UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

GUSTAVO SENGLING FAVARO

DESENVOLVIMENTO DE UM SISTEMA GERADOR DE MÚSICA SIMBÓLICA PARA JOGOS ELETRÔNICOS DO GÊNERO RPG

CAMPO MOURÃO

2025

GUSTAVO SENGLING FAVARO

DESENVOLVIMENTO DE UM SISTEMA GERADOR DE MÚSICA SIMBÓLICA PARA JOGOS ELETRÔNICOS DO GÊNERO RPG

Development of a symbolic music generator system for RPG themed video games

Trabalho de Conclusão de Curso de Graduação como requisito para obtenção do título de Bacharel em do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.
Orientador(a): Prof. Dr. Juliano Henrique Foleis

CAMPO MOURÃO

2025

GUSTAVO SENGLING FAVARO

DESENVOLVIMENTO DE UM SISTEMA GERADOR DE MÚSICA SIMBÓLICA PARA JOGOS ELETRÔNICOS DO GÊNERO RPG

Trabalho de Conclusão de Curso de Graduação como requisito para obtenção do título de Bacharel em do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 17 / junho / 2025

Juliano Henrique Foleis
Doutorado em Engenharia Elétrica com ênfase em Engenharia da Computação
Universidade Tecnológica Federal do Paraná

Marcos Silvano Almeida
Doutorado em Ciência da Computação
Universidade Tecnológica Federal do Paraná

Rodrigo Hübner
Doutorado em Engenharia Elétrica com ênfase em Engenharia da Computação
Universidade Tecnológica Federal do Paraná

CAMPO MOURÃO

2025

AGRADECIMENTOS

Gostaria de expressar minha sincera gratidão a todas as pessoas que me ajudaram e estiveram comigo, tanto na realização deste trabalho, quanto no decorrer do curso inteiro.

Agradeço, primeiramente, à minha família, pelo amor e apoio incondicional ao longo de toda minha trajetória acadêmica, que acreditaram em um adolescente que simplesmente decidiu viver por anos longe de todos para seguir seu sonho, e que hoje podem ver os frutos de tanto tempo de dedicação sendo colhidos.

Ao meu orientador, Prof. Dr. Juliano Henrique Foleis, por ter sido mais do que um professor, mas sim um amigo. Uma pessoa que esteve junto durante grande parte do curso, que compartilhou momentos de alegria, frustração, conquistas. Que me apoiou no meu momento mais difícil no final do curso, nunca desistiu de mim e me acompanhou até o fim deste trabalho. Muito obrigado pelo conhecimento, pelas memórias, pelas conversas, mas principalmente por acreditar em mim.

A todos os amigos e colegas da universidade, com quem compartilhei desafios, aprendizados e todos os momentos bons e ruins ao longo do curso. Em especial aos meus amigos: Fabricio Flavio Martins Damasceno, Fernando Zhen, Getúlio Coimbra Régis, Igor Lara de Oliveira e Lucca Toledo Bordim. Muito obrigado por serem vocês mesmos e por compartilhar a caminhada de vocês junto a minha. Se eu tivesse escolhido outro rumo na minha vida com pessoas completamente diferentes, nada nem nunca teria sido melhor do que o que vivemos juntos. Que a universidade não seja nossa última lembrança juntos.

A todos os envolvidos na instituição. Aos professores do Departamento Acadêmico de Computação, aos profissionais responsáveis pelos demais departamentos que compõem a universidade, a todos que trabalham arduamente para que a instituição seja referência como é. Obrigado pelo aprendizado, pela infraestrutura e pela experiência acadêmica que pude adquirir nestes anos.

E, por fim, agradeço a Deus. Pela força concedida ao longo desta caminhada, pela presença constante nos momentos de dúvida, angústia e superação, e por tornar possível cada passo desta trajetória. A Ele, minha eterna gratidão por ter me permitido viver alegrias, a construir laços e memórias que levarei comigo para sempre.

A todas as pessoas envolvidas de qualquer maneira, muito obrigado por tudo.

RESUMO

Este trabalho detalha o processo de desenvolvimento de um sistema de geração de melodias com temática de jogos do gênero RPG de *video-games* legados, empregando técnicas de aprendizagem de máquina. O sistema pode oferecer estas melodias tanto em arquivo de áudio (WAV) quanto em representação MIDI. O objetivo principal deste sistema é atuar como uma ferramenta de inspiração para compositores superarem possíveis bloqueios criativos. Para o desenvolvimento, foi utilizada uma base de dados de arquivos MIDI, que representam peças musicais de jogos deste gênero, especificamente do *video-game Nintendo Entertainment System* (NES). Para o processamento de dados, foi realizada a separação das melodias de cada música em rótulos diferentes, para melhor contextualização para o modelo de aprendizagem, no qual foi empregado o método de Redes Neurais, em específico as LSTMs. O sistema conta com uma interface gráfica, na qual, através dela, o usuário do sistema insere uma melodia parcial e solicita a geração de uma melodia nova, baseada na entrada do usuário e na emoção selecionada. A implementação deste sistema resultou em melodias com certo grau de organização estrutural e baixa dissonância entre as notas, porém sem padrões musicais claramente definidos. Este trabalho visa demonstrar a sua utilidade para futuras implementações que visem aprimorar a integração da inteligência artificial com a produção musical, podendo ser expandido para diferentes gêneros ou aspectos de estrutura musical.

Palavras-chave: aprendizagem de máquina; redes neurais; jogos eletrônicos; geração de música simbólica; composição musical.

ABSTRACT

This work details the development process of a melody generation system, focused on RPG games for legacy video game consoles, employing machine learning techniques. The system offers the generated melodies in both audio (WAV) and MIDI formats. Its primary goal is to serve as a tool for inspiring composers, helping them overcome potential creative blocks. To build this system, a database of MIDI files was used, representing musical pieces from games of this genre, specifically from the Nintendo Entertainment System (NES) game console. During data preprocessing, the melodies from each track were separated into different labels to improve contextualization for the learning model, which employed a Neural Network approach, specifically Long Short-Term Memory (LSTM) networks. The system includes a graphical user interface, through which users input a partial melody and request the generation of a new one, based on their input and the selected emotion. The implementation produced melodies with a certain degree of structural organization and low dissonance between notes, although without clearly defined musical patterns. This project aims to demonstrate its utility for future implementations that seek to enhance the integration of artificial intelligence within musical production, potentially expanding to other genres or different aspects of musical structure.

Keywords: machine learning; neural networks; video games; symbolic music generation; musical composition.

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ARPG	*Action Role-Playing Game*
DPCM	*Differential Pulse-Code Modulation*
HMM	*Hidden Markov Models*
LSTM	*Long Short-Term Memory*
MIDI	*Musical Instrument Digital Interface*
MSE	*Mean Squared Error*
NES	*Nintendo Entertainment System*
NES-MDB	*The NES Music DataBase*
RNA	Redes Neurais Artificiais
RNN	*Recurrent Neural Network*
RPG	*Role-Playing Game*
WAVE	*Waveform Audio File Format*

SUMÁRIO

1 INTRODUÇÃO

A música desempenha um papel fundamental em produções artísticas, como filmes, desenhos animados e reportagens jornalísticas. No caso de jogos eletrônicos, a trilha sonora se torna um elemento principal em suas narrativas, pois ela possui a função principal de descrever o cenário e o ambiente onde o personagem de uma história narrada por este jogo se encontra. Em jogos do gênero *Role-Playing Game* (RPG), por exemplo, as músicas podem possuir temas específicos, como uma fantasia medieval ou futurista, nos quais as músicas têm mais foco na descrição do cenário onde o personagem se encontra, podendo demandar composições com diferentes enfoques rítmicos e temáticos.

A criação de trilhas sonoras para jogos representa uma tarefa desafiadora. Segundo Plut e Pasquier (2020), os jogos frequentemente seguem uma estrutura musical linear, caracterizada por uma sequência contínua de seções musicais projetadas para serem repetidas infinitamente. Cada peça é produzida para descrever um cenário específico. Este processo pode ser ainda mais desafiador para compositores que não possuem conhecimento amplo em teoria musical ou que enfrentam barreiras criativas que restringem sua capacidade de produção.

Este trabalho tem como objetivo o desenvolvimento de um sistema capaz de gerar melodias com o uso de ferramentas de inteligência artificial generativa. O sistema utiliza como base de dados músicas de jogos do gênero RPG do console *Nintendo Entertainment System* (NES). A partir de uma sequência inicial fornecida pelo usuário, o sistema gera novas melodias, com a finalidade de complementar suas ideias ou inspirá-lo a criar composições originais. A etapa de treinamento do modelo é opcional ao usuário, dado que o sistema já possui um modelo compilado sobre esta base de dados.

O fluxo deste sistema pode ser resumido em quatro etapas, sendo elas:

- **Processo de treinamento do modelo por rótulo (opcional):** Com base nas peças musicais em formato *Musical Instrument Digital Interface* (MIDI) coletadas e manualmente separadas por emoções, o sistema pode treinar um modelo de aprendizagem de máquina com músicas de uma emoção específica.

- **Coleta da entrada do usuário:** O usuário pode fornecer uma melodia tanto por meio da interface do programa quanto por carregar um arquivo MIDI com esta melodia. Também informa a emoção desejada, o número de melodias a serem geradas e a quantidade de notas que as melodias resultantes terá.

- **Geração de melodias baseada na entrada:** O sistema carrega o modelo correspondente à emoção selecionada, utiliza a melodia fornecida e gera uma quantidade de melodias, especificada na entrada do programa.

- **Visualização gráfica e exportação das melodias geradas:** Através da própria interface gráfica, o usuário pode visualizar o comportamento das notas na representação "*Piano Roll*". As melodias também podem ser exportadas tanto em formato de áudio,

como o *Waveform Audio File Format* (WAVE), quanto em formato MIDI, podendo ser utilizadas em diferentes plataformas de áudio, como *softwares* de produção musical.

Este trabalho está estruturado como segue. O Capítulo 2 apresenta o referencial teórico, contendo um breve histórico de algoritmos de composição musical e apresentando alguns conceitos utilizados neste trabalho. O Capítulo 3 descreve o processo de desenvolvimento realizado, detalhando as etapas de pré-processamento de dados, a implementação do modelo de aprendizagem e geração de melodias e a interface de usuário empregada para este sistema. O Capítulo 4 apresenta os resultados obtidos do desenvolvimento do sistema como um todo. Por fim, o Capítulo 5 apresenta o fechamento do trabalho, concluindo o assunto e citando abordagens alternativas para a proposta apresentada.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta o referencial teórico relacionado à geração automática de melodias. Inicialmente, são discutidas abordagens clássicas que não dependem de aprendizado de máquina, como os jogos de dados musicais, baseados em regras combinatórias, e os algoritmos genéticos, que exploram processos evolutivos para compor melodias. Em seguida, são apresentadas abordagens baseadas em aprendizado de máquina, com foco em Modelos de Markov Ocultos (HMMs) e Redes Neurais Recorrentes (RNNs), que permitem capturar padrões musicais a partir de dados. O objetivo é contextualizar o histórico da composição algorítmica no decorrer do tempo, tanto como as técnicas utilizadas no desenvolvimento do trabalho, e destacar suas principais características e limitações.

2.1 Abordagens Clássicas

2.1.1 Jogos de Dados Musicais

As primeiras aplicações de algoritmos voltados à composição musical remontam ao século XVIII, com os chamados jogos de dados musicais. Nessas práticas, o usuário realizava o sorteio de valores por meio de dados e, com base no resultado, selecionava aleatoriamente trechos musicais para compor a partitura. Cada barra de compasso seguia regras específicas, permitindo que a combinação dos fragmentos resultasse em uma sequência musical estruturada.

O método publicado no livro *Der allezeit fertige Polonoisen- und Menuettenkomponist* [O compositor pronto de polonaises e minuetos], proposto por Kirnberger (1767), é considerado um dos precursores dos jogos de dados musicais. A proposta prometia ao leitor a possibilidade de compor melodias sem a necessidade de conhecimentos avançados em teoria musical. Outros compositores também desenvolveram versões próprias desses jogos, como Carl Philipp Emanuel Bach, em 1758, e Maximilian Stadler, em 1780. Há ainda versões não autênticas atribuídas a Joseph Haydn e Wolfgang Amadeus Mozart no final do século XVIII, conforme descrito por Nierhaus (2009).

O método de Kirnberger (1767) oferece compassos, compostos pelo autor, distribuídos em tabelas diferentes. O jogador seleciona um número total de compassos que ele deseja para compor a melodia. Para um número c de compassos, o jogador deve sortear um valor pelos dados c vezes, correspondendo a uma tabela t específica para cada compasso. A junção destes trechos sorteados garante a formação de uma melodia única.

Um exemplo de uma melodia composta por este método pode ser descrito utilizando 4 compassos, juntamente a um dado de 6 faces. Considerando um cenário onde os números 3, 5, 1 e 4 são sorteados, o método irá gerar uma melodia composta pelos compassos c_1t_3, c_2t_5, c_3t_1 e c_4t_4.

2.1.2 Gramáticas Gerativas

O método das gramáticas generativas, desenvolvido por Noam Chomsky em 1957, são estruturas formais que têm como propósito a descrição de uma linguagem, podendo ser composta por frases ou expressões. Estas gramáticas podem ser compostas por frases ou expressões que contém uma cadeia de símbolos, regendo um conjunto de regras específicas para uma linguagem. Segundo Chomsky (2002), uma expressão de uma linguagem é composta de uma combinação de unidades (ou símbolos), e com uma sintaxe para esta linguagem pode-se verificar a corretude destas expressões, fazendo com que seja possível a geração de construções linguísticas com precisão. Estes conceitos podem ser aplicados em diversas áreas da computação, como analisadores sintáticos em compiladores, processamento de linguagem natural, entre outros.

Uma possível aplicação de gramáticas gerativas no contexto de composição musical, segundo Nierhaus (2009), implica em um sistema gerar e analisar elementos musicais, quando é fornecido ao mesmo estruturas musicais diferentes como sua entrada. Estes sistemas podem ser utilizados para a descrição de estilos musicais autênticos. Pode-se utilizar como exemplo o *jazz*, no qual é possível utilizar este método para uma geração de progressões de acordes, com base na entrada do modelo e em conjunto com as regras da harmonização do gênero musical.

2.1.3 Algoritmos Genéticos

Os algoritmos genéticos constituem uma classe de algoritmos de otimização e busca, que utilizam conceitos inspirados na seleção natural e na genética. Segundo Mitchell (1998), um algoritmo genético opera com uma população de cromossomos (representados por dados), uma função de aptidão (*fitness*), um processo de seleção dos cromossomos mais aptos da população e a geração de uma nova população com base na aptidão definida pela função do algoritmo.

Uma aplicação comum de algoritmos genéticos para composição musical é a utilização de acordes como entrada do algoritmo, como realizado por Horner e Goldberg (1991). Nesse experimento, foram selecionados cinco acordes diferentes e um padrão de referência para o cálculo da aptidão dos cromossomos em cada geração, submetendo-os a transformações como *crossover* e mutação. O resultado foi a geração de uma sequência de acordes, correspondente à saída do algoritmo genético a cada nova população.

Os métodos clássicos, no geral, oferecem vantagens como a simplicidade de implementação e a geração de resultados coerentes, com estilos musicais bem definidos. No entanto, apresentam desvantagens significativas, como a rigidez estrutural dos jogos de dados que limita a diversidade composicional, as gramáticas requerem conhecimento explícito e detalhado das regras musicais, e os algoritmos genéticos frequentemente demandam funções de aptidão cuidadosamente projetadas, que podem não capturar adequadamente nuances musicais subjetivas.

Diante dessas limitações, optou-se neste trabalho por um modelo baseado em aprendizagem de máquina, que permite capturar padrões e relacionamentos melódicos de maneira mais flexível, aprendendo diretamente das músicas sem depender de regras codificadas manualmente.

2.2 Abordagens Baseadas em Aprendizagem de Máquina

2.2.1 Modelos Ocultos de Markov

Os *Hidden Markov Models* (HMM) são modelos estatísticos probabilísticos, utilizados para representar sequências temporais ou lineares, geradas por uma sequência de estados ocultos. Segundo Eddy (1996), as HMMs são especialmente úteis para representar sistemas cujas estruturas internas são invisíveis (ocultas), porém com saídas observáveis que seguem padrões estatísticos previsíveis, capazes de capturar regularidades em sequências. No contexto de composição algorítmica, as probabilidades de transição de estados do modelo de Markov são geradas de acordo com os parâmetros da sua estrutura ou calculadas no processo de imitação de estilos pela análise da entrada do sistema.

O uso de HMMs foi explorado por Hirzel e Soukup (2000), no qual foi possível gerar seções de improviso de *jazz* baseadas em pequenos padrões processados pelo modelo oculto. O sistema recebeu a peça *Forest Flower (Sunrise)*, composta por Charles Lloyd, como entrada, juntamente a algumas progressões harmônicas. Com base nessas entradas, foi capaz de gerar uma sucessão de padrões melódicos. Esses padrões representavam os estados ocultos do sistema, formando sua saída.

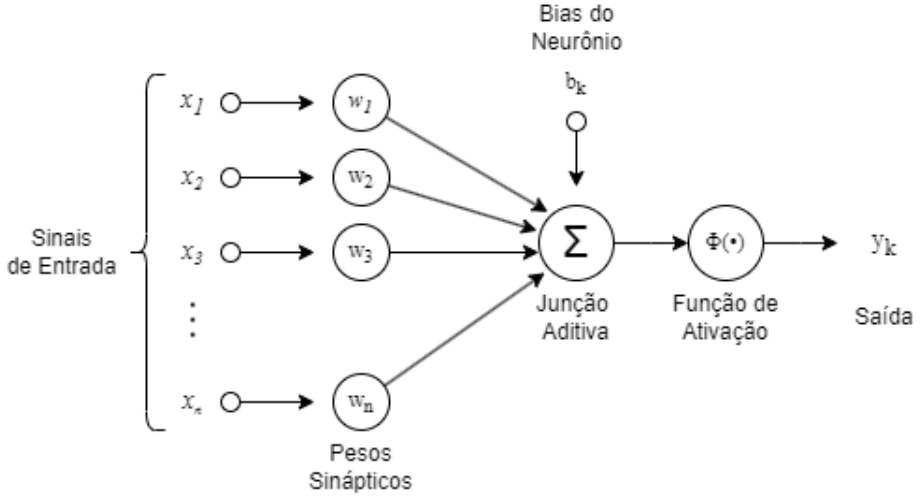
2.2.2 Redes Neurais Artificiais

As Redes Neurais Artificiais (RNA) são sistemas computacionais inspirados na estrutura do cérebro humano, compostos por unidades chamadas neurônios artificiais, organizados em camadas interligadas, como definem Haykin (2001) e Faceli *et al.* (2021). Cada conexão entre neurônios possui um peso ajustado por algoritmos de aprendizado, com base em dados de entrada, permitindo a identificação de padrões e a realização de previsões. A Figura 1 mostra um modelo que representa a arquitetura de um neurônio artificial.

2.2.3 Redes Neurais Recorrentes

Uma Rede Neural Recorrente, ou *Recurrent Neural Network* (RNN) é uma arquitetura de RNA que possui pelo menos um laço de realimentação, ou seja, um dado de uma saída de uma camada de neurônios pode ser transmitida para outras camadas anteriores da rede, segundo explica Haykin (2001), permitindo a modelagem de sequências. A aplicação desta técnica é útil

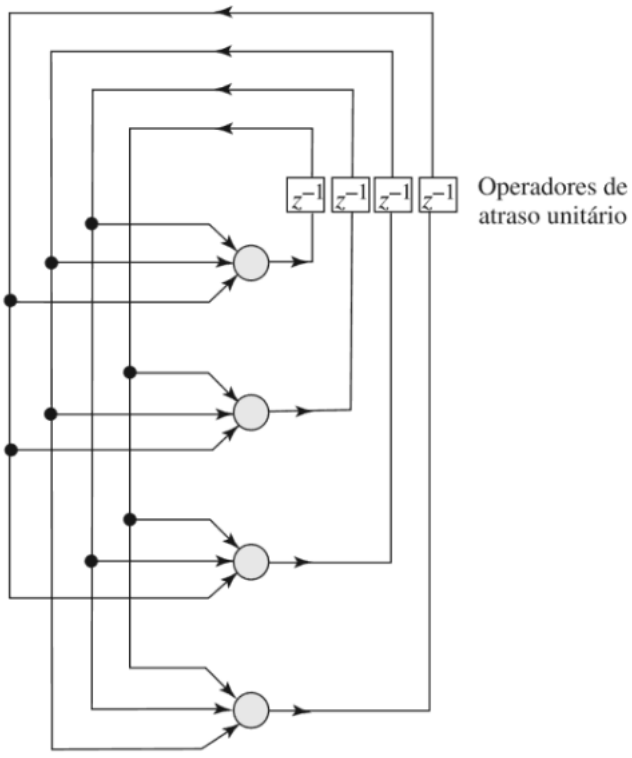
Figura 1 – Arquitetura de um neurônio artificial



Fonte: Adaptado de Haykin (2001).

para aplicar dinamismo no processamento de uma informação, trazendo vantagens em aplicações de sistemas que variam seus estados em relação ao tempo. Segundo SILVA, SPATTI e FLAUZINO (2016), as RNNs podem ser utilizadas para previsão de séries temporais, controle de processos e otimização e identificação de sistemas. A Figura 2 apresenta um exemplo de um modelo de rede recorrente, onde a saída dos neurônios é utilizada como entrada por outros neurônios de camadas distintas.

Figura 2 – Exemplo de RNN com laços de realimentação



Fonte: Haykin (2001).

Embora as RNNs sejam eficazes no processamento de dados sequenciais, elas apresentam limitações significativas, como a dificuldade em manter informações de longo prazo. Para resolver essas falhas, Hochreiter e Schmidhuber (1997) propuseram o modelo *Long Short-Term Memory* (LSTM), que introduz o conceito de células de memória capazes de preservar informações relevantes por períodos mais longos. O LSTM utiliza três portas: de entrada, de saída e de esquecimento. Essas portas regulam, respectivamente, a introdução de novos dados, a liberação de informações para as próximas etapas e a exclusão de dados desnecessários, permitindo um controle mais eficiente sobre o fluxo de informações na rede.

Segundo Hochreiter e Schmidhuber (1997), Jurgen Schmidhuber e Douglas Eck desenvolveram um sistema com a aplicação de LSTM na geração de melodias sobre progressões de acordes, mas, segundo Michael Mozer, os resultados apresentaram baixa coerência musical e pouca organização temática e rítmica, podendo ser relacionados à dificuldade do modelo em lidar com peças musicais mais longas. Porém, em um trabalho recente feito utilizando LSTM, Mauthes (2018) observa que, com um número relativamente maior de épocas de treinamento, o modelo implementado é capaz de gerar notas mais coerentes com a escala fornecida e com menor ocorrência de dissonâncias.

2.3 *Musical Instrument Digital Interface* (MIDI)

O MIDI é um protocolo de comunicação de dados, que descreve um meio de troca de informações e sinais de controle entre instrumentos e sistemas digitais. Enquanto em partituras musicais, a representação é mais voltada a notações musicais, o MIDI descreve instruções específicas para que um circuito eletrônico possa reproduzir estes sons, como explica Rothstein (1995).

Cada instrução transmitida por este protocolo é chamada de Mensagem MIDI. Estas mensagens contém eventos que descrevem como um instrumento ou dispositivo digital deve interpretar um dado comando e o reproduzir. Estes comandos geralmente são mensagens de controle de tempo, controle de nota, controle de volume, entre outros.

Um exemplo de uma mensagem é o comando `note_on`, composta por parâmetros que controlam as notas e suas propriedades. O campo *pitch* contém um número inteiro que representa a nota musical a ser reproduzida. O campo *velocity* indica a "força" que a nota é pressionada, influenciando no seu volume de reprodução, podendo ser comparado a uma medida de pressão em uma tecla de piano. Por exemplo, o valor 100 para *velocity* pode indicar que a nota foi pressionada até o fim, enquanto o valor 0 indica que a nota parou de ser pressionada. O parâmetro *time* é utilizado para referenciar o tempo de execução da nota em relação à ultima. O uso deste campo não é comum para o controle em tempo real de dispositivos MIDI, mas pode ser utilizado em casos de controle automatizado. Por exemplo, a primeira mensagem geralmente possui *time* igual a 0, quando ocorre o início da sequência. As mensagens subsequentes terão valores de tempo que indicam o intervalo, em unidades de *ticks*, desde o evento

Listagem 1 – Mensagens MIDI do *jingle* "1-Up"

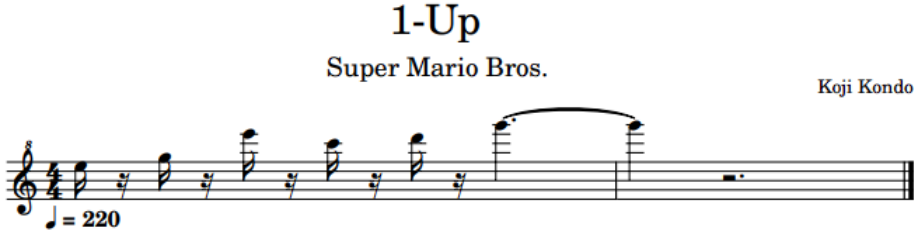
```
1  note_on channel=0 note=88  velocity=100 time=0
2  note_on channel=0 note=88  velocity=0   time=0.0681
3  note_on channel=0 note=91  velocity=100 time=0.0681
4  note_on channel=0 note=91  velocity=0   time=0.0681
5  note_on channel=0 note=100 velocity=100 time=0.0681
6  note_on channel=0 note=100 velocity=0   time=0.0681
7  note_on channel=0 note=96  velocity=100 time=0.0681
8  note_on channel=0 note=96  velocity=0   time=0.0681
9  note_on channel=0 note=98  velocity=100 time=0.0681
10 note_on channel=0 note=98  velocity=0   time=0.0681
11 note_on channel=0 note=103 velocity=100 time=0.0681
12 note_on channel=0 note=103 velocity=0   time=0.6818
```

Fonte: Autoria própria (2023).

anterior. Isso significa que o valor de *time* representa um deslocamento temporal relativo no fluxo da sequência das Mensagems MIDI.

A Listagem 1 apresenta como uma sequência de Mensagens MIDI `note_on` realiza a entrada de notas musicais através de sua interface, e a Figura 3 a mesma representação em uma partitura.

Figura 3 – Partitura do *jingle* "1-Up"



Fonte: Autoria própria (2023).

3 FUNDAMENTO E DESENVOLVIMENTO DO SISTEMA

Neste capítulo, é apresentado o sistema desenvolvido com o objetivo de auxiliar na produção de melodias para RPGs de jogos eletrônicos retrô. As etapas de desenvolvimento do sistema, tanto como a estrutura do mesmo, são apresentadas nas seções a seguir, abrangendo desde a coleta e preparação dos dados até a modelagem do sistema de aprendizagem de máquina e a construção da interface gráfica do usuário.

Para a implementação deste sistema, foi utilizada a linguagem de programação *Python* (versão 3.12.10), criada por Rossum e Jr (1995), que oferece uma vasta disponibilidade de ferramentas para processamento de dados, facilitando o processo de manipulação e pré-processamento dos arquivos MIDI e seus atributos. Além disso, foi empregada a biblioteca *TensorFlow*, desenvolvida por Abadi *et al.* (2016), para a construção, treinamento e avaliação do modelo de rede neural, permitindo a aplicação de técnicas de aprendizado profundo (*deep learning*) com eficiência e escalabilidade.

3.1 Organização Estrutural do Sistema

A estrutura de diretórios e arquivos do sistema foi organizada com o objetivo de promover a modularidade, a clareza e a manutenibilidade do código-fonte. A divisão dos componentes segue uma separação funcional entre dados, lógica de processamento, interface e recursos auxiliares, sendo descrita pelos itens a seguir:

- **`assets/`**: Contém componentes para a execução do programa, sendo categorizados nos diretórios:
  - `midi/`: Contém os arquivos MIDI que compõem a base de dados, separados em subpastas para cada rótulo, como *battle*, *peaceful*, *melancholic*, entre outros.
  - `models/`: Modelos treinados salvos no formato `.keras` para cada emoção.
  - `sounds/`: Arquivos de áudio utilizados na interface gráfica, como notas musicais para o teclado virtual interativo (ex: `C4.mp3`, `D#5.mp3`) e o efeito sonoro para o metrônomo (ex: `tick.mp3`).
- **`core/`**: Núcleo lógico do sistema, contendo códigos-fonte para a execução dos módulos de *machine learning* e outras funcionalidades, sendo eles:
  - `generate.py`: Responsável pela geração de melodias com base nos modelos treinados.
  - `midi_handle.py`: Utilitários para manipulação de arquivos MIDI.
  - `state.py`: Gerencia o estado interno da aplicação.

  - `train_model.py`: Executa o treinamento dos modelos de aprendizado de máquina.
  - `utils.py`: Funções auxiliares utilizadas por diferentes partes do sistema.

- **`ui/`**: Módulos da interface gráfica do usuário, sendo eles:
  - `piano_ui.py`: Interface gráfica do teclado virtual interativo.
  - `handlers.py`, `keyboard_input.py`, `log_panel.py`: Controladores de eventos, entrada via teclado físico e painel de *log* textual, respectivamente.

- **`temp/`**: Diretório temporário utilizado para armazenar arquivos temporários gerados durante o processo de treinamento, sendo criados e deletados pelo próprio *script* de treino.

- **`output/`**: Armazena os arquivos de saída gerados pela aplicação, obtendo as melodias exportadas tanto em formato MIDI quanto em formato sonoro (`.wav`), à escolha do usuário.

- **`main.py`**: Código-fonte principal de entrada da aplicação, responsável por inicializar os componentes do sistema.

3.2 Coleta de dados

As peças musicais utilizadas para compor a base de dados do sistema provém do *dataset The NES Music DataBase* (NES-MDB), desenvolvida por Donahue, Mao e McAuley (2018), contendo músicas do sistema legado de jogos da *Nintendo*, NES, em formato simbólico (MIDI). Esta base de dados contém aproximadamente 5.000 músicas correspondentes a cerca de 400 jogos diferentes deste console.

Considerando que o foco principal do sistema implementado é a geração musical voltada para jogos do gênero RPG, foi feita uma seleção de peças musicais específicas de jogos deste gênero, conforme uma listagem de títulos publicados para o NES, disponibilizada pelo *website* Nintendo Fandom (2025). A seleção das músicas, comparando os jogos citados na lista e os jogos presentes na base de dados, resultou em cerca de 600 faixas musicais para o treinamento do sistema.

A justificativa para utilizar apenas jogos de RPG se dá pela diversidade das músicas presentes em cada jogo, contendo emoções distintas para cada tema. A utilização de outros gêneros pode implicar em um desbalanceamento na base de dados, ao realizar a separação manual das peças musicais em emoções diferentes, cujo processo é descrito posteriormente neste capítulo. Por exemplo, jogos de plataforma podem apresentar um maior volume de músicas com ritmo acelerado, e se forem inclusos na base de dados final pode causar uma minoridade de arquivos em outras emoções, como músicas mais calmas.

A estrutura dos arquivos MIDI utilizados nesta base de dados se assemelha aos canais do *chip* de som do NES, que contém cinco faixas distintas. As duas primeiras correspondem às ondas de pulso, geralmente usadas em melodias e efeitos sonoros, a terceira à onda triangular, comum em linhas de baixo e alguns elementos de percussão, a quarta ao canal de ruído, destinada à percussão e efeitos sonoros, e a quinta, ao canal denominado *Differential Pulse-Code Modulation* (DPCM) que, segundo Donahue *et al.* (2019), reproduz *samples* de áudio gravados e comprimidos nos cartuchos dos jogos, podendo adicionar elementos à música e efeitos sonoros. Para o propósito de recolher apenas as melodias de cada peça musical, foram utilizadas as duas primeiras faixas de cada música, por conterem, em sua maioria, as melodias principais dentro de seus arranjos.

3.3 Organização e Normalização da base de dados

3.3.1 Filtragem das músicas

Após a obtenção dos MIDIs, foi feita uma separação manual de todas as músicas coletadas anteriormente em emoções distintas, para melhor contextualizar o cenário da peça de acordo com a melodia utilizada. A utilização de diferentes emoções como Ação/Tensão, Aventura, Batalha, Mistério e Tranquilidade dão um contexto específico às melodias, e sua separação faz o modelo ser capaz de capturar elementos como estrutura e escalas musicais de acordo com a situação descrita. A Tabela 1 mostra as emoções que foram utilizadas para a separação das peças musicais, assim como a descrição de elementos usadas como critério para a separação.

Tabela 1 – Descrição das emoções utilizados como rótulos para as músicas

Ação	Músicas que evocam movimento, com um rítmo acelerado.
Aventura	Músicas que descrevem cenários, que podem evocar um sentimento de inspiração, jornada.
Batalha	Músicas com rítmos acelerados e intensos, evocam um sentimento de conflito, perigo.
Castelo	Músicas com ritmos calmos, exaltando sentimentos de nobreza.
Cômico	Músicas que evocam um sentimento cômico, eventos que causam humor.
Heroico	Músicas que exaltam heroísmo, glória, protagonismo.
Melancólico	Músicas com temática triste, com rítmos lentos e poucas notas na melodia.
Mistério	Músicas com rítmos lentos, que evocam o sentimento de medo, suspense.
Tranquilidade	Músicas calmas e alegres, evocam um sentimento de paz, de estar em um ambiente familiar, calmo.

Fonte: Autoria Própria (2025).

Em seguida, foi feita uma segunda filtragem das músicas coletadas do *dataset* NES-MDB, removendo músicas de títulos que não agregavam com uma diversificação de cenários em suas trilhas sonoras. Por exemplo, algumas músicas, por serem de jogos do subgênero

Action Role-Playing Game (ARPG), são mais focadas em ação do que descrição de cenário, possuindo menos variedade de emoções em suas trilhas sonoras. Devido a isto, músicas de jogos mencionados na lista composta por Nintendo Fandom (2025) como *Spelunker*, *Ys - The Vanished Omens* e *Zelda II: The Adventure of Link* foram retiradas da base de dados, sobrando um total de 317 peças musicais. A Tabela 2 mostra a distribuição dos arquivos MIDI para cada rótulo.

Tabela 2 – Distribuição das músicas para cada rótulo

Rótulo	Quantidade de músicas
Ação	41
Aventura	37
Batalha	63
Castelo	15
Cômico	11
Heroico	28
Melancólico	34
Mistério	47
Tranquilidade	41
Total	**317**

Fonte: Autoria Própria (2025).

3.3.2 Leitura e pré-processamento de arquivos MIDI

Para o tratamento da base de dados, foi utilizada a biblioteca *"pretty_midi"*, que permite a leitura, manipulação e extração de informações de arquivos no formato MIDI. Dado um arquivo de entrada, a ferramenta consegue extrair o tom da nota de cada faixa (*pitch*), o tempo que a nota começa a ser tocada (*start*) e o tempo que a nota termina (*end*), ambos em relação à duração da música.

A grande maioria desta base de dados possui a melodia de suas peças na primeira faixa de seus MIDIs, porém em outros arquivos, a melodia principal se encontra na segunda faixa de alguns jogos em específico. Para evitar uma eventual dissonância no treinamento do modelo, foi selecionada a segunda faixa das músicas dos jogos *Castlevania II: Simon's Quest*, *Chaos World*, *Dark Lord*, *Famicom Jump II: Saikyō no Shichinin*, *Heracles no Eikou II: Titan no Metsubo* e *Mother*, pois a melodia principal de suas peças está presente nessa faixa. Como os arquivos MIDI têm os três primeiros caracteres de seus nomes representando o código de um determinado jogo, esses caracteres foram usados para identificar as músicas que requerem a segunda faixa como melodia principal, garantindo que a melodia principal sempre seja selecionada corretamente para a composição do modelo de aprendizagem.

As melodias presentes na base de dados variam significativamente tanto em relação à altura tonal quanto em escalas utilizadas para as mesmas. A junção destas melodias, sem um tratamento adequado para as mesmas, pode comprometer a consistência durante o treinamento do modelo, como a geração de notas discrepantes que não transmitem nenhum sentido musical. Para mitigar esse problema, foi implementada uma função para normalizar as melodias

durante o carregamento. Essa função realiza a transposição das notas para um tom específico, de acordo com seu contexto. Por exemplo, Dó para melodias em tom maior e Lá para melodias em tom menor. Além disso, aplica uma segunda transposição para posicionar as melodias dentro de um *range* específico. Neste caso, as notas foram ajustadas para que a sua maioria fique entre C5 e C6.

Para facilitar este processo, foi criado um arquivo de texto contendo os metadados de cada música de uma dada emoção, contendo seu nome e seu tom, se é maior ou menor. A Listagem 2 (no Apêndice A) mostra a função `normalize_notes`, que realiza a transposição das melodias de acordo com seus metadados, também presente no código-fonte `midi_handle.py`.

Para o processamento das notas, foi feita uma adaptação na entrada dos dados, convertendo os campos *start* e *end* para *step* e *duration*. Este formato é mais favorável para o treinamento do modelo, pois são marcações de tempo relativas à distância entre as notas, o que facilita o aprendizado de padrões temporais relativos, tornando-o mais robusto a variações de tempo, conforme explica Macaluso (2023).

Para que haja uma consistência maior no treinamento do modelo, foi aplicado um agrupamento específico para as notas, para que o total seja um número divisível pelo valor definido na variável `seq_length`. Esta variável determina o número de notas a serem agrupadas para definir cada sequência. Isto é necessário para que o modelo trate os dados como sequências, o que encaixa no contexto do uso da LSTM, que é "treinada" com o uso de dados sequenciais.

Esta subdivisão permite que cada sequência separada no treinamento tenha um contexto bem definido, sem misturar notas distintas e transições de melodias diferentes em uma única sequência.

Esse método pode descartar notas presentes no final da melodia. Esse descarte pode não afetar melodias mais longas, mas pode eliminar partes importantes de melodias mais curtas. Para tratar esse problema, foi aplicado um método que aumenta a duração da melodia conforme sua extensão em relação ao parâmetro `seq_length`. Esse método aproveita a estrutura da peça, que é feita para ser executada em repetição, garantindo que todas as notas sejam consideradas sem perder o contexto musical da melodia.

A Listagem 4 (no Apêndice A) do código-fonte `midi_handle.py` mostra a função `midi_notes_to_panda`. Esta função realiza o pré-processamento das melodias e as converte em um *DataFrame Pandas* para a utilização no treinamento do modelo de aprendizagem de máquina.

3.4 Implementação do Sistema de Aprendizagem de Máquina

3.4.1 Treinamento

A etapa de treinamento é responsável por realizar a carga dos dados da base, executar o pré-processamento e preparar os dados para que possam ser utilizados no treinamento da rede

neural. Após o treinamento, o modelo resultante é salvo em disco. Esse modelo será utilizado posteriormente na etapa de geração de melodias.

O processo de treinamento é implementado no *script* `train_model.py`, o qual pode ser executado pela linha de comando. Para isso, foi utilizada a biblioteca *Argparse*, que permite a entrada de parâmetros diretamente via terminal, tais como a emoção e hiperparâmetros, como o comprimento das sequências de entrada (`seq_length`) e o número de épocas de treinamento (`epochs`). A Listagem 3 mostra um exemplo de como executar o *script* para que ele realize o treinamento de um modelo.

Após receber os comandos de entrada, o *script* realiza a busca do diretório que possui os arquivos MIDI, dada a emoção inserida pelo usuário e as carrega pelo método "*midi_to_pandas_notes*", acumulando as melodias em um *DataFrame Pandas*, para logo a seguir serem carregadas para um *Dataset Tensorflow*.

Com os dados devidamente carregados, é necessário realizar a separação das notas em sequências. Essa separação é importante porque o treinamento utiliza modelos de redes neurais, exigindo a definição de sequências de entrada com tamanho fixo. Essa abordagem permite que o modelo aprenda padrões locais ao longo do tempo. Segundo Goodfellow, Bengio e Courville (2016), esse tipo de estrutura sequencial é essencial para que modelos de aprendizado profundo, especialmente redes recorrentes, possam capturar dependências temporais nos dados.

A otimização de hiperparâmetros é considerada uma boa prática no desenvolvimento de modelos de aprendizagem de máquina. No entanto, ela não foi diretamente implementada no código-fonte do treinamento. Esse tipo de otimização exige a execução de múltiplos treinamentos com diferentes combinações de parâmetros. Isso aumenta significativamente o custo computacional e o tempo de execução. Diante das limitações de tempo e da capacidade computacional disponível, optou-se por realizar a otimização de parâmetros específicos, como o `seq_length` e o valor de épocas durante o treinamento. Esta otimização foi realizada para compor os modelos já presentes na compilação deste projeto, cujo processo está descrito no Capítulo 4, no qual foram discutidos os resultados deste projeto.

Segundo Ycart, Benetos *et al.* (2017), configurações simples para a modelagem de dados podem ser eficazes na captura de padrões temporais e harmônicos, especialmente em tarefas de geração musical com redes LSTM, tornando desnecessária, em certos contextos, a complexidade adicional da otimização de hiperparâmetros. Portanto, para este treinamento, serão utilizados valores únicos para parâmetros como `learning_rate`, `batch_size`, entre outros.

A Listagem 5 (no Apêndice A) mostra a rotina de sequenciamento de dados utilizada para a construção das sequências, baseada na implementação realizada por Macaluso (2023) no tutorial da *TensorFlow*. O processo de sequenciamento pode ser resumido em três etapas. A primeira consiste na divisão das notas, armazenadas em um vetor único após o pré-processamento, em janelas deslizantes. A segunda etapa realiza a normalização os dados em um *range* de $[0, 1]$. Por fim, a terceira etapa realiza a segregação das notas destas subsequên-

cias em entrada e rótulo, com a entrada contendo $n-1$ notas e o rótulo contendo a última nota como o elemento a ser previsto a partir desta subsequência.

O sequenciamento em janelas deslizantes pode ser exemplificado, considerando uma base de dados com 100 notas e utilizando um valor de `seq_length` igual a 25. Essa rotina realiza a segmentação dos dados em pequenas subsequências de 25 notas, com sobreposição entre elas. Cada subsequência corresponde a uma janela deslizante ao longo da sequência original. Assim, a primeira sequência s_1 conterá as notas n_1 a n_{25}, a segunda sequência s_2 conterá as notas n_2 a n_{26}, e assim sucessivamente, até que a última janela possível seja gerada.

Antes da definição do modelo e do início do processo de treinamento, é necessário preparar o conjunto de dados sequenciados. Essa preparação garante que os dados possam ser processados de forma eficiente durante as iterações da rede neural. A Listagem 6 (no Apêndice A) apresenta o *pipeline* de preparação do *dataset*. Esse processo inclui o embaralhamento das sequências, a definição do tamanho de *batch* e o armazenamento dos dados em *cache*. Além disso, os dados são carregados antecipadamente durante o treinamento (*prefetching*), o que contribui para a redução de gargalos de entrada e saída de dados, aprimorando o aproveitamento de recursos computacionais.

A arquitetura do modelo foi definida de forma experimental e incremental durante o desenvolvimento do trabalho. A seguir, descreve-se a arquitetura final. Foram utilizadas duas camadas LSTM com 128 unidades (referentes ao alcance dos valores do campo *pitch*), com ativação *tanh*. Essas camadas foram seguidas por camadas de *dropout* com taxa de 20%, com o objetivo de evitar *overfitting*. Em seguida, foi adicionada uma camada densa com 100 unidades e ativação *ReLU*. As funções de ativação *tanh* e *ReLU* foram aplicadas em camadas distintas. A função *tanh* contribui para estabilizar os estados internos da rede, sendo adequada ao processamento de dados sequenciais. Já a função *ReLU* oferece maior eficiência computacional no treinamento e ajuda a evitar a saturação de gradientes.

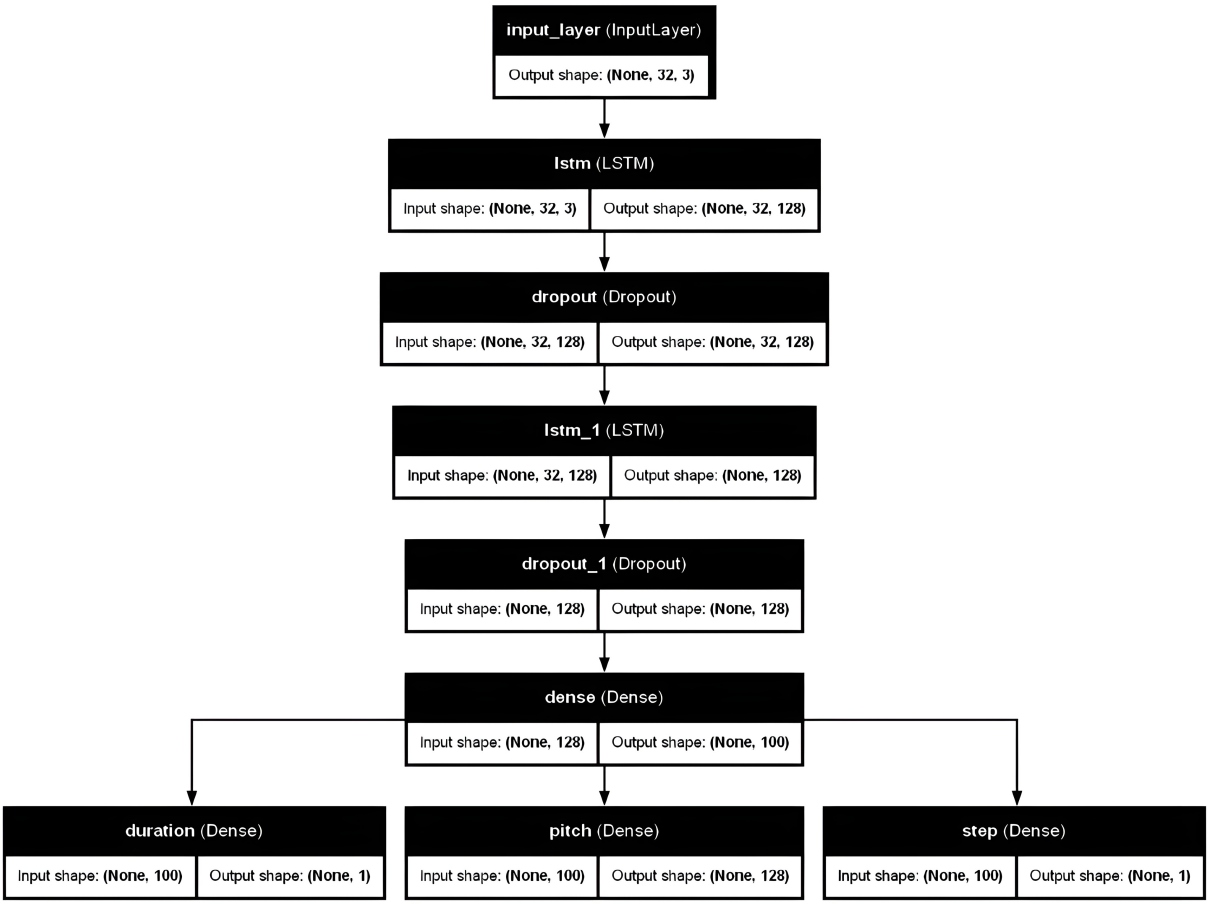
A Figura 4 mostra uma representação visual da arquitetura do modelo de treinamento, como o fluxo do mesmo e as formas de entrada e saída de cada camada descrita anteriormente.

O modelo utiliza duas funções de perda para os três tipos de dados introduzidos. A função `SparseCategoricalCrossentropy` foi aplicada ao campo *pitch*, sendo adequada para rótulos com valores inteiros. Para os campos *step* e *duration*, foi utilizada a função `mse_with_positive_pressure`, uma versão do *Mean Squared Error* (MSE) customizada para penalizar valores negativos durante o treinamento, conforme proposta por Macaluso (2023) em seu tutorial.

Além das funções de perda, o modelo adota um balanceamento específico para o campo *pitch*. Isso se deve ao fato de que, segundo os experimentos realizados, esse componente é o que mais contribui para o aumento da taxa de perda total, que é composta pela soma das perdas de *pitch*, *step* e *duration*. Para mitigar esse impacto, foi atribuído o valor de 0,05 como peso da perda para o *pitch*, enquanto os campos *step* e *duration* mantiveram o peso padrão igual a 1.

As Listagens 7 e 8 (ambas no Apêndice A) mostram, respectivamente, a implementação do modelo e a função de *loss* utilizada para os valores *step* e *duration*.

Figura 4 – Representação visual da arquitetura do modelo.



Fonte: Autoria Própria (2025).

Para o treinamento do modelo, foi utilizado o valor de épocas fornecido na entrada do programa. Como padrão, foi adotado o valor 100, sustentando o argumento de Mauthes (2018), que obteve resultados mais satisfatórios com um maior número de épocas. Durante o treinamento, foram utilizadas *callbacks* para customizar o processo. Entre elas, destacam-se os *checkpoints*, que salvam os pesos mais relevantes de cada época, e o *EarlyStopping*, que interrompe o treinamento caso não haja redução na perda após um número definido de épocas. Essa configuração é apresentada na Listagem 9 (no Apêndice A). Ao final do treinamento, o modelo é salvo em disco no formato *.keras*, juntamente com seus metadados no formato *json*, para posterior uso na rotina de geração de melodias.

3.4.2 Geração de Melodias

A etapa de geração tem como responsabilidade o carregamento dos modelos treinados previamente e a geração das melodias baseadas na entrada do usuário. Essas funcionalidades estão implementadas no *script* `generate.py`.

Para isso, o método `generate_notes` carrega os dados de treinamento correspondentes à emoção selecionada pelo usuário. Em seguida, prepara os dados de entrada, inse-

rindo valores vazios até que tenham o mesmo tamanho das sequências utilizadas no modelo de treino.

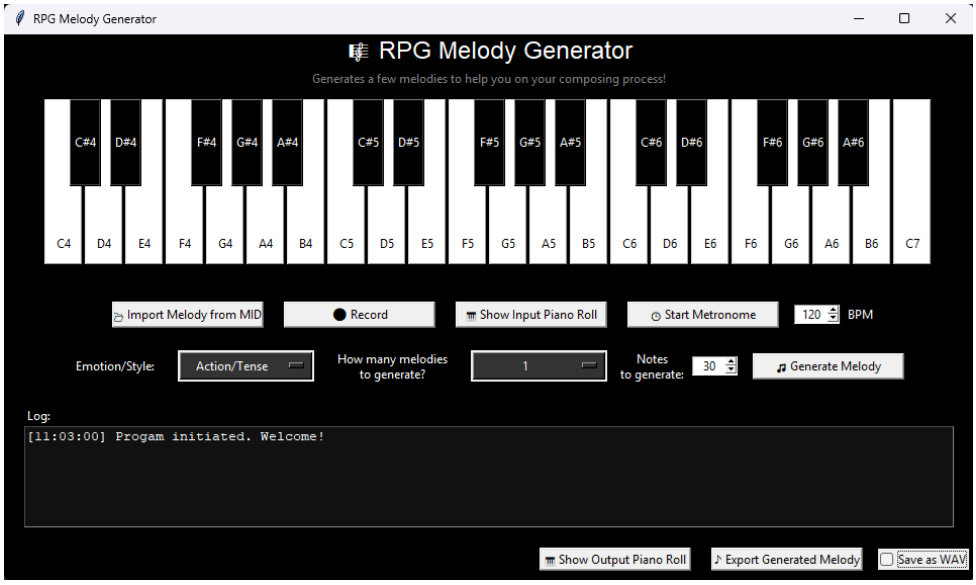
Por fim, realiza a predição das notas por meio da função `predict_next_note`, conforme o número de notas especificado pelo usuário.

A função `predict_next_note` é responsável por realizar a predição de uma nova nota musical com base na entrada fornecida ao modelo. A partir das saídas do modelo para *pitch*, *step* e *duration*, a função aplica uma amostragem probabilística na saída de *pitch*, ajustada pelo parâmetro de *temperature*, que controla a variabilidade das predições. Em seguida, os valores são convertidos para tipos primitivos e normalizados para garantir que sejam não negativos, retornando assim uma nova nota representada como tupla. A Listagem 10 (no Apêndice A) mostra o código-fonte desta rotina de geração de uma nota também inspirada no tutorial de Macaluso (2023).

3.5 Implementação da Interface Gráfica

A implementação da interface gráfica foi realizada utilizando a biblioteca `Tkinter`, apresentando uma interface simples e intuitiva para guiar o usuário nas funcionalidades do sistema. A Figura 5 apresenta a interface gráfica desenvolvida, ilustrando a organização dos principais componentes do aplicativo.

Figura 5 – Tela principal da interface gráfica.



Fonte: Autoria Própria (2025).

Conforme visto na figura, a interface foi dividida em quatro partes, sendo elas:

- **Entrada de dados:** Esta parte é composta por um teclado virtual interativo, que permite ao usuário introduzir uma melodia usando o *mouse* ou o teclado. Também oferece opções de apoio à entrada, como controle de gravação e metrônomo. Além disso, o sistema disponibiliza a opção de carregar uma melodia a partir de um arquivo MIDI.

- **Parâmetros para a geração:** Nesta parte, o usuário pode configurar os parâmetros de geração da melodia a partir da entrada fornecida. É possível definir o estilo, a quantidade de melodias a serem geradas e o número de notas.
- **Caixa de texto para *logging* de eventos:** esta parte possui uma caixa de texto para registrar os eventos do sistema na interface.
- **Gerenciamento pós-geração:** Nesta parte, o sistema oferece opções para visualização gráfica e exportação das melodias geradas para arquivos MIDI, permitindo seu uso em outras finalidades, como a manipulação em *softwares* de edição de áudio.

3.5.1 Teclado Virtual Interativo

O código-fonte `piano_ui.py` implementa um teclado virtual que permite ao usuário inserir uma melodia inicial de forma interativa. A interface simula o funcionamento de um piano, fornecendo retorno sonoro em tempo real e facilitando a experimentação musical. A Listagem 11 (no Apêndice A) descreve como é feito o desenho da interface dentro da classe `PianoUI`.

A função deste teclado virtual não é apenas sonorizar a entrada, mas também registrá-la no sistema. As funções `self.on_press` e `self.on_release` registram as notas inseridas, reproduzem ou interrompem sua execução sonora e realizam a conversão para os formatos *pitch*, *step* e *duration*, permitindo a visualização na interface gráfica e o processamento na geração de novas melodias.

Juntamente ao teclado virtual, a interface principal inclui a função de gravação, que registra a entrada do usuário no sistema, e a função de metrônomo, que reproduz batidas sonoras regulares para auxiliar na manutenção do ritmo durante a execução da melodia, facilitando a precisão temporal da entrada musical.

3.5.2 Fluxo de Dados

O gerenciamento dos dados de entrada e saída do sistema é centralizado na biblioteca `state`, que funciona como uma estrutura global que compartilha os dados pelas funcionalidades da interface gráfica. Por meio dele, são mantidas em variáveis a melodia de entrada fornecida pelo usuário e as melodias geradas pelo sistema, permitindo que essas informações estejam acessíveis de forma consistente em toda a aplicação. A Listagem 12 (no Apêndice A), originada do *script* `handlers.py`, que contém o funcionamento de diversos botões presentes na interface, mostra um exemplo de como esta estrutura é utilizada pela interface responsável pelo teclado digital para armazenar a melodia de entrada do usuário.

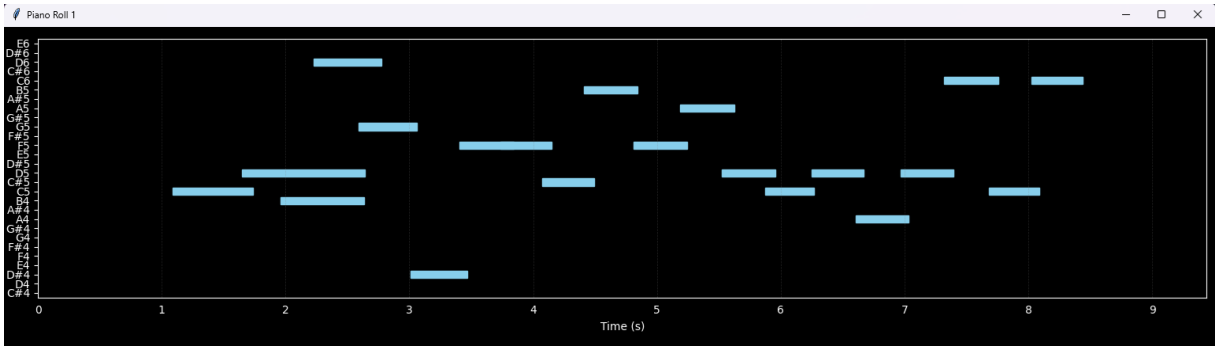
3.5.3 Configuração de Parâmetros para a Geração

A interface permite ao usuário configurar os principais parâmetros que influenciam diretamente o processo de geração de melodias. Entre eles estão a escolha do estilo ou emoção desejada, que define qual modelo treinado será utilizado para orientar a geração musical. O usuário também pode definir o número de melodias a serem geradas em uma única execução (de uma a cinco) e a quantidade de notas em cada melodia (entre dez e cinquenta). Após configurar esses atributos e acionar o botão `Generate Melody`, o módulo de geração é ativado, recebendo os dados da interface gráfica. As Listagens 13 e 14 (ambas no Apêndice A) exemplificam, respectivamente, a configuração do botão `Generate Melody` e a função responsável por sua rotina, localizada no *script* `handlers.py`.

3.5.4 Visualização de Dados

O sistema oferece uma visualização gráfica das melodias por meio do formato *Piano Roll*, implementada com o auxílio da biblioteca `Matplotlib`. Essa visualização está disponível tanto para a melodia de entrada fornecida pelo usuário quanto para as melodias geradas pelo modelo, permitindo uma análise clara da disposição temporal e da distribuição das notas. Além disso, o sistema possibilita a exportação das melodias geradas, tanto no formato de som WAVE de forma que elas possam ser reproduzidas, quanto no formato MIDI, tornando-as compatíveis com diversos *softwares* de edição e produção musical, o que possibilita o uso e a manipulação das melodias geradas. A Listagem 15 (no Apêndice A) e a Figura 6 mostram, respectivamente, um trecho do código onde é gerada a representação das notas e um exemplo de saída do *software*.

Figura 6 – Representação de uma melodia gerada pelo sistema.



Fonte: Autoria Própria (2025).

4 RESULTADOS

Neste capítulo são avaliados os efeitos do pré-processamento no tamanho da base de dados, o processo de treinamento do modelo e a sonoridade das melodias geradas. O parâmetro avaliado foi `seq_length`. Este parâmetro se refere ao tamanho das sequências nas quais as notas contabilizadas no total foram separadas, mantendo a propriedade de aprendizado sequencial da LSTM. Foram avaliados os valores 25, 32 e 50 para `seq_length`.

O parâmetro `seq_length` influencia no resultado por reunir uma quantidade específica de notas em uma única sequência, podendo influenciar no aprendizado do modelo. Por exemplo, músicas lentas e de pequena duração podem possuir poucas notas, contendo padrões em sequências menores. Músicas de maior duração e um arranjo acelerado podem possuir várias notas, dando possibilidade do modelo aprender com maior precisão através de sequências maiores.

4.1 Análise do Pré-processamento

Ao todo, a base de dados conta com 317 arquivos MIDI distribuídos e categorizados entre os nove rótulos, totalizando uma média de 5.246 notas por rótulo a serem processadas. O rótulo Batalha possui o maior número de notas antes do processamento, com 12.557 notas, enquanto o rótulo Cômico possui o menor número de notas, contabilizando 1.147 notas.

4.1.1 Analise Quantitativa das Melodias

Após o pré-processamento utilizando `seq_length = 25`, observou-se uma redução total de 23,38% na quantidade de notas, com variações entre aproximadamente 9,54% (Melancólico) e 28,86% (Batalha) entre os rótulos. Com `seq_length = 32`, a redução total foi de 21,45%, com a variação individual entre aproximadamente 2,35% (Cômico) e 28,65% (Batalha). No entanto, ao utilizar `seq_length = 50`, a perda do total de notas foi visivelmente menor: a redução total foi de 16,44%, com a variação percentual por emoção entre aproximadamente 2,91% (Aventura) e 25,55% (Batalha).

Além disso, o uso de `seq_length = 50` implicou no descartamento de aproximadamente 4% dos arquivos originais, por não atingirem o comprimento mínimo exigido. Esse foi o maior índice de descarte entre os três experimentos, comparado a nenhum descarte com `seq_length = 25` e 1,5% com `seq_length = 32`. Assim, pode-se dizer que o parâmetro `seq_length = 50` foi o que menos perdeu notas para geração entre os outros dois valores, apesar do descarte de melodias mais curtas.

4.1.2 Analise Qualitativa das Melodias

A análise dos dados pré-processados revelou variações significativas na proporção de melodias que passaram por modificações, como transposição tonal e transposição de oitavas, dado o parâmetro `seq_length`.

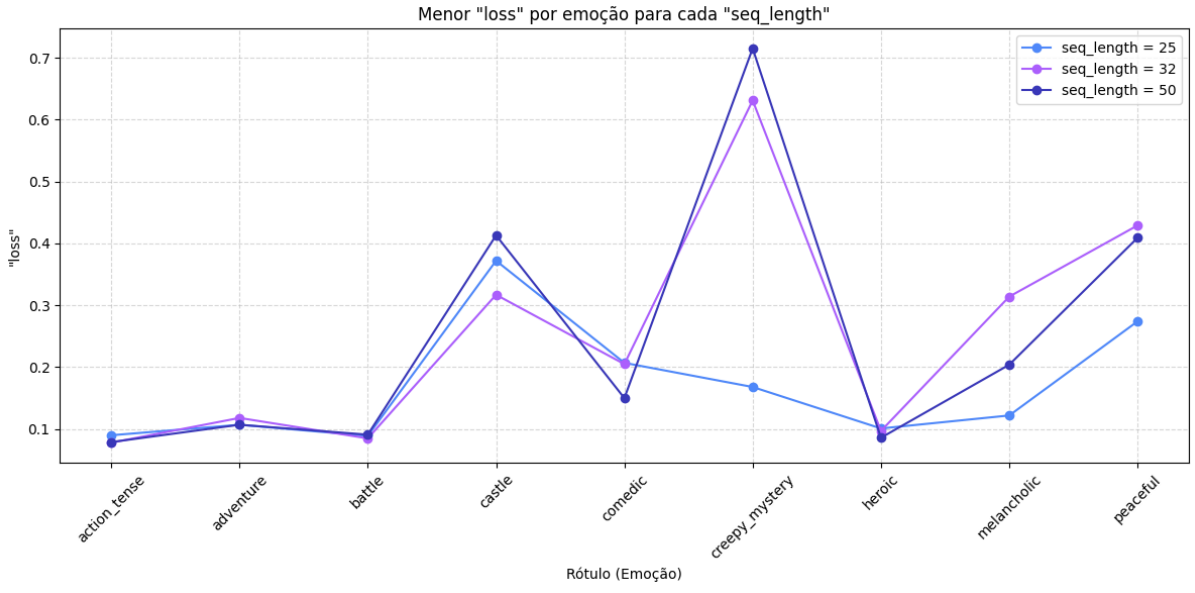
Para `seq_length` = 25, cerca de 85% das melodias foram modificadas: 45% passaram por transposição tonal, 7% apenas por transposição de oitavas e 33% por ambos os tipos. Com `seq_length` = 32, 86% das melodias foram alteradas, sendo 46% transpostas apenas em tonalidade, 8% apenas em oitava e 32% em ambas as transposições. Já para `seq_length` = 50, 87% das melodias foram alteradas, com 48% sofrendo transposição tonal, 9% apenas de oitava e 30% em ambas as dimensões.

Os resultados indicam que valores maiores de `seq_length` podem exigir uma normalização maior das melodias, por mais equilibrado que o apuramento das melodias se apresentam, para as adequar à estrutura temporal imposta por esse parâmetro. Pode-se concluir isto considerando que melodias longas podem ter maior variação em suas estruturas do que melodias curtas.

4.2 Desempenho do Treinamento do Modelo

Para avaliar o desempenho geral do modelo, foi feita a sua execução utilizando os três valores distintos para o parâmetro `seq_length` com os nove rótulos do modelo, totalizando 27 execuções. A Figura 7 mostra uma representação visual comparando o desempenho do modelo para cada valor de `seq_length`. É importante salientar que, quanto menor o *loss* (perda), melhor é o modelo.

Figura 7 – *Loss* dos modelos treinados



Fonte: Autoria própria (2025).

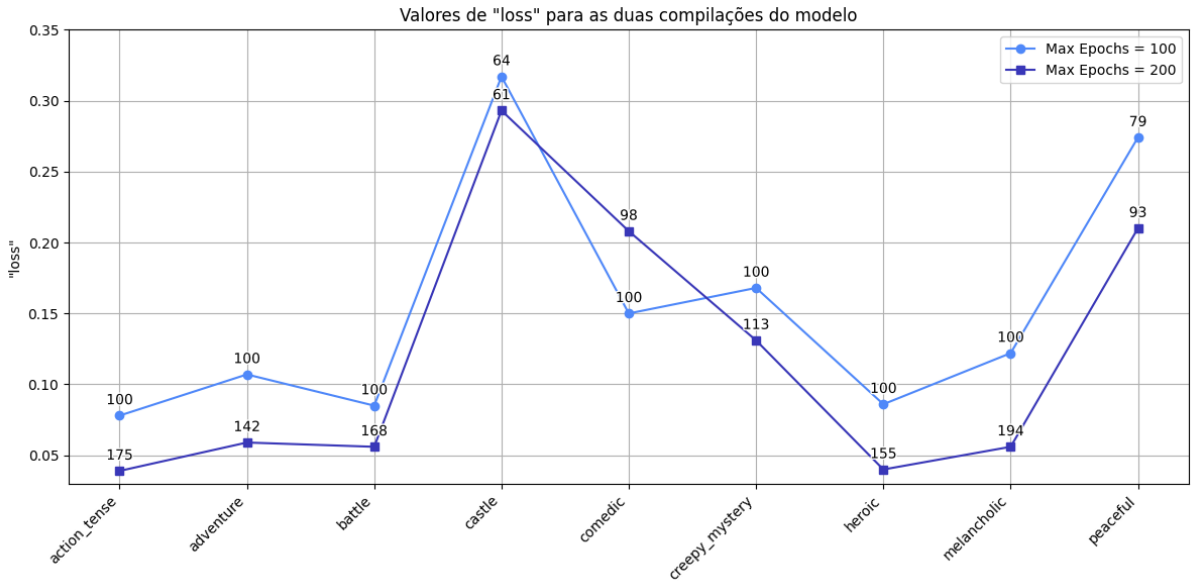
Após a execução dos treinamentos para cada valor de `seq_length`, é possível notar os rótulos que obtiveram o melhor desempenho para cada valor desta variável. Com `seq_length = 25`, os rótulos que apresentaram menor perda no treinamento foram as emoções de Aventura, Mistério, Melancólico e Tranquilidade. Já o valor de `seq_length = 32`, os melhores modelos foram Ação, Batalha e Castelo. Por fim, os melhores modelos para `seq_length = 50` foram Cômico e Heroico.

Juntamente às taxas de perda, pode-se analisar que dentre as execuções realizadas, 40,7% destas tiveram seu melhor desempenho obtido antes do modelo atingir o total de 100 épocas, cujo valor é definido por padrão. Dentre estas ocorrências, 45,4% atingiram seu melhor resultado antes das 60 primeiras épocas. Isto pode ocorrer tanto devido à limitação da base de dados de um rótulo em específico, ao possuir uma quantidade menor de arquivos à disposição, quanto pelo parâmetro `seq_length` que, utilizando o rótulo Mistério como exemplo, desempenhou bem com 25 valores por sequência, mas não desempenhou bem com os valores 32 e 50.

Um exemplo para o primeiro caso é o rótulo Castelo, que é uma das emoções com a menor quantidade de músicas, com apenas 15 arquivos MIDI à disposição do modelo. Para cada valor de `seq_length` (25, 32 e 50) o modelo considerou o treinamento das épocas, respectivamente, 64, 65 e 43. Um exemplo que se refere ao segundo caso é o rótulo Mistério, que, para cada valor, as melhores épocas foram, respectivamente, 100, 45 e 23, indicando que o modelo se aproveitou melhor da representação mais curta das melodias.

Em uma segunda etapa de otimização, foram usados os casos que apresentaram menor taxa de perda adquiridos na avaliação anterior, e juntamente a isto, foi considerado o valor máximo de épocas para 200, a fim de que o treinamento dos modelos tivesse um aproveitamento maior com seus valores otimizados do parâmetro `seq_length`. O gráfico descrito na Figura 8 mostra a comparação do desempenho dos melhores modelos para cada iteração com seus valores máximos de épocas, juntamente ao número da época que as taxas de perda foram consideradas (número acima dos pontos).

Figura 8 – Gráfico de desempenho das duas execuções do modelo.



Fonte: Autoria própria (2025).

Após esta nova etapa de treinamento, os modelos apresentaram uma melhora significativa nas taxas de *loss* para a maioria dos modelos. Junto a isto, pode-se observar que, dos nove modelos treinados, 66% tiveram seus melhores resultados obtidos após a época 100.

4.3 Análise da Sonoridade das Melodias Geradas

Após a compilação dos modelos, eles foram utilizados para gerar melodias de acordo com o rótulo desejado. Embora os resultados na construção dos modelos tenham sido positivos, a sonoridade das melodias geradas não foi satisfatória. Para a geração, foram utilizadas tanto a interface do piano virtual interativo quanto melodias da base de dados. A Tabela 3 apresenta as características das melodias obtidas neste experimento, destacando suas principais qualidades e limitações.

A análise das melodias geradas pelo modelo revelou alguns comportamentos recorrentes entre diferentes rótulos. Um deles foi a repetição de notas ao final de cada melodia. Outro padrão observado foi o início com notas mais graves, seguido por notas mais agudas, mantendo tons constantes e com pouca variação.

Pode-se notar a quantidade insuficiente de dados para os rótulos como um dos fatores que contribuem para a dissonância e a geração de estruturas desorganizadas. Observando os resultados do rótulo *Castelo*, que é o segundo com menor número de amostras disponíveis para o modelo, percebe-se a presença de desorganização e dissonância nas notas geradas. Entretanto, ao comparar com o rótulo *Cômico*, que possui a menor quantidade de amostras entre todos os rótulos, nota-se que, apesar da inconsistência no posicionamento das notas, os resultados se aproximam mais de uma melodia do que os obtidos para o rótulo *Castelo*.

Tabela 3 – Análise qualitativa das melodias para cada rótulo

Estilo/Emoção	Estrutura	Notas
Ação	Se assemelha a uma melodia	Têm variedade mas não trazem sentido musical
Aventura	Se assemelha a uma melodia	Pouca variedade nos tons, rapidamente se estabilizam em sequencias de uma nota só
Batalha	Desorganizada, notas se sobrepõem	Tons dissonantes e fora de escala
Castelo	Desorganizada, notas se sobrepõem	Tons concentrados no primeiro segundo de execução, durações aleatórios
Cômico	Se assemelha a uma melodia	Algumas notas se sobrepõem, tons têm variedade e apresentam escala musical
Heroico	Desorganizada, se assemelha a um acorde	Notas concentradas, com diversos tons sem sentido musical
Melancólico	Se assemelha a uma melodia	Pouca variedade nos tons, apresentam escala musical, e rapidamente se estabilizam em sequências de uma nota só
Mistério	Se assemelha a uma melodia	Notas curtas, andamento da melodia acelera e se estabiliza em sequências de uma nota só
Tranquilidade	Se assemelha a uma melodia	Pouca variedade nos tons, rapidamente se estabilizam em sequencias de uma nota só

Fonte: Autoria própria (2025).

Considerando isso, a quantidade insuficiente de dados para os rótulos pode ser considerada um dos fatores que contribuem para as inconsistências, mas não se pode generalizar essa causa para todos os rótulos.

Para apresentar algumas das saídas geradas pelo sistema, foi criado um repositório[1] que disponibiliza amostras em formato de áudio (WAVE), contendo três exemplos para cada rótulo da base de dados.

[1] https://github.com/gustavofavaro/rpg_melody_generator_samples

5 CONCLUSÕES

O objetivo deste trabalho foi desenvolver uma aplicação que usa inteligência artificial para auxiliar na composição de melodias. A aplicação permite ao usuário selecionar uma emoção que gostaria de evocar com a melodia e entrar com uma sequência inicial de notas. A partir destas entradas, o modelo gera as próximas notas.

Apesar das limitações encontradas no decorrer do desenvolvimento, pode-se concluir que é possível construir um sistema gerador de melodias. Durante esse processo, foi possível compreender a estrutura dos dados que compõem a base do sistema. Essa compreensão permitiu seu uso no treinamento dos modelos de geração de melodias, com a aplicação de conhecimentos musicais e técnicos, visando obter resultados mais próximos do desejado. Mesmo com as etapas de otimização no processamento dos dados e na implementação do modelo de aprendizagem, os resultados não foram satisfatórios. As melodias geradas apresentaram sons que fogem da estrutura musical esperada e incluíram notas sem padrões consistentes extraídos da base de dados.

5.1 Trabalhos Futuros

Para a resolução dos problemas apresentados neste trabalho e a implementação do sistema, existem caminhos alternativos que podem levar a soluções diferentes e possivelmente melhores. Uma das recomendações é a utilização de outras arquiteturas de redes neurais, como o *Transformer*, que também foi projetado para lidar com sequências. O *Transformer* apresenta vantagens em relação ao LSTM, como maior adaptabilidade a bases de dados extensas, como sua maior capacidade de generalização e paralelismo no processamento de sequências, como afirma Khan *et al.* (2022). No entanto, neste trabalho optou-se pelo uso do LSTM devido ao seu custo computacional substancialmente menor em comparação com uma arquitetura baseada em *Transformer*.

Outra possibilidade seria a inclusão de mais bases de dados com músicas de jogos do gênero RPG de outros *video-games* legados. A adição de mais melodias pode fornecer maior variedade de dados a serem explorados e mais material para o modelo de geração se basear. Isso é especialmente relevante, já que a escassez de dados foi um dos principais fatores que influenciaram os resultados insatisfatórios do sistema.

A combinação do uso do *Transformer* com a ampliação da base de dados pode trazer resultados melhores em trabalhos que abordem a geração de música simbólica. Esse modelo é relativamente superior ao LSTM quando se trata de bases de dados mais extensas e apresenta desempenho geral mais consistente.

5.2 Considerações Finais

O desenvolvimento deste trabalho exigiu um esforço significativo para compreender o uso de ferramentas de tratamento e geração de dados com inteligência artificial. Esse processo representou uma oportunidade de aprofundar o aprendizado em uma área em crescimento, tanto no mercado de trabalho quanto no meio acadêmico, contribuindo para a consolidação de conhecimentos e habilidades essenciais para futuras atuações e pesquisas no campo.

REFERÊNCIAS

ABADI, M. *et al.* Tensorflow: A system for large-scale machine learning. *In*: 12TH USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION (OSDI 16). 2016. **Anais [...]** [*s.n.*], 2016. p. 265–283. Disponível em: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

CHOMSKY, N. **Syntactic structures**. [*S.l.*]: Mouton de Gruyter, 2002.

DONAHUE, C. *et al.* Lakhnes: Improving multi-instrumental music generation with cross-domain pre-training. **arXiv preprint arXiv:1907.04868**, „, 2019.

DONAHUE, C.; MAO, H. H.; MCAULEY, J. The nes music database: A multi-instrumental dataset with expressive performance attributes. **arXiv preprint arXiv:1806.04278**, „, 2018.

EDDY, S. R. Hidden markov models. **Current Opinion in Structural Biology**, v. 6, n. 3, p. 361–365, 1996. ISSN 0959-440X. Disponível em: https://www.sciencedirect.com/science/article/pii/S0959440X9680056X.

FACELI, K. *et al.* Inteligência artificial: uma abordagem de aprendizado de máquina, „,. 2021.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [*S.l.*]: MIT Press, 2016. http://www.deeplearningbook.org.

HAYKIN, S. **Redes neurais: princípios e prática**. [*S.l.*]: Bookman Editora, 2001.

HIRZEL, M.; SOUKUP, D. Project writeup for csci 5832 natural language processing. **University of Colorado**, „, 2000.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, MIT press v. 9, n. 8, p. 1735–1780, 1997.

HORNER, A.; GOLDBERG, D. E. **Genetic algorithms and computer-assisted music composition**. [*S.l.*]: Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 1991. v. 51.

KHAN, S. *et al.* Transformers in vision: A survey. **ACM computing surveys (CSUR)**, ACM New York, NY v. 54, n. 10s, p. 1–41, 2022.

KIRNBERGER, J. P. **Der allezeit fertige Polonoisen-und Menuettenkomponist**. [*S.l.*: *s.n.*], 1767.

MACALUSO, J. **Generate music with an RNN**, . 2023. Disponível em: https://www.tensorflow.org/tutorials/audio/music_generation\#create_and_train_the_model.

MAUTHES, N. Vgm-rnn: Recurrent neural networks for video game music generation. **San José State University**, „, 2018. Disponível em: https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1606&context=etd_projects.

MITCHELL, M. **An introduction to genetic algorithms**. [*S.l.*]: MIT press, 1998.

NIERHAUS, G. **Algorithmic composition: paradigms of automated music generation**. [*S.l.*]: Springer Science & Business Media, 2009.

Nintendo Fandom. **List of Nintendo Entertainment System role-playing games**, . 2025. https://nintendo.fandom.com/wiki/List_of_Nintendo_Entertainment_System_role-playing_games. [Ultimo acesso em: 19 maio 2025].

PLUT, C.; PASQUIER, P. Generative music in video games: State of the art, challenges, and prospects. **Entertainment Computing**, Elsevier v. 33„ p. 100337, 2020.

ROSSUM, G. V.; JR, F. L. D. **Python tutorial**. [*S.l.*]: Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995. v. 620.

ROTHSTEIN, J. **MIDI: A comprehensive introduction**. [*S.l.*]: AR Editions, Inc., 1995. v. 7.

SILVA, I. D.; SPATTI, D.; FLAUZINO, R. **REDES NEURAIS ARTIFICIAIS PARA ENGENHARIA E: CIENCIAS APLICADAS - CURSO PRÁTICO**. ARTLIBER, 2016. ISBN 9788588098534. Disponível em: https://books.google.com.br/books?id=w2VHbwAACAAJ.

YCART, A.; BENETOS, E. *et al.* A study on lstm networks for polyphonic music sequence modelling. *In*: ISMIR. 2017. **Anais [...]** [*S.l.*], 2017.

APÊNDICE A – Códigos-fonte do sistema desenvolvido

Listagem 2 – Normaliza o tom e o *range* das notas

```python
def normalize_notes(pitches: list[int], key_mode: str):
    pitch_classes = np.array(pitches) % 12
    histogram = np.bincount(pitch_classes, minlength=12)

    # Define template e alvo de pitch class conforme o modo
    if key_mode == 'major':
        template = np.array([1 if i in [0, 2, 4, 5, 7, 9, 11] else 0 for i in range(12)])
        target_pc = 0  # C
    elif key_mode == 'minor':
        template = np.array([1 if i in [0, 2, 3, 5, 7, 8, 10] else 0 for i in range(12)])
        target_pc = 9  # A

    # Estima o tom atual da melodia (pitch class dominante)
    best_score = -np.inf
    best_key = 0
    for i in range(12):
        rotated = np.roll(template, i)
        score = np.dot(histogram, rotated)
        if score > best_score:
            best_score = score
            best_key = i

    # Aplica a transposicao para a tonica desejada
    tonal_shift = (target_pc - best_key) % 12
    transposed = [p + tonal_shift for p in pitches]

    # Ajusta para a oitava predominante (oitava 5)
    octaves = [(p // 12) - 1 for p in transposed]
    predominant_octave = max(set(octaves), key=octaves.count)
    target_octave = 5
    octave_shift = (target_octave - predominant_octave) * 12
    transposed = [p + octave_shift for p in transposed]

    return transposed
```

Fonte: Autoria própria (2025).

Listagem 3 – Exemplo de execução do *script* de treinamento via terminal

```
$ python core/train_model.py --emotion battle --seq_length 32 --epochs 100
```

Fonte: Autoria própria (2025).

Listagem 4 – Carga e Pre-processamento de um Arquivo da Base de Dados

```python
def midi_to_pandas_notes(midi_file: str, key_mode: str = 'major', seq_length: int = 16):
    music = pretty_midi.PrettyMIDI(midi_file)
    note_list = collections.defaultdict(list)
    prev_start = 0

    title_codes = ['047', '050', '061', '104', '155', '233', '246']
    code = os.path.basename(midi_file)[:3]

    instrument = 1 if code in title_codes and len(music.instruments) > 1 else 0
    notes = music.instruments[instrument].notes

    for note in notes:
        start = note.start
        end = note.end
        if end - start < 0.02: continue

        note_list['pitch'].append(note.pitch)
        note_list['step'].append(start - prev_start)
        note_list['duration'].append(end - start)
        prev_start = start

    n_filtered_notes = len(note_list['pitch'])
    if n_filtered_notes < seq_length * 3:
        added_notes = notes if n_filtered_notes <= seq_length else notes[:int(len(notes)/2)]
        end_time = music.get_end_time()

        for note in added_notes:
            start = note.start + end_time
            end = note.end + end_time
            if end - start < 0.05: continue

            note_list['pitch'].append(note.pitch)
            note_list['step'].append(start - prev_start)
            note_list['duration'].append(end - start)
            prev_start = start

    # Tira as notas que excedem um valor multiplo do seq_length
    note_diff = len(note_list['pitch']) % seq_length
    if note_diff:
        for key in note_list:
            note_list[key] = note_list[key][:-note_diff]

    if len(note_list['pitch']) < seq_length:
        return pd.DataFrame()

    note_list['pitch'] = normalize_notes(note_list['pitch'], key_mode)

    return pd.DataFrame(note_list)
```

Fonte: Autoria própria (2025).

Listagem 5 – Rotina de criação de sequências de dados do *dataset*

```
def create_sequences(dataset: tf.data.Dataset, seq_length: int, vocab_size =
    128) -> tf.data.Dataset:
    seq_length += 1

    windows = dataset.window(seq_length, shift = 1, stride = 1,
    drop_remainder = True)

    flatten = lambda x: x.batch(seq_length, drop_remainder = True)
    sequences = windows.flat_map(flatten)

    def scale_pitch(x):
        x = x/[vocab_size,1.0,1.0]
        return x

    def split_labels(sequences):
        inputs = sequences[:-1]
        labels_dense = sequences[-1]
        labels = {key: labels_dense[i] for i, key in enumerate(key_order)}

        return scale_pitch(inputs), labels

    return sequences.map(split_labels, num_parallel_calls=tf.data.AUTOTUNE)

vocab_size = 128 # Range de valores para o campo pitch
seq_ds = create_sequences(notes_ds, seq_length, vocab_size)
```

Fonte: Adaptado de Macaluso (2023).

Listagem 6 – Preparação do *dataset*

```
batch_size = 64
buffer_size = len(all_notes) - seq_length
train_ds = (seq_ds
            .shuffle(buffer_size)
            .batch(batch_size, drop_remainder=True)
            .cache()
            .prefetch(tf.data.experimental.AUTOTUNE))
```

Fonte: Adaptado de Macaluso (2023).

Listagem 7 – Implementação do modelo de treinamento

```
input_shape = (seq_length, 3)
learning_rate = 0.005

inputs = tf.keras.Input(shape=input_shape)

# Primeira camada LSTM
x = tf.keras.layers.LSTM(128, return_sequences=True, activation='tanh')(
    inputs)
x = tf.keras.layers.Dropout(0.2)(x)
# Segunda camada LSTM
x = tf.keras.layers.LSTM(128, activation='tanh')(x)
x = tf.keras.layers.Dropout(0.2)(x)
# Camada intermediaria
x = tf.keras.layers.Dense(100, activation='relu')(x)

outputs = {
  'pitch': tf.keras.layers.Dense(128, name='pitch')(x),
  'step': tf.keras.layers.Dense(1, name='step')(x),
  'duration': tf.keras.layers.Dense(1, name='duration')(x),
}

model = tf.keras.Model(inputs, outputs)

loss = {
      'pitch': tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
    True),
      'step': mse_with_positive_pressure,
      'duration': mse_with_positive_pressure,
}

loss_weights = {
    'pitch': 0.05,
    'step': 1.0,
    'duration': 1.0,
}

optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

model.compile(
    loss=loss_functions,
    loss_weights=loss_weights,
    optimizer=optimizer,
)
```

Fonte: Autoria própria (2025).

Listagem 8 – Função de perda *Mean Squared Error* com punição a valores negativos

```
@register_keras_serializable()
def mse_with_positive_pressure(y_true: tf.Tensor, y_pred: tf.Tensor):
    mse = (y_true - y_pred) ** 2
    positive_pressure = 10 * tf.maximum(-y_pred, 0.0)
    return tf.reduce_mean(mse + positive_pressure)
```

Fonte: Adaptado de Macaluso (2023).

Listagem 9 – Execução do modelo de treinamento

```
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath='./temp/checkpoints/ckpt_{epoch}.weights.h5',
        save_weights_only=True),
    tf.keras.callbacks.EarlyStopping(
        monitor='loss',
        patience=10,
        verbose=1,
        restore_best_weights=True),
]

history = model.fit(
    train_ds,
    epochs=epochs,
    callbacks=callbacks,
)
```

Fonte: Autoria própria (2025).

Listagem 10 – Rotina de predição de uma nota

```python
def predict_next_note(notes: np.ndarray, model: tf.keras.Model, temperature: float = 2.0) -> tuple[int, float, float]:
    assert temperature > 0

    predictions = model.predict(notes, verbose=0)
    pitch_logits = predictions['pitch']
    step = predictions['step']
    duration = predictions['duration']

    # pitch
    pitch_logits /= temperature  # controla aleatoriedade
    pitch = tf.random.categorical(pitch_logits, num_samples=1)
    pitch = tf.squeeze(pitch, axis=-1)

    # step e duration
    step = tf.squeeze(step, axis=-1)
    duration = tf.squeeze(duration, axis=-1)

    # Adiciona ruido proporcional a temperatura
    step += tf.random.normal(shape=step.shape, mean=0.0, stddev=0.01 * temperature)
    duration += tf.random.normal(shape=duration.shape, mean=0.0, stddev=0.01 * temperature)

    # garante step e duration nao-negativos
    step = tf.maximum(0, step)
    duration = tf.maximum(0, duration)

    return int(pitch), float(step), float(duration)
```

Fonte: Adaptado de Macaluso (2023).

Listagem 11 – Código de geração da interface do teclado virtual

```
def create_keys(self):
    for i, key in enumerate(white_keys):
        btn = tk.Button(self.frame, text=key, bg='white', fg='black',
                        height=white_key_h // 20,
                        width=white_key_w // 10,
                        anchor='s', pady=10)
        btn.place(x=i * white_key_w, y=0)
        self.note_buttons[key] = btn
        btn.bind('<ButtonPress-1>', lambda e,
                 note=key: self.on_press(note))
        btn.bind('<ButtonRelease-1>', lambda e,
                 note=key: self.on_release(note))

    for i, key in enumerate(black_keys):
        if key: # Pula teclas pretas vazias ("")
            btn = tk.Button(self.frame, text=key, bg='black', fg='white',
                            height=black_key_h // 20,
                            width=black_key_w // 10)
            btn.place(x=(i*white_key_w)+(white_key_w-black_key_w // 2),
                      y=0)
            self.note_buttons[key] = btn
            btn.bind('<ButtonPress-1>', lambda e,
                     note=key: self.on_press(note))
            btn.bind('<ButtonRelease-1>', lambda e,
                     note=key: self.on_release(note))
```

Fonte: Autoria própria (2025).

Listagem 12 – Rotina do botão `Stop Recording`

```
def handle_stop_recording(piano, record_btn, stop_btn, log):
    piano.stop_recording()                  # controle do piano_ui
    stop_recording(record_btn, stop_btn)  # controle da UI principal

    state.input_melody = piano.get_normalized_notes()
    log("Recording stopped and saved.")
```

Fonte: Autoria própria (2025).

Listagem 13 – Atribuição do *frame* para geração e botão para gerar melodias

```
# Frame de parametros para a geracao
general_control_frame = tk.Frame(root, bg="black")
general_control_frame.pack(side="top", pady=10)

# ... #

# Botao de gerar melodias
tk.Button(
    general_control_frame, text="Generate Melody", width=20,
    command=lambda: handle_generate_melody(selected_emotion.get(),
    melody_count.get(), notes_var.get(), log)
).pack(side="right", padx=10)
```

Fonte: Autoria própria (2025).

Listagem 14 – *Handler* do botão `Generate Melody`

```
def handle_generate_melody(sentiment, melody_count, num_notes, log):
    def handle():
        state.output_melodies = []

        if not state.input_melody:
            messagebox.showerror('Error', 'No input melody to generate from.')
            return

        log(f'Generating {melody_count} melodies with {sentiment} emotion...')
        for _ in range(melody_count):
            generated_melody = generate_notes(sentiment, state.input_melody, num_predictions=num_notes)
            generated_melody['pitch'] = transpose_octave(generated_melody['pitch'].tolist())
            state.output_melodies.append(generated_melody)

        log(f'{melody_count} melodies generated.')
        messagebox.showinfo('Complete', f'{melody_count} melodies generated.')

    threading.Thread(target=handle, daemon=True).start()
```

Fonte: Autoria própria (2025).

Listagem 15 – Plotagem das notas em formato *Piano Roll*

```
def handle_generate_melody(sentiment, melody_count, num_notes, log):
    def handle():
        state.output_melodies = []

        if not state.input_melody:
            messagebox.showerror('Error', 'No input melody to generate from.')
            return

        log(f'Generating {melody_count} melodies with {sentiment} emotion...')
        for _ in range(melody_count):
            generated_melody = generate_notes(sentiment, state.input_melody, num_predictions=num_notes)
            generated_melody['pitch'] = transpose_octave(generated_melody['pitch'].tolist())
            state.output_melodies.append(generated_melody)

        log(f'{melody_count} melodies generated.')
        messagebox.showinfo('Complete', f'{melody_count} melodies generated.')

    threading.Thread(target=handle, daemon=True).start()
```

Fonte: Autoria própria (2025).