CECS **Centro de Engenharia, Modelagem e Ciências Sociais Aplicadas**

Trabalho de Graduação

Engenharia de Informação

Título do projeto: **Estudo sobre geração de efeitos de áudio usando aprendizado de máquina e métodos clássicos**

Nome do aluno: **Denes Leal dos Santos**

RA do aluno: **11032714**

E-mail do aluno: **denes.leal@aluno.ufabc.edu.br**

Nome do orientador: **Prof Dr Murilo Bellezoni Loiola**

E-mail do orientador: **murilo.loiola@ufabc.edu.br**

Palavras-chave do projeto: **Filtros Digitais, Áudio, Redes Neurais, Machine Learning, DSP**

Resumo

Uma das ferramentas mais utilizadas no campo de produção e tratamento de áudio são os chamados *plug-ins* de áudio, que são softwares que agem como filtros de sinal produzindo um tratamento específico a partir de uma entrada de áudio. Como exemplo, existem *plug-ins* que produzem efeitos como reverberação e *delay* ou que realizam a simulação das características sônicas de um equipamento específico.

Estes *plug-ins* costumam ser construídos utilizando linguagem C/C++ através de plataformas como o JUCE, que utiliza conceitos clássicos de processamento de sinais. Porém, recentemente estão surgindo novas tecnologias que utilizam redes neurais para a construção dos *plug-ins* a partir de gravações de um determinado equipamento ou ambiente para o treinamento do algoritmo. Este trabalho, portanto, visou construir *plug-ins* de Reverberação e *Overdrive* utilizando ambas as técnicas, para então comparar a qualidade do resultado obtido, obtendo sucesso com os *plug-ins* em C++, porém não obtendo resultados satisfatórios com técnicas de aprendizado de máquina em Python por limitações de recursos computacionais, tempo e expertise. Foram feitos estudos e experimentações com três modelos: um regressor linear, um classificador simples e um autoencoder variacional (VAE, do inglês *Variational Autoencoder*), além da reprodução de um artigo usando redes convolucionais temporais (TCN, do inglês *Temporal Convolutional Network*). Este projeto apontou a possibilidade de desenvolvimentos futuros na adaptação destes modelos, em especial com o VAE e TCN.

Sumário

1 Introdução

1.1 *Plug-ins* de áudio

Plug-ins de áudio são softwares utilizados para modificar a característica sônica de um sinal, seja através de DSPs (*Digital Signal Processing*) ou uso de sintetizadores (WILMERING et al., 2020). Estes softwares são muito utilizados na indústria da música, pois possibilitam o tratamento de áudio sem a necessidade dos equipamentos analógicos utilizados nos grandes estúdios, sendo então grandes responsáveis pela descentralização da indústria, ao lado de outras tecnologias como o MP3. Isto porque estúdios menores passaram a ser capazes de competir com as grandes gravadoras, sendo capazes de produzir material de qualidade a menores custos (HUGHES; LANG, 2014). Hoje em dia, as maiores produções contam tanto com o uso de equipamentos clássicos, cuja característica sônica é reconhecida como padrão de qualidade, quanto com o uso de *plug-ins*, que muitas vezes são mais práticos e permitem manipulações que não são possíveis ou são mais difíceis com equipamentos analógicos.

Há diferentes tipos de *plug-ins* no mercado, sendo que os primeiros surgiram através da companhia Steinberg Media Technologies em 1996, chamados *Virtual Studio Technology*, ou *VSTs*, como são popularmente conhecidos (EKEROOT, 2003). Estes softwares podem ser desenvolvidos através de um *Software Development Kit*, ou SDK, disponibilizado pela própria Steinberg, ou através de frameworks de terceiros, como o JUCE (JUCE, 2021).

Os *plug-ins* VST são utilizados para processamento de sinais de áudio em tempo real, utilizando o suporte de uma DAW (*Digital Audio Workstation*), pois não podem ser executados por si só, necessitando do controle de entradas e saídas da DAW. Como qualquer atraso no sinal de áudio em uma produção pode introduzir um cancelamento de fase, prejudicando a qualidade, é preferencia que qualquer latência introduzida seja mínima. Para tanto, os *plug-ins* costumam ser codificados em baixo nível, através da linguagem C++, com algumas facilidades introduzidas pelos frameworks específicos e bibliotecas prontas (EKEROOT, 2003).

Porém, recentemente surgiram iniciativas para experimentar a capacidade das tecnologias de inteligência artificial para simular efeitos antigos ou mesmo criar novos tipos de distorção (STEINMETZ; REISS, 2021). Há estudos sobre o uso de redes neurais para a criação de diversos filtros de áudio, como equalizadores, compressores, *flangers*, *reverbs*, entre outros tipos de efeito (RAMíREZ; BENETOS; REISS, 2020). Além disso, existe também o uso das redes neurais para simular equipamentos reais, como é o caso da companhia Neural DSP (NEURALDSP, 2021).

No trabalho (STEINMETZ; REISS, 2021) por exemplo, são utilizadas redes neurais e TCNs (*Temporal Convolutional Networks*) para a obtenção de efeitos de *Overdrive* de maneira diferente das tecnologias utilizadas até então, com uma série de possibilidades de parametrização para gerar timbres diferentes.

Dado que existem tantas possibilidades para a criação destes filtros de áudio digitais, convém investigar suas diferentes implementações, e comparar suas qualidades, vantagens e desvantagens. Para tanto, escolheram-se dois tipos muito usados na indústria: o *Reverb* e o *Overdrive*.

1.2 *Reverb*

Citando (KOO; PAIK; LEE, 2021), Reverberação, ou *reverb*, é a combinação de múltiplas reflexões e difrações sonoras, que provém percepção espacial. É a característica sonora que diferencia uma voz falando em ambientes diferentes, como uma sala, um quarto, uma catedral ou um banheiro. Esta diferença se dá por conta das reflexões do ambiente, que podem ser mais rápidas ou mais lentas, se repetir mais ou menos vezes, e ter diferentes atenuações em diferentes frequências, conforme exemplificado pela resposta ao impulso ao longo do tempo de uma sala na figura 1. Reverberações são parte importante de qualquer produção musical, por adicionar diferentes texturas aos sons, e por ajudar a criar um panorama entre os diferentes instrumentos, para que seja mais fácil para o ouvinte discernir entre as diferentes fontes sonoras.

Fonte: (KOO; PAIK; LEE, 2021)

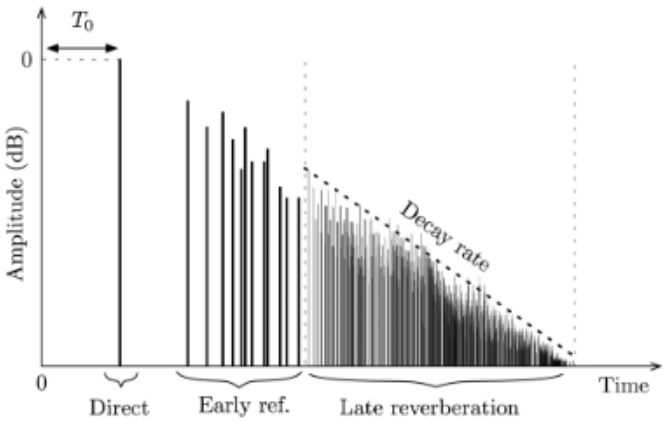

Figura 1: Exemplo generico de resposta ao impulso de uma sala.

Em geral, existem três tipos de algoritmos de reverb, ou híbridos entre estes (VALIMAKI et al., 2012):

- **Redes de *Delay*:**

  Neste método, o sinal original é atrasado, modificado de acordo com parâmetros estabelecidos e realimentados ao sinal original numerosas vezes com parâmetros ligeiramente diferentes, de forma a simular o efeito das reflexões sonoras no ambiente.

- ***Reverb* Convolucional:**

  Funciona através da operação de convolução do sinal de entrada com a resposta ao impulso correspondente ao ambiente que deseja-se simular. Os parâmetros de resposta ao impulso podem ser gravados no próprio ambiente ou simulados.

- **Acústica Computacional:**

  É feita a simulação computacional do comportamento da dispersão da energia acústica em um dado ambiente.

As primeiras utilizações de reverberação artificial ocorreram na década de 1920, através das chamadas câmaras de eco. Uma gravação "seca" era tocada em uma sala preparada especialmente para capturar sua reverberação, com objetos posicionados de forma a diminuir frequências indesejadas, e um microfone posicionado para captar de

forma ótima as reverberações. O som captado era então misturado ao som original, em proporções medidas de acordo com os propósitos artísticos ou perceptuais. Os resultados obtidos desta forma possuem alta qualidade, porém é um método custoso e pouco prático comparado com outros mais modernos (VALIMAKI et al., 2012). Ao longo do tempo, começaram a surgir dispositivos para a reverberação análogica, como os reverberadores de mola e os BBDs (*"Bucket-Brigade" Device*), até a década de 1960, quando começaram a surgir os primeiros reverberadores digitais.

Em 1979, Moorer publicou um estudo didático sobre algoritmos para reverberação digital (MOORER, 1979), o que em conjunto com os trabalhos publicados por Schroeder no começo da década (SCHROEDER; LOGAN, 1961), foi um importante passo no lançamento de diversos equipamentos de *reverb* digital, sendo que o primeiro foi o Lexicon Delta T-101 em 1971. Uma das novidades que vieram com os *reverbs* digitais foi a possibilidade de controle de parâmetros e da característica de decaimento da energia sônica, característica que viria a ser um diferencial do produto. Recentemente, a popularização de placas de processamento gráfico (GPUs) permitiu o uso de *reverbs* digitais mais poderosos, que exigem mais computacionalmente (VALIMAKI et al., 2012).

Os algoritmos utilizados nos *plug-ins* de hoje em dia se baseiam nestes mesmos princípios, sendo que neste trabalho será explorado o método de redes de *delay*.

1.3 Drive

Um dos efeitos mais utilizados na música é o chamado *Overdrive*, ou simplesmente *Drive*, que consiste na distorção da amplitude de um sinal de áudio através de um ganho alto o suficiente em um amplificador, resultando na saturação do sinal que chega em um determinado nível, gerando a retificação da forma de onda nestes picos. Este efeito é facilmente reconhecido nas distorções de guitarra que ficaram populares na música ocidental (BROWNING, 1997). A figura 2 exemplifica uma possível forma de onda com *drive*.

Fonte: (BROWNING, 1997)

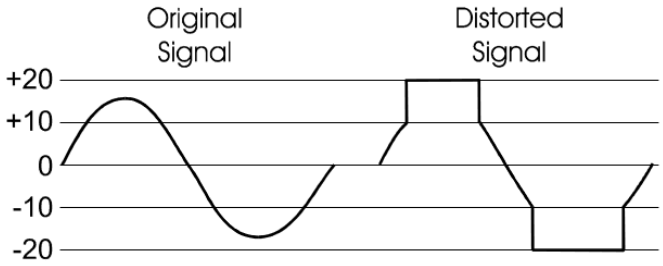

Figura 2: Exemplo de distorção de um sinal em um gráfico de amplitude por tempo sem escala.

Este efeito tem sua origem junto à criação da guitarra elétrica na década de 1930, que foi criada para que o som da guitarra pudesse competir com a sessão de metais das bandas da época. Estas guitarras necessitam de amplificadores que, por questão de redução de custos, eram construídos de forma que o sinal sofria saturação se utilizado acima de 50% da intensidade sonora para qual o equipamento foi projetado. Na década de 1940, guitarristas de blues começaram a utilizar volumes acima da especificação dos amplificadores deliberadamente para alterar o timbre do instrumento (WILMERING et al., 2020).

Hoje em dia, apesar de amplificadores ainda possuirem distorções características, geralmente os efeitos são utilizados através de pedais de distorção, ou *plug-ins* aplicados posteriormente na gravação do instrumento. Muitos músicos têm preferência por pedais que utilizam componentes analógicos para gerar este tipo de efeito, sendo que um dos motivos para tanto é que muitos dos métodos digitais não replicam a parte não linear da distorção, o que resulta em uma diferença na qualidade do timbre. Neste trabalho utilizaremos modelos mais simples de processamento digital de sinais para a recriação deste efeito, dado que a simulação do processamento não linear aumenta significativamente a complexidade do projeto (YEH; ABEL; SMITH, 2007).

2 Objetivo

O objetivo do trabalho é desenvolver filtros de efeitos de áudio utilizando duas abordagens diferentes e comparar os resultados obtidos. A proposta é construir *plug-ins* de áudio com os efeitos *Reverb* e *Overdrive* utilizando métodos clássicos de análise de sinais digitais, na linguagem C++, e com algoritmos de redes neurais e machine learning, para então comparar a qualidade das diferentes implementações.

3 Justificativa

Produções de áudio no geral, sejam para música, podcasts, filmes ou outras formas de mídia, têm se apoiado cada vez mais no uso de softwares de tratamento de som. Portanto, vê-se como necessidade explorar a capacidade de diferentes tecnologias para a implementação de tais softwares. Além disso, visto que este é um trabalho de conclusão de curso, este tema é efetivo por sua interdisciplinaridade e uso de diversos conceitos aprendidos durante o curso.

4 Metodologia

Para a realização deste trabalho, foi feita a implementação dos *Plug-ins* de *Reverb* e *Overdrive* utilizando métodos de DSP normais e diversas tentativas empíricas de desenvolver o equivalente com inteligência artificial.

4.1 Obtenção dos Dados

A obtenção dos dados para o teste de ambos os *plug-ins* e para o treinamento das redes neurais foi feita através da gravação de equipamentos que possuem os efeitos. O efeito de *Overdrive* foi obtido através de um amplificador Fender Rumble 40, que possui uma configuração com a distorção, e o efeito de *reverb* foi obtido através de um pedal de efeitos Vox Stomplab IB, também usado em conjunto com o amplificador citado

anteriormente, porém dessa vez em configuração *flat*. Ambos são equipamentos de contra-baixo, portanto foram usados com um instrumento adequado, um contra-baixo elétrico Tagima Classic Series Millenium 5.

Figura 3: Representação da gravação de áudios sem efeito ou *clean*.

Os áudios foram gravados tanto em linha, através de uma saída do amplificador, quanto com um microfone condensador Behringer C1 através do software Reaper, da Cockos. Para treinar e testar o algoritmo são necessários dois tipos de áudio: o limpo, ou *clean*, e seu correspondente tratado com efeito. Para o áudio clean, como mostrado na figura 3, foram feitas diversas gravações com os efeitos em questão desligados e, posteriormente, as mesmas gravações foram realimentadas nos equipamentos de áudio com o efeito correspondente ligado e gravada novamente, exemplificado na figura 4, de forma a capturar a mudança introduzida pelo respectivo efeito, sem a possibilidade de introduzir diferenças de performance do músico. Após as gravações, os áudios foram selecionados, tratados para alinhamento de fase entre cada áudio clean e seu correspondente distorcido e exportados para serem utilizados nos algoritmos.

O conteúdo das gravações variou entre execução de escalas musicais no instrumento, reprodução de músicas e gravação de técnicas de efeito do instrumento como *slides*, *slap* e *double thumb*. Os arquivos resultantes totalizaram cerca de 3 Gigabytes em arquivos WAV.

Figura 4: Representação da gravação de áudios com efeito aplicado através da realimentação do áudios limpos gravados anteriormente.

4.2 Implementação dos *Plug-ins*

Para a implementação dos *plug-ins* DSP sem uso de técnicas de machine learning, foi utilizado o framework JUCE, que é baseado na linguagem C++. O JUCE consiste em uma coleção de bibliotecas e estruturas de arquivos prontas para serem trabalhados em uma IDE (*Integrated Development Environment*) da preferência do desenvolvedor, com uma série de facilidades para a criação de *plug-ins*, tanto no processamento digital de sinais, quanto no front-end, onde o usuário poderá alterar parâmetros para a modificação do som em questão (JUCE, 2021).

A versão com uso de machine learning pode ser implementada utilizando bibliotecas como Sklearn, PyTorch e TensorFlow, que são bibliotecas de inteligência artificial que contam com diversas interfaces para a criação de modelos para treinamento, sendo que o PyTorch também conta com uma API (*Application Programming Interface*) da linguagem C++, o que possibilita utilizar os algoritmos desenvolvidos no próprio JUCE, para facilitar a criação de um Front-End e as integrações necessárias para o *plug-in*.

4.3 Avaliação dos *Plug-ins*

O principal critério para avaliação dos *plug-ins* é verificar se estes cumprem seu objetivo proposto, ou seja, se adicionam a um sinal de entrada os efeitos de *reverb* e *Overdrive*, respectivamente, através de seus devidos algoritmos. Para tanto, pode-se utilizar testes perceptuais, análises de distribuição de energia por frequência no tempo e também analises comparativas entre as formas de onda de entrada e saída, tanto entre os próprios algoritmos, quanto com *plug-ins* já existentes, como o Reaverb da Cockos, para o *reverb*, e o NA808 *Overdrive* Pro da Nembrini Audio.

5 Desenvolvimento

O desenvolvimento do trabalho foi feito nas linguagens C++ e Python. Para ambos os métodos, foram utilizadas bases de conhecimento e tutoriais públicos, disponíveis na documentação dos frameworks e de alguns trabalhos independentes, que serão referenciados de acordo. Trechos de código desenvolvidos neste projeto estão disponíveis no apêndice para referência, e os repositórios de código, ilustrações complementares e exemplos de áudio gerados encontram-se no GitHub (SANTOS, 2023).

5.1 Desenvolvendo Reverb em C++

O primeiro efeito implementado foi o *reverb*, desenvolvido através do framework JUCE, que contém diversas bibliotecas prontas para o tratamento de sinais e seu encapsulamento na forma de um *plug-in*. Este framework disponibiliza uma biblioteca específica para processamento digital de sinais, contendo diversas funções como, por exemplo, a convolução de diversos sinais em cadeia. Foi utilizado um módulo de reverberação presente neste pacote, em que é possível acrescentar as modificações na forma de onda características deste efeito e controlar estas mudanças através de parâmetros, como o tamanho da sala em que o som irá refletir e a quantidade de elementos atenuadores de frequências agudas, como móveis, quadros etc.

O código foi construído com o auxílio da documentação do JUCE (JUCE, 2022) e de um código de exemplo público para referência (KENGO, 2022). Os parâmetros requeridos pela biblioteca são os seguintes:

- ***roomSize:***

  O tamanho da sala em questão. Quanto maior a sala em que um som reverbera, maior o tempo entre o som e cada uma de suas reflexões nas paredes do ambiente. Portanto, este parâmetro controla a quantidade efetiva de reflexões e a sensação de estar em um espaço amplo, como uma catedral, ou pequeno, como um quarto.

- ***damping:***

  O nível de absorção de frequências agudas em cada reflexão. Este parâmetro simula a absorção natural de frequências efetuada por objetos que se encontram no ambiente em que o som é reproduzido. Quanto maior o parâmetro, mais frequências são absorvidas.

- ***wet e dry:***

  Em áudio, usa-se o termo *wet*, ou molhado, para se referir a um som que é modificado por algum efeito, e *dry*, ou seco, para o som puro sem processamento. Neste caso, esses parâmetros se referem à proporção em que é misturado o som processado com o som original, de forma a controlar a quantidade de influência que o efeito tem sobre o som na saída. No caso da implementação feita neste trabalho, ambos os parâmetros foram combinados em uma única variável chamada *dw*, que quando igual a zero, resulta em um som completamente seco, e quando igual a 1 em um som completamente processado.

- ***width:***

  Parâmetro relacionado à imagem estéreo do *reverb*. Como estamos tratando apenas com mono, este parâmetro não é tão relevante.

Estes parâmetros são coletados a partir de um layout que o usuário pode interagir em tempo real, alterando dinâmicamente o áudio obtido. Os parâmetros são então processados pelo módulo juce::dsp::reverb, que efetua a transformação do sinal com as características requeridas.

O resultado obtido a partir do código é um *plug-in* que pode ser compilado tanto para ser um .exe standalone, quanto um arquivo no formato VST, VST2 ou VST3, que pode ser utilizado em qualquer DAW para processar áudio em tempo real, de acordo com os parâmetros controlados conforme estabelecido.

5.2 Desenvolvendo *Overdrive* em C++

Para o *plug-in* de *Overdrive*, também foram utilizadas bibliotecas do JUCE e a ajuda de tutoriais disponíveis na documentação do framework, porém não foi utilizado um módulo próprio para *Overdrive*. Ao invés disso, foi utilizado o pacote juce::dsp::waveshaper, que permite aplicar uma função para transformar a forma de onda de um determinado sinal.

Este pacote é utilizado como um filtro em uma cadeia, acompanhado de um ganho de entrada no começo da cadeia para possibilitar o ajuste da saturação de sinal e um ganho de saída no final para ajuste do volume final do áudio obtido.

Como detalhado anteriormente, o que confere ao efeito de *overdrive* sua característica sonora é a saturação do sinal, que torna a forma de onda achatada nos picos. Com a função waveshaper, podemos recriar essa forma de onda matematicamente. Uma forma seria limitar o valor para um pico de 1 ou -1. Porém, isso torna os transientes abruptos, o que resulta em um som mais ríspido. Portanto, foi utilizada uma função tangente hiperbólica, que introduz a mesma limitação dos picos do sinal, porém com uma transição mais suave entre o meio e o pico da onda.

O framework utilizado contém toda uma estrutura pronta com layouts padrão para o front-end dos *plug-ins*, com possibilidade de personalização do mesmo conforme mostrado no *plug-in* de *reverb*. A ideia original era incluir parâmetros para modificação do ganho de entrada e de saída na interface, porém houve dificuldade na inserção de

parâmetros externos dentro da fórmula de tangente hiperbólica do *waveshaper*. Portanto, o resultado final é um *plug-in* de *Overdrive* que é funcional e produz o resultado esperado, mas que para aumentar o nível de distorção, ou o volume de saída, é necessário alterar parâmetros dentro do próprio código.

5.3 Resultados JUCE

A figura 5 mostra exemplos de formas de onda obtidas ao aplicar os efeitos desenvolvidos em um determinado trecho de áudio. Percebe-se que no caso do Overdrive há um efeito de compressão por conta da limitação da forma de onda, removendo parcialmente a dinâmica entre os trechos de diferentes intensidades sonoras. O resultado é uma forma de onda parcialmente retificada, o que gera a característica desejada no efeito.

Já no reverb podemos observar a dispersão temporal da intensidade, que é sutil no exemplo em gerado com os parâmetros do plug-in em posição *flat*, mas é exacerbado no exemplo de simulação de ambiente maior.

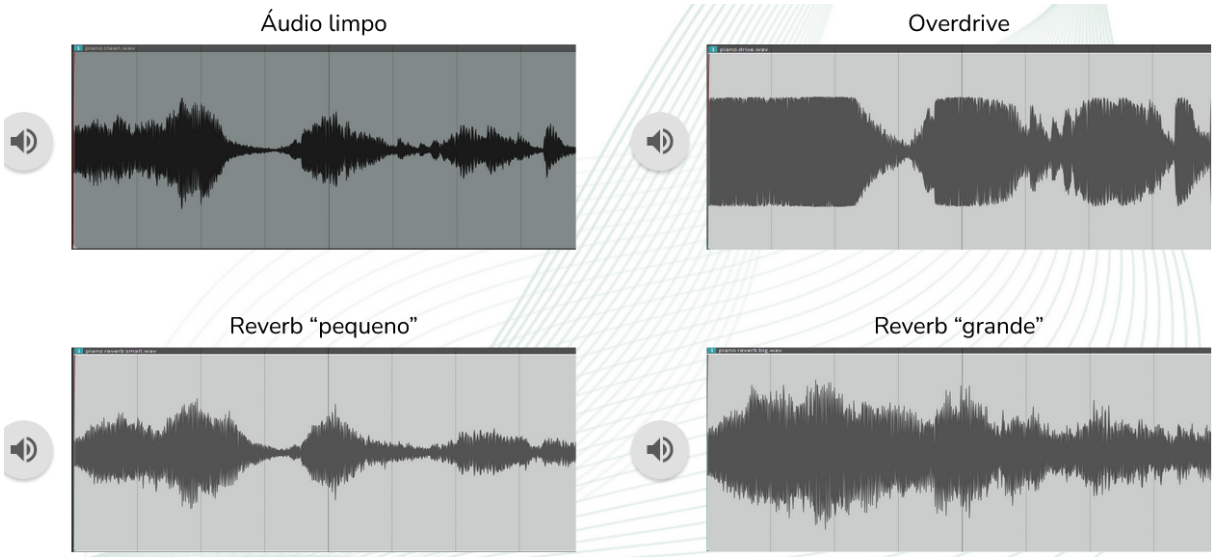

Figura 5: Formas de onda resultantes da aplicação dos plug-ins desenvolvidos em um dado trecho de áudio.

Podemos observar se os plug-ins atendem as características esperadas comparando as formas de onda obtidas com o produzido por outros plug-ins disponíveis comercialmente. Para a comparação do Overdrive, foi usado o plug-in Apocrita da eXe Consulting, enquanto o ReaVerbate da Cockos foi utilizado para o Reverb. Ambos estão representados na figura 6.

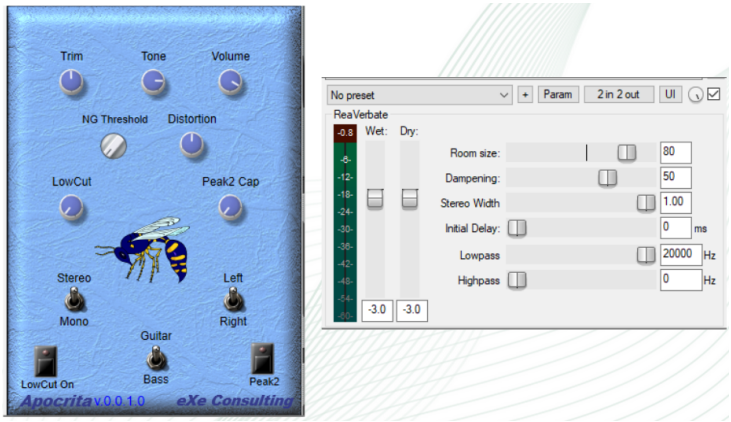

Figura 6: Plug-ins disponíveis comercialmente usados para comparação, Apocrita (esquerda) e ReaVerbate (direita). Os parâmetros utilizados nos exemplos correspondem ao mostrado na figura.

Em ambos os casos podemos observar formas de ondas bastante semelhantes, com diferenças sutis causadas por parâmetros extras configuráveis nos plug-ins comerciais e outras diferenças de implementação. Por exemplo, podemos observar que o Overdrive desenvolvido no projeto tem um decaimento mais abrupto nos trechos de menor intensidade sonora, enquanto o obtido através do Apocrita tem amplitude mais uniforme.

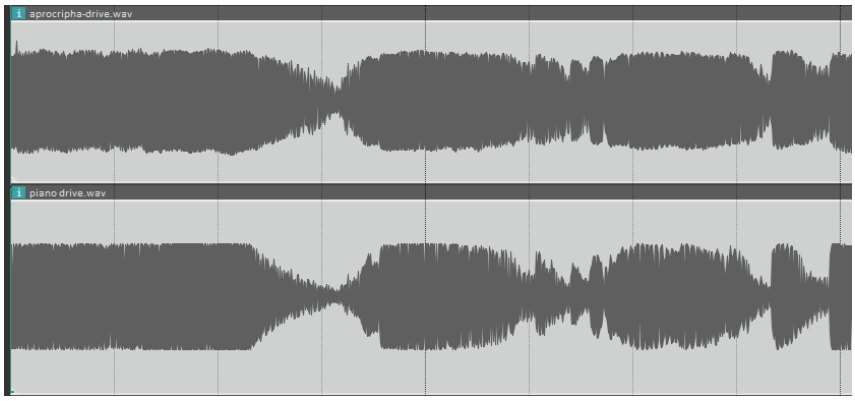

Figura 7: Comparação da forma de onda resultante do plug-in Apocrita (em cima) e desenvolvido no projeto (abaixo).

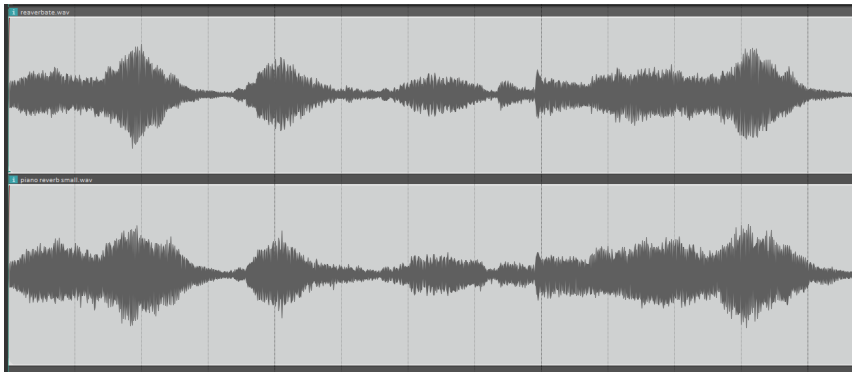

Figura 8: Comparação da forma de onda resultante do plug-in ReaVerbate (em cima) e desenvolvido no projeto (abaixo).

5.4 Desenvolvendo Efeitos de Áudio com Machine Learning

Para explorar a possibilidade de tratamento de áudio usando técnicas de Machine Learning, foi necessário desenvolver e avaliar diversas arquiteturas, adaptando seus parâmetros, entradas e saídas de forma a testar o aprendizado dos modelos e a posterior reconstrução do áudio a partir do resultado obtido.

As técnicas utilizadas foram aplicadas, a princípio, com o intuito de gerar o efeito de *Overdrive*, sendo que uma vez encontrada uma arquitetura funcional para este efeito, seria possível variar os dados de treinamento e verificar se isto seria suficiente para gerar um *reverb*. Porém, o problema se mostrou mais complexo que antecipado e não foi possível obter os resultados esperados. Portanto, o detalhamento a seguir se refere apenas ao efeito *Overdrive*. Os seguintes modelos foram desenvolvidos ou avaliados:

- Modelo de regressão linear da biblioteca sklearn;
- Rede neural classificadora adaptada;
- Autoencoder Variacional (*Variational Autoencoder*, VAE)
- Reprodução do artigo (STEINMETZ; REISS, 2022)

Estes modelos foram selecionados por conta da disponibilidade de materiais detalhados, que facilitaram o estudo e entendimento das características por trás do funcionamento de redes neurais construídas para aplicações de áudio. Vale notar que as arquiteturas em questão seguem uma ordem crescente de complexidade, com o estudo de cada modelo e o entendimento de suas limitações guiando a escolha do próximo.

5.5 Tipos de Dados Utilizados

Além da arquitetura em si, em alguns casos também foi necessário avaliar diferentes tipos de dados nas entradas e saídas do algoritmo, a fim de avaliar diferenças na qualidade do áudio obtido ao final do processo e do tempo de processamento. Os tipos de dados avaliados foram:

- Áudio bruto em formato WAV

- MFCCs

- Espectrogramas

Áudio bruto, nesse contexto, refere-se ao arquivo de áudio no formato WAV, que guarda a informação do áudio na forma de uma coleção de pontos de forma fiel à sua conversão analógico-digital, sem nenhuma forma de compressão, conforme exemplificado na figura 9. Este tipo de arquivo pode ser monaural ou estéreo, codificando informações distintas para o canal esquerdo e direito ou guardando apenas um único canal. Também pode armazenar o áudio codificado em diferentes frequências de amostragem, sendo o padrão 44kHz. Neste projeto, foram tratados apenas áudios mono codificados na taxa padrão.

A vantagem de utilizar dados brutos no treinamento é a garantia de que todas as informações necessárias para a manipulação e reconstrução do áudio estão presentes, mesmo quando essas características ainda não foram definidas para a aplicação em questão. Além disso, também vale destacar o fato de ser desnecessário usar algoritmos para extração das *features* requeridas e posterior reconstrução do áudio a partir dos dados obtidos na saída da rede neural. Isto pode acarretar na perda de informações importantes para a obtenção do efeito desejado no caso da extração de features ou mesmo ter a qualidade do resultado final reduzida por conta das características do algoritmo de reconstrução de áudio utilizado.

Embora o uso de arquivos WAV garantam que não tenhamos que lidar com estes possíveis pontos de falha, por se tratar de arquivos não comprimidos estes podem conter um volume de dados massivo dependendo da duração do áudio em questão e da frequência de codificação utilizada, podendo facilmente chegar a centenas de MegaBytes com poucos minutos de duração. Este volume de dados faz o processo de treinamento ser extremamente custoso em tempo e em recursos computacionais, o que pode vir a tornar o uso do modelo treinado lento demais para aplicações em tempo real, sendo esse um dos principais desafios na construção de *plug-ins* usando redes neurais (STEINMETZ; REISS, 2022).

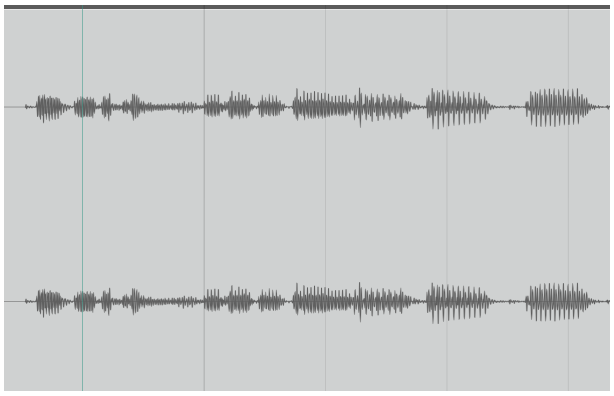

Figura 9: Exemplo de visualização de uma arquivo WAV estéreo, com cada linha representando o áudio de um canal.

Outro tipo de dado utilizado foram os chamados *Mel Frequency Cepstral Coefficients* (Coeficientes Ceptrais de Mel), ou MFCCs (LOGAN et al., 2000), que são um conjunto de características que podem ser medidas a cada momento de um arquivo de áudio e que representam características como timbre, textura e sonoridade do som em questão, sendo frequentemente usados em algoritmos de classificação de gênero músical e de reconhecimento de voz. A figura 10 exemplifica um possível conjunto de MFCCs na forma de um mapa de calor.

Estes coeficientes podem ser extraídos do arquivo de áudio usando algoritmos próprios e definindo uma janela de captura para a extração, sendo o resultado final como uma sequência de "fotos", mostrando a variação dos coeficientes ao longo do arquivo.

A tentativa de utilizar estes coeficientes justifica-se por ser um arquivo muito mais compacto que o WAV original, tendo melhor desempenho no treinamento e execução do algoritmo, além de conter informações pertinentes ao timbre, que é exatamente a característica principal que buscamos afetar ao construir efeitos e *plug-ins* de áudio.

Fonte: (VELARDO, 2020)

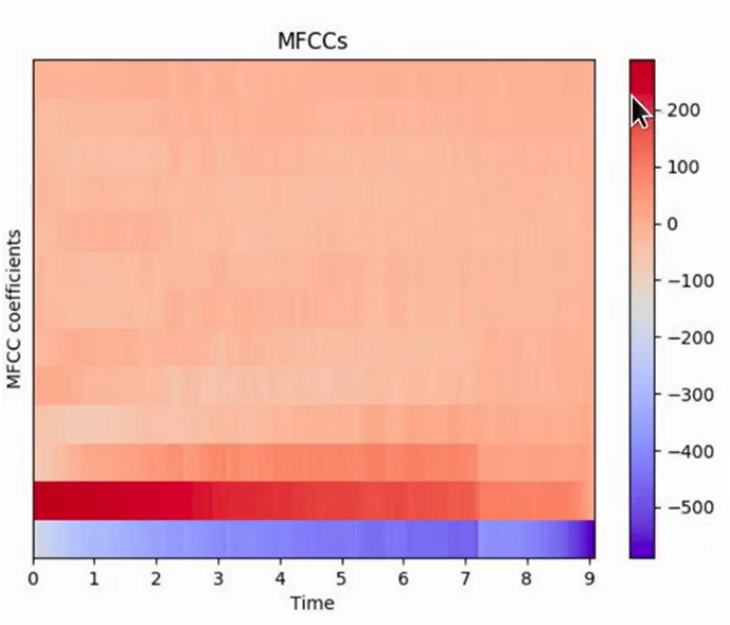

Figura 10: Exemplo de visualização de MFCCs

Porém, por ser um recorte muito específico das características de um dado áudio, a reconstrução de um som a partir dos MFCCs pode não ter a fidelidade necessária para uma dada aplicação.

Por fim, os espectrogramas são imagens representando graficamente a distribuição espectral de um som em um dado intervalo de tempo através de um mapa de calor, mostrando a intensidade sonora de cada uma das frequências presentes, podendo ter foco no espectro audível pelo ser humano (WYSE, 2017). A figura 11 é um exemplo de espectrograma.

No contexto deste projeto, os espectrogramas podem ser vistos como um intermediário entre os áudios brutos e MFCCs, sendo ainda um arquivo mais leve do que o WAV, mas contendo mais informações que os coeficientes cepstrais, com uma capacidade maior para a reconstrução do áudio posteriormente. Porém, como os espectrogramas são extraídos definindo um "*frame*", pode-se perder informações referentes aos transientes entre um frame e o próximo, dificultando a reconstrução do áudio (JUVELA et al., 2018).

Fonte: (VELARDO, 2020)

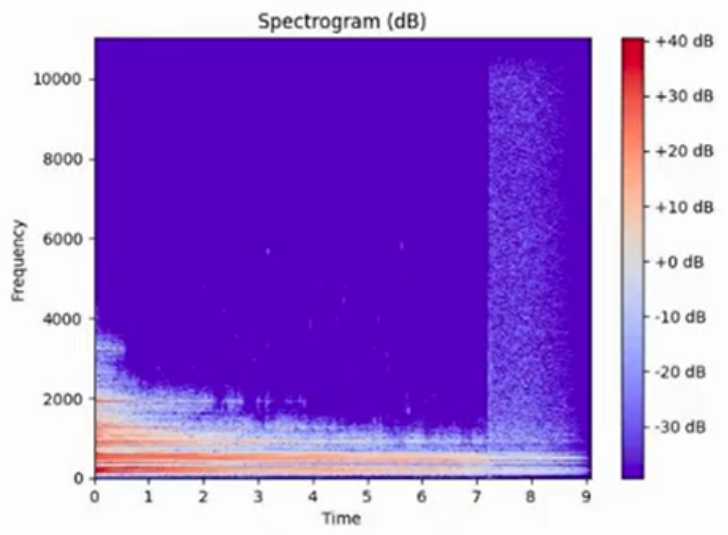

Figura 11: Exemplo de visualização de um espectrograma

5.6 Regressão Linear com Sklearn

O desenvolvimento do primeiro modelo foi feito utilizando a ferramenta Google Colab, com desenvolvimento na linguagem Python. O objetivo é importar áudios que servirão para treino e teste do algoritmo, estruturá-los em forma de um vetor que possa ser interpretado pela biblioteca neural_network do pacote sklearn, treinar o algoritmo MLPRegressor, cuja função é executar uma regressão linear para estabelecer uma relação entre os dados de entrada e saída.

Primeiramente, foi feita a importação dos áudios utilizados para treinamento, teste e predição. Os dados foram então extraídos destes arquivos, e convertidos para PCM (do inglês, *Pulse-Code Modulation*).

Os dados obtidos são então normalizados e convertidos em uma matriz bidimensional, efetivamente representando a disposição de cada amostra do áudio digital ao longo do tempo. Uma vez tratados os dados, estes são utilizados para treinar o algoritmo, que posteriormente é usado para inferir a saída a partir de outros áudios de entrada. A predição é então exportada para um arquivo WAV, que é executado para avaliação do resultado.

Os resultados obtidos nesta fase não foram satisfatórios. Os melhores arquivos gerados até então por esse método continham a característica sonora desejada do *Overdrive*, porém o áudio obtido é duas vezes mais longo que o original, e uma oitava abaixo, ou seja, mais grave que o esperado. Tentativas de ajustar valores como a taxa de amostragem e a forma de conversão para PCM resultaram em outros problemas, como frequências altas exarcebadas, áudio excessivamente saturado e ruído alto no geral.

5.7 Modelo Classificador Adaptado

Seguindo na exploração de métodos para realizar o tratamento de áudio usando machine learning, foi feito um estudo através de cursos disponibilizados na plataforma Youtube. O primeiro deles, intitulado "Deep Learning (for Audio) with Python" (VELARDO, 2020), aborda diversos conceitos de ML com foco em aplicações de áudio, concluindo com a construção de um classificador de cinco camadas utilizando a linguagem Python e o Framework Tensorflow.

TensorFlow (TENSORFLOW, 2023) é uma plataforma open-source para machine learning, que funciona através da criação e manipulação de tensores, que são matrizes multidimensionais usadas como representações de neurônios em redes neurais.

O algoritmo apresentado no decorrer do curso funciona recebendo como dados de entrada uma matriz de MFCCs extraídas de um arquivo de áudio, que são avaliadas pelo modelo treinado de forma a classificar o áudio em questão como pertencente a um dado gênero musical. Na prática, o algoritmo funciona de forma semelhante a um classificador de imagens, sendo cada matriz de MFCC obtida associada à uma imagem bidimensional.

O código em questão foi adaptado com o objetivo de verificar a capacidade da arquitetura construída para aplicar corretamente os efeitos de áudio. Isso foi feito de duas formas distintas: a primeira foi alterar a quantidade de neurônios em cada camada do algoritmo, de forma a obter um arquivo WAV na saída, ao invés de apenas o resultado da classificação. Isto pode ser feito tornando a dimensão dos tensores de saída proporcionais a quantidade de dados por segundo esperada em um arquivo de áudio.

A segunda forma foi alterar não apenas a saída, mas também a entrada do algoritmo para utilizar o arquivo de áudio bruto ao invés de suas MFCCs, de forma a avaliar se as informações contidas nas MFCCs são suficientes para a transformação acurada do áudio em relação ao arquivo bruto.

5.8 Autoencoder Variacional

Uma das arquiteturas avaliadas foi o chamado Autoencoder Variacional ou VAE (do inglês, *Variational Autoencoder*), um algoritmo construído para extrair automaticamente as informações de maior relevância para um dado problema a partir de um conjunto de dados, além de ser conhecido pela capacidade de generalizar resultados (PANDEY; KUMAR; NAMBOODIRI, 2018).

O VAE consiste de duas partes simétricas: um codificador e um decodificador. O codificador consiste em um funil de camadas de neurônios que recebe em sua entrada dados de áudio bruto e fornece na saída um arquivo de dimensões muito reduzidas em relação a entrada, efetivamente realizando a compressão do arquivo.

O decodificador, por sua vez, é o exato oposto: um funil invertido que recebe um arquivo da saída do codificador e o transforma novamente em um arquivo de áudio. Quando juntamos a saída do codificador e a entrada do decodificador em uma mesma arquitetura, temos o chamado Autoencoder, representado na figura 12.

O autoencoder variacional difere do tradicional na forma em que a codificação é feita, pois seu resultado é um espaço latente de probabilidade Gaussiana, ao invés de um simples mapeamento. Isso permite que o algoritmo possa inferir as características de qualquer lacuna presente no mapeamento através de operações probabilísticas (DOERSCH, 2016).

A vantagem deste algoritmo é que o processo de codificar o áudio faz com que a rede neural identifique automaticamente as características mais relevantes para o treinamento em questão, sendo capaz de generalizar os resultados do treinamento com muita eficácia.

Fonte: (TOWARDSDATASCIENCE.COM, 2023)

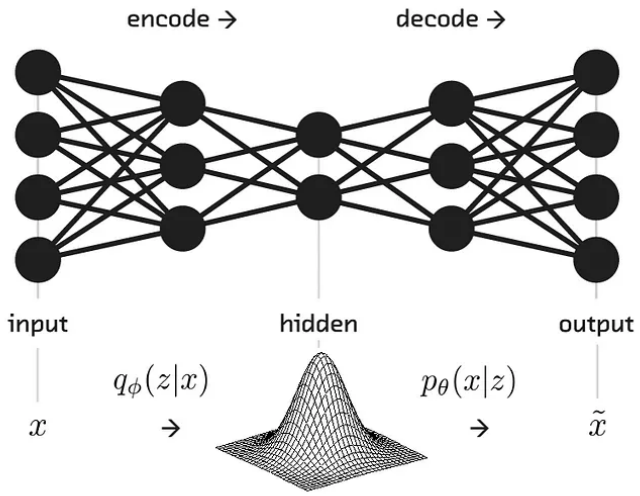

Figura 12: Representação de um VAE, onde cada círculo representa um neurônio ou tensor.

O código desta arquitetura foi feito através da adaptação do algoritmo desenvolvido no curso "Generating Sound with Neural Networks" (VELARDO, 2021), com variações nos parâmetros de entrada e de saída. Foram desenvolvidos e avaliados os seguintes métodos para geração de áudio com efeito usando o VAE:

- Entrada e saída com áudio bruto;
- Entrada e saída com MFCCs;
- Entrada e saída com espectrogramas;

No caso de tentativas com a saída do algoritmo em forma de MFCCs e espectrogramas, foi necessário aplicar um algoritmo de reconstrução do áudio a partir dos dados em questão para se obter o resultado final.

O treinamento do algoritmo foi feito utilizando fragmentos de um arquivo de áudio "limpo", i.e., sem distorção na entrada, e fragmentos correspondentes aos da entrada, porém com efeito de distorção aplicado na saída, de forma que o algoritmo possa identificar os padrões envolvidos na transformação do sinal através do treinamento.

Também foram feitas tentativas com a normalização dos dados de entrada e/ou saída no treinamento e ajustes como sincronização de fase, aplicação de efeitos usando *plug-ins* nos dados de treinamento para eliminar variáveis decorrentes da gravação e também a substituição total das gravações por novos áudios gerados através de MIDI (LOY, 1985). No entanto, estas tentativas não surtiram mudanças significativas, indicando a necessidade de maiores ajustes no modelo.

5.9 Reprodução do modelo MicroTCN

O último experimento foi uma reprodução do artigo "Efficient neural networks for real-time modeling of analog dynamic range compression" (STEINMETZ; REISS, 2022), através de código no Github, modelos pré-treinados e dataset de áudio para treinamento disponibilizados pelos autores.

Fonte: (STEINMETZ; REISS, 2022)

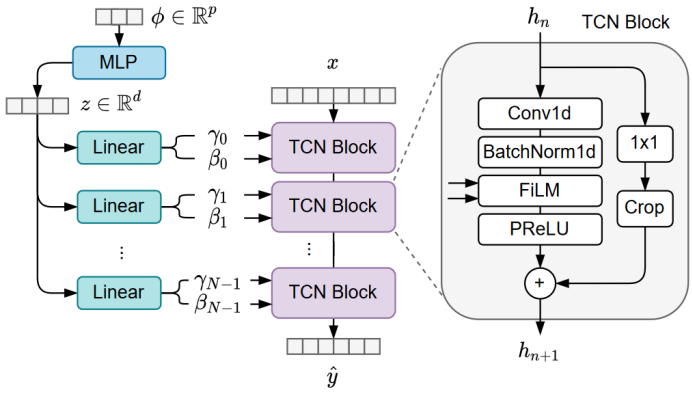

Figura 13: Arquitetura do modelo microTCN

O artigo estuda a criação de um *plug-in* de emulação do compressor analógico LA-2A usando adaptações do modelo TCN (*Temporal Convolutional Network*), com o intuito de verificar a possibilidade de criação de um modelo que consiga reproduzir as características do equipamento em questão e que seja capaz de processar áudio em tempo real. Para tanto, são feitas comparações de diversas versões causais e não causais do

TCN, cuja arquitetura é mostrada na figura 13, com outro modelo estabelecido, o LSTM-32 proposto em (WRIGHT et al., 2019).

A reprodução dos resultados do artigo foi executada de acordo com as instruções do arquivo ReadMe do código disponibilizado, sendo executado em um ambiente Linux na distribuição Pop OS 22.04 LTS, utilizando Python nas versões 3.8, 3.9 e 3.10.

Para cada uma das versões foram executados os seguintes passos:

- Criação de ambiente virtual para instalação de dependências e execução do código;
- Instalação de dependências especificadas no arquivo Requirements.txt, adaptando versões especificadas que se encontram indisponíveis para versão mais próxima encontrada;
- Execução de modelos pré-treinados através do script comp.py conforme instruções;
- Avaliação dos áudios gerados através de escuta e análise da forma de onda no software REAPER.

No caso do Python 3.8 também foi feito o treinamento do modelo uTCN-300 a partir do Dataset disponibilizado, para comparação com execução dos modelos pré-treinados.

6 Discussão e considerações finais

Como esperado, a implementação utilizando métodos clássicos, com um framework bem estabelecido no mercado especializado em desenvolvimento voltado para áudio, como é o JUCE, se mostrou muito mais simples e direto do que a implementação com redes neurais, mesmo sem conhecimento extensivo da linguagem C++. Por outro lado, a implementação com técnicas de Machine Learning se mostrou muito mais complexa do que o esperado, tomando a maior parte do desenvolvimento do trabalho. Isto porque, apesar da implementação do algoritmo em si ser muitas vezes simples quando comparado com o

equivalente em C++, estes sistemas possuem muitos parâmetros ajustáveis, como quantidade de camadas, tipos de operações aplicadas em cada camada, quantidade de neurônios, taxa de aprendizagem, funções de erro, entre outros.

Além disso, ajustes feitos fora do sistema, como no tratamento de dados de entrada e saída, também podem exercer forte influência sobre o resultado final. Tudo isso somado a testes muito custosos, tanto em recursos computacionais quanto em tempo, fazem com que esta abordagem exija um estudo muito mais aprofundado do funcionamento dos sistemas em questão quando comparado ao JUCE.

Os resultados obtidos com os *plug-ins* desenvolvidos em C++ são bastante satisfatórios e compatíveis com as opções de *plug-ins* disponíveis no mercado, principalmente os mais simples, disponibilizados como parte de conjuntos maiores. Porém, há espaço para maior exploração dentro desse tema verificando, por exemplo, diferenças sonoras em diferentes transformações matemáticas aplicadas através da função waveshaper ou mesmo verificar as diferentes possibilidades de parametrização das variáveis envolvidas.

No caso do *reverb* especificamente, é importante notar que, por se tratar de um efeito mais complexo envolvendo a dispersão temporal causada pelos reflexos de um som em um ambiente, foi necessário utilizar uma biblioteca pronta para a geração deste efeito. Há espaço para exploração em futuros projetos nesse tema, testando diferentes implementações e comparando suas diferenças.

Figura 14: Exemplo de áudio gerado pelo modelo VAE

De forma geral, os resultados desenvolvidos com redes neurais não foram satisfatórios. A grande maioria dos testes resultou apenas em modelos que geram ruído, de forma semelhante ao mostrado na figura 14. Porém, houveram duas exceções notáveis: o modelo de regressão linear que gera áudios mais lentos e com frequência deslocada em uma oitava, como mostrado na figura 15, e a tentativa de reprodução do microTCN, que satura excessivamente qualquer sinal na entrada, conforme mostrado na figura 16.

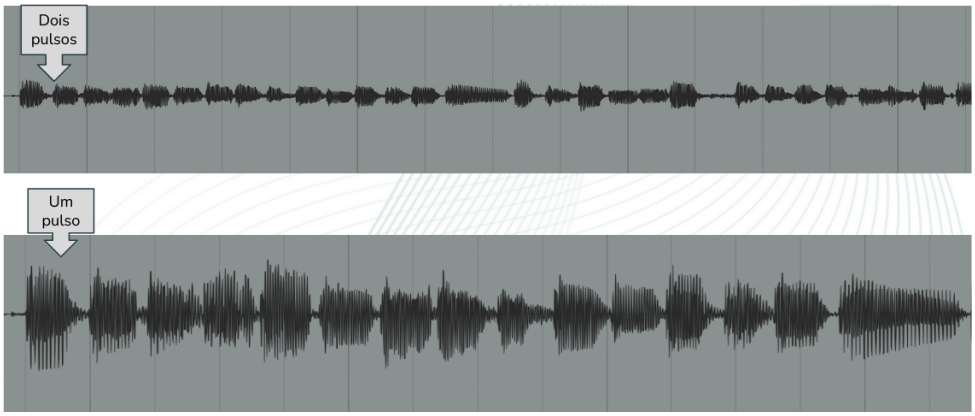

Figura 15: Comparação entre a forma onda obtida na saída do modelo de regressão linear (embaixo) e sua entrada (em cima).

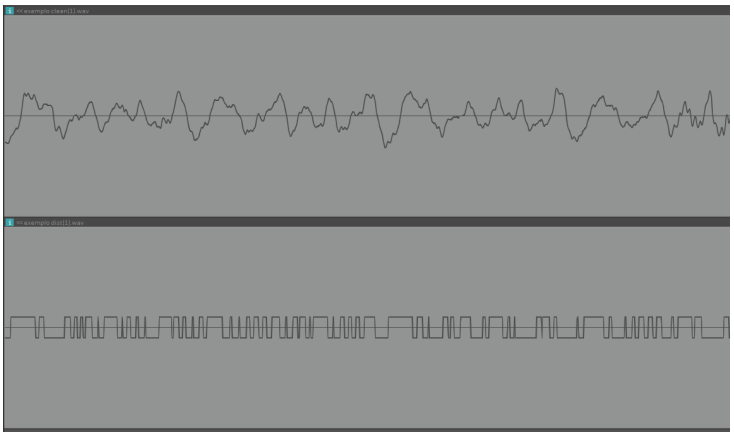

Figura 16: Exemplo de áudio gerado pela reprodução do modelo microTCN-300 (embaixo) e sua entrada (em cima). Nota-se que a forma de onda de saída é retangular, distorcendo o som como um efeito de *Overdrive*

Seria necessária uma análise mais detalhada para averiguar o motivo exato por trás de cada um desses resultados, mas para o primeiro caso, a principal hipótese é que isto seria uma tentativa do algoritmo de adaptar um efeito não-linear através de uma regressão linear em que não foram definidos os parâmetros de dimensionamento dos dados de saída, permitindo que o resultado tenha uma taxa de amostragem maior que o esperado e, portanto, resulte em um áudio mais longo quando amostrado em 44kHz.

No caso do microTCN, uma possível explicação seria alguma anomalia introduzida pelas modificações nas versões das dependências utilizadas no código, que precisaram ser adaptadas devido a não estarem disponíveis no gerenciador de pacotes do sistema utilizado. Há uma anotação no repositório do código feita por outro usuário que tentou reproduzi-lo, apontando que o código funciona sem necessidade de modificar versões no sistema operacional Ubuntu 18. Portanto, uma possibilidade seria tentar executar o código em um sistema do tipo para verificar se há diferença.

Já para o classificador e o autoencoder, seriam necessários mais testes antes de ter um resultado conclusivo, mas o desenvolvimento deste projeto e a comparação com outros desenvolvimentos presentes na literatura indicam que estas arquiteturas não contêm as operações necessárias para a aplicação destes tipos de efeito de áudio, porém é possível que versões modificadas possam ter resultados mais satisfatórios para aplicações que não sejam em tempo real.

Dos modelos estudados, o VAE foi o que teve a maior concentração de esforços para desenvolvimento e teste. Isto porque foi o modelo em que mais foram feitos experimentos com a variação dos tipos de dados de treinamento. Apesar dos resultados encontrados não atenderem a expectativa do projeto, foi possível observar as seguintes características:

- Treinamento com áudio bruto: treinamento demorado, exigindo muito processamento. A diferença entre os valores esperados e obtidos permanece constante independente de quantidade de dados ou quantidade de ciclos de treinamento;

- Treinamento com MFCCs: treino mais curto e com parâmetros melhores, porém a reconstrução do áudio deixa muito a desejar. Possível investigação futura envolve combinar MFCCs com áudio bruto para tentar aplicar o "perfil" dos coeficientes em um dado áudio;

- Treinamento com espectrogramas: semelhante ao com MFCCs, porém com resultado final um pouco melhor, mas ainda insatisfatório devido à limitações no algoritmo de reconstrução do áudio através de IFFT;

É importante notar que para o objetivo de construir um *plug-in* que aplica um filtro para produzir um determinado efeito em um áudio em tempo real, é necessário que o tratamento seja feito com o áudio bruto, pois adicionar etapas adicionais de conversão no processo pode resultar em latência, o que inviabilizaria o resultado para este tipo de aplicação.

7 Conclusão

A proposta original deste projeto era a de construir quatro *plug-ins* de áudio: dois *Overdrives* e dois *reverb*s, cada um sendo desenvolvido usando frameworks clássicos ou técnicas de Machine Learning e redes neurais, para posterior comparação entre os resultados obtidos usando cada um dos métodos. O resultado obtido de fato são os dois *plug-ins* em C++ propostos e uma série de estudos, experimentos e tentativas com quatro modelos de redes neurais que acabaram por não ter resultados satisfatórios.

Em termos de avaliação dos *plug-ins* que foram construídos, como o objeto de comparação acabou por não ter sucesso, não foram feitas tentativas de avaliar formalmente a qualidade dos mesmos. Apenas foi constatado que os *plug-ins* de fato aplicam os efeitos propostos conforme suas respectivas especificações e, portanto, o resultado obtido foi considerado satisfatório.

Quanto aos modelos de Machine Learning, apesar da literatura existente mostrar que é possível construir *plug-ins* usando estas técnicas conforme proposto, não foi possível criar um modelo capaz de produzir os efeitos de forma adequada por limitações em re-

cursos computacionais, tempo e expertise necessária para guiar as modificações e testes relevantes, sendo que este se mostrou um assunto muito mais complexo que o determinado pelo estudo preliminar do projeto.

Apesar da segunda parte do projeto não ter o resultado esperado, a exploração do assunto mostrou diversas possibilidades de expansão da investigação feita, especialmente no que se refere aos tipos de dados usados, sendo a combinação de MFCCs com dados brutos para criar uma parametrização com a qual tratar os dados uma possibilidade interessante. Também seria promissor investigar modificações nos modelos VAE e TCN para verificar possíveis diferenças nos resultados obtidos.

Referências

BROWNING, P. L. Audio digital signal processing in real time. 1997. Disponível em: ⟨http://docshare02.docshare.tips/files/16091/160913495.pdf⟩.

DOERSCH, C. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

EKEROOT, J. Implementing a parametric eq plug-in in c++ using the multi-platform vst specification. 2003. Disponível em: ⟨https://www.diva-portal.org/smash/get/diva2:1031081/FULLTEXT01.pdf⟩.

HUGHES, J.; LANG, K. R. If i had a song:the culture of digital community networks and its impact on the music industry. 2014. Disponível em: ⟨https://www.tandfonline.com/doi/abs/10.1080/14241270309390033⟩.

JUCE. 2021. Disponível em: ⟨https://juce.com/discover/stories/projucer-manual⟩.

JUCE. 2022. Disponível em: ⟨https://docs.juce.com/master/tutorial_dsp_introduction.html⟩.

JUVELA, L. et al. Speech waveform synthesis from mfcc sequences with generative adversarial networks. p. 5679–5683, 2018.

KENGO, S. 2022. Disponível em: ⟨https://github.com/szkkng/simple-reverb⟩.

KOO, J.; PAIK, S.; LEE, K. Reverb conversion of mixed vocal tracks using an end-to-end convolutional deep neural network. 2021. Disponível em: ⟨https://arxiv.org/pdf/2103.02147.pdf⟩.

LOGAN, B. et al. Mel frequency cepstral coefficients for music modeling. v. 270, n. 1, p. 11, 2000.

LOY, G. Musicians make a standard: The midi phenomenon. *Computer Music Journal*, JSTOR, v. 9, n. 4, p. 8–26, 1985.

MOORER, J. A. About this reverberation business. *Computer music journal*, JSTOR, p. 13–28, 1979.

NEURALDSP. 2021. Disponível em: ⟨https://neuraldsp.com/⟩.

PANDEY, L.; KUMAR, A.; NAMBOODIRI, V. Monoaural audio source separation using variational autoencoders. In: *Interspeech*. [S.l.: s.n.], 2018. p. 3489–3493.

RAMíREZ, M. A. M.; BENETOS, E.; REISS, J. D. Modeling plate and spring reverberation using a dsp-informed deep neural network. 2020. Disponível em: ⟨https://ieeexplore.ieee.org/abstract/document/9053093⟩.

SANTOS, D. L. dos. 2023. Disponível em: ⟨https://github.com/denes27/TG-Info-Juce⟩.

SCHROEDER, M. R.; LOGAN, B. F. "colorless"artificial reverberation. *IRE Transactions on Audio*, IEEE, n. 6, p. 209–214, 1961.

STEINMETZ, C. J.; REISS, J. D. Randomized overdrive neural networks. 2021. Disponível em: ⟨https://arxiv.org/pdf/2010.04237.pdf⟩.

STEINMETZ, C. J.; REISS, J. D. Efficient neural networks for real-time modeling of analog dynamic range compression. 2022.

TENSORFLOW. 2023. Disponível em: ⟨https://www.tensorflow.org/guide/basics⟩.

TOWARDSDATASCIENCE.COM. 2023. Disponível em: ⟨https://towardsdatascience.com/what-the-heck-are-vae-gans-17b86023588a⟩.

VALIMAKI, V. et al. Fifty years of artificial reverberation. 2012. Disponível em: ⟨https://ieeexplore.ieee.org/abstract/document/6161610⟩.

VELARDO, V. 2020. Disponível em: ⟨https://youtube.com/playlist?list=PL-wATfeyAMNrtbkCNsLcpoAyBBRJZVlnf⟩.

VELARDO, V. 2021. Disponível em: ⟨https://youtube.com/playlist?list=PL-wATfeyAMNpEyENTc-tVH5tfLGKtSWPp⟩.

WILMERING, T. et al. A history of audio effects. 2020. Disponível em: ⟨https://www.mdpi.com/2076-3417/10/3/791⟩.

WRIGHT, A. et al. Real-time black-box modelling with recurrent neural networks. In: *22nd international conference on digital audio effects (DAFx-19)*. [S.l.: s.n.], 2019. p. 1–8.

WYSE, L. Audio spectrogram representations for processing with convolutional neural networks. *arXiv preprint arXiv:1706.09559*, 2017.

YEH, D. T.; ABEL, J. S.; SMITH, J. O. Simplified, physically-informed models of distortion and overdrive guitar effects pedals. 2007. Disponível em: ⟨https://dafx.labri.fr/main/papers/p189.pdf⟩.

8 Apêndices:

8.1 Trechos do código do plug-in de Reverb em C++

```cpp
void ReverbExemploAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

    params.roomSize = *apvts.getRawParameterValue("size");
    params.damping = *apvts.getRawParameterValue("damp");
    params.width = *apvts.getRawParameterValue("width");
    params.wetLevel = *apvts.getRawParameterValue("dw");
    params.dryLevel = 1.0f - *apvts.getRawParameterValue("dw");
    params.freezeMode = *apvts.getRawParameterValue("freeze");

    leftReverb.setParameters(params);
    rightReverb.setParameters(params);

    juce::dsp::AudioBlock<float> block(buffer);

    auto leftBlock = block.getSingleChannelBlock(0);
    auto rightBlock = block.getSingleChannelBlock(1);

    juce::dsp::ProcessContextReplacing<float> leftContext(leftBlock);
    juce::dsp::ProcessContextReplacing<float> rightContext(rightBlock);

    leftReverb.process(leftContext);
    rightReverb.process(rightContext);
}
```

Figura 17: Implementação de Reverb no framework JUCE

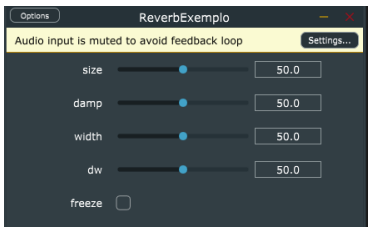

Figura 18: Interface do plug-in de Reverb

```cpp
private:
    juce::dsp::Reverb::Parameters params;
    juce::dsp::Reverb leftReverb, rightReverb;
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (ReverbExemploAudioProcessor)
};
```

Figura 19: Módulo Juce::DSP::Reverb que recebe os parâmetros

```
juce::AudioProcessorValueTreeState::ParameterLayout ReverbExemploAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;

    layout.add(std::make_unique<juce::AudioParameterFloat>("size",
        "size",
        juce::NormalisableRange<float>(0.0f, 1.0f, 0.001f, 1.0f),
        0.5f,
        juce::String(),
        juce::AudioProcessorParameter::genericParameter,
        [](float value, int) {
            if (value * 100 < 10.0f)
                return juce::String(value * 100, 2);
            else if (value * 100 < 100.0f)
                return juce::String(value * 100, 1);
            else
                return juce::String(value * 100, 0); },
        nullptr));

    layout.add(std::make_unique<juce::AudioParameterFloat>("damp",
        "damp",
        juce::NormalisableRange<float>(0.0f, 1.0f, 0.001f, 1.0f),
        0.5f,
        juce::String(),
        juce::AudioProcessorParameter::genericParameter,
        [](float value, int) {
            if (value * 100 < 10.0f)
                return juce::String(value * 100, 2);
            else if (value * 100 < 100.0f)
                return juce::String(value * 100, 1);
            else
                return juce::String(value * 100, 0); },
        nullptr));


    layout.add(std::make_unique<juce::AudioParameterFloat>("width",
        "width",
        juce::NormalisableRange<float>(0.0f, 1.0f, 0.001f, 1.0f),
        0.5f,
        juce::String(),
        juce::AudioProcessorParameter::genericParameter,
        [](float value, int) {
            if (value * 100 < 10.0f)
                return juce::String(value * 100, 2);
            else if (value * 100 < 100.0f)
                return juce::String(value * 100, 1);
            else
                return juce::String(value * 100, 0); },
        nullptr));

    layout.add(std::make_unique<juce::AudioParameterFloat>("dw",
        "dw",
        juce::NormalisableRange<float>(0.0f, 1.0f, 0.001f, 1.0f),
        0.5f,
        juce::String(),
```

Figura 20: Implementação de parâmetros de Reverb no layout

8.2 Trechos do código do plug-in de Overdrive em C++

```
private:
    //==============================================================================
    enum {
        filterIndex,
        preGainIndex,
        waveshaperIndex,
        postGainIndex
    };

    using Filter = juce::dsp::IIR::Filter<float>;
    using FilterCoefs = juce::dsp::IIR::Coefficients<float>;

    juce::dsp::ProcessorChain<juce::dsp::ProcessorDuplicator<Filter, FilterCoefs>, juce::dsp::Gain<float>, juce::dsp::WaveShaper<float>, juce::dsp::Gain<float>> processorChain;
};
```

Figura 21: Cadeia de filtros para efeito de Overdrive

```
class Distortion
{
public:
    //==============================================================================
    Distortion(float amp, float pregain, float postgain)
    {
        auto& waveshaper = processorChain.template get<waveshaperIndex>();

        //onda quadrada
        /* waveshaper.functionToUse = [](Type x)
        {
            return juce::jlimit(Type(-0.5), Type(0.5), x);
        };*/

        //tangente hiperbolica
        waveshaper.functionToUse = [] (float x)
        {
            return std::tanh(x);
        };

        auto& preGain = processorChain.template get<preGainIndex>();
        preGain.setGainDecibels(30.0f);

        auto& postGain = processorChain.template get <postGainIndex>();
        postGain.setGainDecibels(0.0f);

    }
```

Figura 22: Classe Distortion, responsável pela transformação do sinal

8.3 Trechos do código do modelo de regressão linear em Python

Importação e tratamento dos dados

```
# Dados de Treino
x_audio = AudioSegment.from_file("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Clean/1.wav")
y_audio = AudioSegment.from_file("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Drive/1D.wav")

#x_rate, X = wavfile.read("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Clean/1.wav")
#y_rate, Y = wavfile.read("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Drive/1D.wav")

#x_audio, y_audio = cutsignal(x_audio, y_audio)

x_data = x_audio._data
y_data = y_audio._data
X = topcm(x_data)
Y = topcm(y_data)
# X = x_data
# Y = y_data

# Dados de Teste
w_audio = AudioSegment.from_file("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Clean/2.wav")
w_data = w_audio._data
z_audio = AudioSegment.from_file("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Drive/2D.wav")
z_data = z_audio._data
W = topcm(w_data)
Z = topcm(z_data)
# W = w_data
# Z = z_data

#w_rate, W = wavfile.read("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Clean/2.wav")
#z_rate, Z = wavfile.read("/content/drive/MyDrive/UFABC/TG/TG/Python/Audios/Audios Drive/2D.wav")
```

Figura 23: Importação dos arquivos para treinamento da rede neural

Normalização e padronização dos dados

```
# Dados de Treino
X = normalize(np.array(X))
Y = normalize(np.array(Y))
X = X.reshape(-1,1)
Y = Y.reshape(-1,1)

# # Dados de Teste
W = normalize(np.array(W))
Z = normalize(np.array(Z))
W = W.reshape(-1,1)
Z = Z.reshape(-1,1)

#X, Y = cutsignal(X,Y)
#W,Z = cutsignal(W,Z)

# Renomeação das variáveis
X_train, X_test, y_train, y_test = X, W, Y, Z
```

Figura 24: Importação dos arquivos para treinamento da rede neural

Treinamento do algorítimo

```
mlp = MLPRegressor(hidden_layer_sizes=(10), solver='lbfgs', random_state=1)
mlp.fit(X_train, y_train)
y_pred = mlp.predict(X_test)
```

Figura 25: Tratamento de dados para treinamento da rede neural

8.4 Trechos do código do modelo classificador em Python

```
N_NEURONS_L1 = 1280
N_NEURONS_L2 = 1080
N_NEURONS_L3 = 862
INPUT_SHAPE = tf.TensorShape([256, 862])
RESHAPE_LAYER_SHAPE = (256, 862)
LEARNING_RATE = 0.0001
CLIP_NORM = 0.001


 Denes Leal *
def generateModel():
    # Initializing Device Specification
    device_spec = tf.DeviceSpec(job="localhost", replica=0, device_type="GPU")
    name = ''
    # Specifying the device
    with tf.device(device_spec):
        # build network topology
        model = keras.Sequential([

            # 1st dense layer
            keras.layers.Dense(N_NEURONS_L1, input_shape=INPUT_SHAPE, activation='relu'),

            # 2nd dense layer
            keras.layers.Dense(N_NEURONS_L2, activation='relu'),

            # 3rd dense layer
            keras.layers.Dense(N_NEURONS_L3, activation='relu'),

            # output layer
            keras.layers.Reshape(RESHAPE_LAYER_SHAPE)
        ])
        for layer in model.layers:
            print(layer.output_shape)

        # compile model
        optimiser = keras.optimizers.Adam(learning_rate=LEARNING_RATE, clipnorm=CLIP_NORM)
        model.compile(optimizer=optimiser,
                      loss="mse",
                      metrics=['accuracy'])

        model.summary()

        return model
```

Figura 26: Código do modelo classificador adaptado

8.5 Trechos do código do modelo classificador em Python

```
Denes Leal
def _build(self):
    self._build_encoder()
    self._build_decoder()
    self._build_autoencoder()

Denes Leal
def _build_autoencoder(self):
    model_input = self._model_input
    model_output = self.decoder(self.encoder(model_input))
    self.model = Model(model_input, model_output, name="autoencoder")

Denes Leal
def _build_decoder(self):
    decoder_input = self._add_decoder_input()
    dense_layer = self._add_dense_layer(decoder_input)
    reshape_layer = self._add_reshape_layer(dense_layer)
    conv_transpose_layers = self._add_conv_transpose_layers(reshape_layer)
    decoder_output = self._add_decoder_output(conv_transpose_layers)
    self.decoder = Model(decoder_input, decoder_output, name="decoder")
```

Figura 27: Fragmento do código do modelo VAE detalhando sua composição através de um encoder e decoder