Curva de Perda e Avaliação por Epoch (Transformers com *dataset* Maestro)



Fonte: (Autoria Própria)

O gráfico para o cenário Transformer + MAESTRO (Figura 23) mostra o modelo em seu estágio mais inicial de treinamento, também interrompido após 2 épocas. Ambas as curvas, de treinamento (linha azul) e de avaliação (linha vermelha), apresentam uma leve queda, indicando que o modelo iniciou o processo de aprendizado. No entanto, o treinamento foi interrompido antes que qualquer convergência significativa pudesse ocorrer.

4.2.3 Análise Qualitativa e Avaliação Musical

Após a análise dos aspectos computacionais, esta seção se aprofunda foca na avaliação da qualidade musical das composições geradas. Diferentemente das métricas quantitativas de performance, a qualidade artística é inerentemente subjetiva. Para abordar essa subjetividade de forma estruturada, foi conduzido um painel de avaliação com cinco indivíduos com diferentes formações e experiências no universo musical: um vocalista, um baterista, um produtor musical, um músico amador e um estudante de conservatório.

A cada avaliador foram apresentados os quatro áudios gerados de forma cega, sem a identificação do modelo ou do *dataset* de origem. Os avaliadores foram instruídos a atribuir uma nota de 1 (muito fraco) a 5 (excelente) para cada peça, com base em três critérios: Coerência Melódica, Riqueza Harmônica e Estrutura Rítmica. A "Nota Final", inspirada na Matriz GUT, foi calculada como o produto das três notas médias, visando ponderar o desempenho equilibrado entre os critérios.

4.2.3.1 Cenário 1: LSTM + Chopin

O primeiro áudio, com duração de 50 segundos, foi gerado pela arquitetura LSTM treinada no *dataset* estilisticamente focado de Frédéric Chopin.

São feitas as seguintes observações sobre o áudio gerado:

a) A análise auditiva revela uma composição com pulso constante e moderado;

b) A melodia se apresenta de forma fragmentada, com arpejos curtos e repetição de notas que não se consolidam em frases musicais claras;

c) O aspecto mais deficiente é a harmonia, que soa predominantemente atonal e dissonante, sem seguir progressões de acordes funcionais, distanciando-se drasticamente do estilo Romântico do *dataset* de treinamento;

d) O ritmo, embora estável, é mecânico e carece de qualquer variação, baseando-se em uma figura rítmica única e ininterrupta;

e) O resultado geral é o de um exercício técnico errático, que demonstra a capacidade do modelo de gerar sequências de notas, mas falha em capturar a estrutura musical complexa de Chopin.

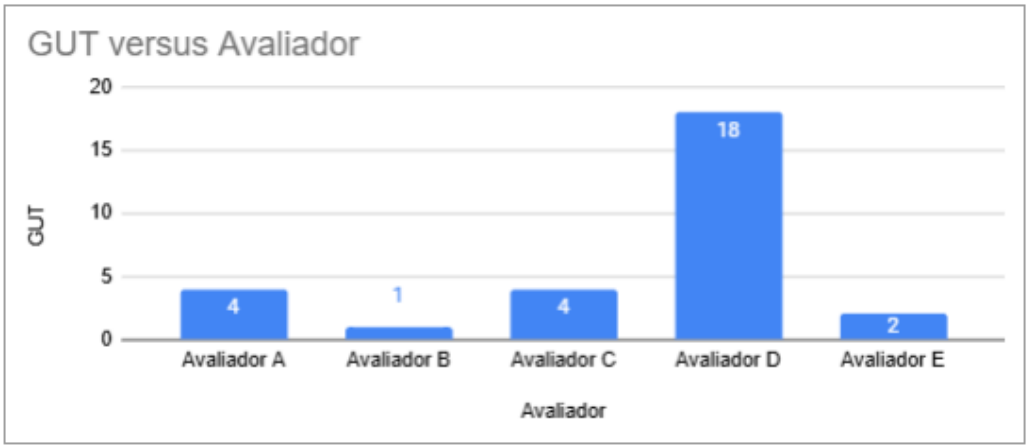
Na Figura 24 são apresentadas as notas elencadas por cada um dos avaliadores.

Figura 24 - Avaliação Qualitativa (LSTM + Chopin)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	2	1	2	4	A melodia é "quebrada" e impossível de cantar; a harmonia soa errada.
Avaliador B	Baterista	1	1	1	1	Ritmo excessivamente simples, "de maquininha", sem nenhuma dinâmica ou variação.
Avaliador C	Produtor Musical	2	1	2	4	Somente ruído digital estruturado; sem coerência harmônica ou melódica para ser aproveitado.
Avaliador D	Músico Amador	3	2	3	18	Reconhece que há notas de piano e um ritmo, mas a música "não vai para lugar nenhum".
Avaliador E	Conservatório	1	1	2	2	Critica a total ausência de funcionalidade harmônica e o desenvolvimento melódico nulo.

Fonte: (Autoria Própria)

Figura 25 – GUT vs Avaliador (LSTM + Chopin)



Fonte: (Autoria Própria)

A Figura 25 ilustra a variação nas percepções do painel sobre este áudio. Fica evidente a divergência significativa do Avaliador D (Músico Amador), que atribuiu uma

nota final (GUT) de 18, que contrasta fortemente com os demais avaliadores (A, B, C e E), cujas notas finais foram 4 ou menos. Analisando os dados da Figura 24, esta pontuação elevada foi impulsionada por uma percepção mais positiva dos critérios de Melodia (3) e Ritmo (3), sugerindo que a peça, embora tecnicamente falha para especialistas, foi percebida como musicalmente aceitável por um ouvinte com um perfil diferente.

4.2.3.2 Cenário 2: LSTM + MAESTRO

O segundo áudio, também com duração de 50 segundos, foi gerado pelo mesmo modelo LSTM, mas treinado no vasto e variado *dataset* MAESTRO.

São feitas as seguintes observações sobre o áudio gerado:

a) Surpreendentemente, o resultado sonoro é muito similar ao do cenário anterior;

b) A melodia continua fragmentada;

c) A harmonia permanece atonal e dissonante; e

d) O ritmo é igualmente mecânico e uniforme.

Esta semelhança notável sugere que a arquitetura LSTM, com sua representação simbólica simplista, atingiu um teto em sua capacidade de aprendizado. Mesmo exposto à riqueza do *dataset* MAESTRO, o modelo não foi capaz de internalizar e reproduzir padrões musicais mais complexos, indicando que a limitação principal pode residir na própria arquitetura, e não nos dados de treinamento.

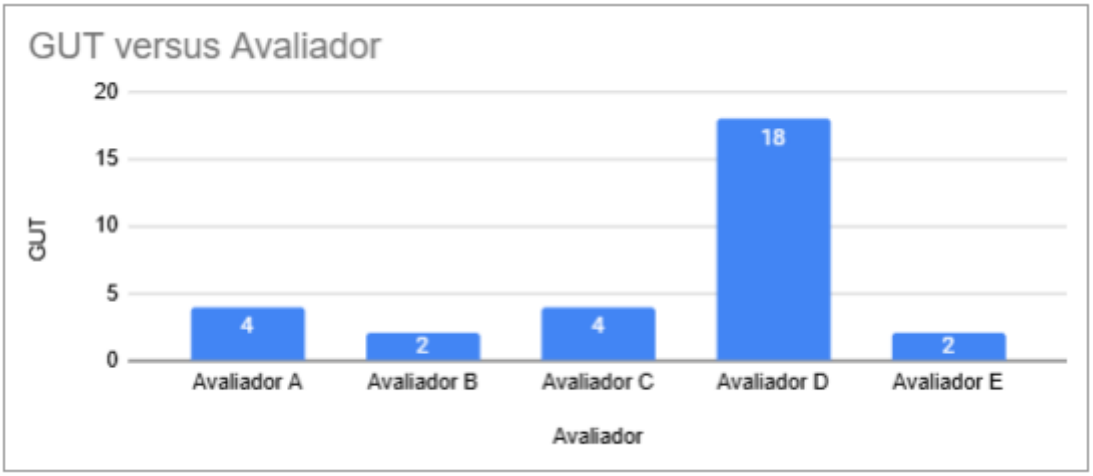
Na Figura 26 são apresentadas as notas elencadas por cada um dos avaliadores.

Figura 26 - Avaliação Qualitativa (LSTM + Maestro)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	1	2	2	4	Melodia "sem pé nem cabeça", impossível de seguir ou cantar.
Avaliador B	Baterista	2	1	1	2	Ritmo novamente "travado" e sem vida, como um metrônomo quebrado.
Avaliador C	Produtor Musical	2	1	2	4	Estruturalmente pobre, harmonicamente inutilizável.
Avaliador D	Músico Amador	3	2	3	18	Soa como "música de videogame antigo que deu defeito", repetitivo.
Avaliador E	Conservatório	1	1	2	2	Nenhuma melhora; continua atonal e com pobreza rítmica e melódica.

Fonte: (Autoria Própria)

Figura 27 – GUT vs Avaliador (LSTM + MAESTRO)



Fonte: (Autoria Própria)

A Figura 27, referente ao cenário LSTM + MAESTRO, exibe um padrão de avaliação notavelmente similar ao do experimento com o *dataset* Chopin. Novamente, há uma forte divergência na percepção do Avaliador D (Músico Amador), que atribuiu uma nota final de 18, enquanto os demais quatro avaliadores mantiveram suas notas em um patamar muito baixo (entre 2 e 4). Esta consistência nos resultados reforça a seguinte conclusão da análise técnica: a arquitetura LSTM, com esta representação de dados, atingiu um teto de capacidade e produziu resultados musicalmente pobres, independentemente da riqueza do *dataset* de treinamento.

4.2.3.3 Cenário 3: Transformer + Chopin

O terceiro áudio, com 5 minutos e 16 segundos, foi gerado pela arquitetura Transformer treinada no *dataset* de Chopin.

São feitas as seguintes observações sobre o áudio gerado:

a) O resultado é radicalmente diferente dos anteriores;

b) A peça abandona qualquer semelhança com o estilo de Chopin, assumindo um caráter de música eletrônica experimental;

c) Não há melodia tradicional, mas sim uma textura sonora densa, criada por uma sucessão rápida e pontilhista de notas;

d) A harmonia é complexa e consistentemente dissonante;

e) O destaque é o ritmo, que é frenético, fluido e altamente sincopado, em total contraste com a rigidez do LSTM; e

f) O resultado é artisticamente polarizador: não é uma imitação de Chopin, mas uma reinterpretação "alienígena" dos dados, demonstrando a capacidade do Transformer de identificar e complexificar micro-padrões rítmicos em detrimento da estrutura melódico-harmônica tradicional.

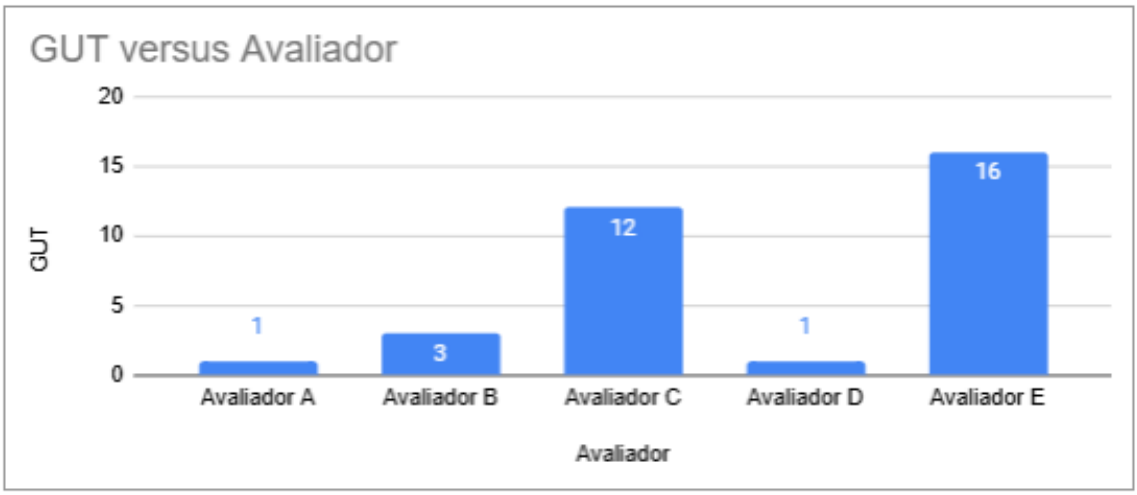
Na Figura 28 são apresentadas as notas elencadas por cada um dos avaliadores.

Figura 28 - Avaliação Qualitativa (Transformers + Chopin)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	1	1	1	1	"Isso não é música, é só um monte de barulho aleatório."
Avaliador B	Baterista	1	1	3	3	Acha o ritmo "interessante e quebrado", mas sem um "beat" para seguir.
Avaliador C	Produtor Musical	2	2	3	12	Reconhece a complexidade e a textura. Pode ver uso como um efeito sonoro, mas não como música.
Avaliador D	Músico Amador	1	1	1	1	Acha a peça "desagradável" e "sem sentido", muito longe de algo que reconhece como música.
Avaliador E	Conservatório	2	2	4	16	Acha a peça academicamente "interessante" como música eletroacústica ou textural, elogia a complexidade rítmica, mas critica a falta de estrutura formal.

Fonte: (Autoria Própria)

Figura 29 – GUT vs Avaliador (Transformers + Chopin)



Fonte: (Autoria Própria)

A Figura 29, referente ao cenário Transformer + Chopin, evidencia a forte polarização nas avaliações. Fica clara a divisão no painel: os Avaliadores A e D atribuíram as notas mais baixas (GUT 1), enquanto os Avaliadores C e E concederam as mais altas (GUT 12 e 16) . Essa divergência está alinhada aos perfis dos ouvintes, conforme detalhado na Figura 28: as pontuações elevadas vieram dos avaliadores com backgrounds mais técnicos (Produtor e Conservatório), que valorizaram a complexidade rítmica e textural. Em contrapartida, as notas baixas vieram dos avaliadores com foco em melodia tradicional (Vocalista e Músico Amador). O gráfico, portanto, valida visualmente a análise de que a peça possui um caráter experimental, que agrada a uns e desagrada a outros.

4.2.3.4 Cenário 4: Transformer + MAESTRO

O áudio final, com duração de 8 minutos e 31 segundos, foi gerado pelo Transformer treinado com o *dataset* MAESTRO. Considera-se que este cenário produziu o resultado musicalmente mais coerente e bem-sucedido de todos.

São feitas as seguintes observações sobre o áudio gerado:

a) A peça apresenta melodias claras, "cantáveis" e com desenvolvimento temático;
b) A harmonia é tonal, funcional e agradável, suportando a melodia de forma eficaz;
c) O ritmo, embora simples, é consistente e serve como uma base sólida para a composição;
d) O resultado é uma peça musical estruturada e esteticamente agradável, que soa como uma composição intencional, talvez para uma trilha sonora.

Este cenário demonstra o potencial da combinação de uma arquitetura eficiente (Transformer) com um *dataset* grande e de alta qualidade (MAESTRO).

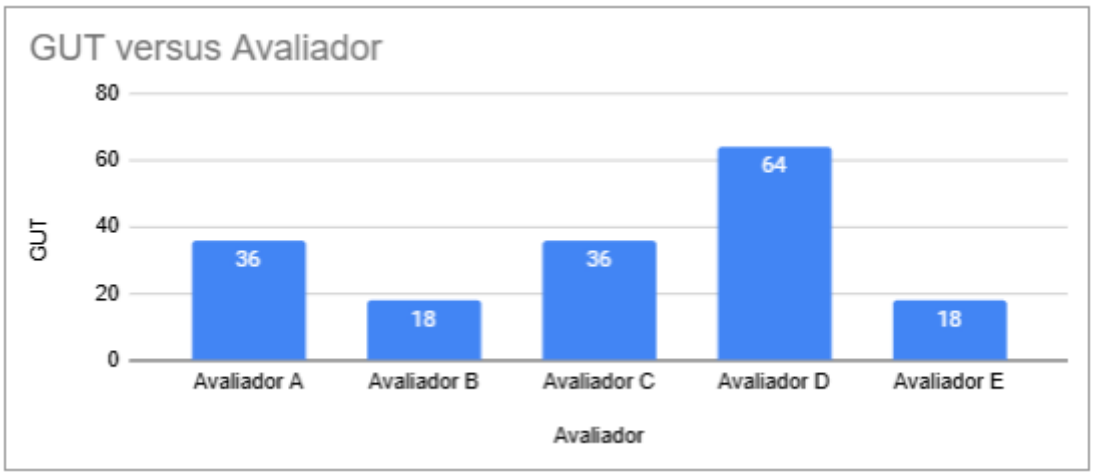
Na Figura 30 são apresentadas as notas elencadas por cada um dos avaliadores.

Figura 30 - Avaliação Qualitativa (Transformers + Maestro)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	4	3	3	36	"Uma melodia de verdade! É bonita, fácil de seguir e cantarolar."
Avaliador B	Baterista	3	3	2	18	O ritmo é simples, mas tem um "feeling" e funciona bem com o resto. Não é mais um robô.
Avaliador C	Produtor Musical	4	3	3	36	"Isso é totalmente utilizável. Soa como uma boa trilha de fundo para um vídeo ou jogo. É coeso."
Avaliador D	Músico Amador	4	4	4	64	Acharia a música "linda" e "inspiradora", a mais musical de todas.
Avaliador E	Conservatório	3	3	2	18	Elogiaria a clareza da estrutura tonal e o desenvolvimento melódico. Criticaria a simplicidade rítmica.

Fonte: (Autoria Própria)

Figura 31 – GUT vs Avaliador (Transformers + Maestro)



Fonte: (Autoria Própria)

A Figura 31, referente ao cenário Transformer + MAESTRO, demonstra o forte consenso positivo em relação à qualidade deste áudio. Em nítido contraste com os outros três cenários, todas as notas GUT são substancialmente altas, variando de 18 a 64. O Avaliador D ("Músico Amador") atribuiu a pontuação mais elevada (64), alinhando-se à sua justificativa de ser a "mais musical de todas" (Figura 30). Mesmo os avaliadores com perfis mais técnicos e críticos (B e E), que atribuíram as notas mais baixas (18), ainda pontuaram esta peça muito acima das outras. O gráfico, portanto, valida visualmente a seguinte conclusão da análise técnica: esta combinação de arquitetura e *dataset* foi a única a produzir um resultado reconhecido como musical e coeso.

4.2.3.5 Síntese da Análise Qualitativa

A análise comparativa dos quatro resultados da rodada de linha de base permite extrair conclusões diretas.

A arquitetura LSTM, independentemente do *dataset*, demonstrou dificuldades em aprender estruturas musicais complexas. Conforme observado nas avaliações apresentadas nas Figura 24 e Figura 26, os áudios gerados foram avaliados como repetitivos e harmonicamente incoerentes. Houve um consenso entre os avaliadores, refletido nas notas baixas para os critérios de Harmonia e Ritmo, alinhando-se à análise técnica.

A arquitetura Transformer mostrou-se altamente sensível aos dados de treinamento. Com o *dataset* MAESTRO (Figura 30), foi a combinação que gerou música estruturada e recebeu um consenso positivo dos avaliadores. Em contrapartida, o cenário Transformer + Chopin (Figura 28) apresentou divergências na avaliação. O laudo técnico (Seção 4.2.3.3) descreveu o resultado como textural e ritmicamente complexo, mas atonal. As notas do painel (Figura 28) refletiram essa dualidade: avaliadores com foco técnico (B e E) atribuíram as notas mais altas do painel ao critério "Ritmo", enquanto outros avaliadores (A e D) atribuíram as notas mais baixas. Esta divergência sugere que o modelo, em seu estado *underfit*, focou em uma dimensão musical (ritmo) em detrimento das outras.

4.3 AJUSTE DOS MODELOS: APROFUNDAMENTO E TREINAMENTO ESTENDIDO

Após a análise da primeira rodada experimental, que estabeleceu uma linha de base (*baseline*) para o desempenho de cada arquitetura, esta seção detalha as estratégias de ajustes implementadas. O objetivo é investigar se um aumento na capacidade do modelo (profundidade da rede) e/ou no tempo de exposição aos dados (duração do treinamento) pode resultar em uma melhora significativa na qualidade musical das composições geradas.

A estratégia de geração determinística (*greedy search*) é mantida idêntica à da primeira rodada, a fim de isolar o impacto das mudanças na arquitetura e no treinamento. Os ajustes são aplicados aos modelos treinados com o *dataset* de Chopin, e os correspondentes resultados são analisados na seção subsequente.

4.3.1 PROPOSTAS DE AJUSTES PARA O MODELO LSTM + CHOPIN

A análise inicial do áudio "LSTM + Chopin" revelou uma composição ritmicamente mecânica e harmonicamente incoerente. Os seguintes ajustes propostos visam aumentar a capacidade de representação do modelo e refinar seu processo de convergência:

a) Ajuste 1: Aumento da Profundidade e Largura da Rede

   I. Hipótese: A arquitetura de linha de base (duas camadas com 512 e 256 unidades) pode não ter a capacidade representacional suficiente para capturar a complexidade harmônica e melódica da música de Chopin.

   II. Ação Proposta: Realizar experimentos para aumentar a complexidade do modelo, explorando duas vias:

      i. Aprofundamento: Adicionar uma terceira camada LSTM à arquitetura (ex: 512 -> 256 -> 128 unidades), permitindo que o modelo aprenda características em um nível de abstração adicional.

      ii. Alargamento: Manter as duas camadas, mas aumentar o número de neurônios em cada uma (ex: 1024 -> 512 unidades), ampliando a capacidade de cada camada de aprender padrões.

b) Ajuste 2: Extensão do Tempo de Treinamento

   I. Hipótese: As 50 épocas de treinamento da linha de base podem ser insuficientes para que o modelo convirja satisfatoriamente.

   II. Ação Proposta: Aumentar o número de épocas de treinamento (ex: para 100 ou 150 épocas). A curva de perda do conjunto de validação será monitorada para identificar o ponto de melhor generalização e evitar o *overfitting*.

c) Ajuste 3: Alteração do Otimizador

I. Hipótese: O otimizador Adamax pode não ser a escolha ideal para esta tarefa específica, e um otimizador diferente poderia encontrar um mínimo melhor na avaliação de perda.

II. Ação Proposta: Substituir o Adamax pelo otimizador Adam, que é o padrão mais utilizado em muitas aplicações de *deep learning*, para verificar se há alguma melhora na velocidade de convergência ou na qualidade do resultado final.

4.3.2 Propostas de Ajuste para o Modelo Transformer + Chopin

O resultado do "Transformer + Chopin" apresentou uma composição focada na textura sonora e na complexidade rítmica, em vez de em uma melodia tradicional. O áudio resultante mostrou-se musicalmente atonal e distante do estilo de referência. Os seguintes ajustes visam tentar que o modelo aprenda estruturas musicais mais coesas:

a) Ajuste 1: Aumento da Profundidade da Rede

I. Hipótese: A arquitetura de 6 camadas do Transformer pode ser aprofundada para capturar relações hierárquicas mais complexas presentes na música de Chopin.

II. Ação Proposta: Aumentar o número de camadas do decodificador do Transformer (o hiperparâmetro nlayers), por exemplo, de 6 para 8 ou 12.

b) Ajuste 2: Extensão Radical do Tempo de Treinamento

I. Hipótese: O resultado "alienígena" da primeira rodada é um sintoma claro de *underfitting* severo. Considera-se que as 2 épocas de treinamento foram insuficientes para o modelo ir além de aprender padrões texturais superficiais.

II. Ação Proposta: Aumentar o número de épocas de treinamento (ex: para 50, 60 ou mais). Considera-se que este é o passo mais crítico para permitir que o modelo tenha tempo suficiente para convergir e aprender as regras de melodia e harmonia, em vez de apenas a complexidade rítmica.

c) Ajuste 3: Alteração do Otimizador

I. Hipótese: O otimizador Adam padrão pode ser aprimorado.

II. Ação Proposta: Substituir o otimizador Adam pelo AdamW. O AdamW implementa uma técnica de decaimento de peso (*weight decay*) de forma mais eficaz do que o Adam tradicional, o que pode levar a uma melhor generalização e é frequentemente a escolha preferida para o treinamento de Transformers (LOSHCHILOV; HUTTER, 2019).

4.3.3 Propostas de Ajustes para o Modelo LSTM + MAESTRO

A análise inicial do "LSTM + MAESTRO" mostrou que, mesmo exposto a um vasto e rico *dataset*, o modelo produziu um resultado musicalmente pobre, similar ao do *dataset* de Chopin. Considera-se que isso sugere que a arquitetura de linha de base está subutilizando a riqueza dos dados e operando abaixo de sua capacidade potencial.

Os seguintes ajustes propostos visam dar ao modelo mais capacidade e melhores condições para aprender com o complexo *dataset* MAESTRO.

a) Ajuste 1: Aumento da Profundidade e Largura da Rede

I. Hipótese: A arquitetura de duas camadas com 512 e 256 unidades é muito simples para capturar e modelar a diversidade de estilos e a complexidade harmônica presentes no *dataset* MAESTRO;

II. Ação Proposta: Aumentar a capacidade de representação do modelo. Como o gargalo de tempo foi identificado na preparação dos dados e não no treinamento, experimentar com uma rede maior é uma estratégia promissora. Propõe-se testar uma arquitetura mais profunda e mais larga, como a Opção C definida para o *dataset* de Chopin (três camadas LSTM com 1024, 512 e 256 unidades), para verificar se um modelo com mais parâmetros consegue extrair padrões mais significativos do vasto *dataset*.

b) Ajuste 2: Extensão do Tempo de Treinamento

I. Hipótese: As 50 épocas podem ser insuficientes para que um modelo, mesmo o de linha de base, convirja em um *dataset* do tamanho do MAESTRO.

II. Ação Proposta: Aumentar o número de épocas de treinamento (ex: para 100 ou mais).

c) Ajuste 3: Ajuste Fino da Taxa de Aprendizado e do Otimizador

I. Hipótese: A combinação de otimizador e taxa de aprendizado pode não ser a ideal para um treinamento tão longo e complexo.

II. Ação Proposta: Trocar o otimizador Adamax pelo Adam e experimentar uma taxa de aprendizado menor (ex: 0.001). Uma taxa menor pode permitir que o modelo navegue pela complexa paisagem de perda do MAESTRO de forma mais cuidadosa, encontrando mínimos de erro melhores ao longo de um treinamento mais extenso.

4.3.4 Propostas de Ajustes para o Modelo Transformer + MAESTRO

O cenário "Transformer + MAESTRO" produziu a composição mais bem-sucedida da rodada inicial, com melodia clara, harmonia funcional e boa estrutura.

Os seguintes ajustes propostos para esta etapa visam, portanto, o refinamento do modelo, com o objetivo de explorar se é possível extrair ainda mais nuances e complexidade do rico *dataset* MAESTRO:

a) Ajuste 1 (Principal): Extensão e Refinamento do Tempo de Treinamento

I. Hipótese: Embora o treinamento inicial tenha sido mais longo que o do Chopin, ele pode não ter sido suficiente para uma convergência satisfatória do modelo no massivo *dataset* MAESTRO.

II. Ação Proposta: Aumentar o número de épocas de treinamento de forma controlada (ex: para 10, 15 ou mais), avaliando a qualidade do áudio gerado em diferentes pontos de verificação (*checkpoints*). Isso permitirá identificar o ponto ótimo em que o modelo atinge seu pico de performance antes de começar a apresentar sinais de *overfitting*.

b) Ajuste 2: Implementação de um Agendador de Taxa de Aprendizado (Scheduler)

I. Hipótese: Uma taxa de aprendizado fixa, mesmo que pequena, pode não ser a estratégia ideal para um treinamento longo em um *dataset* complexo.

Uma taxa de aprendizado que se ajusta dinamicamente pode levar a uma convergência mais estável e a um mínimo de perda melhor;

II. Ação Proposta: Implementar um agendador de taxa de aprendizado no *loop* de treinamento. A abordagem padrão para Transformers, que consiste em uma fase de "aquecimento" (*warm-up*) seguida por um decaimento gradual (linear ou cosseno), será testada. Essa técnica permite que o modelo se estabilize no início do treinamento com passos menores e depois acelere o aprendizado, reduzindo a taxa novamente no final para um ajuste fino.

c) Ajuste 3: Alteração do Otimizador para AdamW

I. Hipótese: O otimizador Adam pode ser substituído por uma variante mais moderna que demonstrou melhores resultados de generalização em modelos Transformer;

II. Ação Proposta: Substituir o otimizador Adam pelo AdamW. Conforme proposto por Loshchilov e Hutter (2019), o AdamW desacopla o mecanismo de decaimento de peso (weight decay) da atualização do gradiente, o que frequentemente resulta em modelos mais bem generalizados.

4.4 ANÁLISE DA SEGUNDA RODADA EXPERIMENTAL (PÓS- AJUSTES)

Após a análise dos resultados da linha de base, foram implementadas as estratégias de ajustes detalhadas na Seção 4.3, com o objetivo de aprimorar a qualidade musical das composições geradas. Os códigos-fonte modificados para esta segunda rodada, refletindo as otimizações de arquitetura e treinamento, estão disponíveis nos Adendos E a H. Esta seção apresenta e analisa os resultados desta segunda rodada experimental, focando novamente tanto no impacto computacional das otimizações quanto na avaliação da qualidade musical resultante.

4.4.1 DESEMPENHO COMPUTACIONAL PÓS- AJUSTES

4.4.1.1 Cenário LSTM com ajustes + *dataset* Chopin

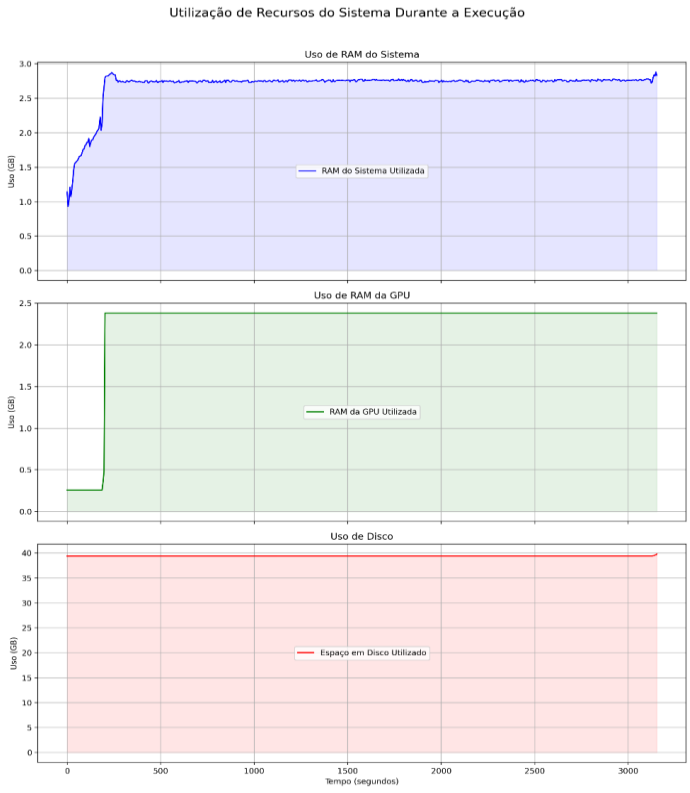
A análise do perfil de recursos para o cenário LSTM com ajustes + *dataset* Chopin (Figura 32) evidencia o custo computacional associado ao aumento da capacidade do modelo com as seguintes observações:

a) O "Uso de RAM do Sistema" segue o padrão de aumento inicial durante o pré-processamento, estabilizando-se em um platô de aproximadamente 2.8 GB;

b) A mudança mais significativa ocorre no "Uso de RAM da GPU", que salta para um patamar estável de 2.4 GB durante o treinamento, que é um aumento substancial em comparação com os 1.3 GB utilizados pelo modelo de linha de base. Este resultado confirma que a arquitetura mais profunda e mais larga exige significativamente mais memória de GPU para ser treinada;

c) O "Uso de Disco" permaneceu estável.

Figura 32 - Utilização de Recursos do Sistema (LSTM com ajustes + *dataset* Chopin)

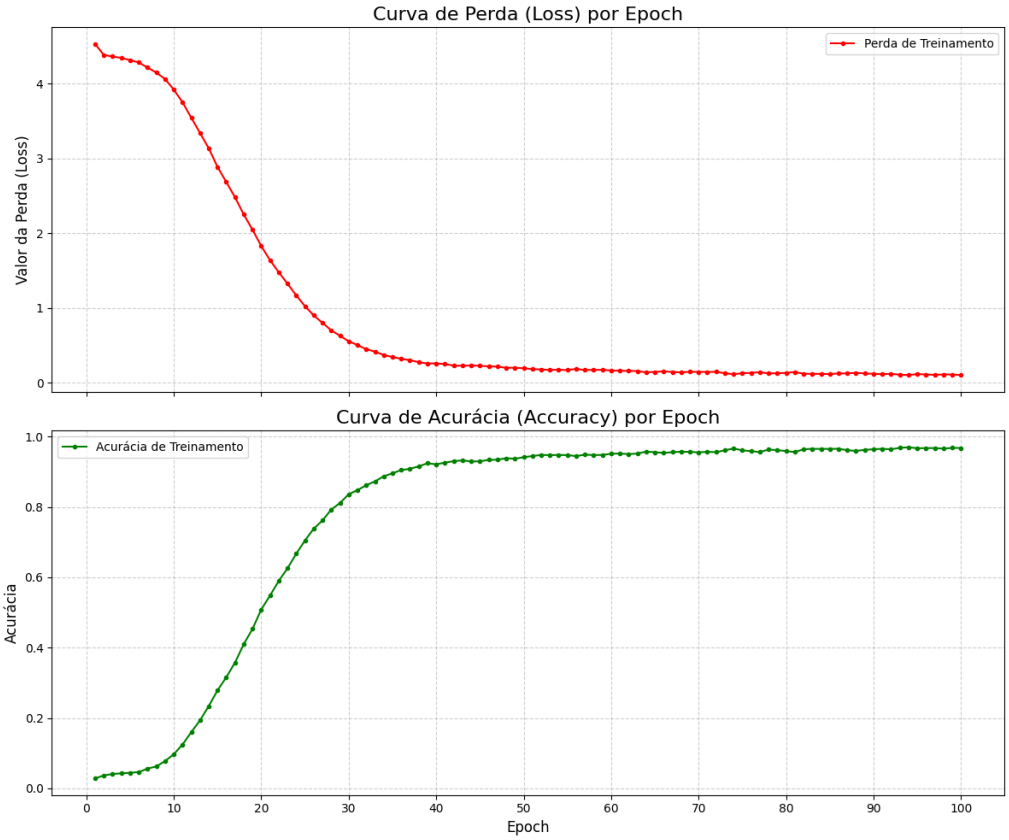


Fonte: (Autoria Própria)

Além do consumo de recursos, as curvas de aprendizado da rodada com ajustes (Figura 33) fornecem *insights* sobre a eficácia do treinamento. O gráfico de perda (*loss*) do cenário LSTM com ajustes + *dataset* Chopin demonstra um processo de convergência bem-sucedido. A perda de treinamento inicia alta (próxima a 4.7) e, após uma breve estabilização, inicia uma descida íngreme entre as épocas 10 e 40, indicando a principal fase de aprendizado do modelo

A curva de acurácia espelha esse comportamento, subindo de valores próximos de zero para mais de 90% no mesmo período. Nas 50 épocas finais, ambas as curvas se estabilizam em um platô de alto desempenho (Perda < 0.2, Acurácia > 95%). Esta alta acurácia quantitativa justifica a melhora observada na análise qualitativa: o modelo tornou-se muito proficiente em prever os padrões do corpus, resultando em uma peça musical tonal e coerente.

Figura 33 – Curva de Perda e Acurácia (LSTM com ajustes + *dataset* Chopin)



Fonte: (Autoria Própria)

4.4.1.2 Cenário LSTM com ajustes + *dataset* Maestro

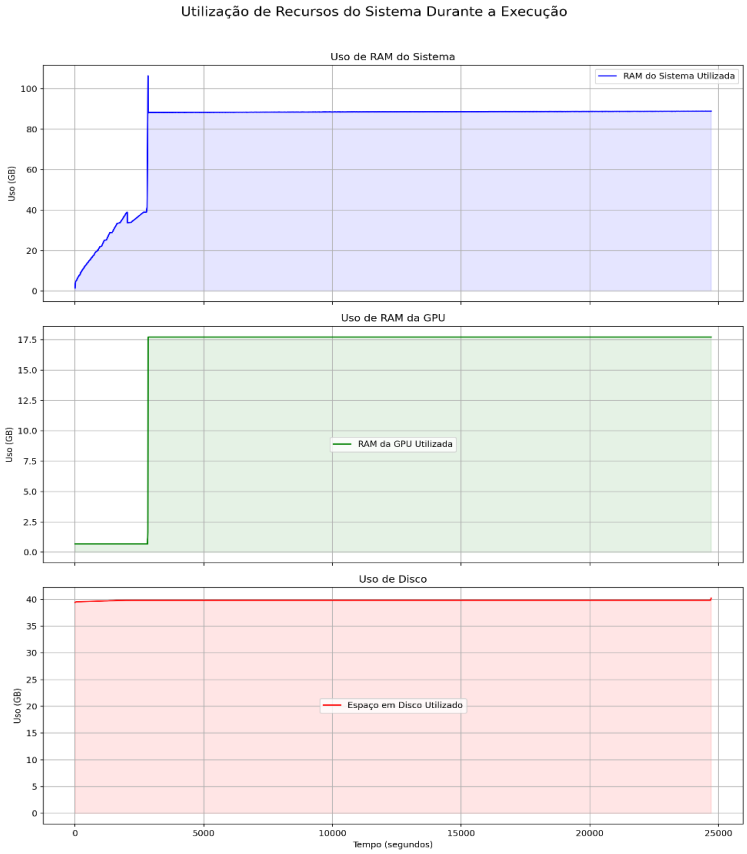
A análise do perfil de recursos para o cenário LSTM com ajustes + *dataset* MAESTRO (Figura 34) mostra uma demanda massiva por memória de sistema e GPU com as seguintes observações:

a) O "Uso de RAM do Sistema" exibe um comportamento de "saturação", com um pico abrupto que atinge e se mantém em um patamar superior a 80 GB durante a fase final de preparação dos dados e o início do treinamento;

b) O "Uso de RAM da GPU" também salta para um nível de utilização muito elevado, próximo a 17.5 GB, refletindo o grande tamanho do modelo e do vocabulário. Este perfil de consumo extremo, que leva os recursos de hardware ao seu limite, posiciona este cenário como o mais custoso do estudo, evidenciando os desafios de escalar arquiteturas LSTM para *datasets* de grande porte;

c) O "Uso de Disco" permaneceu estável.

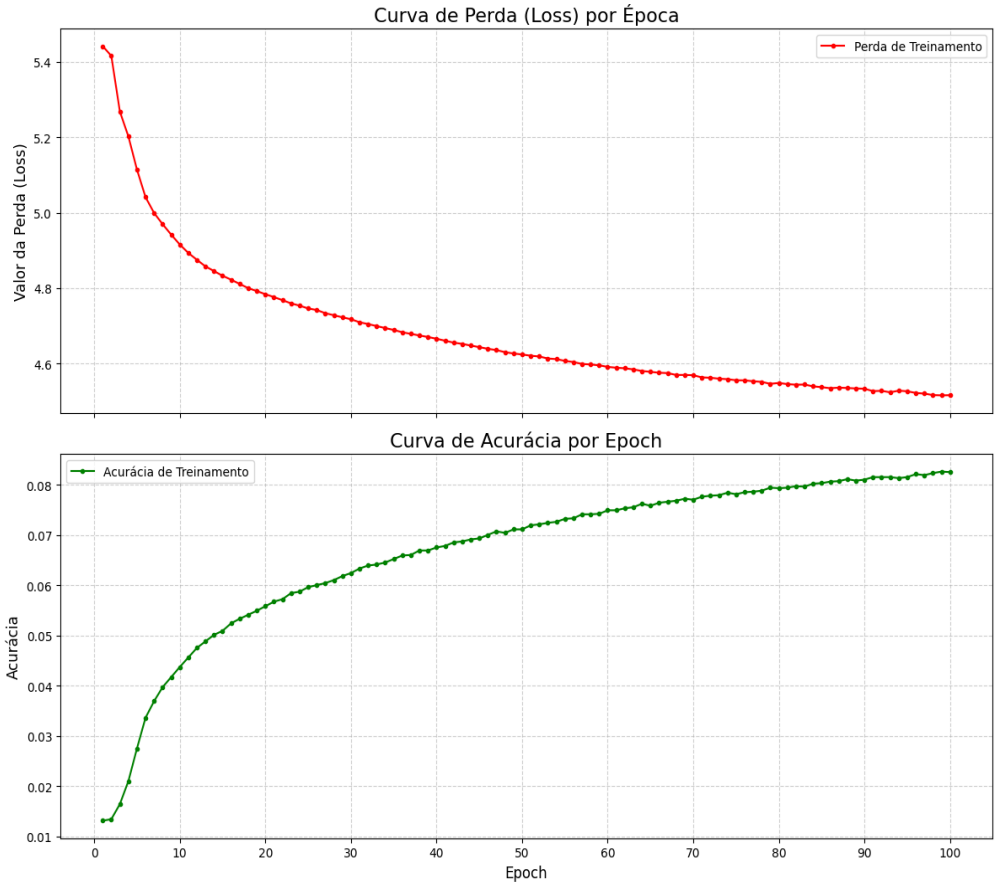
Figura 34 - Utilização de Recursos do Sistema (LSTM com ajustes + *dataset* Maestro)



Fonte: (Autoria Própria)

As curvas de aprendizado do cenário LSTM com ajustes + *dataset* Maestro, apresentadas na Figura 35, são a principal evidência quantitativa que explica o fracasso qualitativo deste experimento. O gráfico de Perda (*Loss*) mostra uma descida lenta, mas contínua, ao longo das 100 épocas. No entanto, o gráfico de Acurácia (*Accuracy*) revela o problema central: o modelo, mesmo com uma arquitetura mais profunda e um treinamento mais longo, falhou em aprender os padrões do *dataset* MAESTRO.

Figura 35 – Curva de Perda e Acurácia (LSTM com ajustes + *dataset* Maestro)



Fonte: (Autoria Própria)

A acurácia inicia em um nível próximo do aleatório (cerca de 1.3%) e, após 100 épocas, atinge um teto de performance extremamente baixo, em torno de 8.2%. Isso indica que o modelo não está sofrendo de *overfitting* (onde a acurácia de treinamento seria alta), mas sim de um severo *underfitting* ou uma falha de convergência. O modelo foi incapaz de modelar a complexidade e a vasta diversidade do *dataset*.

Esta falha em aprender justifica o "colapso de modelo" observado no áudio gerado. A rede neural, ao não conseguir encontrar padrões musicais complexos, convergiu para um mínimo local "preguiçoso": a estratégia estatisticamente mais simples para minimizar a perda, que foi a repetição incessante de um único evento musical.

4.4.1.3 Cenário Transformer com ajustes + *dataset* Chopin

Da análise do perfil de recursos para o cenário Transformer com ajustes + *dataset* Chopin (Figura 36) tem-se as seguintes observações:

a) O uso de RAM do Sistema se estabiliza em um platô em torno de 2.5 GB após a fase inicial de carregamento;

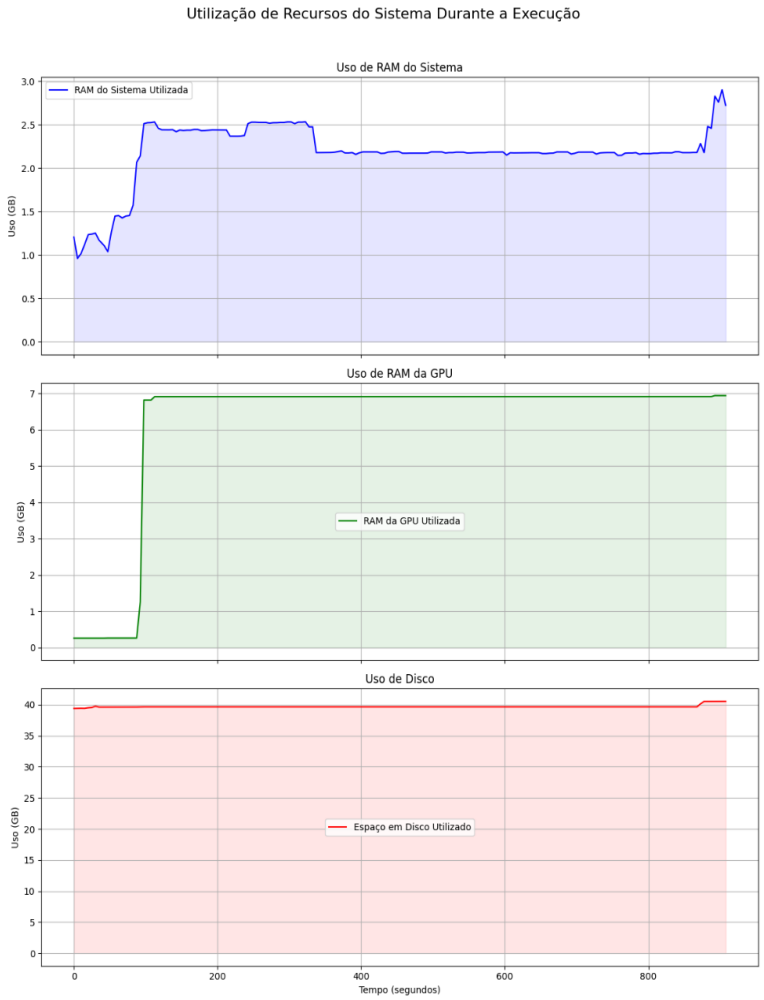
b) O "Uso de RAM da GPU" demonstra novamente o perfil característico da arquitetura, com um salto abrupto para um nível de utilização máxima de aproximadamente 7 GB assim que o treinamento se inicia. O consumo permanece nesse patamar elevado durante toda a execução, confirmando que a demanda de VRAM do Transformer é primariamente ditada por sua complexidade arquitetural;

c) O "Uso de Disco" permaneceu estável.

d) A Perda de Treinamento (linha azul) decresce de forma consistente ao longo das 50 épocas, indicando que o modelo estava aprendendo e se ajustando continuamente ao corpus de Chopin. No entanto, a Perda de Avaliação (linha vermelha) conta uma história diferente: ela diminui apenas nas primeiras épocas, atingindo seu ponto mínimo por volta da Época 7, e a partir daí começa a subir de forma constante.

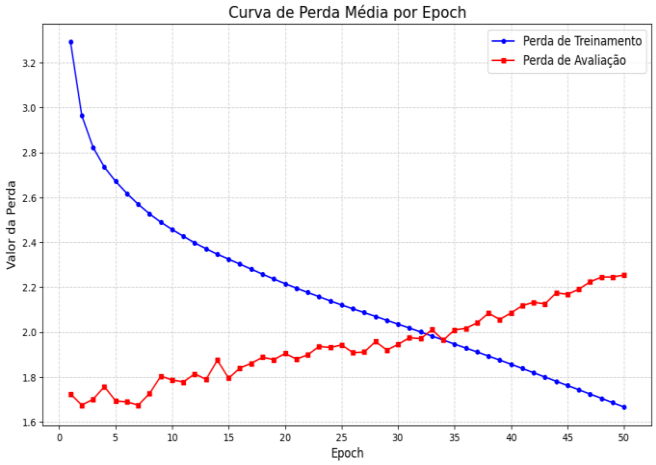
e) Este comportamento divergente é o sintoma claro de que o modelo começou a "memorizar" os dados de treinamento, perdendo sua capacidade de generalização para dados não vistos. O modelo final, salvo na Época 50, estava, portanto, em um estado de overfit. Isso justifica o resultado musical: o modelo convergiu para uma solução "segura" e estatisticamente provável dentro do corpus (um arpejo tonal simples), mas que não reflete a complexidade ou a capacidade de generalização que um modelo treinado até o "ponto ideal" (Época 7) poderia ter oferecido.

Figura 36 - Utilização de Recursos do Sistema (Transformers com *dataset* Chopin)



Fonte: (Autoria Própria)

Figura 37 – Curva de Perda e Avaliação (Transformers com ajustes + dataset Chopin)



Fonte: (Autoria Própria)

4.4.1.4 Cenário Transformer com ajustes + *dataset* MAESTRO

Da análise do perfil de recursos para o cenário Transformer com ajustes + *dataset* MAESTRO (Figura 38) têm-se as seguintes observações:

a) Ocorre um pico no uso de RAM do Sistema que atinge aproximadamente 4.5 GB durante a fase de carregamento dos dados, estabilizando-se em um platô em torno de 2.5 GB durante o longo período de treinamento;

b) O "Uso de RAM da GPU" segue o padrão esperado para a arquitetura, com um salto abrupto para um nível de utilização máximo de quase 9 GB assim que o treinamento se inicia, permanecendo nesse patamar por mais de 4 horas. Este consumo elevado e sustentado de VRAM confirma o Transformer como a arquitetura mais intensiva em memória de GPU;

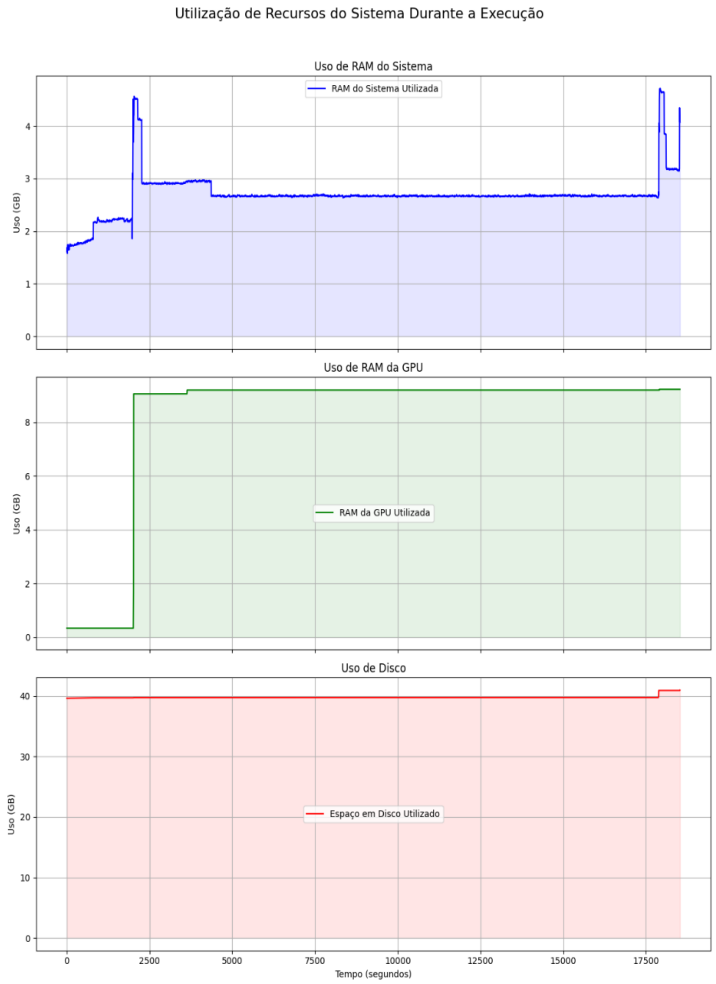
c) O "Uso de Disco" permaneceu estável.

As curvas de aprendizado do cenário Transformer com ajustes + dataset MAESTRO, apresentadas na Figura 39, validam o sucesso da otimização e justificam o resultado qualitativo superior. O gráfico exibe um processo de convergência muito rápido e estável.

A Perda de Treinamento (linha azul) mostra uma queda acentuada nas primeiras 5 épocas, passando de 0.486 para 0.479, e então estabiliza em um platô, com o modelo continuando a fazer apenas ajustes mínimos. De forma similar, a Perda de Avaliação (linha vermelha), apesar de alguma volatilidade inicial (com um pico na Época 3), também encontra um platô estável e baixo.

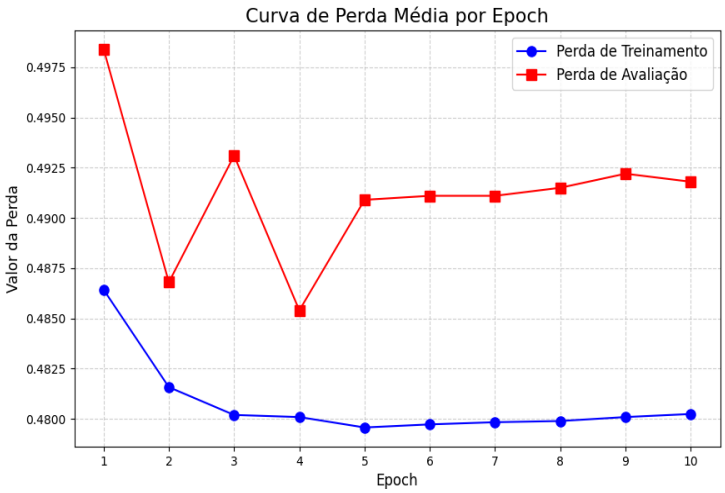
O fato de ambas as curvas de perda caírem rapidamente e se manterem baixas e estáveis, sem sinais de divergência (como no LSTM+MAESTRO) ou de *overfitting* claro (como no Transformer+Chopin), indica que o modelo atingiu uma boa convergência. Ele aprendeu os padrões complexos do *dataset* MAESTRO de forma eficiente e generalizada, o que se traduziu diretamente na geração de música coesa, melódica e harmonicamente funcional.

Figura 38 - Utilização de Recursos do Sistema (Transformer com ajustes + *dataset* MAESTRO)



Fonte: (Autoria Própria)

Figura 39 – Curva de Perda e Avaliação (Transformer com ajustes + *dataset* MAESTRO)



Fonte: (Autoria Própria)

4.4.1.5 Análise Comparativa do Custo Computacional (Rodada 1 vs. Rodada 2)

Em síntese, a rodada de otimizações, conforme detalhado nos cenários de 4.4.1.1 a 4.4.1.4, confirmou o custo computacional direto dos ajustes implementados. Ao comparar estes resultados com os da linha de base (Seção 4.2.1), observa-se um aumento substancial no tempo total de execução em todos os quatro cenários. O cenário LSTM + Chopin, por exemplo, saltou de 10 minutos e 50 segundos para 52 minutos e 35 segundos na rodada com ajustes, um aumento de 385%. De forma similar, o Transformer + MAESTRO aumentou de 1 hora e 23 minutos para 5 horas e 8 minutos. Este aumento é uma consequência direta da decisão de estender o número de épocas e aprofundar as arquiteturas, demonstrando o claro *trade-off* entre o investimento em tempo de treinamento e a busca por resultados qualitativamente superiores.

4.4.2 Avaliação Musical Comparativa (Rodada 1 vs. Rodada 2)

Após a implementação das estratégias de ajustes, foi conduzida uma nova rodada de avaliação qualitativa com o mesmo painel de cinco avaliadores, seguindo os critérios de Melodia, Harmonia e Ritmo. Esta seção analisa o impacto das otimizações na qualidade musical das composições, comparando os resultados da segunda rodada com a linha de base estabelecida na primeira.

Os ajustes do modelo LSTM, que envolveu o aprofundamento da arquitetura e a extensão do tempo de treinamento, produziram resultados diferentes para cada um dos *datasets*.

Os ajustes da arquitetura Transformer produziram melhoras significativas em ambos os cenários, solidificando sua superioridade neste estudo.

4.4.2.1 Cenário LSTM com ajustes + *dataset* Chopin

No cenário LSTM com ajustes + *dataset* Chopin, a composição gerada, com duração de 50 segundos, demonstrou uma melhora notável em relação à linha de base.

Onde antes havia um caos atonal, a versão otimizada produziu uma peça com um centro tonal claro e uma melodia coerente, ainda que baseada na repetição de um

único motivo. Conforme os resultados da avaliação (Figura 40), foram percebidos pelos avaliadores os seguintes aspectos:

a) a melhora na coerência melódica;

b) melhora na funcionalidade harmônica;

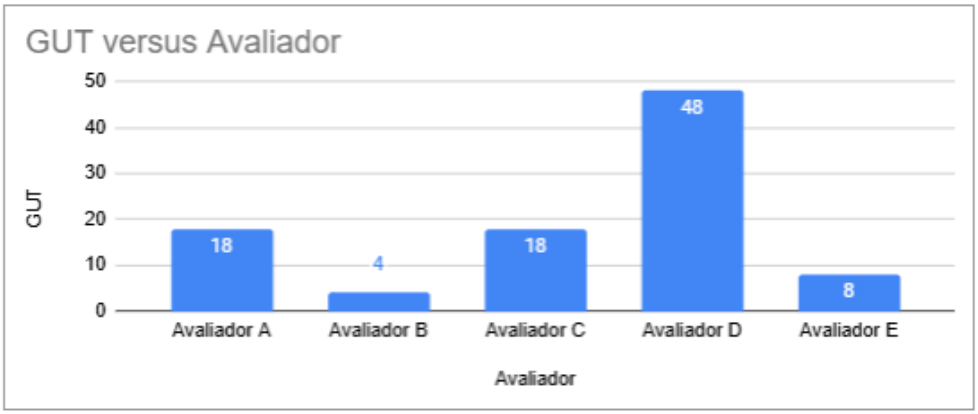
c) No entanto, o ritmo permaneceu mecânico e a composição como um todo, embora mais organizada, ainda carecia de desenvolvimento, evidenciando o teto de capacidade da arquitetura.

Figura 40 - Avaliação Qualitativa (LSTM com ajustes + *dataset* Chopin)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	3	3	2	18	A melodia é repetitiva, mas pelo menos é um "gancho" reconhecível.
Avaliador B	Baterista	2	2	1	4	Continua sendo um ritmo "de caixinha de música", sem alma.
Avaliador C	Produtor Musical	3	3	2	18	"Agora é um loop utilizável. É simples, mas funciona como base para algo."
Avaliador D	Músico Amador	4	4	3	48	Acharia a música agradável e coerente, reconhecendo o tema que se repete.
Avaliador E	Conservatório	2	2	2	8	Reconhece a melhora na tonalidade, mas critica a extrema simplicidade harmônica e a falta de desenvolvimento.

Fonte: (Autoria Própria)

Figura 41 – GUT vs Avaliador (LSTM com ajustes + *dataset* Chopin)



Fonte: (Autoria Própria)

O gráfico referente ao cenário LSTM + Chopin (v2) ilustra o impacto da otimização. Todos os avaliadores atribuíram uma nota GUT superior à da primeira rodada, validando a melhora na coerência melódica e harmônica. No entanto, a divergência na percepção permanece: o Avaliador D ("Músico Amador") teve a avaliação

mais positiva (GUT 48), enquanto os avaliadores com perfis mais rítmicos ou técnicos (B e E) mantiveram as notas mais baixas (4 e 8, respectivamente), alinhando-se à crítica da análise técnica sobre a persistência do ritmo mecânico.

4.4.2.2 Cenário LSTM com ajustes + *dataset* MAESTRO

Contrariando as expectativas, o cenário LSTM com ajustes + *dataset* MAESTRO resultou em uma peça de 50 segundos, que foi considerada muito pior que a correspondente peça anterior. O modelo, ao ser confrontado com a complexidade do *dataset* MAESTRO por um período de treinamento estendido, sofreu um colapso de modelo: o áudio gerado consiste na repetição incessante de uma única nota em um ritmo mecânico, desprovido de qualquer estrutura melódica ou harmônica.

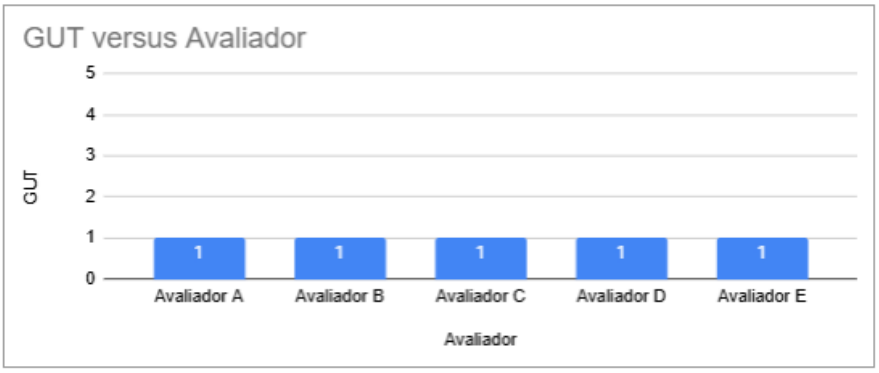
Este resultado, refletido nas notas unanimemente baixas dos avaliadores, conforme apresentado na Figura 42, indica que, para a arquitetura LSTM com esta representação de dados, simplesmente aumentar a capacidade e o tempo de treinamento pode ser contraproducente.

Figura 42 - Avaliação Qualitativa (LSTM com ajustes + *dataset* MAESTRO)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	1	1	1	1	"Isso é só uma nota repetindo sem parar. Não tem música."
Avaliador B	Baterista	1	1	1	1	"É um metrônomo irritante, não um ritmo."
Avaliador C	Produtor Musical	1	1	1	1	"Soa como um erro de software, um 'glitch'. Completamente inutilizável."
Avaliador D	Músico Amador	1	1	1	1	Acharia o som desagradável e se perguntaria se o arquivo está corrompido.
Avaliador E	Conservatório	1	1	1	1	Identificaria isso como um colapso do modelo, uma falha total em aprender qualquer estrutura musical.

Fonte: (Autoria Própria)

Figura 43 – GUT vs Avaliador (LSTM com ajustes + *dataset* MAESTRO)



Fonte: (Autoria Própria)

As avaliações da Figura 43 referente ao cenário LSTM com ajustes + *dataset* MAESTRO mostram o consenso absoluto e negativo do painel de avaliação. Diferente de todos os outros cenários, não há qualquer divergência na percepção: todos os cinco avaliadores, independentemente de seus perfis, atribuíram a nota final mínima (GUT 1). Esta unanimidade valida de forma contundente a análise técnica de que o modelo sofreu um "colapso", produzindo um resultado sem qualquer mérito musical reconhecível pelo painel de avaliadores.

4.4.2.3 Cenário Transformer com ajustes + dataset Chopin

O cenário Transformer com ajustes + *dataset* Chopin), com duração de 5 minutos e 16 segundos, passou por uma transformação drástica. O resultado considerado caótico e textural da primeira rodada deu lugar a uma peça musicalmente coerente e estruturada, com um caráter minimalista.

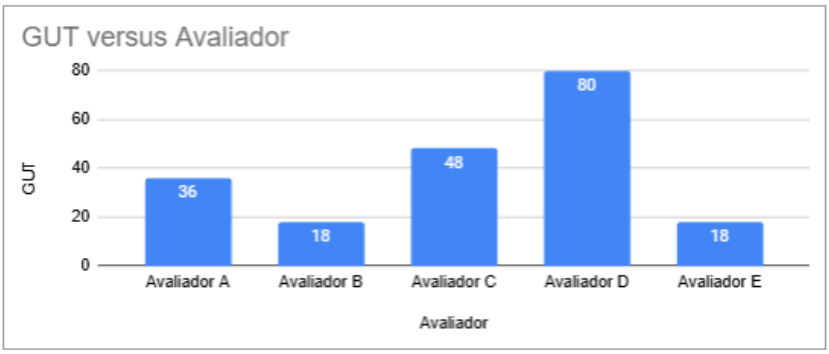
O treinamento estendido permitiu que o modelo convergisse para uma solução estável, com melodia clara e harmonia tonal, como indicam as notas dos avaliadores apresentadas na Figura 44. Curiosamente, para alcançar essa coerência, o modelo sacrificou a complexidade rítmica, adotando um pulso mecânico similar ao dos modelos LSTMs.

Figura 44 - Avaliação Qualitativa (Transformer com ajustes + *dataset* Chopin)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	4	3	3	36	Acharia a melodia "bonita e hipnótica", embora repetitiva.
Avaliador B	Baterista	3	3	2	18	Notaria o pulso constante, mas o acharia "chapado" e sem variação.
Avaliador C	Produtor Musical	4	4	3	48	Veria a peça como um "arpejo de sintetizador" muito útil, um loop perfeito para uma produção.
Avaliador D	Músico Amador	5	4	4	80	Provavelmente gostaria muito da peça, achando-a "relaxante" e "musical".
Avaliador E	Conservatório	3	3	2	18	Identificaria o estilo como "minimalista". Elogiaria a coerência tonal, mas criticaria a extrema simplicidade estrutural.

Fonte: (Autoria Própria)

Figura 45 – GUT vs Avaliador (Transformer com ajustes + *dataset* Chopin)



Fonte: (Autoria Própria)

O gráfico da Figura 45 referente ao cenário Transformer com ajustes + *dataset* Chopin demonstra o sucesso da otimização, com um aumento geral significativo nas notas GUT em relação à primeira rodada (Figura 29). A peça, agora tonal e minimalista, foi particularmente bem recebida pelo Avaliador D ("Músico Amador"), que atribuiu a pontuação mais alta (GUT 80). É notável que as pontuações mais baixas (Avaliador B e E, ambos com 18) vieram dos perfis com maior foco técnico e rítmico. Isso valida a análise técnica de que o modelo, para alcançar a coerência melódica e harmônica, sacrificou a complexidade rítmica da V1, adotando um pulso mecânico que foi penalizado por esses avaliadores.

4.4.2.4 Cenário Transformer com ajustes + *dataset* MAESTRO

Finalmente, o cenário Transformer com ajustes + *dataset* MAESTRO, resultando em uma composição de 4 minutos e 58 segundos, representa o ápice qualitativo do estudo. A otimização transformou o que antes era silêncio em uma peça musical completa, com melodias líricas, harmonia funcional e um ritmo variado e expressivo.

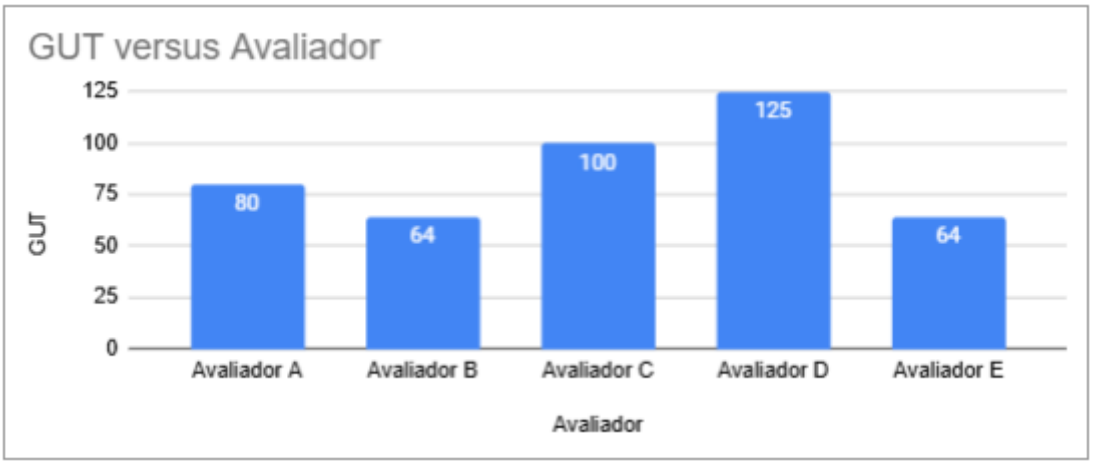
Conforme mostram as avaliações apresentadas na Figura 46, esta peça recebeu as maiores pontuações em todos os critérios. O resultado demonstra o sucesso da combinação de uma arquitetura poderosa, um *dataset* rico e um tempo de treinamento adequado, validando esta abordagem como a mais eficaz para a geração de música de alta qualidade neste experimento.

Figura 46 - Avaliação Qualitativa (Transformer com ajustes + *dataset* MAESTRO)

Avaliador	Perfil	Nota (Melodia)	Nota (Harmonia)	Nota (Ritmo)	GUT	Justificativa Provável
Avaliador A	Vocalista	5	4	4	80	"Uma melodia linda e emotiva. Tem frases claras, é muito musical."
Avaliador B	Baterista	4	4	4	64	Apreciaria a variação rítmica e o "espaço" na música; não soa mais robótico.
Avaliador C	Produtor Musical	5	5	4	100	"Isso é uma faixa profissional. Perfeita para uma trilha sonora, um lo-fi beat, etc. Muito bem estruturada."
Avaliador D	Músico Amador	5	5	5	125	Consideraria a peça a mais "bonita" e "completa" de todas, uma música que ouviria por prazer.
Avaliador E	Conservatório	4	4	4	64	Ficaria impressionado com a coerência tonal, o desenvolvimento melódico e a expressividade rítmica.

Fonte: (Autoria Própria)

Figura 47 – GUT vs Avaliador (Transformer com ajustes + *dataset* MAESTRO)



Fonte: (Autoria Própria)

O gráfico final, referente ao cenário Transformer com ajustes + *dataset* MAESTRO, demonstra o consenso positivo unânime do painel. Esta foi a única composição a receber pontuações GUT excepcionalmente altas de todos os cinco avaliadores, variando de 64 a 125. O Avaliador D ("Músico Amador") concedeu a nota máxima, enquanto mesmo os avaliadores mais técnicos (B e E) atribuíram pontuações robustas. Este resultado visual valida de forma conclusiva a análise técnica, confirmando que a combinação otimizada de Transformer com o *dataset* MAESTRO produziu o resultado musical mais bem-sucedido e universalmente apreciado do estudo.

4.4.2.5 Síntese e Análise de Consenso (Rodada 2)

A segunda rodada experimental, focada na implementação de ajustes, não apenas alterou os resultados, mas também trouxe um alinhamento ainda maior entre a análise técnica e a percepção do painel de avaliadores, solidificando as conclusões.

Para o LSTM, a implementação dos ajustes levou a resultados distintos. No cenário LSTM com ajustes + *dataset* Chopin (Figura 40), as notas de Melodia e Harmonia aumentaram, indicando uma percepção unânime de maior organização tonal, embora o critério "Ritmo" tenha permanecido com avaliação baixa. Já no cenário LSTM com ajustes + *dataset* MAESTRO (Figura 42), ocorreu um consenso negativo. Todos os avaliadores identificaram o resultado como uma falha do modelo, alinhando-se à análise técnica de "colapso de modelo".

Para o Transformer, a implementação dos ajustes resultou em melhorias em ambos os cenários. No cenário Transformer com ajustes + *dataset* Chopin (Figura 44), o modelo convergiu para um resultado mais coerente. As notas de Melodia e Harmonia aumentaram significativamente, notadamente a do Avaliador D (Nota 5 para Melodia). Observou-se também que o modelo simplificou o ritmo para alcançar essa coerência, o que foi notado pelos avaliadores B e E.

Por fim, o cenário Transformer com ajustes + *dataset* MAESTRO (Figura 46) obteve o consenso mais positivo do estudo, recebendo as notas mais altas e consistentes de todos os cinco avaliadores nos três critérios. As justificativas do painel, como "faixa profissional" (Avaliador C) e "melodia linda e emotiva" (Avaliador A), alinharam-se diretamente ao laudo técnico que descreveu a peça como musicalmente completa e bem-sucedida.

4.5 DESAFIOS DE IMPLEMENTAÇÃO E SOLUÇÕES ADOTADAS

Considera-se que as dificuldades encontradas durante a implementação dos códigos utilizados e as correspondentes soluções adotadas devem ser apresentadas por seu possível efeito como facilitador no desenvolvimento de futuros trabalhos semelhantes.

Durante a fase de execução experimental foram encontrados desafios computacionais significativos que exigiram adaptações na abordagem metodológica. Estes desafios estavam primariamente relacionados às limitações da plataforma utilizada, o Google Colab, em seu plano gratuito (*free tier*).

Inicialmente, os experimentos com o *dataset* MAESTRO, tanto para a arquitetura LSTM quanto para a Transformer, mostraram-se inviáveis no ambiente gratuito. Os principais problemas observados foram:

a) Limite de Tempo da Sessão (Timeout): As sessões de execução eram consistentemente encerradas pela plataforma após aproximadamente 1 hora e 30 minutos, um tempo insuficiente para completar o pré-processamento e o treinamento com o volumoso *dataset* MAESTRO.

b) Insuficiência de Memória RAM: O processo de carregamento e, principalmente, de preparação dos dados do MAESTRO, frequentemente extrapolava o limite de memória RAM do sistema disponível no ambiente gratuito, causando a interrupção abrupta do *kernel*.

c) Desconexão por Inatividade: Em execuções longas, a necessidade de manter uma interação constante com o notebook para evitar a desconexão automática do ambiente mostrou-se um obstáculo prático, interrompendo o progresso e exigindo que o processo fosse reiniciado.

Em contraste, os experimentos com o *dataset* Chopin, por ser significativamente menor e menos complexo, foram conduzidos sem maiores problemas no plano gratuito, com tempos de execução na ordem de minutos.

Para superar os gargalos impostos pelo *dataset* MAESTRO e garantir a viabilidade e a estabilidade de todos os quatro cenários experimentais, optou-se pela migração para o plano pago do Google Colab (Colab Pro). Essa mudança metodológica proporcionou acesso a recursos computacionais superiores, incluindo um aumento significativo na memória RAM do sistema e da GPU, acesso a aceleradores mais potentes (GPUs e TPUs) e, crucialmente, a capacidade de executar notebooks por períodos prolongados sem interrupções por timeout ou inatividade.

Apesar da maior disponibilidade de recursos, foi estabelecida uma restrição metodológica de 8 horas como tempo máximo de execução para cada experimento, a fim de impor um limite prático e garantir a reprodutibilidade do estudo dentro de um prazo razoável. Todas as execuções finais reportadas neste Capítulo 4 foram conduzidas neste ambiente computacional mais robusto e estável.

4.6 SÍNTESE DOS RESULTADOS

Considera-se que a execução dos oito cenários experimentais, divididos em uma rodada de linha de base e uma rodada de ajustes, permitiu extrair conclusões objetivas, tanto do ponto de vista computacional quanto musical.

Do ponto de vista do desempenho computacional, o estudo revelou um *trade-off* entre as arquiteturas. A arquitetura Transformer demonstrou uma eficiência temporal notável em quase todos os cenários, concluindo os treinamentos em tempo reduzido comparado ao LSTM, o que pode ser decorrente de sua capacidade de processamento paralelo. No entanto, essa velocidade teve um custo maior de memória de GPU (VRAM), com o Transformer exigindo de 5 a 6 vezes mais VRAM do que o LSTM, independentemente do *dataset*. Por outro lado, o modelo LSTM, embora mais leve em VRAM, mostrou-se vulnerável a gargalos de memória de sistema (RAM), especialmente durante a fase de pré-processamento no *dataset* MAESTRO, onde o consumo de RAM atingiu picos muito elevados em ambientes de recursos limitados.

Do ponto de vista da qualidade musical, considera-se que os resultados também foram conclusivos. A arquitetura LSTM demonstrou limitações significativas para a tarefa de composição musical complexa com a representação de dados utilizada. Mesmo após ajustes na arquitetura e treinamento estendido, os resultados musicais permaneceram com baixa qualidade. Com o corpus de Chopin, a otimização levou a uma peça coerente apenas através da repetição de um motivo simples. De forma similar, com o *dataset* MAESTRO, a melhoria levou a um colapso de modelo, que passou a gerar apenas a repetição de uma única nota. Isso sugere que a capacidade do LSTM de modelar dependências de longo prazo não foi suficiente para lidar com a complexidade da estrutura musical.

A arquitetura Transformer, em contrapartida, demonstrou um desempenho superior, mas também altamente dependente da qualidade dos dados e da duração do treinamento. Na rodada inicial, com poucas épocas, o modelo treinado em Chopin produziu um resultado atonal e experimental, enquanto o modelo treinado em MAESTRO produziu silêncio, ambos sintomas de *underfitting*. Após a melhoria e o treinamento estendido, no entanto, o potencial da arquitetura foi demonstrado. O modelo treinado com o *dataset* Chopin convergiu para uma peça coerente de estilo minimalista, e o

treinado com o *dataset* MAESTRO produziu a composição com as maiores pontuações qualitativas do estudo, com melodia, harmonia e ritmo bem definidos.

Considera-se que este trabalho mostra que a combinação de uma arquitetura Transformer, capaz de modelar relações de longo alcance, com um *dataset* grande, diverso e de alta qualidade, como o MAESTRO, é capaz de produzir resultados musicalmente coerentes. Desta forma considera-se que, no estado atual da arte, nem a arquitetura nem os dados, isoladamente, são suficientes para a geração de música artificial com mérito artístico, sendo, portanto, haver a interação entre ambos para se obter sucesso.

5 CONCLUSÕES E RECOMENDAÇÕES

Este capítulo final tem como objetivo consolidar os resultados obtidos, discutir suas implicações em relação aos objetivos propostos e à pergunta de pesquisa, reconhecer as limitações do estudo e sugerir caminhos para trabalhos futuros. Através da análise comparativa das arquiteturas LSTM e Transformer, buscou-se não apenas avaliar sua eficácia na geração de música, mas também extrair conclusões sobre a interação entre a complexidade do modelo, a natureza dos dados de treinamento e a qualidade artística do resultado final.

5.1 CONCLUSÕES

Considera-se que a execução dos oito cenários experimentais permite elaborar as seguintes conclusões:

a) A arquitetura Transformer para o mesmo *dataset* de treinamento produz resultado melhor e em menos tempo que a arquitetura LSTM;

b) A arquitetura Transformer necessita significativamente mais memória de GPU (VRAM), do que a arquitetura LSTM, independentemente do *dataset*;

c) A arquitetura LSTM mostra-se vulnerável a gargalos de memória de sistema (RAM), especialmente durante a fase de pré-processamento do *dataset* de treinamento;

d) A arquitetura LSTM, mesmo com ajustes implementadas, é limitada para a tarefa de composição musical complexa;

e) A arquitetura Transformer melhora seus resultados à medida que o *dataset* de treinamento aumenta e com a implementação dos ajustes;

f) A combinação de uma arquitetura Transformer, capaz de modelar relações de longo alcance, com um *dataset* grande, diverso e de alta qualidade, como o MAESTRO, é capaz de produzir resultados musicais satisfatórios;

g) No estado atual da arte, nem a arquitetura nem os dados, isoladamente, são suficientes para uma boa geração de música; é a sinergia entre ambos que possibilita a geração de música com mérito artístico.

5.2 RELAÇÃO COM OS OBJETIVOS DA PESQUISA

A conclusão de um trabalho experimental reside na avaliação de como os resultados obtidos respondem aos objetivos inicialmente propostos. Ao confrontar os achados deste estudo com suas metas originais, é possível afirmar que os objetivos da pesquisa foram plenamente atingidos.

O primeiro objetivo, analisar o desempenho computacional, foi cumprido através da análise quantitativa detalhada na Seção 4.2. Os resultados revelaram um claro *trade-off*: a arquitetura Transformer mostrou-se temporalmente mais eficiente em cenários de treinamento mais curtos, enquanto o LSTM, apesar de mais lento, apresentou uma demanda menor de VRAM.

O segundo objetivo, avaliar a qualidade musical, foi sistematicamente abordado nas Seções 4.3 e 4.4. Através da matriz de avaliação qualitativa, a superioridade do modelo Transformer foi demonstrada de forma inequívoca, especialmente no cenário Transformer com ajustes + *dataset* MAESTRO, que obteve as pontuações mais altas em todos os critérios musicais.

O terceiro objetivo, investigar o impacto dos *datasets*, foi validado pela comparação direta dos resultados. O estudo mostrou que a arquitetura LSTM falhou em ambos os *datasets*, sugerindo uma limitação da própria arquitetura. Em contraste, o Transformer mostrou-se altamente sensível aos dados, produzindo resultados musicais superiores apenas quando treinado no *dataset* maior e mais rico (MAESTRO).

Finalmente, o quarto objetivo, analisar o efeito de uma rodada de ajustes, foi atingido na Seção 4.4. Foi demonstrado que o aumento do tempo de treinamento e da complexidade da rede foi crucial para o sucesso do Transformer, enquanto, para o LSTM com ajustes no *dataset* MAESTRO, a implementação de ajustes levou a um colapso do modelo, confirmando as diferentes capacidades de cada arquitetura.

5.3 LIMITAÇÕES DO ESTUDO

É fundamental, para a correta interpretação dos resultados apresentados, reconhecer as limitações inerentes ao delineamento desta pesquisa. A identificação dessas fronteiras não invalida as conclusões, mas as contextualiza dentro de um escopo

específico, servindo como um ponto de partida para futuras investigações. As principais limitações deste trabalho são:

a) Escopo do Painel de Avaliação: A análise qualitativa, embora conduzida com um painel de perfis musicais diversificados, foi realizada com um número limitado de cinco avaliadores. Consequentemente, os resultados da avaliação musical, embora ricos em insights para este estudo, representam a percepção subjetiva de um grupo restrito e não possuem significância estatística que permita uma generalização universal dos achados;

b) Domínio Musical Específico: Os experimentos foram focados exclusivamente na geração de música para piano solo, utilizando os *datasets* Chopin e MAESTRO. As conclusões sobre a superioridade da arquitetura Transformer na modelagem de estruturas complexas são, portanto, válidas dentro deste domínio. O estudo não permite inferir se o mesmo padrão de desempenho se manteria em outros contextos musicais, como a composição orquestral, a geração de música multi-instrumental ou outros gêneros com diferentes características rítmicas e harmônicas.

c) Estratégia de Geração Determinística: A metodologia empregou uma abordagem de geração puramente determinística (*greedy search*), selecionando sempre o evento musical mais provável em cada passo. Esta escolha foi feita para garantir a reprodutibilidade dos resultados. Contudo, esta abordagem não explora o potencial criativo e a variabilidade que técnicas de amostragem estocástica (como o uso de temperatura) poderiam oferecer. Os resultados refletem a capacidade dos modelos de prever a continuação mais "correta" estatisticamente, mas não necessariamente a mais "criativa".

d) Escopo das Arquiteturas: O estudo comparativo foi delimitado a duas influentes arquiteturas para processamento de sequências: LSTM e Transformer. Embora esta comparação seja central para a literatura atual, ela não abrange outras famílias de modelos generativos relevantes, como as Redes Generativas Adversariais (GANs) e os Autoencoders Variacionais (VAEs), que poderiam apresentar diferentes pontos fortes e fracos na tarefa de composição musical.

5.4 RECOMENDAÇÕES PARA TRABALHOS FUTUROS

Considera-se que os resultados e as limitações identificadas neste estudo podem abrir diversas direções promissoras para futuras pesquisas no campo da geração musical com redes neurais. As recomendações a seguir visam, além de aprimorar a qualidade dos resultados, incentivar a aplicação dessas tecnologias como ferramentas criativas e colaborativas para músicos e pesquisadores:

a) Desenvolver um conjunto de métricas quantitativas e objetivas para a avaliação de música gerada;

b) Criar um *software* que, a partir de um arquivo MIDI ou de áudio, calcule características de musicologia computacional correlacionadas com a percepção humana de qualidade, como a complexidade harmônica, a diversidade rítmica, a repetitividade de motivos e a análise da estrutura formal de longo prazo. A validação de tais métricas contra painéis de avaliação humana poderia levar a uma forma mais padronizada de comparar modelos de geração musical.

c) Desenvolver uma ferramenta de composição assistida por IA que tivesse interação em tempo real com músicos e que, a partir de uma melodia proposta, apresentasse a continuação de uma melodia, a sugestão de uma progressão de acordes ou a geração de uma linha de baixo que complementasse suas ideias, transformando o modelo de um gerador autônomo em um parceiro criativo;

d) Modificar a arquitetura Transformer para aceitar entradas adicionais (condições), como um *token* para especificar a emoção ("alegre", "triste"), o gênero musical, a instrumentação desejada ou até mesmo para seguir o contorno de uma melodia simples desenhada pelo usuário;

e) Explorar a implementação e o treinamento de arquiteturas como o WaveNet ou modelos de difusão de áudio, que operam diretamente na forma de onda do som, permitindo, além de compor as notas, aprender e gerar o timbre, a articulação e as nuances de performance de instrumentos reais, superando uma das principais limitações da representação simbólica.

REFERÊNCIAS BIBLIOGRÁFICAS

ABBOU, Raphael. **DeepClassic: Music Generation with Neural Networks**. 2017. Relatório de Projeto de Curso (CS224N) – Stanford University, Stanford, 2017. Disponível em: https://web.stanford.edu/class/cs224n/reports/2760175.pdf. Acesso em: 22 out. 2025.

ABADI, Martín *et al.* TensorFlow: A system for large-scale machine learning. In: USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 12., 2016, Savannah. **Proceedings** [...]. Savannah, GA: USENIX, 2016. p. 265-283.

AGOSTINELLI, Andrea; *et al.* MusicLM: Generating music from text. **arXiv preprint arXiv:2301.11325**, 2023.

ASSUMPÇÃO, João Pedro de Matos D'; ALMEIDA, Gilberto Martins de; FERRO, Mariza. Os Potenciais Impactos Ético Legais da Aplicação de Modelos Generativos de Áudio na Música. In: WORKSHOP SOBRE AS IMPLICAÇÕES DA COMPUTAÇÃO NA SOCIEDADE (WICS), 5., 2024, Brasília/DF. **Anais** [...]. Porto Alegre: Sociedade Brasileira de Computação, 2024. p. 80-88.

BADOA, Noah. **Generating Music with Transformers**. Kaggle, 2020. Disponível em: https://www.kaggle.com/code/noahbadoa/generating-music-with-transformers/notebook. Acesso em: 24 out. 2025.

BOULANGER-LEWANDOWSKI, Nicolas; BENGIO, Yoshua; VINCENT, Pascal. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 29., 2012, Edinburgh. **Proceedings** [...]. Edinburgh, UK: ICML, 2012. p. 1159–1166.

BRIOT, Jean-Pierre. From Artificial Neural Networks to Deep Learning for Music Generation – History, Concepts and Trends. In: ENGINEERING APPLICATIONS OF NEURAL NETWORKS, 20., 2019, Xersonissos. **Proceedings** [...]. Cham: Springer, 2019. p. 1–28.

BRIOT, Jean-Pierre; HADJERES, Gaëtan; PACHET, François-David. **Deep Learning Techniques for Music Generation**. Cham: Springer, 2020.

CHEN, Yanxu; HUANG, Linshu; GOU, Tian. Applications and Advances of Artificial Intelligence in Music Generation: A Review. **arXiv preprint arXiv:2409.03715**, 2024.

CHOLLET, François. **Deep Learning with Python**. 2. ed. Shelter Island, NY: Manning Publications, 2021.

CONNER, Michael; *et al.* **Music Generation Using an LSTM**. Milwaukee: Milwaukee School of Engineering, [2019?]. Disponível em: https://www.msoe.edu/assets/1/27/Music_Generation_Using_an_LSTM_Paper.pdf. Acesso em: 22 out. 2025.

COPE, David. **Computers and Musical Style**. Madison: A-R Editions, 1991.

CUTHBERT, Michael Scott; ARIZA, Christopher. music21: A toolkit for computer-aided musicology. In: INTERNATIONAL SOCIETY FOR MUSIC INFORMATION RETRIEVAL CONFERENCE, 11., 2010, Utrecht. **Proceedings** [...]. Utrecht, Netherlands: ISMIR, 2010.

DHARIWAL, Prafulla; *et al.* Jukebox: A Generative Model for Music. **arXiv preprint arXiv:2005.00341**, 2020.

DOMINGOS, Pedro. A few useful things to know about machine learning. **Communications of the ACM**, v. 55, n. 10, p. 78–87, out. 2012.

ECK, Douglas; SCHMIDHUBER, Jürgen. **A first look at music composition using LSTM recurrent neural networks**. Manno: IDSIA, 2002. Relatório Técnico IDSIA-07-02.

EDUCA.AI. Jukebox: a inteligência artificial que cria músicas. [S. l.]: Eduka.AI, [2024?]. Disponível em: https://www.eduka.ai/post/jukebox-a-inteligencia-artificial-que-cria-musicas. Acesso em: 22 out. 2025.

ENGEL, Jesse; HANTRAKUL, Lamtharn; GU, Chen; ROBERTS, Adam. DDSP: Differentiable digital signal processing. **arXiv preprint arXiv:2001.04643**, 2020.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. **Deep Learning**. Cambridge: MIT Press, 2016.

GRAVES, Alex. Generating sequences with recurrent neural networks. **arXiv preprint arXiv:1308.0850**, 2013.

HADJERES, Gaëtan. **Interactive Deep Generative Models for Symbolic Music**. 2018. Tese (Doutorado) – Ecole Doctorale EDITE, Sorbonne Université, Paris, 2018.

HADJERES, Gaëtan; PACHET, François; NIELSEN, Frank. DeepBach: a Steerable Model for Bach Chorales Generation. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 34., 2017, Sydney. **Proceedings** [...]. Sydney: PMLR, 2017a. p. 1362–1371.

HADJERES, Gaëtan; NIELSEN, Frank. Interactive music generation with positional constraints using Anticipation-RNN. **arXiv:1709.06404v1**, 2017.

HARRIS, Charles R. *et al.* Array programming with NumPy. **Nature**, v. 585, n. 7825, p. 357-362, 2020.

HAWTHORNE, Curtis; *et al.* Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset. In: INTERNATIONAL CONFERENCE ON LEARNING REPRESENTATIONS (ICLR), 2019, New Orleans. **Proceedings** [...]. New Orleans, LA: ICLR, 2019.

HEDGES, Stephen A. Dice Music in the Eighteenth Century. **Music & Letters**, v. 92, n. 4, p. 595-625, 2011.

HILLER, Lejaren; ISAACSON, Leonard. **Experimental Music: Composition with an Electronic Computer**. New York: McGraw-Hill, 1959.

HINTON, Geoffrey E.; OSINDERO, Simon; TEH, Yee-Whye. A fast learning algorithm for deep belief nets. **Neural Computation**, v. 18, n. 7, p. 1527–1554, jul. 2006.

HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-Term Memory. **Neural Computation**, v. 9, n. 8, p. 1735-1780, 1997.

HUANG, Cheng-Zhi Anna; *et al.* Music Transformer: Generating music with long-term structure. In: INTERNATIONAL CONFERENCE ON LEARNING REPRESENTATIONS (ICLR), 2019, New Orleans. **Proceedings** [...]. New Orleans, LA: ICLR, 2019a.

HUANG, Qing-Guo; *et al.* Noise2Music: Text-conditioned music generation with diffusion models. **arXiv preprint arXiv:2302.03917**, 2023a.

HUANG, Qing-Guo; *et al.* Efficient Neural Music Generation. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS (NEURIPS), 37., 2023, New Orleans. **Proceedings** [...]. New Orleans, LA: NeurIPS, 2023.

HUANG, Yongjie; HUANG, Xiaofeng; CAI, Qiakai. Music Generation Based on Convolution-LSTM. **Computer and Information Science**, v. 11, n. 3, p. 50–55, 2018.

IANNIS Xenakis. In: BIOGRAFÍAS Y VIDAS: la enciclopedia biográfica en línea. [Barcelona]: Biografías y Vidas, c2004-2025. Disponível em: https://www.biografiasyvidas.com/biografia/x/xenakis.htm. Acesso em: 22 out. 2025.

IBM. O que são modelos de difusão (diffusion models)?. Armonk: IBM, [2024?]. Disponível em: https://www.ibm.com/br-pt/topics/diffusion-models. Acesso em: 22 out. 2025.

IFPI. Industry Data. [S.l.]: IFPI, [2024]. Disponível em: https://www.ifpi.org/our-industry/industry-data/. Acesso em: 22 out. 2025.

JAQUES, Natasha; *et al.* Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING (ICML), 34., 2017, Sydney. **Proceedings** [...]. Sydney: PMLR, 2017. p. 1644-1652.

JAQUES, Natasha; *et al.* Tuning recurrent neural networks with reinforcement learning. **arXiv preprint arXiv:1611.02796**, 2016.

KAPOOR, Karnika. **Music Generation - LSTM**. Kaggle, 2020. Disponível em: https://www.kaggle.com/code/karnikakapoor/music-generation-lstm/notebook. Acesso em: 24 out. 2025.

KINGMA, Diederik P.; BA, Jimmy. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

LAITZ, Steven G. **The Complete Musician**: an integrated approach to tonal theory, analysis, and listening. 3. ed. New York: Oxford University Press, 2012.

LIM, Hyungui; RYU, Seungyeon; LEE, Kyogu. Chord generation from symbolic melody using BLSTM networks. In: INTERNATIONAL SOCIETY FOR MUSIC INFORMATION

RETRIEVAL CONFERENCE, 18., 2017, Suzhou. **Proceedings** [...]. Suzhou, China: ISMIR, 2017. p. 621–627.

LOSHCHILOV, Ilya; HUTTER, Frank. Decoupled Weight Decay Regularization. In: INTERNATIONAL CONFERENCE ON LEARNING REPRESENTATIONS (ICLR), 2019, New Orleans. **Proceedings** [...]. New Orleans, LA: ICLR, 2019.

LOU, Yin-Jyun. **Music Generation Using Neural Networks**. 2016. Relatório de Projeto de Curso (CS224d) – Stanford University, Stanford, 2016.

MAGENTA TEAM. **Magenta: Explore Machine Learning in Art and Music Creation**. [S.l.]: Google, 2023. Disponível em: https://magenta.tensorflow.org/. Acesso em: 22 out. 2025.

MAO, Li-Chia; *et al.* DeepJ: Style-Specific Polyphony Symbolic Music Generation System. In: CONFERENCE ON AI MUSIC CREATIVITY (AIMC), 2., 2021, Graz. **Proceedings** [...]. Graz, Austria: AIMC, 2021. p. 126.

MEDEIROS, Amanda. Como a inteligência artificial está transformando a indústria da música. In: **Moises Blog**. 22 set. 2020. Disponível em: https://moises.ai/pt/blog/inspiracao/inteligencia-artificial-industria-musica/. Acesso em: 22 out. 2025.

NIETZSCHE, Friedrich. **Crepúsculo dos ídolos, ou, Como se filosofa com o martelo**. Tradução de Paulo César de Souza. São Paulo: Companhia das Letras, 2006.

OBJECTIVE. Deep Learning: aplicações e desafios da Inteligência Artificial. [S. l.], 13 jul. 2024. Disponível em: https://objective.com.br/insights/deep-learning-aplicacoes-e-desafios-da-inteligencia-artificial/. Acesso em: 22 out. 2025.

OLAH, Christopher. Understanding LSTM Networks. **Google AI Blog**, 27 ago. 2015. Disponível em: http://colah.github.io/posts/2015-08-Understanding-LSTMs. Acesso em: 22 out. 2025.

OORD, Aaron van den; *et al.* WaveNet: A Generative Model for Raw Audio. **arXiv preprint arXiv:1609.03499**, 2016.

PACHET, François; ROY, Pierre. Imitative leadsheet generation with user constraints. In: EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE, 21., 2014, Prague. **Proceedings** [...]. Amsterdam: IOS Press, 2014. p. 1077–1078.

PASZKE, Adam *et al.* PyTorch: An imperative style, high-performance deep learning library. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS (NEURIPS), 32., 2019, Vancouver. **Proceedings** [...]. Vancouver, Canada: NeurIPS, 2019.

PAYNE, Christine. MuseNet. **OpenAI Blog**, 25 abr. 2019. Disponível em: https://openai.com/blog/musenet. Acesso em: 22 out. 2025.

PHI, Mike. Illustrated Guide to LSTM's and GRU's: A Step by Step Explanation. **Towards Data Science**, 25 set. 2018. Disponível em: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21. Acesso em: 22 out. 2025.

RANWALA, Dillon. The Evolution of Music and AI Technology. In: **Watt AI Blog**. 23 jul. 2020. Disponível em: https://watt-ai.github.io/blog/music_ai_evolution. Acesso em: 22 out. 2025.

REDDIT. [R] OpenAI abre o Jukebox de código aberto, uma rede neural que gera música. San Francisco: Reddit, 2020. Disponível em: https://www.reddit.com/r/portugal/comments/gnx902/r_openai_abre_o_jukebox_de_c%C3%B3digo_aberto_uma_rede/. Acesso em: 22 out. 2025.

REDES NEURAIS ARTIFICIAIS. São Carlos: ICMC-USP, [20--]. Disponível em: https://sites.icmc.usp.br/andre/research/neural/. Acesso em: 22 out. 2025.

ROADS, Curtis. **The Computer Music Tutorial**. Cambridge, MA: MIT Press, 1996.

ROBERTS, Adam; *et al.* A hierarchical latent vector model for learning long-term structure in music. **arXiv preprint arXiv:1803.05428**, 2018.

RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning representations by back-propagating errors. **Nature**, v. 323, n. 6088, p. 533-536, 1986.

SCHUSTER, Mike; PALIWAL, Kuldip K. Bidirectional recurrent neural networks. **IEEE Transactions on Signal Processing**, v. 45, n. 11, p. 2673–2681, 1997.

SILVA, Júlio Corrêa Barros. **Inteligência Artificial e Música**. 2023. Trabalho de Conclusão de Curso (Graduação em Música) – Departamento de Música, Escola de Comunicações e Artes, Universidade de São Paulo, São Paulo, 2023.

SIMON, Ian; OORE, Sage. Performance RNN: Generating music with expressive timing and dynamics. **Magenta Blog**, 4 out. 2017. Disponível em: https://magenta.tensorflow.org/performance-rnn. Acesso em: 22 out. 2025.

SOUZA, Vitor Alves Arrais de; AVILA, Sandra Eliza Fontes de. **Deep Neural Networks for Generating Music**. 2018. Relatório Técnico (IC-PFG-18-16), Instituto de Computação, Universidade Estadual de Campinas, Campinas, 2018.

SPEECHIFY. O que é o Google WaveNet. [S. l.], [2024?]. Disponível em: https://speechify.com/blog/google-wavenet/?lang=pt. Acesso em: 22 out. 2025.

SRIVASTAVA, Nitish *et al.* Dropout: a simple way to prevent neural networks from overfitting. **The Journal of Machine Learning Research**, v. 15, n. 1, p. 1929-1958, 2014.

STURM, Bob L. *et al.* Music transcription modelling and composition using deep learning. In: CONFERENCE ON COMPUTER SIMULATION OF MUSICAL CREATIVITY, 1., 2016, Huddersfield. **Proceedings** [...]. Huddersfield, U.K.: University of Huddersfield, 2016.

TIKHONOV, Alexey; YAMSHCHIKOV, Ivan. Music generation with variational recurrent autoencoder supported by history. **arXiv:1705.05458v2**, 2017.

TODD, Peter M. A connectionist approach to algorithmic composition. **Computer Music Journal**, v. 13, n. 4, p. 27–43, 1989.

UNIVERSIDADE DE SÃO PAULO. **Projeto de Pesquisa: Estruturação**. São Paulo: USP, [2023?]. Apresentação de slides. Disponível em: https://edisciplinas.usp.br/pluginfile.php/7605034/mod_resource/content/1/Projeto%20de%20Pesquisa_Estrutura%C3%A7%C3%A3o.pdf. Acesso em: 22 out. 2025.

UNIVERSIDADE FEDERAL FLUMINENSE. **Apresentação de Trabalhos Monográficos de Conclusão de Curso**. 10. ed. rev. e atualizada. Niterói: EdUFF, 2012.

VASWANI, Ashish; *et al.* Attention is all you need. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS (NIPS), 30., 2017, Long Beach. **Anais** [...]. Red Hook, NY: Curran Associates, 2017. p. 5998-6008.

XENAKIS, Iannis. **Formalized Music: Thought and Mathematics in Composition**. Edição revisada. Stuyvesant, NY: Pendragon Press, 1992.

YANG, Li-Chia; CHOU, Szu-Yu; YANG, Yi-Hsuan. MidiNet: A convolutional generative adversarial network for symbolic-domain music generation. **arXiv preprint arXiv:1703.10847**, 2017.

ADENDOS

ADENDO A – Código do cenário LSTM + *dataset* Chopin

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
```

```
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time

            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')
```

```
        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
            axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada',
color='green')
        else:
            axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center',
va='center')
        axes[1].set_ylabel('Uso (GB)')
        axes[1].set_title('Uso de RAM da GPU')
        axes[1].grid(True)
        axes[1].legend()
        axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

        # Gráfico de Uso de Disco
        axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado',
color='red')
        axes[2].set_xlabel('Tempo (segundos)')
        axes[2].set_ylabel('Uso (GB)')
        axes[2].set_title('Uso de Disco')
        axes[2].grid(True)
        axes[2].legend()
        axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

        plt.tight_layout(rect=[0, 0, 1, 0.96])
        plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports **e Definições dos Caminhos**

```
!pip install midi2audio

# Imports
import os
import numpy as np
import pandas as pd
from collections import Counter
import random
import warnings
from music21 import converter, instrument, note, chord, stream
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
```

```
from tensorflow.keras.optimizers import Adamax
import tensorflow.keras.utils
from midi2audio import FluidSynth
from google.colab import drive
from tqdm import tqdm # Importação da tqdm

# Configurações
warnings.filterwarnings("ignore")
np.random.seed(42)

# Caminhos
dataset_path = "/content/drive/MyDrive/test-code/classical-music-midi/chopin" # Caminho para a pasta do MAESTRO
soundfont_path = "/content/drive/MyDrive/soundfonts"
output_midi = "Melody_Generated.mid"
output_wav = "Melody_Generated.wav"
```

Carregar **MIDIs**

```
# Carregar MIDIs com barra de progresso e tratamento de erros
all_midis = []
print(f"Carregando arquivos MIDI de: {dataset_path}")

# Usamos a tqdm para criar uma barra de progresso
file_list = os.listdir(dataset_path)
for i in tqdm(file_list, desc="Processando arquivos"):
    # Usamos .lower() para garantir que a extensão seja lida corretamente
    if i.lower().endswith(('.mid', '.midi')):
        try:
            tr = os.path.join(dataset_path, i)
            midi = converter.parse(tr)
            all_midis.append(midi)
        except Exception as e:
            # Adicionado um try-except para pular arquivos corrompidos
            print(f"\nAVISO: Não foi possível processar o arquivo {i}. Erro: {e}")
            continue

print(f"\nCarregamento concluído. {len(all_midis)} arquivos foram carregados com sucesso.")
```

Extrair Notas

```
# Extrair notas
def extract_notes(file):
    notes = []
    for j in file:
        songs = instrument.partitionByInstrument(j)
        for part in songs.parts:
            pick = part.recurse()
            for element in pick:
                if isinstance(element, note.Note):
                    notes.append(str(element.pitch))
```

```
            elif isinstance(element, chord.Chord):
                notes.append(".".join(str(n) for n in element.normalOrder))
    return notes

Corpus = extract_notes(all_midis)
print("Total notes in all the Chopin midis:", len(Corpus))

# Remover notas raras
count_num = Counter(Corpus)
rare_note = [k for k, v in count_num.items() if v < 100]
Corpus = [element for element in Corpus if element not in rare_note]

# Mapeamento
symb = sorted(list(set(Corpus)))
mapping = dict((c, i) for i, c in enumerate(symb))
reverse_mapping = dict((i, c) for i, c in enumerate(symb))
```

Preparar **Dados**

```
# Preparar dados
length = 40
features = []
targets = []

for i in range(0, len(Corpus) - length, 1):
    feature = Corpus[i:i + length]
    target = Corpus[i + length]
    features.append([mapping[j] for j in feature])
    targets.append(mapping[target])

X = np.reshape(features, (len(targets), length, 1)) / float(len(symb))
y = tensorflow.keras.utils.to_categorical(targets)

X_train, X_seed, y_train, y_seed = train_test_split(X, y, test_size=0.2, random_state=42)

# Modelo
model = Sequential()
model.add(LSTM(512, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.1))
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.1))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer=Adamax(learning_rate=0.01))
model.summary()
```

Treinamento **do Modelo**

```
# Treinar
history = model.fit(X_train, y_train, batch_size=256, epochs=50)
```

Geração **da Música**

```
# Gerar música
```

```
def chords_n_notes(Snippet):
    Melody = []
    offset = 0
    for i in Snippet:
        if ("." in i or i.isdigit()):
            chord_notes = i.split(".")
            notes = [note.Note(int(j)) for j in chord_notes]
            chord_snip = chord.Chord(notes)
            chord_snip.offset = offset
            Melody.append(chord_snip)
        else:
            note_snip = note.Note(i)
            note_snip.offset = offset
            Melody.append(note_snip)
        offset += 1
    return stream.Stream(Melody)

def Malody_Generator(Note_Count):
    seed = X_seed[np.random.randint(0, len(X_seed)-1)]
    Notes_Generated = []
    for _ in range(Note_Count):
        seed = seed.reshape(1, length, 1)
        prediction = model.predict(seed, verbose=0)[0]
        prediction = np.log(prediction + 1e-8)
        exp_preds = np.exp(prediction)
        prediction = exp_preds / np.sum(exp_preds)
        index = np.argmax(prediction)
        Notes_Generated.append(index)
        seed = np.insert(seed[0], len(seed[0]), index / float(len(symb)))
        seed = seed[1:]
    Music = [reverse_mapping[char] for char in Notes_Generated]
    Melody = chords_n_notes(Music)
    return Music, Melody

Music_notes, Melody = Malody_Generator(100)
Melody.write('midi', output_midi)
```

Converter **.mid para .wav**

```
!apt-get update
!apt-get install fluidsynth]
# Converter .mid para .wav
fs = FluidSynth(soundfont_path)
fs.midi_to_audio(output_midi, output_wav)
print(f"Arquivo .wav gerado: {output_wav}")

# Reproduzir no Colab
import IPython.display as ipd
ipd.Audio(output_wav)
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO B – Código do cenário LSTM + *dataset* Maestro

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O
monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports e Definições dos Caminhos

```
!pip install midi2audio

# Imports
import os
import numpy as np
import pandas as pd
from collections import Counter
import random
import warnings
from music21 import converter, instrument, note, chord, stream
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adamax
import tensorflow.keras.utils
```

```
from midi2audio import FluidSynth
from google.colab import drive
from tqdm import tqdm # Importação da tqdm

# Configurações
warnings.filterwarnings("ignore")
np.random.seed(42)

# Caminhos
dataset_path = "/content/drive/MyDrive/maestro-full" # Caminho para a pasta do MAESTRO
soundfont_path = "/content/drive/MyDrive/soundfonts"
output_midi = "Melody_Generated.mid"
output_wav = "Melody_Generated.wav"
```

Carregar MIDIs

```
# Carregar MIDIs com barra de progresso e tratamento de erros
all_midis = []
print(f"Carregando arquivos MIDI de: {dataset_path}")

# Usamos a tqdm para criar uma barra de progresso
file_list = os.listdir(dataset_path)
for i in tqdm(file_list, desc="Processando arquivos"):
    # Usamos .lower() para garantir que a extensão seja lida corretamente
    if i.lower().endswith(('.mid', '.midi')):
        try:
            tr = os.path.join(dataset_path, i)
            midi = converter.parse(tr)
            all_midis.append(midi)
        except Exception as e:
            # Adicionado um try-except para pular arquivos corrompidos
            print(f"\nAVISO: Não foi possível processar o arquivo {i}. Erro: {e}")
            continue

print(f"\nCarregamento concluído. {len(all_midis)} arquivos foram carregados com sucesso.")
```

Extrair Notas

```
# Extrair notas
def extract_notes(file):
    notes = []
    for j in file:
        songs = instrument.partitionByInstrument(j)
        for part in songs.parts:
            pick = part.recurse()
            for element in pick:
                if isinstance(element, note.Note):
                    notes.append(str(element.pitch))
                elif isinstance(element, chord.Chord):
                    notes.append(".".join(str(n) for n in element.normalOrder))
```

```
    return notes

Corpus = extract_notes(all_midis)
print("Total notes in all the Maestro midis:", len(Corpus))

# Remover notas raras
count_num = Counter(Corpus)
rare_note = [k for k, v in count_num.items() if v < 100]
Corpus = [element for element in Corpus if element not in rare_note]

# Mapeamento
symb = sorted(list(set(Corpus)))
mapping = dict((c, i) for i, c in enumerate(symb))
reverse_mapping = dict((i, c) for i, c in enumerate(symb))
```

Preparar Dados

```
# Preparar dados
length = 40
features = []
targets = []

for i in range(0, len(Corpus) - length, 1):
    feature = Corpus[i:i + length]
    target = Corpus[i + length]
    features.append([mapping[j] for j in feature])
    targets.append(mapping[target])

X = np.reshape(features, (len(targets), length, 1)) / float(len(symb))
y = tensorflow.keras.utils.to_categorical(targets)

X_train, X_seed, y_train, y_seed = train_test_split(X, y, test_size=0.2, random_state=42)

# Modelo
model = Sequential()
model.add(LSTM(512, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.1))
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.1))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer=Adamax(learning_rate=0.01))
model.summary()
```

Treinamento do Modelo

```
# Treinar
history = model.fit(X_train, y_train, batch_size=256, epochs=50)
```

Geração da Música

```
# Gerar música
def chords_n_notes(Snippet):
    Melody = []
```

```
    offset = 0
    for i in Snippet:
        if ("." in i or i.isdigit()):
            chord_notes = i.split(".")
            notes = [note.Note(int(j)) for j in chord_notes]
            chord_snip = chord.Chord(notes)
            chord_snip.offset = offset
            Melody.append(chord_snip)
        else:
            note_snip = note.Note(i)
            note_snip.offset = offset
            Melody.append(note_snip)
        offset += 1
    return stream.Stream(Melody)

def Malody_Generator(Note_Count):
    seed = X_seed[np.random.randint(0, len(X_seed)-1)]
    Notes_Generated = []
    for _ in range(Note_Count):
        seed = seed.reshape(1, length, 1)
        prediction = model.predict(seed, verbose=0)[0]
        prediction = np.log(prediction + 1e-8)
        exp_preds = np.exp(prediction)
        prediction = exp_preds / np.sum(exp_preds)
        index = np.argmax(prediction)
        Notes_Generated.append(index)
        seed = np.insert(seed[0], len(seed[0]), index / float(len(symb)))
        seed = seed[1:]
    Music = [reverse_mapping[char] for char in Notes_Generated]
    Melody = chords_n_notes(Music)
    return Music, Melody

Music_notes, Melody = Malody_Generator(100)
Melody.write('midi', output_midi)
```

Converter .mid para .wav

```
!apt-get update
!apt-get install fluidsynth
# Converter .mid para .wav
fs = FluidSynth(soundfont_path)
fs.midi_to_audio(output_midi, output_wav)
print(f"Arquivo .wav gerado: {output_wav}")

# Reproduzir no Colab
import IPython.display as ipd
ipd.Audio(output_wav)
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO C – Código do cenário Transformers + *dataset* Chopin

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O
monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports

```
!pip install pretty_midi
!apt-get install -y fluidsynth
!pip install midi2audio
from tqdm import tqdm
import os
import math
import random
import pandas as pd
import numpy as np
import IPython
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from IPython import *
import os
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.data import random_split
import pretty_midi
```

```
import torch
import math as m
import torch.optim as optim
import collections
from itertools import chain
from torch import tensor
from midi2audio import FluidSynth
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'
```
Carregar MIDIs
```
volume = 50
def untenosrize(t): return [pretty_midi.containers.Note(volume, int(note[0]), float(note[1]),
float(note[2])) for note in t]
def tenosrize(r):
    a = pretty_midi.PrettyMIDI(r)
    tnotes = []
    for b in a.instruments:
        b = b.notes
        notes =  tensor([[c.pitch, c.start, c.get_duration()] for c in b])
        tnotes.append(notes)
    return tnotes[0]
def dic(t, dics):
    shape = t.shape
    t = t.clone()
    for a in range(3):
        for b in tqdm(range(len(t[:, a]))):
            t[b, a] = dics[a].index([t[b, a].item()])
    t = t.type(torch.int)
    return t.reshape(shape)
def dic(t, dics):
    t = t.clone()
    for a in range(3):
        t[:, a] = tensor([dics[a].index([t[b, a].item()]) for b in range(len(t[:, a]))])
    return t.type(torch.int)
def undic(t, dics):
    shape = t.shape
    l = []
    for a in range(3):
        l.append(tensor(list(map(dics[a].__getitem__, t[:, a]))))
    l = torch.stack(l, dim=1)
    return l.reshape(shape)
def featurize(t):
    index = torch.argsort(t[:, 1], dim=0)
    t = torch.stack([t[int(a)] for a in index])
    out = t[1:, 1] - t[:-1, 1]
    t[1:, 1]= out
```

```
    return t
def unfeaturize(t):
    t = t.clone()
    for a in range(len(t)-1):
        out = t[a+1, 1]+ t[a, 1]
        t[a+1, 1]= out
    out = t[:, 2]+t[:, 1]
    t[:, 2] = out
    return t
# Substitua o bloco que cria a lista 'data' por este:

import os
from tqdm import tqdm # Garanta que tqdm está importado aqui

path = '/content/drive/MyDrive/test-code/classical-music-midi/chopin' # Verifique se este
caminho está 100% correto
data = []

# Este novo loop procura arquivos diretamente na pasta 'path'
for filename in os.listdir(path):
    if filename.lower().endswith(('.mid', '.midi')):
        data.append(os.path.join(path, filename))

# Adicione esta linha de verificação para ter certeza:
print(f"VERIFICAÇÃO: Foram encontrados {len(data)} arquivos MIDI.")
```

Extrair Notas, Construção e Reconstrução de Audios Teste

```
# Função de construção
def get_dics(directory):
    song = []
    for t in tqdm(directory):
        t = featurize(tenosrize(t))
        t[:, 1] = torch.clamp(t[:, 1], max=3.9687)
        t[:, 2] = torch.clamp(t[:, 2], min=1/time_step , max=4)
        song.append(t)
    t = torch.cat(song)
    t[:, [1,2]] = torch.round(t[:, [1,2]]*time_step)/time_step
    dics = [list(np.array(torch.unique(t[:, a]).type(torch.float))) for a in range(3)]
    dics[0] = [np.float32(a) for a in range(128)]
    return t, dics

# Rodar tudo do zero (sem usar arquivos salvos)

time_step = 32
t, dics = get_dics(data)  # <- 'data' deve estar carregada

songs = dic(t, dics)      # <- essa função também precisa estar definida corretamente
```

```
# Função de reconstrução
def unlatent(t, dics=dics):
    t = undic(t, dics)
    t = untenosrize(unfeaturize(t))
    return t
batch_size = 32
sequence_len = 128
split = torch.split(songs, sequence_len)[:-1]
x , y = torch.stack(split)[1:], torch.stack(split)[:-1]
dslen = len(x)//10
xtrain, ytrain = x[:dslen*9], y[:dslen*9]
xtest, ytest = x[dslen*9:], y[dslen*9:]
class trainset(Dataset):
    def __init__(self, data):
        self.x, self.y = data
    def __len__(self): return len(self.x)
    def __getitem__(self, index):
        x = self.x[index] # Input tokens
        y_tokens = self.y[index].long() # Target tokens, shape (seq_len, 3)

        # Codifica cada atributo com seu vocabulário correto
        y_pitch = nn.functional.one_hot(y_tokens[:, 0], num_classes=len(dics[0]))
        y_dtime = nn.functional.one_hot(y_tokens[:, 1], num_classes=len(dics[1]))
        y_dur = nn.functional.one_hot(y_tokens[:, 2], num_classes=len(dics[2]))

        # Retorna o input e uma tupla com as 3 etiquetas codificadas
        return x, (y_pitch, y_dtime, y_dur)

train, test = trainset([xtrain, ytrain]), trainset([xtest, ytest])
train, test = DataLoader(train, batch_size=batch_size, shuffle = True), DataLoader(test,
batch_size=batch_size, shuffle = True)

t, tok = get_dics(data[:2])
lat = dic(t, tok)
qwe = unlatent(lat, tok)
mid = pretty_midi.PrettyMIDI(data[0])
p1 = "test_uncompressed.mid"
p2 = "test_compressed.mid"
mid.write(p1)
mid.instruments[0].notes = qwe
mid.write(p2)
fs = 44000
# original
mid = pretty_midi.PrettyMIDI(p1)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
#compressed
mid = pretty_midi.PrettyMIDI(p2)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

```
p = data[0]
numiter = range(999).__iter__()
def gener(gen, x, dis= None, sequences=100, escape_count=10):
    output = []
    gen.to(device)
    output = []
    for a in range(sequences):
        if dis==None:
            x = gen(x, generate=True)
        else:
            dis.to(device)
                samples = [gen(x.type(torch.int), generate=True).type(torch.float32) for a in
range(escape_count)]
            score = [dis(samples[a]) for a in range(escape_count)]
            x = samples[score.index(max(score))]
        output.append(x)
    return undic(torch.cat(output, dim=1).squeeze().type(torch.int), dics)
def make_song(gen, p=p, sequence_len=sequence_len, dis=None, sequences= 25,
escape_count=10, evalu=False):
    with torch.no_grad():
        if evalu:
            model.eval()
        else: model.train()
        x,y = next(iter(train))
        x = x[0].unsqueeze(dim=0)
                preds = gener(gen, x.to(device), dis=dis, sequences=sequences,
escape_count=escape_count)
        out = untenosrize(unfeaturize(preds.squeeze()))
        mid = pretty_midi.PrettyMIDI(p)
        mid.instruments[0].notes = out
        itera = str(numiter.__next__())
        mid.write("song " + itera + '.mid')
```

Modelo

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = sequence_len):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(batch_size, max_len, 1, d_model)
        pe[:, :, 0, 0::2] = torch.sin(position * div_term)
        pe[:, :, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

```
def compose(f, x): return f(x)
class transformer(nn.Module):
   def __init__(self, d_model=1024, nhead=16, d_hid=16, nlayers=6, dropout= 0.25, nembeds=128):
      super().__init__()

      self.pos_encoder = PositionalEncoding(d_model, dropout)
      self.embeds = nn.Embedding(nembeds, d_model)
      self.pos_embeds = PositionalEncoding(d_model, dropout)
      layers = nn.TransformerEncoderLayer(d_model*3, nhead, d_hid, dropout, batch_first=True)
      self.transformer_encoder = nn.TransformerEncoder(layers, nlayers)
      self.decoder = nn.Linear(d_model*3, nembeds*3)

      self.dropout = nn.Dropout(dropout)
      self.d_model = d_model
      self.mask = torch.triu(torch.ones(sequence_len, sequence_len) * float('-inf'), diagonal=1).to(device)

   def forward(self, x, generate=False):
      x = self.embeds(x)* math.sqrt(self.d_model)
      x = self.pos_embeds(x)
      sp = x.shape
      x = torch.reshape(x, (sp[0], sp[1], (self.d_model*3)))
      x = self.transformer_encoder(x, self.mask)
      x= self.decoder(x)
      shap = x.shape
      x = torch.reshape(x, (shap[0], shap[1], 3, shap[2]//3))
      if generate:
         x = torch.argmax(x, dim=3)
      return x
# CÓDIGO NOVO - SUBSTITUA SUA FUNÇÃO 'eval' INTEIRA POR ESTA

def eval(model, dis=None):
   model.eval()
   model.to(device)
   with torch.no_grad():
      loss, false_preds, true_preds = [], [], []
      count = 0
      for x_batch, y_batch in test: # Renomeei as variáveis

         # --- CORREÇÃO APLICADA AQUI ---
         x = x_batch.to(device)
         y = (y_batch[0].to(device), y_batch[1].to(device), y_batch[2].to(device))
         # ----------------------------

         if count == 0:
            shape = torch.numel(x)
```

```
            count = 1
        preds = model(x)
        if dis != None:
            gen = torch.argmax(preds, dim=3)
            dis.to(device)
            false_preds.append(dis(gen).mean().item())
            true_preds.append(dis(x).mean().item())
        loss.append(creloss(preds, y).item())
    numelloss = round((sum(loss)/len(loss))/shape, 4)
    avloss = round(sum(loss)/len(loss), 4)
    if dis != None:
        false_preds = round(sum(false_preds)/len(false_preds), 4)
        true_preds = round(sum(true_preds)/len(true_preds), 4)
        print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss) + " False "
+ str(false_preds) + " True " + str(true_preds))
    else:
        print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss))
```

Treinamento do Modelo

```
from tqdm import tqdm
# CÓDIGO NOVO
import torch.nn.functional as F

def smax(t):
  # Usando a função softmax nativa do PyTorch, que é mais estável
  return F.softmax(t, dim=-1)

def creloss(preds, targets):
  # preds tem shape: (batch, seq, 3, 128)
  # targets é uma tupla com y_pitch, y_dtime, y_dur
  y_pitch, y_dtime, y_dur = targets

  # Fatiamos as previsões do modelo para corresponder ao tamanho de cada
vocabulário
  preds_pitch = preds[:, :, 0, :y_pitch.shape[-1]]
  preds_dtime = preds[:, :, 1, :y_dtime.shape[-1]]
  preds_dur   = preds[:, :, 2, :y_dur.shape[-1]]

  # Aplicamos softmax para obter as probabilidades
  preds_pitch = smax(preds_pitch)
  preds_dtime = smax(preds_dtime)
  preds_dur   = smax(preds_dur)

  # Calculamos a perda para cada atributo (adicionei 1e-8 para estabilidade numérica)
  loss_pitch = -torch.sum(y_pitch.float() * torch.log(preds_pitch + 1e-8))
  loss_dtime = -torch.sum(y_dtime.float() * torch.log(preds_dtime + 1e-8))
  loss_dur   = -torch.sum(y_dur.float() * torch.log(preds_dur + 1e-8))

  # Retornamos a soma das perdas
```

```
    return loss_pitch + loss_dtime + loss_dur
# CÓDIGO NOVO (PARA COLAR NO LUGAR DO ANTIGO)

def fit(model, dl, epochs=1, lr=0.0001):
    count = 0
    model.to(device)
    lossfunc = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    model.train()
    total_loss = []
    count = 0
    for a in range(epochs):
        for x_batch, y_batch in tqdm(dl): # Renomeei as variáveis para clareza
            if count == 0:
                shape = torch.numel(x_batch)
                count +=1

            # --- CORREÇÃO APLICADA AQUI ---
            # Move o input 'x' para o dispositivo
            x = x_batch.to(device)
            # Move cada tensor DENTRO da tupla 'y' para o dispositivo
            y = (y_batch[0].to(device), y_batch[1].to(device), y_batch[2].to(device))
            # ----------------------------

            preds = model(x)
            loss = creloss(preds, y)
            with torch.no_grad():
                total_loss.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        avloss = sum(total_loss)/len(total_loss)
        print("training batch loss " + str(avloss) + " numel loss " + str(avloss/shape))
        eval(model)
model = transformer()
fit(model, train, lr=0.0001, epochs=2)
torch.save(model, "music_transformer.pkl")
make_song(model)
make_song(model, evalu=True)
```

Geração da Música

```
mid = pretty_midi.PrettyMIDI("song 0.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
mid = pretty_midi.PrettyMIDI("song 1.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
class discriminator(nn.Module):
    def __init__(self, input_dim=3, lin_dim=256, lstm_dim=256, lstm_layers=4,
dropout=0.5):
```

```
        super().__init__()
        self.droupout = nn.Dropout(dropout)
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, lin_dim // 2),
            nn.LeakyReLU(0.2),
            nn.Linear(lin_dim // 2, lin_dim),
            nn.LeakyReLU(0.2))
        self.conv = nn.Sequential(
            nn.Conv1d(lstm_dim, 8, 8, 4),
            nn.LeakyReLU(0,1),
            nn.Conv1d(8, 1, 8, 8),
            nn.LeakyReLU(0,1),
            nn.Linear(3, 1))
        self.act= nn.Sigmoid()
        self.lin = nn.Sequential(nn.Linear(4, 1))
                    self.lstm     =     nn.LSTM(input_size=lin_dim,     hidden_size=lstm_dim,
num_layers=lstm_layers, batch_first=True, dropout=dropout, bidirectional=False)

    def forward(self, x):
        x = x.type(torch.float)
        x = self.mlp(x)
        x, h = self.lstm(x)
        x = self.conv(x.permute(0, 2, 1)).permute(0, 1, 2)
        return self.act(x.squeeze())
def traindis(gen, dis, epochs=1, lr=0.001, noise_scale=10):
    dis_opt =  optim.Adam(dis.parameters(), lr=lr)
    gen.train().to(device)
    dis.train().to(device)
    lossfunc = nn.BCELoss().to(device)
    for a in range(epochs):
        for x_batch, y_batch  in tqdm(train):


            x = x_batch.to(device)

            y_real = x.type(torch.float32)

            dis_opt.zero_grad()
            with torch.no_grad(): # Geramos 'fake' sem calcular gradientes para o gerador
                fake = gen(x, generate=True).type(torch.float32)

            fake_preds = dis(fake)
            real_preds = dis(y_real) # Usamos 'y_real' (que é o input 'x')

            fake_loss = lossfunc(fake_preds, (torch.zeros_like(fake_preds)))
            real_loss = lossfunc(real_preds,(torch.ones_like(real_preds)))
            dis_loss = (fake_loss + real_loss)/2
            dis_loss.backward()
```

```
            dis_opt.step()
        eval(model, dis)
dis = discriminator()
traindis(model, dis)
make_song(model, dis=dis)
mid = pretty_midi.PrettyMIDI("song 2.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Converter .mid para .wav

```
from midi2audio import FluidSynth
fs = FluidSynth('/content/drive/MyDrive/soundfonts/FluidR3_GM.sf2')  # certifique-se que o arquivo .sf2 está presente
fs.midi_to_audio('test.mid', 'test.wav')
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO D – Código do cenário Transformers + *dataset* Maestro

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O
monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports

```
!pip install pretty_midi
!apt-get install -y fluidsynth
!pip install midi2audio
from tqdm import tqdm
import os
import math
import random
import pandas as pd
import numpy as np
import IPython
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from IPython import *
import os
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.data import random_split
import pretty_midi
```

```
import torch
import math as m
import torch.optim as optim
import collections
from itertools import chain
from torch import tensor
from midi2audio import FluidSynth
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'
```
Carregar MIDIs
```
volume = 50
def untenosrize(t): return [pretty_midi.containers.Note(volume, int(note[0]), float(note[1]),
float(note[2])) for note in t]
def tenosrize(r):
    a = pretty_midi.PrettyMIDI(r)
    tnotes = []
    for b in a.instruments:
        b = b.notes
        notes =  tensor([[c.pitch, c.start, c.get_duration()] for c in b])
        tnotes.append(notes)
    return tnotes[0]
def dic(t, dics):
    shape = t.shape
    t = t.clone()
    for a in range(3):
        for b in tqdm(range(len(t[:, a]))):
            t[b, a] = dics[a].index([t[b, a].item()])
    t = t.type(torch.int)
    return t.reshape(shape)
def dic(t, dics):
    t = t.clone()
    for a in range(3):
        t[:, a] = tensor([dics[a].index([t[b, a].item()]) for b in range(len(t[:, a]))])
    return t.type(torch.int)
def undic(t, dics):
    shape = t.shape
    l = []
    for a in range(3):
        l.append(tensor(list(map(dics[a].__getitem__, t[:, a]))))
    l = torch.stack(l, dim=1)
    return l.reshape(shape)
def featurize(t):
    index = torch.argsort(t[:, 1], dim=0)
    t = torch.stack([t[int(a)] for a in index])
    out = t[1:, 1] - t[:-1, 1]
    t[1:, 1]= out
```

```
    return t
def unfeaturize(t):
    t = t.clone()
    for a in range(len(t)-1):
        out = t[a+1, 1]+ t[a, 1]
        t[a+1, 1]= out
    out = t[:, 2]+t[:, 1]
    t[:, 2] = out
    return t
path = '/content/drive/MyDrive/maestro-v3.0.0'
subdir = [x[0] for x in os.walk(path)][1:]
data = []
for files in subdir:
    for file in os.listdir(files):
        data.append(os.path.join(files, file))
time_step = 32
```

Extrair Notas, Construção e Reconstrução de Audios Teste

```
# Função de construção
def get_dics(directory):
    song = []
    for t in tqdm(directory):
        t = featurize(tenosrize(t))
        t[:, 1] = torch.clamp(t[:, 1], max=3.9687)
        t[:, 2] = torch.clamp(t[:, 2], min=1/time_step , max=4)
        song.append(t)
    t = torch.cat(song)
    t[:, [1,2]] = torch.round(t[:, [1,2]]*time_step)/time_step
    dics = [list(np.array(torch.unique(t[:, a]).type(torch.float))) for a in range(3)]
    dics[0] = [np.float32(a) for a in range(128)]
    return t, dics

# Rodar tudo do zero (sem usar arquivos salvos)
t, dics = get_dics(data)  # <- 'data' deve estar carregada

songs = dic(t, dics)      # <- essa função também precisa estar definida corretamente

# Função de reconstrução
def unlatent(t, dics=dics):
    t = undic(t, dics)
    t = untenosrize(unfeaturize(t))
    return t
batch_size = 32
sequence_len = 128
split = torch.split(songs, sequence_len)[:-1]
x , y = torch.stack(split)[1:], torch.stack(split)[:-1]
dslen = len(x)//10
xtrain, ytrain = x[:dslen*9], y[:dslen*9]
xtest, ytest = x[dslen*9:], y[dslen*9:]
```

```
class trainset(Dataset):
    def __init__(self, data):
        self.x, self.y = data
    def __len__(self): return len(self.x)
    def __getitem__(self, index):
        y = self.y[index]
        y = nn.functional.one_hot(y.type(torch.long), num_classes=len(dics[2]))
        return self.x[index], y
train, test = trainset([xtrain, ytrain]), trainset([xtest, ytest])
train, test = DataLoader(train, batch_size=batch_size, shuffle = True), DataLoader(test,
batch_size=batch_size, shuffle = True)
t, tok = get_dics(data[:2])
lat = dic(t, tok)
qwe = unlatent(lat, tok)
mid = pretty_midi.PrettyMIDI(data[0])
p1 = "test_uncompressed.mid"
p2 = "test_compressed.mid"
mid.write(p1)
mid.instruments[0].notes = qwe
mid.write(p2)
fs = 44000
# oringal
mid = pretty_midi.PrettyMIDI(p1)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
#compressed
mid = pretty_midi.PrettyMIDI(p2)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Modelo

```
p = data[0]
numiter = range(999).__iter__()
def gener(gen, x, dis= None, sequences=100, escape_count=10):
    output = []
    gen.to(device)
    output = []
    for a in range(sequences):
        if dis==None:
            x = gen(x, generate=True)
        else:
            dis.to(device)
            samples = [gen(x.type(torch.int), generate=True).type(torch.float32) for a in
range(escape_count)]
            score = [dis(samples[a]) for a in range(escape_count)]
            x = samples[score.index(max(score))]
        output.append(x)
    return undic(torch.cat(output, dim=1).squeeze().type(torch.int), dics)
def make_song(gen, p=p, sequence_len=sequence_len, dis=None, sequences= 25,
escape_count=10, evalu=False):
    with torch.no_grad():
```

```
        if evalu:
            model.eval()
        else: model.train()
        x,y = next(iter(train))
        x = x[0].unsqueeze(dim=0)
        preds = gener(gen, x.to(device), dis=dis, sequences=sequences,
escape_count=escape_count)
        out = untenosrize(unfeaturize(preds.squeeze()))
        mid = pretty_midi.PrettyMIDI(p)
        mid.instruments[0].notes = out
        itera = str(numiter.__next__())
        mid.write("song " + itera + '.mid')
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = sequence_len):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(batch_size, max_len, 1, d_model)
        pe[:, :, 0, 0::2] = torch.sin(position * div_term)
        pe[:, :, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
def compose(f, x): return f(x)
class transformer(nn.Module):
    def __init__(self, d_model=1024, nhead=16, d_hid=16, nlayers=6, dropout= 0.25,
nembeds=128):
        super().__init__()

        self.pos_encoder = PositionalEncoding(d_model, dropout)
        self.embeds = nn.Embedding(nembeds, d_model)
        self.pos_embeds = PositionalEncoding(d_model, dropout)
        layers = nn.TransformerEncoderLayer(d_model*3, nhead, d_hid, dropout,
batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(layers, nlayers)
        self.decoder = nn.Linear(d_model*3, nembeds*3)

        self.dropout = nn.Dropout(dropout)
        self.d_model = d_model
        self.mask = torch.triu(torch.ones(sequence_len, sequence_len) * float('-inf'),
diagonal=1).to(device)

    def forward(self, x, generate=False):
        x = self.embeds(x)* math.sqrt(self.d_model)
        x = self.pos_embeds(x)
```

```
        sp = x.shape
        x = torch.reshape(x, (sp[0], sp[1], (self.d_model*3)))
        x = self.transformer_encoder(x, self.mask)
        x= self.decoder(x)
        shap = x.shape
        x = torch.reshape(x, (shap[0], shap[1], 3, shap[2]//3))
        if generate:
            x = torch.argmax(x, dim=3)
        return x
def eval(model, dis=None):
    model.eval()
    model.to(device)
    with torch.no_grad():
        loss, false_preds, true_preds = [], [], [ ]
        count = 0
        for x,y in test:
            x,y = x.to(device), y.to(device)
            if count == 0:
                shape = torch.numel(x)
                count = 1
            preds = model(x)
            if dis != None:
                gen = torch.argmax(preds, dim=3)
                dis.to(device)
                false_preds.append(dis(gen).mean().item())
                true_preds.append(dis(x).mean().item())
            loss.append(creloss(preds, y).item())
        numelloss = round((sum(loss)/len(loss))/shape, 4)
        avloss = round(sum(loss)/len(loss), 4)
        if dis != None:
            false_preds = round(sum(false_preds)/len(false_preds), 4)
            true_preds = round(sum(true_preds)/len(true_preds), 4)
            print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss) + " False "
+ str(false_preds) + " True " + str(true_preds))
        else:
            print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss))
```

Treinamento do Modelo

```
from tqdm import tqdm
def smax(t): return t.exp()/torch.sum(t.exp(), dim=2, keepdim=True)
def creloss(x, y):
    shap = x.shape
    x = smax(x)
    return -torch.sum(y*torch.log(x))
def fit(model, dl, epochs=1, lr=0.0001):
    count = 0
    model.to(device)
    lossfunc = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
```

```
    model.train()
    total_loss = []
    count = 0
    for a in range(epochs):
        for x,y in tqdm(dl):
            if count == 0:
                shape = torch.numel(x)
                count +=1
            x,y = x.to(device), y.to(device)
            preds = model(x)
            loss = creloss(preds, y)
            with torch.no_grad():
                total_loss.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        avloss = sum(total_loss)/len(total_loss)
        print("training batch loss " + str(avloss) + " numel loss " + str(avloss/shape))
        eval(model)
model = transformer()
fit(model, train, lr=0.0001, epochs=2)
torch.save(model, "music_transformer.pkl")
make_song(model)
make_song(model, evalu=True)
```

Geração da Música

```
mid = pretty_midi.PrettyMIDI("song 0.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
mid = pretty_midi.PrettyMIDI("song 1.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
class discriminator(nn.Module):
        def __init__(self, input_dim=3, lin_dim=256, lstm_dim=256, lstm_layers=4,
dropout=0.5):
        super().__init__()
        self.droupout = nn.Dropout(dropout)
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, lin_dim // 2),
            nn.LeakyReLU(0.2),
            nn.Linear(lin_dim // 2, lin_dim),
            nn.LeakyReLU(0.2))
        self.conv = nn.Sequential(
            nn.Conv1d(lstm_dim, 8, 8, 4),
            nn.LeakyReLU(0,1),
            nn.Conv1d(8, 1, 8, 8),
            nn.LeakyReLU(0,1),
            nn.Linear(3, 1))
        self.act= nn.Sigmoid()
        self.lin = nn.Sequential(nn.Linear(4, 1))
```

```
                self.lstm = nn.LSTM(input_size=lin_dim, hidden_size=lstm_dim,
num_layers=lstm_layers, batch_first=True, dropout=dropout, bidirectional=False)

    def forward(self, x):
        x = x.type(torch.float)
        x = self.mlp(x)
        x, h = self.lstm(x)
        x = self.conv(x.permute(0, 2, 1)).permute(0, 1, 2)
        return self.act(x.squeeze())

def traindis(gen, dis, epochs=1, lr=0.001, noise_scale=10):
    dis_opt =  optim.Adam(dis.parameters(), lr=lr)
    gen.train().to(device)
    dis.train().to(device)
    lossfunc = nn.BCELoss().to(device)
    for a in range(epochs):
        for x,y  in tqdm(train):
            x, y =x.to(device), y.to(device)
            y = torch.argmax(y, dim=3).type(torch.float32)
            dis_opt.zero_grad()
            fake = gen(x, generate=True).type(torch.float32)
            fake_preds = dis(fake)
            real_preds= dis(y)
            fake_loss = lossfunc(fake_preds, (torch.zeros_like(fake_preds)))
            real_loss = lossfunc(real_preds,(torch.ones_like(real_preds)))
            dis_loss = (fake_loss + real_loss)/2
            dis_loss.backward()
            dis_opt.step()
        eval(model, dis)
dis = discriminator()
traindis(model, dis)
make_song(model, dis=dis)
mid = pretty_midi.PrettyMIDI("song 2.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Converter .mid para .wav

```
from midi2audio import FluidSynth
fs = FluidSynth('/content/drive/MyDrive/soundfonts/FluidR3_GM.sf2')  # certifique-se que
o arquivo .sf2 está presente
fs.midi_to_audio('test.mid', 'test.wav')
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
```

```
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO E – Código do cenário Chopin com sjustes + *dataset* Chopin

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O
monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports e Definições dos Caminhos

```
!pip install midi2audio

# Imports
import os
import numpy as np
import pandas as pd
from collections import Counter
import random
import warnings
from music21 import converter, instrument, note, chord, stream
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adamax
import tensorflow.keras.utils
from midi2audio import FluidSynth
```

```
from google.colab import drive
from tqdm import tqdm # Importação da tqdm

# Configurações
warnings.filterwarnings("ignore")
np.random.seed(42)

# Caminhos
dataset_path = "/content/drive/MyDrive/test-code/classical-music-midi/chopin" # Caminho para a pasta do MAESTRO
soundfont_path = "/content/drive/MyDrive/soundfonts"
output_midi = "Melody_Generated.mid"
output_wav = "Melody_Generated.wav"
```

Carregar MIDIs

```
# Carregar MIDIs com barra de progresso e tratamento de erros
all_midis = []
print(f"Carregando arquivos MIDI de: {dataset_path}")

# Usamos a tqdm para criar uma barra de progresso
file_list = os.listdir(dataset_path)
for i in tqdm(file_list, desc="Processando arquivos"):
    # Usamos .lower() para garantir que a extensão seja lida corretamente
    if i.lower().endswith(('.mid', '.midi')):
        try:
            tr = os.path.join(dataset_path, i)
            midi = converter.parse(tr)
            all_midis.append(midi)
        except Exception as e:
            # Adicionado um try-except para pular arquivos corrompidos
            print(f"\nAVISO: Não foi possível processar o arquivo {i}. Erro: {e}")
            continue

print(f"\nCarregamento concluído. {len(all_midis)} arquivos foram carregados com sucesso.")
```

Extrair Notas

```
# Extrair notas
def extract_notes(file):
    notes = []
    for j in file:
        songs = instrument.partitionByInstrument(j)
        for part in songs.parts:
            pick = part.recurse()
            for element in pick:
                if isinstance(element, note.Note):
                    notes.append(str(element.pitch))
                elif isinstance(element, chord.Chord):
                    notes.append(".".join(str(n) for n in element.normalOrder))
    return notes
```

```
Corpus = extract_notes(all_midis)
print("Total notes in all the Chopin midis:", len(Corpus))

# Remover notas raras
count_num = Counter(Corpus)
rare_note = [k for k, v in count_num.items() if v < 100]
Corpus = [element for element in Corpus if element not in rare_note]

# Mapeamento
symb = sorted(list(set(Corpus)))
mapping = dict((c, i) for i, c in enumerate(symb))
reverse_mapping = dict((i, c) for i, c in enumerate(symb))
```
Preparar Dados
```
# Preparar dados
length = 40
features = []
targets = []

for i in range(0, len(Corpus) - length, 1):
    feature = Corpus[i:i + length]
    target = Corpus[i + length]
    features.append([mapping[j] for j in feature])
    targets.append(mapping[target])

X = np.reshape(features, (len(targets), length, 1)) / float(len(symb))
y = tensorflow.keras.utils.to_categorical(targets)

X_train, X_seed, y_train, y_seed = train_test_split(X, y, test_size=0.2, random_state=42)

# Modelo
# OPÇÃO C: MODELO MAIS PROFUNDO E MAIS LARGO
model = Sequential()
# Camada 1 (Larga)
model.add(LSTM(1024, input_shape=(X.shape[1], X.shape[2]),
return_sequences=True))
model.add(Dropout(0.2)) # Aumentei um pouco o dropout para combater o overfitting
# Camada 2 (Intermediária)
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.2))
# Camada 3 (Final Recorrente)
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```
Treinamento do Modelo
```
# Treinar
```

```
history = model.fit(X_train, y_train, batch_size=256, epochs=100)
```
Geração da Música
```
# Gerar música
def chords_n_notes(Snippet):
    Melody = []
    offset = 0
    for i in Snippet:
        if ("." in i or i.isdigit()):
            chord_notes = i.split(".")
            notes = [note.Note(int(j)) for j in chord_notes]
            chord_snip = chord.Chord(notes)
            chord_snip.offset = offset
            Melody.append(chord_snip)
        else:
            note_snip = note.Note(i)
            note_snip.offset = offset
            Melody.append(note_snip)
        offset += 1
    return stream.Stream(Melody)

def Malody_Generator(Note_Count):
    seed = X_seed[np.random.randint(0, len(X_seed)-1)]
    Notes_Generated = []
    for _ in range(Note_Count):
        seed = seed.reshape(1, length, 1)
        prediction = model.predict(seed, verbose=0)[0]
        prediction = np.log(prediction + 1e-8)
        exp_preds = np.exp(prediction)
        prediction = exp_preds / np.sum(exp_preds)
        index = np.argmax(prediction)
        Notes_Generated.append(index)
        seed = np.insert(seed[0], len(seed[0]), index / float(len(symb)))
        seed = seed[1:]
    Music = [reverse_mapping[char] for char in Notes_Generated]
    Melody = chords_n_notes(Music)
    return Music, Melody

Music_notes, Melody = Malody_Generator(100)
Melody.write('midi', output_midi)
```
Converter .mid para .wav
```
!apt-get update
!apt-get install fluidsynth
# Converter .mid para .wav
fs = FluidSynth(soundfont_path)
fs.midi_to_audio(output_midi, output_wav)
print(f"Arquivo .wav gerado: {output_wav}")

# Reproduzir no Colab
```

```
import IPython.display as ipd
ipd.Audio(output_wav)
```
Plot dos Gráficos de Utilização de Recursos
```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO F – Código do cenário LSTM com ajustes + *dataset* Maestro

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports e Definições dos Caminhos

```
!pip install midi2audio

# Imports
import os
import numpy as np
import pandas as pd
from collections import Counter
import random
import warnings
from music21 import converter, instrument, note, chord, stream
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adamax
from tensorflow.keras.optimizers import Adam
```

```
import tensorflow.keras.utils
from midi2audio import FluidSynth
from google.colab import drive
from tqdm import tqdm # Importação da tqdm

# Configurações
warnings.filterwarnings("ignore")
np.random.seed(42)

# Caminhos
dataset_path = "/content/drive/MyDrive/maestro-full" # Caminho para a pasta do MAESTRO
soundfont_path = "/content/drive/MyDrive/soundfonts"
output_midi = "Melody_Generated.mid"
output_wav = "Melody_Generated.wav"
```

Carregar MIDIs

```
import multiprocessing
from functools import partial

# --- CÉLULA DE CARREGAMENTO OTIMIZADA ---

# Função auxiliar que processa um único arquivo.
# Ela precisa ser definida fora do loop principal para funcionar com multiprocessing.
def parse_midi_file(file_path, show_warnings=False):
    try:
        midi = converter.parse(file_path)
        return midi
    except Exception as e:
        if show_warnings:
            # Pega apenas o nome do arquivo para a mensagem de erro
            file_name = os.path.basename(file_path)
            print(f"\nAVISO: Não foi possível processar o arquivo {file_name}. Erro: {e}")
        return None

# Iniciar o cronômetro e o monitoramento
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5)
monitor.start()

# Carregar MIDIs com barra de progresso e tratamento de erros
print(f"Carregando arquivos MIDI de: {dataset_path}")

try:
    file_list = [os.path.join(dataset_path, i) for i in os.listdir(dataset_path) if i.lower().endswith(('.mid', '.midi'))]

    # Define o número de processos (geralmente o número de núcleos da CPU é uma boa escolha)
```

```
    # Deixe None para usar todos os núcleos disponíveis
    num_processes = multiprocessing.cpu_count()
    print(f"Utilizando {num_processes} núcleos para acelerar o carregamento...")

    # Cria um pool de processos para executar a tarefa em paralelo
    # O 'if __name__ == "__main__":' é importante no Jupyter em alguns sistemas
    if __name__ == "__main__":
        with multiprocessing.Pool(processes=num_processes) as pool:
            # Usa pool.imap para processar a lista e manter a barra de progresso (tqdm)
            # O resultado será uma lista de objetos midi (ou None para arquivos com erro)
            results = list(tqdm(pool.imap(parse_midi_file, file_list), total=len(file_list),
desc="Processando arquivos"))

    # Filtra os resultados para remover os arquivos que falharam (valor None)
    all_midis = [midi for midi in results if midi is not None]

    print(f"\nCarregamento concluído. {len(all_midis)} arquivos foram carregados com
sucesso.")

except FileNotFoundError:
    print(f"ERRO: O diretório '{dataset_path}' não foi encontrado. Verifique o caminho na
Célula 3.")
```

Extrair Notas

```
# Extrair notas
def extract_notes(file):
    notes = []
    for j in file:
        songs = instrument.partitionByInstrument(j)
        for part in songs.parts:
            pick = part.recurse()
            for element in pick:
                if isinstance(element, note.Note):
                    notes.append(str(element.pitch))
                elif isinstance(element, chord.Chord):
                    notes.append(".".join(str(n) for n in element.normalOrder))
    return notes

Corpus = extract_notes(all_midis)
print("Total notes in all the Maestro midis:", len(Corpus))

# Remover notas raras
count_num = Counter(Corpus)
rare_note = [k for k, v in count_num.items() if v < 100]
Corpus = [element for element in Corpus if element not in rare_note]

# Mapeamento
symb = sorted(list(set(Corpus)))
mapping = dict((c, i) for i, c in enumerate(symb))
```

```
reverse_mapping = dict((i, c) for i, c in enumerate(symb))
```
Preparar Dados
```
# Preparar dados
length = 40
features = []
targets = []

for i in range(0, len(Corpus) - length, 1):
    feature = Corpus[i:i + length]
    target = Corpus[i + length]
    features.append([mapping[j] for j in feature])
    targets.append(mapping[target])

X = np.reshape(features, (len(targets), length, 1)) / float(len(symb))
y = tensorflow.keras.utils.to_categorical(targets)

X_train, X_seed, y_train, y_seed = train_test_split(X, y, test_size=0.2, random_state=42)

# CÓDIGO NOVO (MAIS PROFUNDO E MAIS LARGO)
model = Sequential()
# Camada 1 (Larga)
model.add(LSTM(1024, input_shape=(X.shape[1], X.shape[2]),
return_sequences=True))
model.add(Dropout(0.2))
# Camada 2 (Intermediária)
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.2))
# Camada 3 (Final Recorrente)
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))

from tensorflow.keras.optimizers import Adam
optimizer = Adam(learning_rate=0.001) # Defina o Adam com a taxa de aprendizado que
planejamos
model.compile(loss='categorical_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
```
Treinamento do Modelo
```
history = model.fit(X_train, y_train, batch_size=256, epochs=100) # Alterado de 50 para
100
```
Geração da Música
 # Gerar música
```
def chords_n_notes(Snippet):
    Melody = []
    offset = 0
    for i in Snippet:
        if ("." in i or i.isdigit()):
```

```
            chord_notes = i.split(".")
            notes = [note.Note(int(j)) for j in chord_notes]
            chord_snip = chord.Chord(notes)
            chord_snip.offset = offset
            Melody.append(chord_snip)
        else:
            note_snip = note.Note(i)
            note_snip.offset = offset
            Melody.append(note_snip)
        offset += 1
    return stream.Stream(Melody)

def Malody_Generator(Note_Count):
    seed = X_seed[np.random.randint(0, len(X_seed)-1)]
    Notes_Generated = []
    for _ in range(Note_Count):
        seed = seed.reshape(1, length, 1)
        prediction = model.predict(seed, verbose=0)[0]
        prediction = np.log(prediction + 1e-8)
        exp_preds = np.exp(prediction)
        prediction = exp_preds / np.sum(exp_preds)
        index = np.argmax(prediction)
        Notes_Generated.append(index)
        seed = np.insert(seed[0], len(seed[0]), index / float(len(symb)))
        seed = seed[1:]
    Music = [reverse_mapping[char] for char in Notes_Generated]
    Melody = chords_n_notes(Music)
    return Music, Melody

Music_notes, Melody = Malody_Generator(100)
Melody.write('midi', output_midi)
```

Converter .mid para .wav

```
!apt-get update
!apt-get install fluidsynth
# Converter .mid para .wav
fs = FluidSynth(soundfont_path)
fs.midi_to_audio(output_midi, output_wav)
print(f"Arquivo .wav gerado: {output_wav}")

# Reproduzir no Colab
import IPython.display as ipd
ipd.Audio(output_wav)
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial
```

```
# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO G – Código do cenário Transformers com ajustes + *dataset* Chopin

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O
monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports

```
!pip install pretty_midi
!apt-get install -y fluidsynth
!pip install midi2audio
from tqdm import tqdm
import os
import math
import random
import pandas as pd
import numpy as np
import IPython
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from IPython import *
import os
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.data import random_split
import pretty_midi
```

```
import torch
import math as m
import torch.optim as optim
import collections
from itertools import chain
from torch import tensor
from midi2audio import FluidSynth
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'
```
Carregar MIDIs
```
volume = 50
def untenosrize(t): return [pretty_midi.containers.Note(volume, int(note[0]), float(note[1]),
float(note[2])) for note in t]
def tenosrize(r):
    a = pretty_midi.PrettyMIDI(r)
    tnotes = []
    for b in a.instruments:
        b = b.notes
        notes =  tensor([[c.pitch, c.start, c.get_duration()] for c in b])
        tnotes.append(notes)
    return tnotes[0]
def dic(t, dics):
    shape = t.shape
    t = t.clone()
    for a in range(3):
        for b in tqdm(range(len(t[:, a]))):
            t[b, a] = dics[a].index([t[b, a].item()])
    t = t.type(torch.int)
    return t.reshape(shape)
def dic(t, dics):
    t = t.clone()
    for a in range(3):
        t[:, a] = tensor([dics[a].index([t[b, a].item()]) for b in range(len(t[:, a]))])
    return t.type(torch.int)
def undic(t, dics):
    shape = t.shape
    l = []
    for a in range(3):
        l.append(tensor(list(map(dics[a].__getitem__, t[:, a]))))
    l = torch.stack(l, dim=1)
    return l.reshape(shape)
def featurize(t):
    index = torch.argsort(t[:, 1], dim=0)
    t = torch.stack([t[int(a)] for a in index])
    out = t[1:, 1] - t[:-1, 1]
    t[1:, 1]= out
```

```
    return t
def unfeaturize(t):
    t = t.clone()
    for a in range(len(t)-1):
        out = t[a+1, 1]+ t[a, 1]
        t[a+1, 1]= out
    out = t[:, 2]+t[:, 1]
    t[:, 2] = out
    return t
# Substitua o bloco que cria a lista 'data' por este:

import os
from tqdm import tqdm # Garanta que tqdm está importado aqui

path = '/content/drive/MyDrive/test-code/classical-music-midi/chopin' # Verifique se este
caminho está 100% correto
data = []

# Este novo loop procura arquivos diretamente na pasta 'path'
for filename in os.listdir(path):
    if filename.lower().endswith(('.mid', '.midi')):
        data.append(os.path.join(path, filename))

# Adicione esta linha de verificação para ter certeza:
print(f"VERIFICAÇÃO: Foram encontrados {len(data)} arquivos MIDI.")
```

Extrair Notas, Construção e Reconstrução de Audios Teste

```
# Função de construção
def get_dics(directory):
    song = []
    for t in tqdm(directory):
        t = featurize(tenosrize(t))
        t[:, 1] = torch.clamp(t[:, 1], max=3.9687)
        t[:, 2] = torch.clamp(t[:, 2], min=1/time_step , max=4)
        song.append(t)
    t = torch.cat(song)
    t[:, [1,2]] = torch.round(t[:, [1,2]]*time_step)/time_step
    dics = [list(np.array(torch.unique(t[:, a]).type(torch.float))) for a in range(3)]
    dics[0] = [np.float32(a) for a in range(128)]
    return t, dics

# Rodar tudo do zero (sem usar arquivos salvos)

time_step = 32
t, dics = get_dics(data)  # <- 'data' deve estar carregada

songs = dic(t, dics)      # <- essa função também precisa estar definida corretamente

# Função de reconstrução
```

```
def unlatent(t, dics=dics):
    t = undic(t, dics)
    t = untenosrize(unfeaturize(t))
    return t
batch_size = 32
sequence_len = 128
split = torch.split(songs, sequence_len)[:-1]
x , y = torch.stack(split)[1:], torch.stack(split)[:-1]
dslen = len(x)//10
xtrain, ytrain = x[:dslen*9], y[:dslen*9]
xtest, ytest = x[dslen*9:], y[dslen*9:]
class trainset(Dataset):
    def __init__(self, data):
        self.x, self.y = data
    def __len__(self): return len(self.x)
    def __getitem__(self, index):
        x = self.x[index] # Input tokens
        y_tokens = self.y[index].long() # Target tokens, shape (seq_len, 3)

        # Codifica cada atributo com seu vocabulário correto
        y_pitch = nn.functional.one_hot(y_tokens[:, 0], num_classes=len(dics[0]))
        y_dtime = nn.functional.one_hot(y_tokens[:, 1], num_classes=len(dics[1]))
        y_dur = nn.functional.one_hot(y_tokens[:, 2], num_classes=len(dics[2]))

        # Retorna o input e uma tupla com as 3 etiquetas codificadas
        return x, (y_pitch, y_dtime, y_dur)

train, test = trainset([xtrain, ytrain]), trainset([xtest, ytest])
train, test = DataLoader(train, batch_size=batch_size, shuffle = True), DataLoader(test,
batch_size=batch_size, shuffle = True)
t, tok = get_dics(data[:2])
lat = dic(t, tok)
qwe = unlatent(lat, tok)
mid = pretty_midi.PrettyMIDI(data[0])
p1 = "test_uncompressed.mid"
p2 = "test_compressed.mid"
mid.write(p1)
mid.instruments[0].notes = qwe
mid.write(p2)
fs = 44000
# original
mid = pretty_midi.PrettyMIDI(p1)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
#compressed
mid = pretty_midi.PrettyMIDI(p2)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Modelo

```
p = data[0]
numiter = range(999).__iter__()
def gener(gen, x, dis= None, sequences=100, escape_count=10):
    output = []
    gen.to(device)
    output = []
    for a in range(sequences):
        if dis==None:
            x = gen(x, generate=True)
        else:
            dis.to(device)
            samples = [gen(x.type(torch.int), generate=True).type(torch.float32) for a in
range(escape_count)]
            score = [dis(samples[a]) for a in range(escape_count)]
            x = samples[score.index(max(score))]
        output.append(x)
    return undic(torch.cat(output, dim=1).squeeze().type(torch.int), dics)
def make_song(gen, p=p, sequence_len=sequence_len, dis=None, sequences= 25,
escape_count=10, evalu=False):
    with torch.no_grad():
        if evalu:
            model.eval()
        else: model.train()
        x,y = next(iter(train))
        x = x[0].unsqueeze(dim=0)
        preds = gener(gen, x.to(device), dis=dis, sequences=sequences,
escape_count=escape_count)
        out = untenosrize(unfeaturize(preds.squeeze()))
        mid = pretty_midi.PrettyMIDI(p)
        mid.instruments[0].notes = out
        itera = str(numiter.__next__())
        mid.write("song " + itera + '.mid')
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = sequence_len):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(batch_size, max_len, 1, d_model)
        pe[:, :, 0, 0::2] = torch.sin(position * div_term)
        pe[:, :, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
def compose(f, x): return f(x)
class transformer(nn.Module):
```

```
   def __init__(self, d_model=1024, nhead=16, d_hid=16, nlayers=6, dropout= 0.25, nembeds=128):
      super().__init__()

      self.pos_encoder = PositionalEncoding(d_model, dropout)
      self.embeds = nn.Embedding(nembeds, d_model)
      self.pos_embeds = PositionalEncoding(d_model, dropout)
      layers = nn.TransformerEncoderLayer(d_model*3, nhead, d_hid, dropout, batch_first=True)
      self.transformer_encoder = nn.TransformerEncoder(layers, nlayers)
      self.decoder = nn.Linear(d_model*3, nembeds*3)

      self.dropout = nn.Dropout(dropout)
      self.d_model = d_model
      self.mask = torch.triu(torch.ones(sequence_len, sequence_len) * float('-inf'), diagonal=1).to(device)

   def forward(self, x, generate=False):
      x = self.embeds(x)* math.sqrt(self.d_model)
      x = self.pos_embeds(x)
      sp = x.shape
      x = torch.reshape(x, (sp[0], sp[1], (self.d_model*3)))
      x = self.transformer_encoder(x, self.mask)
      x= self.decoder(x)
      shap = x.shape
      x = torch.reshape(x, (shap[0], shap[1], 3, shap[2]//3))
      if generate:
         x = torch.argmax(x, dim=3)
      return x
# CÓDIGO NOVO - SUBSTITUA SUA FUNÇÃO 'eval' INTEIRA POR ESTA

def eval(model, dis=None):
   model.eval()
   model.to(device)
   with torch.no_grad():
      loss, false_preds, true_preds = [], [], []
      count = 0
      for x_batch, y_batch in test: # Renomeei as variáveis

         # --- CORREÇÃO APLICADA AQUI ---
         x = x_batch.to(device)
         y = (y_batch[0].to(device), y_batch[1].to(device), y_batch[2].to(device))
         # ----------------------------

         if count == 0:
            shape = torch.numel(x)
            count = 1
         preds = model(x)
```

```
            if dis != None:
                gen = torch.argmax(preds, dim=3)
                dis.to(device)
                false_preds.append(dis(gen).mean().item())
                true_preds.append(dis(x).mean().item())
            loss.append(creloss(preds, y).item())
        numelloss = round((sum(loss)/len(loss))/shape, 4)
        avloss = round(sum(loss)/len(loss), 4)
        if dis != None:
            false_preds = round(sum(false_preds)/len(false_preds), 4)
            true_preds = round(sum(true_preds)/len(true_preds), 4)
            print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss) + " False "
+ str(false_preds) + " True " + str(true_preds))
        else:
            print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss))
```

Treinamento do Modelo

```
from tqdm import tqdm
# CÓDIGO NOVO
import torch.nn.functional as F

def smax(t):
    # Usando a função softmax nativa do PyTorch, que é mais estável
    return F.softmax(t, dim=-1)

def creloss(preds, targets):
    # preds tem shape: (batch, seq, 3, 128)
    # targets é uma tupla com y_pitch, y_dtime, y_dur
    y_pitch, y_dtime, y_dur = targets

    # Fatiamos as previsões do modelo para corresponder ao tamanho de cada
vocabulário
    preds_pitch = preds[:, :, 0, :y_pitch.shape[-1]]
    preds_dtime = preds[:, :, 1, :y_dtime.shape[-1]]
    preds_dur   = preds[:, :, 2, :y_dur.shape[-1]]

    # Aplicamos softmax para obter as probabilidades
    preds_pitch = smax(preds_pitch)
    preds_dtime = smax(preds_dtime)
    preds_dur   = smax(preds_dur)

    # Calculamos a perda para cada atributo (adicionei 1e-8 para estabilidade numérica)
    loss_pitch = -torch.sum(y_pitch.float() * torch.log(preds_pitch + 1e-8))
    loss_dtime = -torch.sum(y_dtime.float() * torch.log(preds_dtime + 1e-8))
    loss_dur   = -torch.sum(y_dur.float() * torch.log(preds_dur + 1e-8))

    # Retornamos a soma das perdas
    return loss_pitch + loss_dtime + loss_dur
# CÓDIGO NOVO (PARA COLAR NO LUGAR DO ANTIGO)
```

```
def fit(model, dl, epochs=1, lr=0.0001):
    count = 0
    model.to(device)
    lossfunc = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=lr)
    model.train()
    total_loss = []
    count = 0
    for a in range(epochs):
        for x_batch, y_batch in tqdm(dl): # Renomeei as variáveis para clareza
            if count == 0:
                shape = torch.numel(x_batch)
                count +=1

            # --- CORREÇÃO APLICADA AQUI ---
            # Move o input 'x' para o dispositivo
            x = x_batch.to(device)
            # Move cada tensor DENTRO da tupla 'y' para o dispositivo
            y = (y_batch[0].to(device), y_batch[1].to(device), y_batch[2].to(device))
            # ----------------------------

            preds = model(x)
            loss = creloss(preds, y)
            with torch.no_grad():
                total_loss.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        avloss = sum(total_loss)/len(total_loss)
        print("training batch loss " + str(avloss) + " numel loss " + str(avloss/shape))
        eval(model)
model = transformer()
fit(model, train, lr=0.0001, epochs=50)
torch.save(model, "music_transformer.pkl")
make_song(model)
make_song(model, evalu=True)
mid = pretty_midi.PrettyMIDI("song 0.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
mid = pretty_midi.PrettyMIDI("song 1.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Geração da Música

```
class discriminator(nn.Module):
    def __init__(self, input_dim=3, lin_dim=256, lstm_dim=256, lstm_layers=4,
dropout=0.5):
        super().__init__()
        self.droupout = nn.Dropout(dropout)
```

```
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, lin_dim // 2),
            nn.LeakyReLU(0.2),
            nn.Linear(lin_dim // 2, lin_dim),
            nn.LeakyReLU(0.2))
        self.conv = nn.Sequential(
            nn.Conv1d(lstm_dim, 8, 8, 4),
            nn.LeakyReLU(0,1),
            nn.Conv1d(8, 1, 8, 8),
            nn.LeakyReLU(0,1),
            nn.Linear(3, 1))
        self.act= nn.Sigmoid()
        self.lin = nn.Sequential(nn.Linear(4, 1))
        self.lstm = nn.LSTM(input_size=lin_dim, hidden_size=lstm_dim,
num_layers=lstm_layers, batch_first=True, dropout=dropout, bidirectional=False)

    def forward(self, x):
        x = x.type(torch.float)
        x = self.mlp(x)
        x, h = self.lstm(x)
        x = self.conv(x.permute(0, 2, 1)).permute(0, 1, 2)
        return self.act(x.squeeze())


def traindis(gen, dis, epochs=1, lr=0.001, noise_scale=10):
    dis_opt =  optim.Adam(dis.parameters(), lr=lr)
    gen.train().to(device)
    dis.train().to(device)
    lossfunc = nn.BCELoss().to(device)
    for a in range(epochs):
        for x_batch, y_batch  in tqdm(train):


            x = x_batch.to(device)

            y_real = x.type(torch.float32)

            dis_opt.zero_grad()
            with torch.no_grad():
                fake = gen(x, generate=True).type(torch.float32)

            fake_preds = dis(fake)
            real_preds = dis(y_real)

            fake_loss = lossfunc(fake_preds, (torch.zeros_like(fake_preds)))
            real_loss = lossfunc(real_preds,(torch.ones_like(real_preds)))
            dis_loss = (fake_loss + real_loss)/2
            dis_loss.backward()
```

```
            dis_opt.step()
        eval(model, dis)
dis = discriminator()
traindis(model, dis)
make_song(model, dis=dis)
mid = pretty_midi.PrettyMIDI("song 2.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Converter .mid para .wav

```
from midi2audio import FluidSynth
fs = FluidSynth('/content/drive/MyDrive/soundfonts/FluidR3_GM.sf2')  # certifique-se que o arquivo .sf2 está presente
fs.midi_to_audio('test.mid', 'test.wav')
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```

ADENDO H – Código do cenário Transformers + *dataset* Maestro

Conexão com Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

Configuração da análise de Hardware

```
# Célula de Configuração do Monitoramento (adicionar no início)

# 1. Instalar bibliotecas para monitoramento
!pip install psutil pynvml

# 2. Importar tudo o que vamos precisar
import time
import threading
import psutil
import pynvml
import pandas as pd
import matplotlib.pyplot as plt

# 3. Definir a classe que fará o monitoramento
class ResourceMonitor:
    def __init__(self, interval=5):
        self.interval = interval
        self.data = []
        self._stop_event = threading.Event()
        self.thread = threading.Thread(target=self.run, daemon=True)

        # Inicializa a NVML para monitoramento da GPU
        try:
            pynvml.nvmlInit()
            self.gpu_count = pynvml.nvmlDeviceGetCount()
        except pynvml.NVMLError:
            self.gpu_count = 0
            print("AVISO: Placa NVIDIA não encontrada ou driver indisponível. O
monitoramento da GPU será desativado.")

    def _get_gpu_ram_usage(self):
        if self.gpu_count == 0:
            return 0
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        info = pynvml.nvmlDeviceGetMemoryInfo(handle)
        return info.used / (1024**3)  # Convertido para GB

    def run(self):
        """O método que roda em segundo plano para coletar dados."""
        start_time = time.time()
        while not self._stop_event.is_set():
            timestamp = time.time() - start_time
```

```
            # Coleta de dados
            sys_ram_used = psutil.virtual_memory().used / (1024**3) # GB
            gpu_ram_used = self._get_gpu_ram_usage() # GB
            disk_used = psutil.disk_usage('/').used / (1024**3) # GB

            self.data.append([timestamp, sys_ram_used, gpu_ram_used, disk_used])
            time.sleep(self.interval)

        if self.gpu_count > 0:
            pynvml.nvmlShutdown()

    def start(self):
        """Inicia o monitoramento."""
        print("Iniciando monitoramento de recursos...")
        self.thread.start()

    def stop(self):
        """Para o monitoramento."""
        self._stop_event.set()
        self.thread.join()
        print("Monitoramento de recursos finalizado.")
        return pd.DataFrame(self.data, columns=['Tempo (s)', 'RAM Sistema (GB)', 'RAM
GPU (GB)', 'Disco (GB)'])

    def plot(self):
        """Plota os dados coletados."""
        df = self.stop()

        if df.empty:
            print("Nenhum dado de monitoramento foi coletado.")
            return

        fig, axes = plt.subplots(3, 1, figsize=(12, 15), sharex=True)
        fig.suptitle('Utilização de Recursos do Sistema Durante a Execução', fontsize=16)

        # Gráfico de RAM do Sistema
        axes[0].plot(df['Tempo (s)'], df['RAM Sistema (GB)'], label='RAM do Sistema
Utilizada', color='blue')
        axes[0].set_ylabel('Uso (GB)')
        axes[0].set_title('Uso de RAM do Sistema')
        axes[0].grid(True)
        axes[0].legend()
        axes[0].fill_between(df['Tempo (s)'], df['RAM Sistema (GB)'], alpha=0.1, color='blue')

        # Gráfico de RAM da GPU
        if self.gpu_count > 0:
```

```
        axes[1].plot(df['Tempo (s)'], df['RAM GPU (GB)'], label='RAM da GPU Utilizada', color='green')
    else:
        axes[1].text(0.5, 0.5, 'Monitoramento de GPU não disponível', ha='center', va='center')
    axes[1].set_ylabel('Uso (GB)')
    axes[1].set_title('Uso de RAM da GPU')
    axes[1].grid(True)
    axes[1].legend()
    axes[1].fill_between(df['Tempo (s)'], df['RAM GPU (GB)'], alpha=0.1, color='green')

    # Gráfico de Uso de Disco
    axes[2].plot(df['Tempo (s)'], df['Disco (GB)'], label='Espaço em Disco Utilizado', color='red')
    axes[2].set_xlabel('Tempo (segundos)')
    axes[2].set_ylabel('Uso (GB)')
    axes[2].set_title('Uso de Disco')
    axes[2].grid(True)
    axes[2].legend()
    axes[2].fill_between(df['Tempo (s)'], df['Disco (GB)'], alpha=0.1, color='red')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

# 4. Iniciar o cronômetro e o monitoramento
# (Coloque estas 2 linhas logo antes do seu código principal começar a rodar)
tempo_inicial = time.time()
monitor = ResourceMonitor(interval=5) # O 'interval' é em segundos
monitor.start()
```

Imports

```
!pip install pretty_midi
!apt-get install -y fluidsynth
!pip install midi2audio
from tqdm import tqdm
import os
import math
import random
import pandas as pd
import numpy as np
import IPython
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from IPython import *
import os
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.data import random_split
import pretty_midi
```

```
import torch
import math as m
import torch.optim as optim
import collections
from itertools import chain
from torch import tensor
from midi2audio import FluidSynth
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'
```
Carregar MIDIs
```
volume = 50
def untenosrize(t): return [pretty_midi.containers.Note(volume, int(note[0]), float(note[1]),
float(note[2])) for note in t]
def tenosrize(r):
    a = pretty_midi.PrettyMIDI(r)
    tnotes = []
    for b in a.instruments:
        b = b.notes
        notes =  tensor([[c.pitch, c.start, c.get_duration()] for c in b])
        tnotes.append(notes)
    return tnotes[0]
def dic(t, dics):
    shape = t.shape
    t = t.clone()
    for a in range(3):
        for b in tqdm(range(len(t[:, a]))):
            t[b, a] = dics[a].index([t[b, a].item()])
    t = t.type(torch.int)
    return t.reshape(shape)
def dic(t, dics):
    t = t.clone()
    for a in range(3):
        t[:, a] = tensor([dics[a].index([t[b, a].item()]) for b in range(len(t[:, a]))])
    return t.type(torch.int)
def undic(t, dics):
    shape = t.shape
    l = []
    for a in range(3):
        l.append(tensor(list(map(dics[a].__getitem__, t[:, a]))))
    l = torch.stack(l, dim=1)
    return l.reshape(shape)
def featurize(t):
    index = torch.argsort(t[:, 1], dim=0)
    t = torch.stack([t[int(a)] for a in index])
    out = t[1:, 1] - t[:-1, 1]
    t[1:, 1]= out
```

```
    return t
def unfeaturize(t):
    t = t.clone()
    for a in range(len(t)-1):
        out = t[a+1, 1]+ t[a, 1]
        t[a+1, 1]= out
    out = t[:, 2]+t[:, 1]
    t[:, 2] = out
    return t
path = '/content/drive/MyDrive/maestro-v3.0.0'
subdir = [x[0] for x in os.walk(path)][1:]
data = []
for files in subdir:
    for file in os.listdir(files):
        data.append(os.path.join(files, file))
time_step = 32
```

Extrair Notas, Construção e Reconstrução de Audios Teste

```
# Função de construção
def get_dics(directory):
    song = []
    for t in tqdm(directory):
        t = featurize(tenosrize(t))
        t[:, 1] = torch.clamp(t[:, 1], max=3.9687)
        t[:, 2] = torch.clamp(t[:, 2], min=1/time_step , max=4)
        song.append(t)
    t = torch.cat(song)
    t[:, [1,2]] = torch.round(t[:, [1,2]]*time_step)/time_step
    dics = [list(np.array(torch.unique(t[:, a]).type(torch.float))) for a in range(3)]
    dics[0] = [np.float32(a) for a in range(128)]
    return t, dics

# Rodar tudo do zero (sem usar arquivos salvos)
t, dics = get_dics(data)  # <- 'data' deve estar carregada

songs = dic(t, dics)      # <- essa função também precisa estar definida corretamente

# Função de reconstrução
def unlatent(t, dics=dics):
    t = undic(t, dics)
    t = untenosrize(unfeaturize(t))
    return t
batch_size = 32
sequence_len = 128
split = torch.split(songs, sequence_len)[:-1]
x , y = torch.stack(split)[1:], torch.stack(split)[:-1]
dslen = len(x)//10
xtrain, ytrain = x[:dslen*9], y[:dslen*9]
xtest, ytest = x[dslen*9:], y[dslen*9:]
```

```
class trainset(Dataset):
    def __init__(self, data):
        self.x, self.y = data
    def __len__(self): return len(self.x)
    def __getitem__(self, index):
        y = self.y[index]
        y = nn.functional.one_hot(y.type(torch.long), num_classes=len(dics[2]))
        return self.x[index], y
train, test = trainset([xtrain, ytrain]), trainset([xtest, ytest])
train, test = DataLoader(train, batch_size=batch_size, shuffle = True), DataLoader(test,
batch_size=batch_size, shuffle = True)
t, tok = get_dics(data[:2])
lat = dic(t, tok)
qwe = unlatent(lat, tok)
mid = pretty_midi.PrettyMIDI(data[0])
p1 = "test_uncompressed.mid"
p2 = "test_compressed.mid"
mid.write(p1)
mid.instruments[0].notes = qwe
mid.write(p2)
fs = 44000
# oringal
mid = pretty_midi.PrettyMIDI(p1)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
#compressed
mid = pretty_midi.PrettyMIDI(p2)
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Modelo

```
p = data[0]
numiter = range(999).__iter__()
def gener(gen, x, dis= None, sequences=100, escape_count=10):
    output = []
    gen.to(device)
    output = []
    for a in range(sequences):
        if dis==None:
            x = gen(x, generate=True)
        else:
            dis.to(device)
            samples = [gen(x.type(torch.int), generate=True).type(torch.float32) for a in
range(escape_count)]
            score = [dis(samples[a]) for a in range(escape_count)]
            x = samples[score.index(max(score))]
        output.append(x)
    return undic(torch.cat(output, dim=1).squeeze().type(torch.int), dics)
def make_song(gen, p=p, sequence_len=sequence_len, dis=None, sequences= 25,
escape_count=10, evalu=False):
    with torch.no_grad():
```

```
        if evalu:
            model.eval()
        else: model.train()
        x,y = next(iter(train))
        x = x[0].unsqueeze(dim=0)
        preds = gener(gen, x.to(device), dis=dis, sequences=sequences,
escape_count=escape_count)
        out = untenosrize(unfeaturize(preds.squeeze()))
        mid = pretty_midi.PrettyMIDI(p)
        mid.instruments[0].notes = out
        itera = str(numiter.__next__())
        mid.write("song " + itera + '.mid')
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = sequence_len):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(batch_size, max_len, 1, d_model)
        pe[:, :, 0, 0::2] = torch.sin(position * div_term)
        pe[:, :, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
def compose(f, x): return f(x)
class transformer(nn.Module):
    def __init__(self, d_model=1024, nhead=16, d_hid=16, nlayers=6, dropout= 0.25,
nembeds=128):
        super().__init__()

        self.pos_encoder = PositionalEncoding(d_model, dropout)
        self.embeds = nn.Embedding(nembeds, d_model)
        self.pos_embeds = PositionalEncoding(d_model, dropout)
        layers = nn.TransformerEncoderLayer(d_model*3, nhead, d_hid, dropout,
batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(layers, nlayers)
        self.decoder = nn.Linear(d_model*3, nembeds*3)

        self.dropout = nn.Dropout(dropout)
        self.d_model = d_model
        self.mask = torch.triu(torch.ones(sequence_len, sequence_len) * float('-inf'),
diagonal=1).to(device)

    def forward(self, x, generate=False):
        x = self.embeds(x)* math.sqrt(self.d_model)
        x = self.pos_embeds(x)
```

```
        sp = x.shape
        x = torch.reshape(x, (sp[0], sp[1], (self.d_model*3)))
        x = self.transformer_encoder(x, self.mask)
        x= self.decoder(x)
        shap = x.shape
        x = torch.reshape(x, (shap[0], shap[1], 3, shap[2]//3))
        if generate:
            x = torch.argmax(x, dim=3)
        return x
def eval(model, dis=None):
    model.eval()
    model.to(device)
    with torch.no_grad():
        loss, false_preds, true_preds = [], [], [ ]
        count = 0
        for x,y in test:
            x,y = x.to(device), y.to(device)
            if count == 0:
                shape = torch.numel(x)
                count = 1
            preds = model(x)
            if dis != None:
                gen = torch.argmax(preds, dim=3)
                dis.to(device)
                false_preds.append(dis(gen).mean().item())
                true_preds.append(dis(x).mean().item())
            loss.append(creloss(preds, y).item())
        numelloss = round((sum(loss)/len(loss))/shape, 4)
        avloss = round(sum(loss)/len(loss), 4)
        if dis != None:
            false_preds = round(sum(false_preds)/len(false_preds), 4)
            true_preds = round(sum(true_preds)/len(true_preds), 4)
            print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss) + " False "
+ str(false_preds) + " True " + str(true_preds))
        else:
            print("Eval batch loss " + str(avloss) + " numel loss " + str(numelloss))
```

Treinamento do Modelo

```
from tqdm import tqdm
def smax(t): return t.exp()/torch.sum(t.exp(), dim=2, keepdim=True)
def creloss(x, y):
    shap = x.shape
    x = smax(x)
    return -torch.sum(y*torch.log(x))
def fit(model, dl, epochs=1, lr=0.0001):
    count = 0
    model.to(device)
    lossfunc = nn.CrossEntropyLoss()
```

```
    optimizer = optim.AdamW(model.parameters(), lr=lr) # Alterado para AdamW
    model.train()
    total_loss = []
    count = 0
    for a in range(epochs):
        for x,y in tqdm(dl):
            if count == 0:
                shape = torch.numel(x)
                count +=1
            x,y = x.to(device), y.to(device)
            preds = model(x)
            loss = creloss(preds, y)
            with torch.no_grad():
                total_loss.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        avloss = sum(total_loss)/len(total_loss)
        print("training batch loss " + str(avloss) + " numel loss " + str(avloss/shape))
        eval(model)
# LINHA ALTERADA
model = transformer(nlayers=8) # Adicionado o parâmetro nlayers=8
fit(model, train, lr=0.0001, epochs=10)
torch.save(model, "music_transformer.pkl")
make_song(model)
make_song(model, evalu=True)
```

Geração da Música

```
mid = pretty_midi.PrettyMIDI("song 0.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
mid = pretty_midi.PrettyMIDI("song 0.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
class discriminator(nn.Module):
    def __init__(self, input_dim=3, lin_dim=256, lstm_dim=256, lstm_layers=4,
dropout=0.5):
        super().__init__()
        self.droupout = nn.Dropout(dropout)
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, lin_dim // 2),
            nn.LeakyReLU(0.2),
            nn.Linear(lin_dim // 2, lin_dim),
            nn.LeakyReLU(0.2))
        self.conv = nn.Sequential(
            nn.Conv1d(lstm_dim, 8, 8, 4),
            nn.LeakyReLU(0,1),
            nn.Conv1d(8, 1, 8, 8),
            nn.LeakyReLU(0,1),
            nn.Linear(3, 1))
        self.act= nn.Sigmoid()
```

```
        self.lin = nn.Sequential(nn.Linear(4, 1))
        self.lstm = nn.LSTM(input_size=lin_dim, hidden_size=lstm_dim,
num_layers=lstm_layers, batch_first=True, dropout=dropout, bidirectional=False)

    def forward(self, x):
        x = x.type(torch.float)
        x = self.mlp(x)
        x, h = self.lstm(x)
        x = self.conv(x.permute(0, 2, 1)).permute(0, 1, 2)
        return self.act(x.squeeze())
def traindis(gen, dis, epochs=1, lr=0.001, noise_scale=10):
    dis_opt =  optim.Adam(dis.parameters(), lr=lr)
    gen.train().to(device)
    dis.train().to(device)
    lossfunc = nn.BCELoss().to(device)
    for a in range(epochs):
        for x,y  in tqdm(train):
            x, y =x.to(device), y.to(device)
            y = torch.argmax(y, dim=3).type(torch.float32)
            dis_opt.zero_grad()
            fake = gen(x, generate=True).type(torch.float32)
            fake_preds = dis(fake)
            real_preds= dis(y)
            fake_loss = lossfunc(fake_preds, (torch.zeros_like(fake_preds)))
            real_loss = lossfunc(real_preds,(torch.ones_like(real_preds)))
            dis_loss = (fake_loss + real_loss)/2
            dis_loss.backward()
            dis_opt.step()
        eval(model, dis)
dis = discriminator()
traindis(model, dis)
make_song(model, dis=dis)
mid = pretty_midi.PrettyMIDI("song 2.mid")
IPython.display.Audio(mid.synthesize(fs=fs), rate=fs)
```

Converter .mid para .wav

```
from midi2audio import FluidSynth
fs = FluidSynth('/content/drive/MyDrive/soundfonts/FluidR3_GM.sf2')  # certifique-se que
o arquivo .sf2 está presente
fs.midi_to_audio('test.mid', 'test.wav')
```

Plot dos Gráficos de Utilização de Recursos

```
# Célula de Finalização e Plotagem (adicionar no final)

# 1. Parar o cronômetro e calcular o tempo total
tempo_final = time.time()
tempo_total_segundos = tempo_final - tempo_inicial

# Formatando o tempo para horas, minutos e segundos
horas = int(tempo_total_segundos // 3600)
```

```
minutos = int((tempo_total_segundos % 3600) // 60)
segundos = int(tempo_total_segundos % 60)

print("\n--- Relatório Final de Execução ---")
print(f"Tempo de Execução Total: {horas}h {minutos}min {segundos}s")

# 2. Parar o monitor e gerar o gráfico de uso de recursos
# A função plot() já chama o stop() e processa os dados
monitor.plot()
```