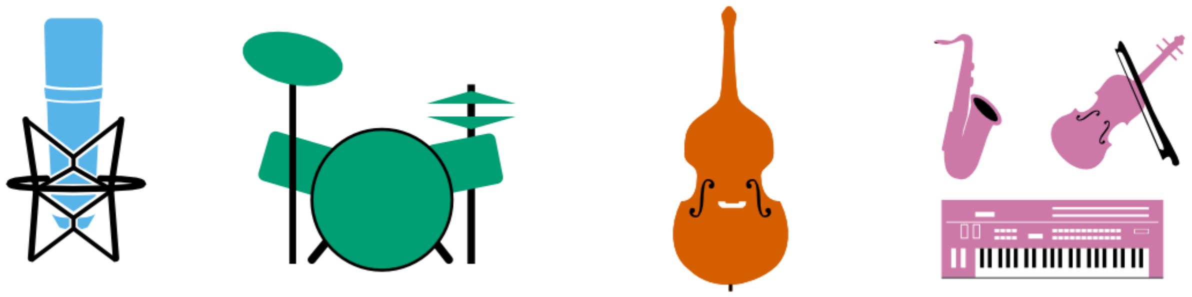
Synthesis Report



Machine Learning for Audio on the Web

—

Maria Clara JACINTHO
Delton VAZ

Computer Science and Management, 5th year
University Tutor: Dr. Chouki TIBERMACINE
Project owner: Fabian-Robert STÖTER

2020 - Montpellier

1. Figures and tables

2. Glossary

Term	Definition
DSP	Digital Signal Processing
Fourier Transform	“The Fourier transform decomposes a function of time (a signal) into its constituent frequencies.”[1]
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform - an algorithm that implements the DFT
STFT	Short-time Fourier Transform
ISTFT	Inverse Short-Time Fourier Transform
LSTM	Long short-term memory - “is an artificial recurrent neural network architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points, but also entire sequences of data.”[2]

[1] https://en.wikipedia.org/wiki/Fourier_transform
[2] https://en.wikipedia.org/wiki/Long_short-term_memory

3. Abstract

In this report, we present our Capstone project for the Course of Computer Science and Management at Polytech Montpellier that we developed in partnership with the SigSep team (Inria, LIRMM) [1]. The project consists of creating an application that allows a user to unmix a song, i.e. to separate each instrument (such as guitar, bass, vocals) from a song, on the web, straight on their browser. In this context, we use JavaScript to develop a program that uses both Digital Signal Processing (DPS) and machine learning techniques and can be run completely on the browser without the need for a server. Our approach consists of manually implementing DSP operations that do not exist in JavaScript, converting a pre-trained model to TensorFlowJS [2] and creating an intuitive and minimalistic UI using Vue.

Keywords: Machine Learning, DSP, Audio, JavaScript

4. Résumé

Dans ce rapport, nous présentons notre Projet de Fin d'études pour le cursus Informatique et Gestion à Polytech Montpellier. Nous avons développé ce projet en partenariat avec l'équipe SigSep (Inria, LIRMM). Le projet consiste à créer une application qui permet à un utilisateur de démixer une chanson, c'est-à-dire de séparer chaque instrument (comme la guitare, la basse, le chant) d'une chanson, sur le web, directement sur son navigateur. Dans ce contexte, nous utilisons JavaScript pour développer un programme qui utilise à la fois le traitement numérique du signal et les techniques d'apprentissage machine et qui peut être exécuté entièrement sur le navigateur sans avoir besoin d'un serveur. Notre approche consiste à implémenter manuellement des opérations DSP qui n'existent pas en JavaScript, à convertir un modèle écrit en TensorFlowJS et à créer une interface utilisateur intuitive et minimaliste à l'aide du framework JavaScript Vue.

5. Context of the project

The main objective of this project is the development of a tool that allows any user, using a browser, to separate the musical instruments from a piece of music, i.e. the user can extract only the sound of the guitar or vocals of a song, without the need of any kind of server for processing.

The project was initially proposed by the SigSep team from the INRIA [1] (National Institute for Research in Digital Science and Technology). They have created a music separation model in Python that reaches state of the art separation performance, called Open Unmix. The model was developed using the machine learning framework PyTorch. They were inspired by a website made by a Japanese company called KoeKestra[3]. The website also allows the user to unmix a song right on the browser, using a different machine learning model called Spleeter [4], created by Deezer [5] and converted to JavaScript.

The music separation pipeline can be divided into 3 parts: pre-processing, prediction and post-processing. The pre and post-processing parts involve Fourier Transforms, operations very common in the Digital Processing Field (DSP). The model prediction part involves transforming a model developed and trained in Python into JavaScript, loading it and using it to predict.

Currently, there are very few libraries available for doing DSP in JavaScript. We were not, in fact, able to find a library that implemented all of the functions that we needed. Our work, therefore, began with learning more about DSP and Fourier Transforms, so that we would be able to adapt existing solutions to our needs. Since there are other languages such as Python that already have such functions implemented (and also used in the original model), those implementations became our basic example and baseline.

Originally, the SigSep team wanted us to use the Spleeter model, since it was originally developed using TensorFlow, and could, in theory, be converted into TensorFlowJS (TFJS), a machine learning framework for JavaScript. There is a built-in converter in TFJS to convert models trained in Python to JS. We later found, however, that was not the case since it used operations that are not yet available on TFJS such as Long short-term memory (LSTM).

After some discussion with Fabian, the project coordinator, it was decided that we would use an early model from the SigSep team that was built using TensorFlow 1.5. We found that we also could not convert that model to TFJS. Finally, after more discussion with the team, it was decided that a new model would be trained specifically for this project. An advantage of this approach is that it resulted in a lighter-weight model, better suited for web applications.

It is also worth noting that both the areas of DSP and Machine Learning on JavaScript are very new and change very quickly. While we were developing the project the TFJS team added the ability to run Python models directly on javascript, but unfortunately only while using Node. Also, many of the methods and libraries we used were developed within the last year, some, like the Fourier Transforms, within the last 6 months. There are many difficulties tied to developing such a bleeding-edge project, such as the lack of testing or documentation for the features we needed, but we still found it an incredibly interesting and enjoyable process.

6. Project Organization

Initially, we set up a meeting with our tutors, to decide how we would organize ourselves during the project. We decided to have weekly meetings, without a fixed day, to show our progress and discuss next steps. We use the following tools for communication and development:

- **Slack** - We used Slack to exchange ideas, ask questions and give updates on the project. The tutor from the SigSep team, Fabian Stöter, was very active on the platform, and usually answered our questions very quickly. We also scheduled meetings with him through the app. We chose Slack because it is a great messaging app, with the functionality of being able to separate the conversation into various channels and threads, that also has some great plug-ins such as GitHub which generates notifications when something is changed in the main repository.

- **Github** - At the beginning of the project, the project tutor created a private repository on Github on the SigSep team page. Once the project is finished, the repository will be made public. In addition, once the website was developed, we used Github Pages, a free service to host web pages. Both students had experience with Github, so there was no need to learn a new tool.
    - **Github Projects** - It is an online Kanban board that allows users to organize their tasks and their state of completion. Since it is integrated with a Github repository, it is able to directly reference Issues, commits, branches etc. We used this tool based on a recommendation from the project tutor.

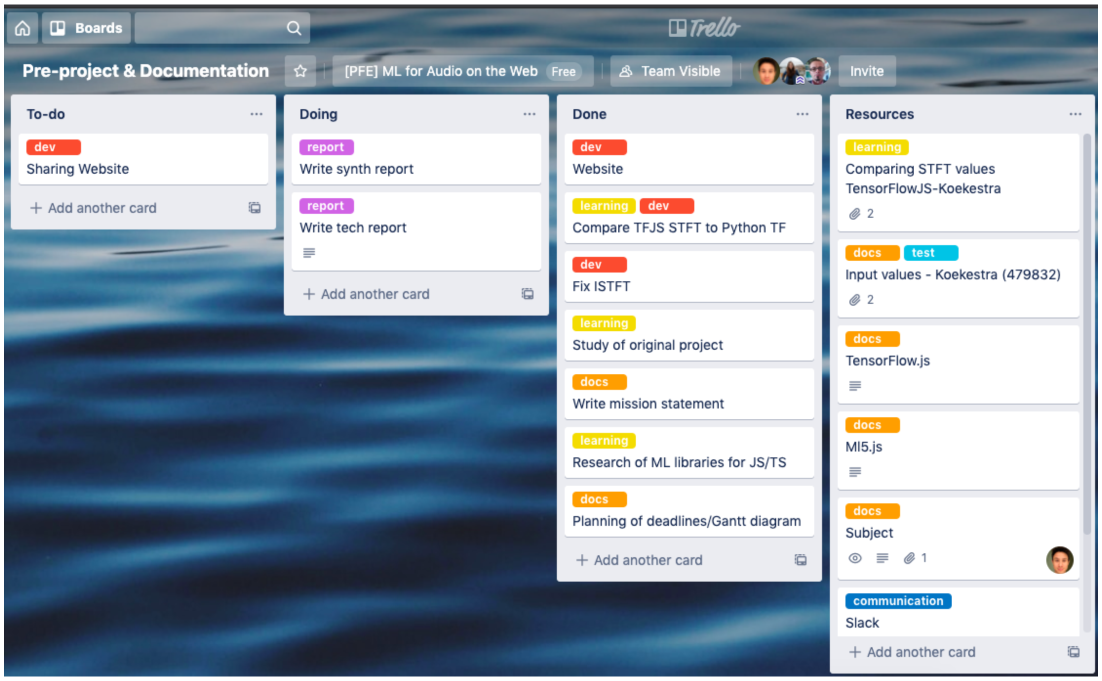
- **Google Drive and Docs** - We used Google Drive to share files among the team, such as sample songs to test the model and interesting papers related to the project. We also used Google Docs to write the reports and mission statement, since it allows instant sharing and concurrent editing of text documents online.

- **VSCode/WebStorm** - Each student used a different IDE for development. VSCode is a lightweight editor that has great debugging tools and WebStorm is a powerful editor.

- **Audacity** - It is an open-source tool that allows us to analyze the waveform of a song or generate a tone, e.g. a sine wave. It was especially useful to analyze the output of the Fourier Transforms and helped us identify many bugs.

- **Google Chrome** - For the testing of the website, we used Chrome since it has great developer tools. It also has support for the WebGL technology, that greatly increases the speed of our app by performing GPU-accelerated operations. Furthermore, it is one of the most popular browsers in the world, and we wanted to make sure the program would run on it.

- **Trello** **-** Web-based Kanban tool that we used to organize important resources from the research that we did during project development such as important links to read.


Boards
Trello
Pre-project & Documentation
[PFE] ML for Audio on the Web Free
Team Visible
Invite

To-do
- dev — Sharing Website
- + Add another card

Doing
- report — Write synth report
- report — Write tech report
- + Add another card

Done
- dev — Website
- learning, dev — Compare TFJS STFT to Python TF
- dev — Fix ISTFT
- learning — Study of original project
- docs — Write mission statement
- learning — Research of ML libraries for JS/TS
- docs — Planning of deadlines/Gantt diagram
- + Add another card

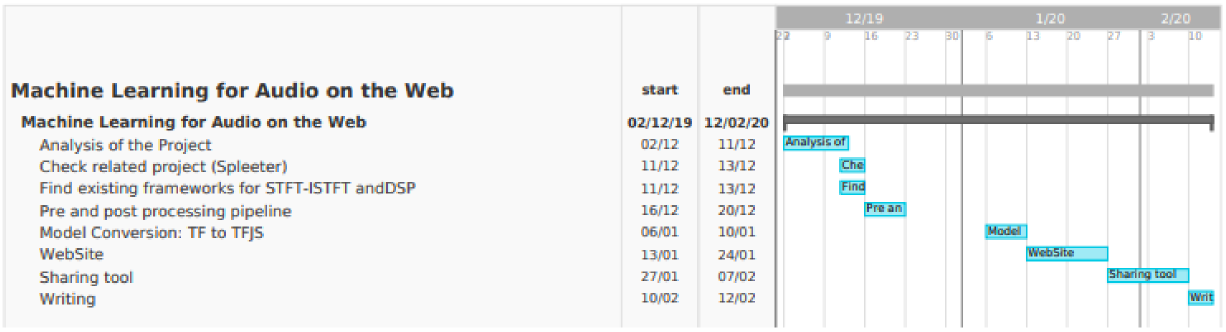
Resources
- learning — Comparing STFT values TensorFlowJS-Koekestra — 2
- docs, test — Input values - Koekestra (479832) — 2
- docs — TensorFlow.js
- docs — Ml5.js
- docs — Subject — 1
- communication — Slack
- + Add another card

Fig. I - Our trello board

6.1 Management methods

Initially, we expected to follow a traditional **Waterfall** development model, where we completely finished each task of the project before beginning the next one. Once we started development, we found that it was better to switch to the **Agile** development model, where we continuously improve upon the same feature and can work on multiple features at once, on a smaller scale. This helped the development as we had more feedback from the project owner and we were able to start working on features (such as the model conversion) even if the previous stage (pre-processing) was not completely finished.

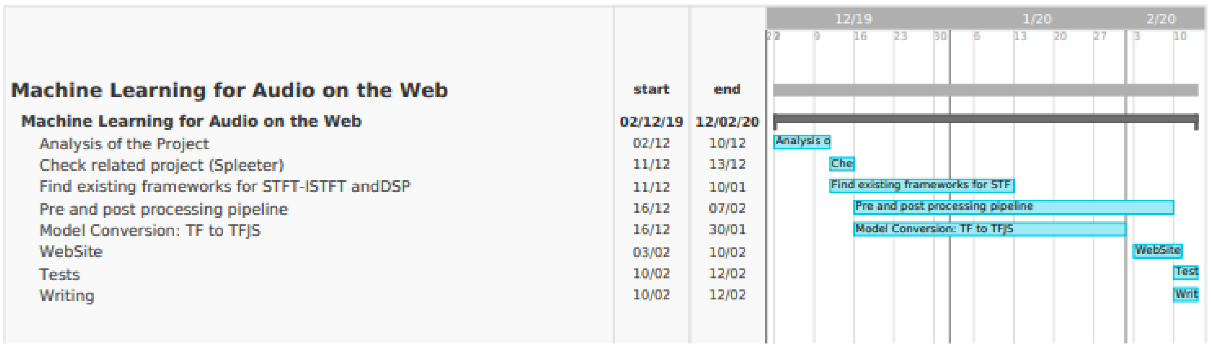
At the beginning of the project, we created a planning equivalent to the following Gantt diagram:



Machine Learning for Audio on the Web	start	end
Machine Learning for Audio on the Web	**02/12/19**	**12/02/20**
Analysis of the Project	02/12	11/12
Check related project (Spleeter)	11/12	13/12
Find existing frameworks for STFT-ISTFT andDSP	11/12	13/12
Pre and post processing pipeline	16/12	20/12
Model Conversion: TF to TFJS	06/01	10/01
WebSite	13/01	24/01
Sharing tool	27/01	07/02
Writing	10/02	12/02

Fig. II - The original Gantt diagram

We arrived at these estimations by using our previous experience in similar research projects and discussing with the project tutor, Fabian Stöter. In the end, this is how our project turned out:



Machine Learning for Audio on the Web	start	end
Machine Learning for Audio on the Web	**02/12/19**	**12/02/20**
Analysis of the Project	02/12	10/12
Check related project (Spleeter)	11/12	13/12
Find existing frameworks for STFT-ISTFT andDSP	11/12	10/01
Pre and post processing pipeline	16/12	07/02
Model Conversion: TF to TFJS	16/12	30/01
WebSite	03/02	10/02
Tests	10/02	12/02
Writing	10/02	12/02

12/19
1/20
2/20
2
9
16
23
30
6
13
20
27
3
10
Analysis o
Che
Find existing frameworks for STF
Pre and post processing pipeline
Model Conversion: TF to TFJS
WebSite
Test
Writ

Fig. II - Updated Gantt diagram

It is easy to see that the biggest underestimation was that of the time that it would take to complete the pre and post-processing pipeline. We originally believed that we would be able to find a library that implemented the operations we needed, but that proved to not be the case. We found out that the field of digital signal processing in JavaScript is very new and underdeveloped. We had to take some time, therefore, to better understand the mathematical concepts behind the operations we needed (STFT/ISTFT) so that we could implement them ourselves.

We also underestimated the time it would take to convert the model, as we did not expect to have to change from the originally-planned Spleeter model to the new OpenUMX model. Fortunately, we were able to parallelize development, not having to stop working on getting the pipeline to work 100% to wait for the model.

6.2 Division of Labour

In the initial phase of research, each student worked individually. Once development started, we often did pair programming, bouncing ideas off each other. A great advantage of pair programming in a math-heavy project such as this one is that it ensures that all participants are on the same level of understanding of the theory behind the programm. A big part of the project was trying out different DSP libraries and trying to find a way to implement the STFT/ISTF operations in JavaScript, so we divided different libraries among ourselves and reported back on whether they did or did not work. Towards the end, we started to parallelize development, and one student was charged with making the website and the other with developing the tests.

7. Development

In this section we will describe the process of development of the project and also give an overview of how the application works.

7.1 Initial phase - An overview of existing technologies

We were tasked, initially, with studying the existing projects, mainly the Japanese company KoeKestra and the open-source unmixing model Spleeter. In this phase we also researched JavaScript libraries that implement DSP operations, such as the Short-Time Fourier Transform (STFT) or its inverse.

We quickly discovered that JavaScript is not a very popular language for the development of DSP or Machine Learning applications. Many DSP calculations are very computing-intensive, and until recently it made no sense to implement them with JavaScript, since for any web application these operations could be run more effectively in another language on a server. However, advancements in browser technologies, such as WebGL, allow for the execution of these operations on client-side.

Regarding machine learning libraries for JavaScript, there are two main ones: TensorFlow.JS and ONXX. We first looked into TFJS, which is, according to their GitHub ReadMe, *"an open-source hardware-accelerated JavaScript library for training and deploying machine learning models"*. It allows you to develop a model from scratch, use an existing Python model or retrain a pre-existing model. It uses WebGL to be able to execute operations on the GPU, optimizing their speed and has a built-in converter tool to convert between Python models and TFJS. ONXX also allows you to build models from scratch and to load pre-trained models. It also uses WebAssembly when WebGL is not available, increasing the performance on CPU. Finally, it also has converter tools to convert between different types of models, e.g. from TensorFlow to ONNX.

In our case, the initial plan was to use the Spleeter model, already developed using TensorFlow on Python, so the project tutor suggested that we use TensorFlowJS. This allows us to more closely "translate" the pre- and post-processing pipeline by using the same operations as the original model. TFJS also has a Node version which allows us to more quickly test our program without developing a UI.

We did not find a library that had both the STFT and the ISTFT. We did find some applications that had manually implemented these operations, but in most cases the implementations were not easily modifiable for our needs. One example of this is the STFT and ISTFT developed by Mikola Lysenko, available on his GitHub. It was suggested to us by the project tutor and was one of the first options we looked into. Unfortunately, the code was created for personal use, and did not have a lot of official documentation. This situation occurred with other STFT/ISTFT implementations, such as the one by KoeKestra, which had no official documentation, only comments on the code in japanese. Here is a brief overview of the libraries we tested and the operations they implemented:

Implementation	FFT	STFT	IFFT	ISTFT	Obs.
FFT.js [7]	✅	❌	✅	❌	
mikolalysenko/stft [8]	✅*	✅	✅*	✅†	No documentation * uses FFT.JS † Works only if you call ISTFT from the STFT callback.
Dsp.js [9]	✅	❌	✅	❌	
Essentia.js [10]	❌	❌	❌	❌	The status of this library, according to its GitHub

					ReadMe: *"Under development, unoptimized and use at your own risk."* ** The functions we would need were commented on as well.*
Magenta [11]	✅*	✅†	✅*	✅†	* uses FFT.js †STFT is from a different package than ISTFT, they are incompatible
TFJS	✅	✅	✅	❌	

Tab. I - An overview of STFT/ISTFT libraries for JavaScript

We decided to also use the TFJS library for DSP, since we were already using it for the model prediction part. For the ISTFT, we built it ourselves based on the Magenta implementation, using the TFJS IFFT.

7.2 Development - Converting the pipeline into JavaScript

7.2.1 Overview

The program follows the following pipeline: An audio file is decoded into a float buffer. We then perform a short-time fourier transform on this data. We transform the result of the STFT, which is an array of complex numbers in cartesian coordinates into polar coordinates (magnitude and phase). We take the magnitude and input it into the model. We use the model's output in conjunction with the phase we calculated before to reconstruct the signal. We then perform an inverse short-time fourier transform on this signal and finally save this result as an audio file.

We will now get into a bit more detail on each phase:

7.2.2 Audio decoding

The initial input a user will give to the program is an audio file. To go from an audio file to an array of floats we use what is called a **decoder**. If you are developing a browser application, for example, you can use the browser's built-in decoder through the **Web Audio API**. If you are developing applications on Node though, you will need to choose a decoder. Since in the beginning we were developing the program on Node, we had to find a decoder ourselves.

The first option we looked into was *audio-decode* [12], that is able to decode many different types of audio file. Unfortunately we found during development that it did not work correctly for our purposes. To still be able to develop our program without having to rely on a decoder, we used the browser's AudioContext to save a *txt file* of a decoded audio file. Later on, we decided to use *lame.js*, another audio decoder that actually worked.

To understand more about how music files are made, please look into the "Digital Audio Introduction" on our Technical Report.

7.2.3 STFT

Once we have the array buffers resulting from the decoding ready we can start preparing it for the model. To do this, we need to perform an operation called Short-Time Fourier Transform. A Fourier Transform is basically a **transform** (a mapping) that takes a signal (*function of time*) and transforms it into a **function of frequency.** It is still the same signal but represented in a different way, as shown in figure IV.

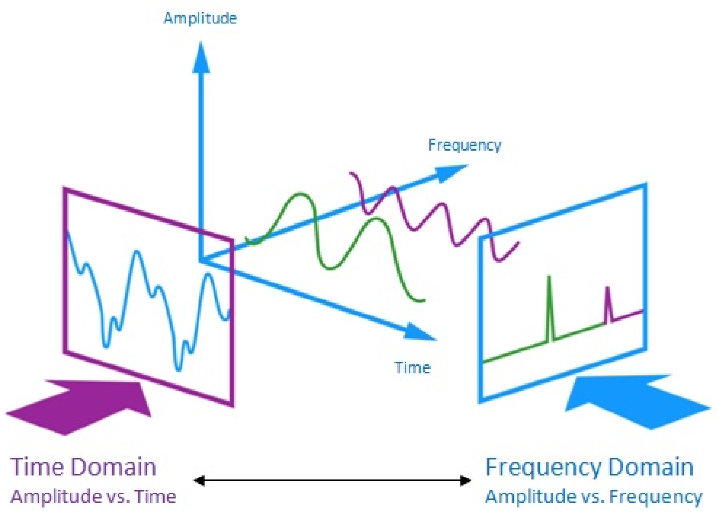


Fig. IV - Time Domain and Frequency Domain [13]

The short-time fourier transform (STFT) is a fourier transform that applies a discrete version of the Fourier Transform (DFT) into windowed segments, as can be seen in fig. V. The segments are windowed using a window function, such as a Hann window (see fig VI). We apply the DFT to these segments and get the results as complex numbers.

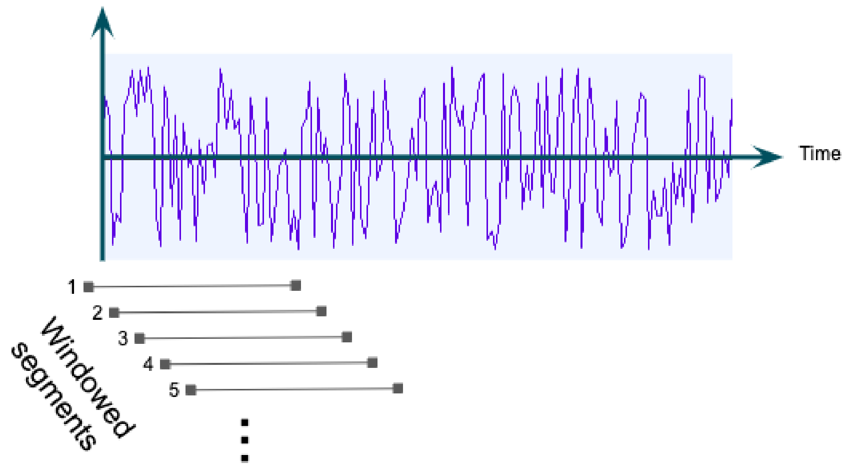


Fig. V - The STFT is essentially the DFT applied in different windowed segments

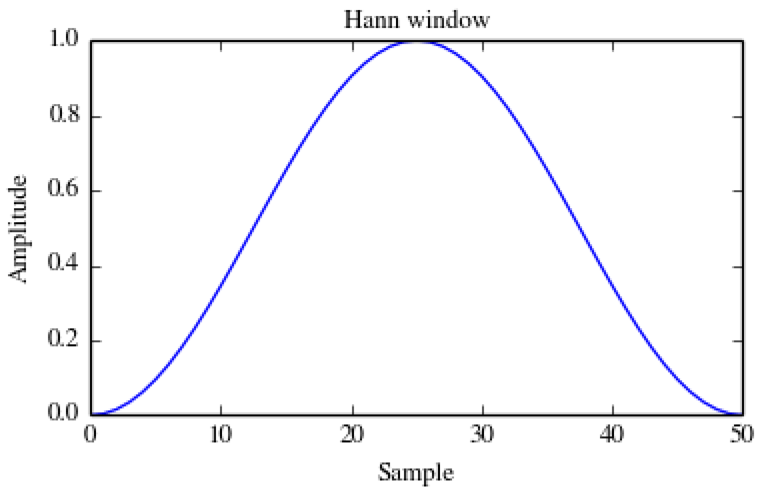


Fig. VI - Hann Window [14]

We chose to use the TensorFlow implementation of this function because it is the same library we used to load the machine learning model. This helps ensure compatibility between the pre-processing pipeline and the model, since, for example the model input has to be a TFJS Tensor. Furthermore, using this library allows us to represent complex numbers without having to resort to using the interleaved representation of complex numbers, as can be seen in the figure below:

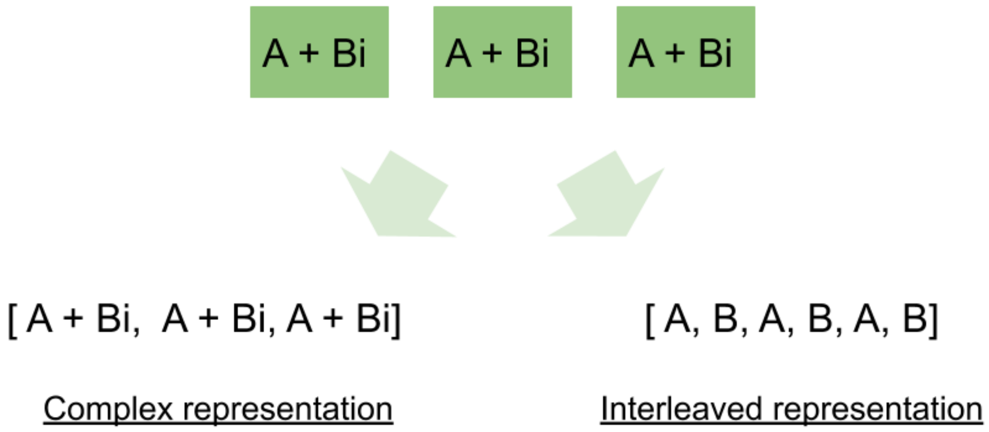


Fig. VII - Visualization of different ways to represent complex numbers

7.2.4 Model

The machine learning model is what allows us to actually separate the track into different instruments. Initially, the project tutor, Fabian Stöter, wanted us to use an open-source model developed by the French company Deezer, called Spleeter. This model was developed in TensorFlow 2.0 and has achieved state-of-the-art music separation results.

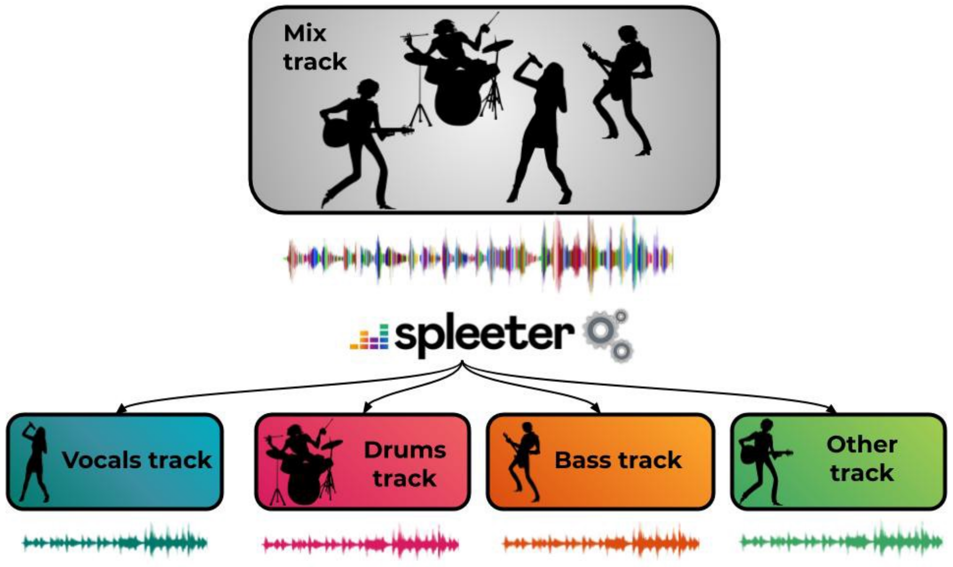


Fig. VIII - A visualization of the Spleeter model [15]

Unfortunately, we were unable to convert the model successfully to JavaScript. The Spleeter model uses many operations that are not yet implemented in TFJS, such as the ISTFT and straddled slice. While we were testing this conversion, the TensorFlowJs team released a new feature, that allows users to directly load Python models (TensorFlow's *SavedModel*), without having to convert them to TFJS (TFJS's *FrozenModel*). Sadly, this option is currently only available for the Node version of the TFJS library, but it still allowed us to test our pipeline.

Since the Spleeter model could not be converted, we tried using a model developed by the SigSep team using TensorFlow 1.5. This model did not have the same accuracy as the one developed in PyTorch, but it was easier to convert to TFJS. Unfortunately, this model also uses operations not yet implemented in TFJS, more specifically the Long Short-Term Memory cells.

We discussed our options with the project tutor and it was decided that he would train a new model that only used operations available on TFJS. This model took about a week to be developed and trained. The new model exchanges the LSTM layers for convolutions instead. While it does not have the same accuracy as the other models, it has other benefits, such as a more light-weight architecture that allow the model to run better on the browser and even on mobile devices. The new model exchanges the LSTM layers for convolutions instead

The basic principle of the model is to filter out of the song the frequencies that do not belong in the target instrument, as illustrated by figure IX. Our model only outputs the vocals of a song, but to get the accompaniment we can simply subtracted the filtered vocals from the original song.

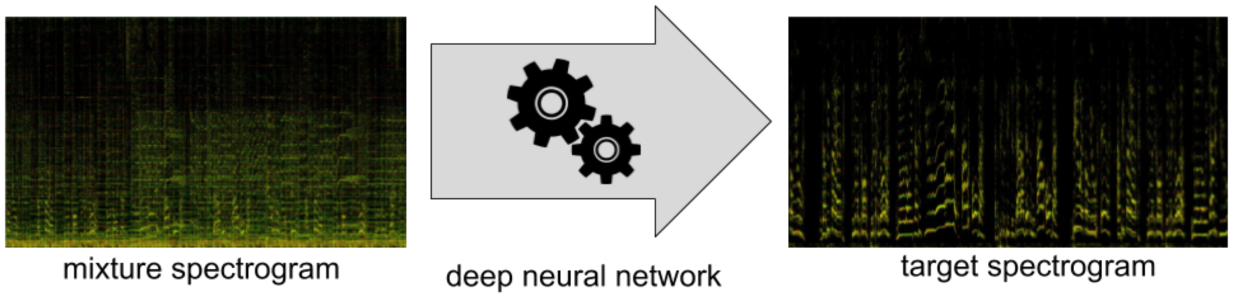


Fig. IX - Visual representation of the model

The input of the model is not simply the output of the STFT, we first need to transform it from cartesian coordinates into polar coordinates, that is magnitude and phase, as shown in figure X. While getting the magnitude is simple, getting the phase was a bit more difficult, as it involved operations that, on WebGL-enabled browsers, do not support complex numbers, e.g. stack. We got around it by separating the complex numbers into its real and imaginary parts, doing the operation and later rebuilding the complex number.

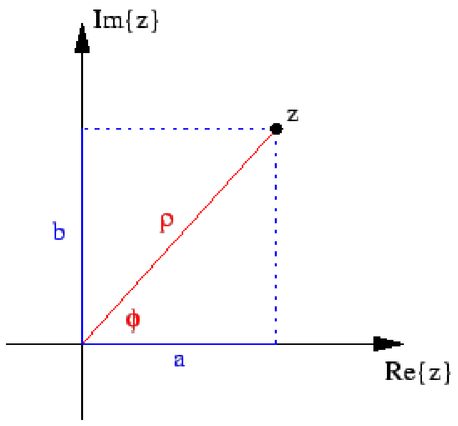


Fig. X - A complex number represented in cartesian and polar coordinates

Once we get the output of the model, we need to do the inverse, that is, go from polar coordinates back to cartesian coordinates. The formula for this is $Z = \rho e^{i\varphi}$, where Z is the resulting number, $\rho$is the magnitude and$\varphi$is the phase. Unfortunately we face the same problem as before, where we cannot do the operation $e^{i\varphi}$ with complex numbers JavaScript on the browser. To get around this, we used Euler's formula, $e^{i\phi} = cos(\phi) + i\,sin(\phi)$. This leaves us with a complex array that can be passed through the Inverse Short-Time Fourier Transform.

7.2.5 ISTFT

The Inverse Short-Time Fourier Transform performs the opposite operation from the STFT, that is, it takes a function of frequency and transforms it back into a function of time. To achieve this, it performs the inverse discrete fourier transform (IDTF) using a windowing function [16].

This was the function that we struggled with the most, since there are very few implementations of it in JavaScript. As mentioned before, we tried many different libraries to no avail. In the end, we decided to base ourselves upon an implementation from the MagentaJS library. This method is private and therefore not exported, so we had to re-implement it in our project. It also uses a fixed ISTFT spectrogram (window length, frame length) and the interleaved complex number representation (see figure VII), both of which are incompatible with our use case.

We, therefore, made some modifications so it would better suit our project, which included implementing a new windowing function and changing the IDFT implementation to TFJS, to maintain compatibility with the rest of the pipeline (STFT/model). These modifications involved a lot of research into the math behind this function, since any error in the window function, padding or overlap would result in incorrect audio outputs. The output of this function is an array of floats, ready to be converted back into audio.

7.2.6 Audio encoding

To transform a buffer back into audio we use an *encoder*. While we were developing on Node we use the library *wavefile*. On the browser, we opted to manually encode it back, allocating the headers, etc.

7.2.7 Tests

Together with the development of the Vue application, the tests were developed to speed up the progress of the project. Being this a project that involves precision testing it was a complex and arduous task. During one of our meetings with Fabian Stoter, we defined that the tests of the STFT-ISTFT functions would be done in an integrated way to observe the mean squared error of the Pre-Post processing.

For that, we did both manual and automated tests. As for the manual tests, we used Audacity to visualize the resulting waves also to check their details, for example, if they were not displaced or with missing frames. Using this wave viewer helped us to identify more quickly where the bugs were. Then we moved on to the automated tests such as the coverage test in order to verify that each branch of the code was being tested and also to the performance to analyze how it behaved with different input sizes.

7.2.8 WebDev - Learning to use Vue

We decided to use Vue because the project tutor had already developed their website using it and had components that could be reused. While we did not have any experience with vue, we already had some experience with other JavaScript frameworks such as React, so it was not very difficult to start developing.

The initial plan for the website was quite simple: build a component where you can upload a file, process it and play back the resulting song. The SigSep team had already developed a player component that allows the user to play more than one track at the same time {17]. We also found other components such as the drag and drop zone that made the development easier.

The project tutor asked us to keep the UI simple and to build a light-weight demo of the project. He asked us to develop something that could be transformed into an embeddable componted, so someone could easily reuse it on another webpage. We therefore opted for a minimalistic design with an intuitive UI.

7.2.9 Sharing website

Initially, we had the intention of creating a website where you would be able to share the results of the music separation. Unfortunately, we were not able to finish it during the project. We have already discussed it with the project tutor, and he agreed that it was best to focus on the unmixing pipeline before starting the sharing platform. Since we plan to continue working on the project, we will help to build it in the near future.

8. Final Results

8.1 File Structure

As a final result we obtained the following file structure:

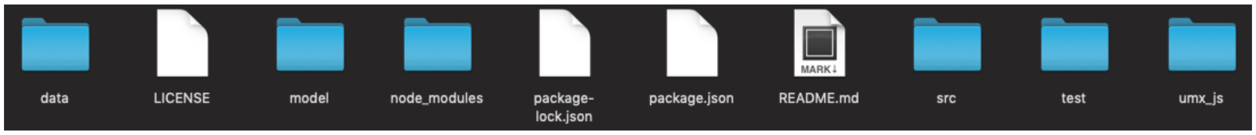


Fig XI - Unmix folder structure

- **data**: It has the files used for testing such as samples of songs and sample waves. Ignored on github.
- **model**: The graph model shards and its .json converted from Python to JavaScript.
- **node_modules**: NPM test dependencies as much as the module that is tested. Ignored on github.
- **src**: source code
- **test**: test files
- **Umx_js**: Vue application.
    - In the *umx_js* folder there is a folder called *bib* where the library is written purely in JavaScript, that is, it does not use any node dependency and the import of TFJS is done via script tag.

8. 2 Error

Mean Squared Error (MSE) tests were performed after the reconstruction of the signals (without passing through the model), that is, doing the STFT and then the ISTFT. These tests resulted in an error smaller than 10^{-10}. In the beginning of the project, the project tutor had informed us that it would be ideal to keep this MSE at less than . We can therefore say that we were successful in this respect.

8.3 Performance

We measured the performance of our program by running the same test in two different environments: a browser (Google Chrome) and on Node. The first test (2^{10}) takes longer

than the following tests because it includes the time it takes for the model to be loaded, which is only done once for each instance of the program.
We can see that, as the input size increases exponentially, so does the execution time. We can also see that the program is much faster on Node. This is probably due to the fact that the whole STFT and prediction pipeline can be run in parallel in Node, which is not possible in the browser since the program runs on a single thread.

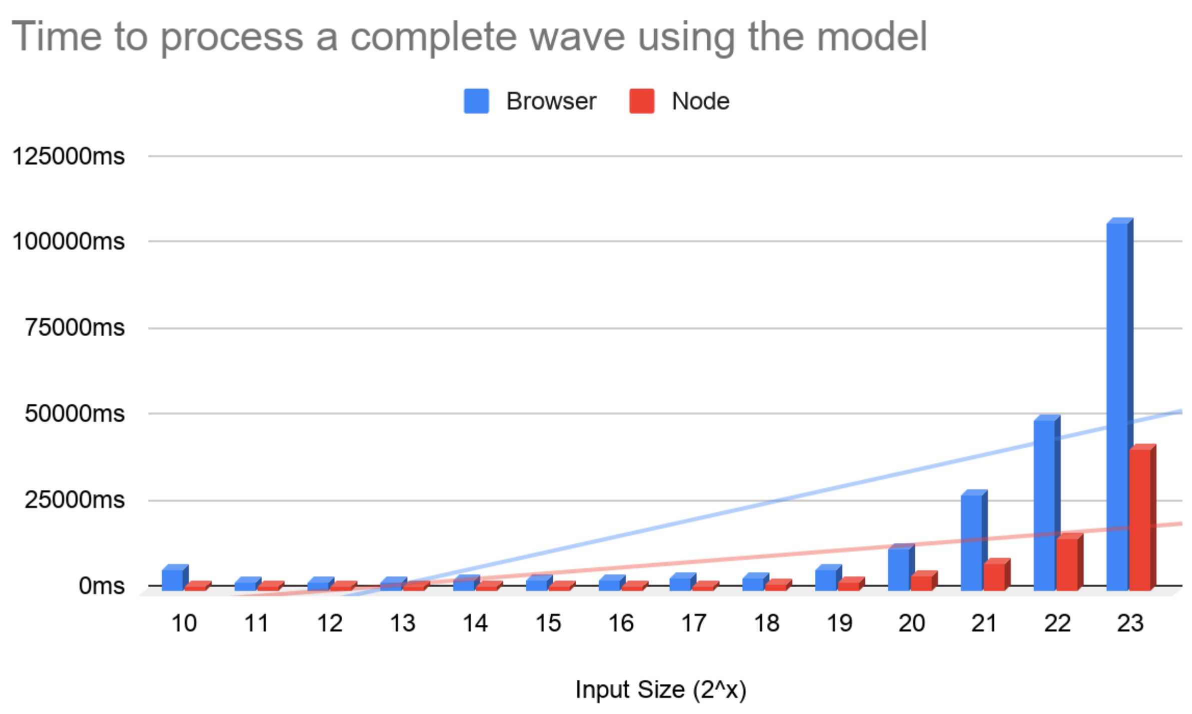


Fig. XII - Difference in execution time, browser vs. Node

Our website is available at https://sigsep.github.io/open-unmix-js/. When the user enters the website, they are presented with the dropzone:

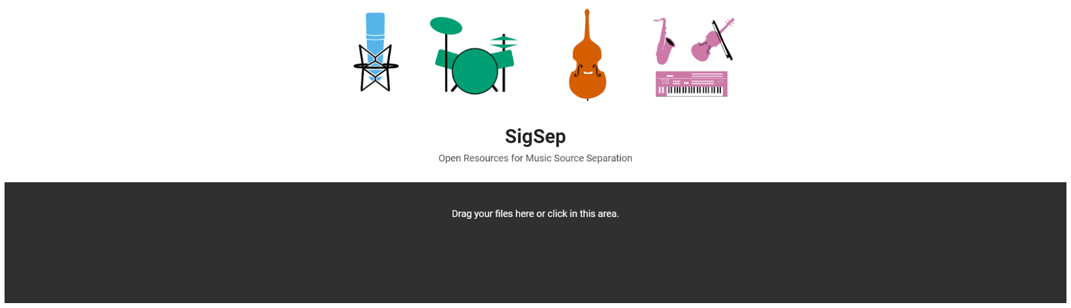

SigSep
Open Resources for Music Source Separation
Drag your files here or click in this area.

Fig. XIII - The site, with the dropzone

They can upload a file of up to 50 MB on the dropzone. Once they drop the file, the player and the “process song” button will appear. Once the audio file has been decoded, the “process song” button will be enabled:

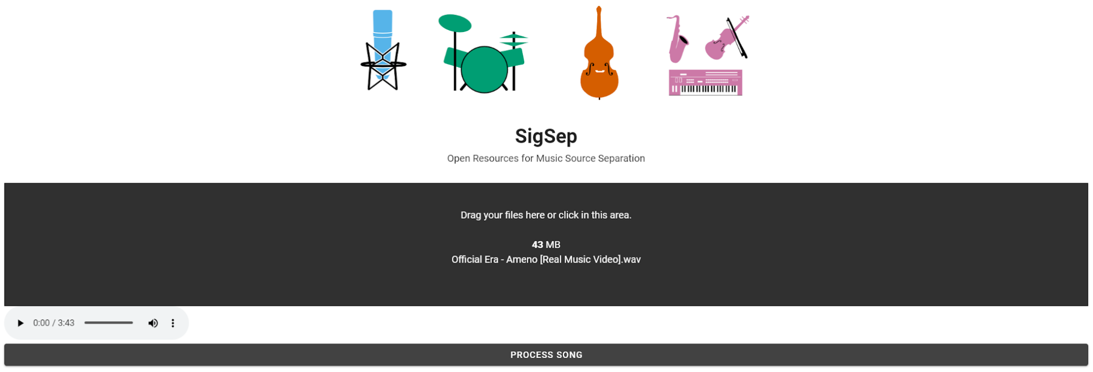

SigSep
Open Resources for Music Source Separation
Drag your files here or click in this area.
43 MB
Official Era - Ameno [Real Music Video].wav
0:00 / 3:43
PROCESS SONG

Fig. XIV - The site with a song fully decoded and button enabled

Once the user presses the “process song” button, this button is put into “loading” mode. Once the song has been unmixed, the dropzone, audio player and button are hidden, and the player is rendered. The player allows the user to mix the song on the fly: they can mute or solo the vocals and accompaniment individually. In addition, the user can download each stem as a wav file by clicking on the download button below the player:

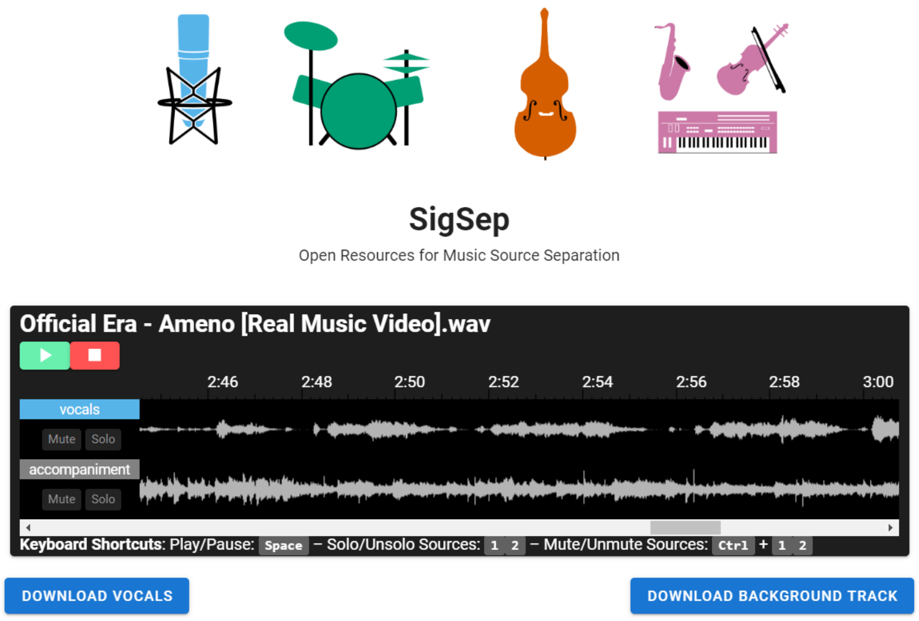

SigSep
Open Resources for Music Source Separation
Official Era - Ameno [Real Music Video].wav
2:46
2:48
2:50
2:52
2:54
2:56
2:58
3:00
vocals
Mute
Solo
accompaniment
Mute
Solo
Keyboard Shortcuts: Play/Pause: Space – Solo/Unsolo Sources: 1 2 – Mute/Unmute Sources: Ctrl + 1 2
DOWNLOAD VOCALS
DOWNLOAD BACKGROUND TRACK

Fig. XV - The results of the unmixing on the player

9. Future Improvements

The next step for this project is the creation of an NPM package containing both the trained and converted machine learning model and our pre and post-pipeline. This would allow other users to use the model in their own web applications. The website we develop will be included in it as a simple demo. As the model continues to be improved, new versions of the package will be published.

Another possible step is helping the TensorFlowJs team implement the ISTFT operation. Since we have developed our ISTFT using TFJS, we could, with some modifications, create a pull-request on their GitHub repository so that this operation would be officially included in the next TensorFlowJS release.

We have discussed with the project tutor our wishes to continue working on the project. We will create the NPM package and also the sharing platform that we were not able to complete on this project. There is also a possibility of writing a paper about the project.

10. Post-Mortem

Overall, this project was a lot more complicated than we initially thought. We did not predict taking so long on the STFT/ISTFT implementation, which led to us overpromising on the mission statement. When we first started our research into DPS libraries for JavaScript, we found many different options, but later we realized none of them fit in the use case of our project.

We were very lucky to have a tutor like Fabian Stöter, since he explained really well how the machine learning model worked, helped us understand some hard concepts in DPS like the STFT/ISTFT, window functions etc. Plus he was always available on Slack, sometimes answering our questions at night or on the weekends.

When we realized we would have to manually implement the ISTFT, we started researching more deeply into the math. We had last studied Fourier Transforms in 2016 in University, so we were quite rusty in those respects. We spent a lot of time realering the Fourier basics and the STFT/ISTFT specifics.

Overall, we would say that, while we were not initially prepared to implement complex mathematical operations, we managed to do so successfully. A more careful initial planning would have helped us correctly set expectations and reduce the scope of the project.

11. Conclusion

This project was an incredible learning opportunity for us. It allowed us to work with bleeding-edge technologies both in the field of machine learning and javascript development. We never imagined in the beginning of the project that we would have implemented all of the things that we did or how challenging and rewarding it would be.

In this project we had the opportunity to discover a bit of the world of Digital Signal Processing. It is a whole field of computer science we had never had any contact with, and probably would never have had the opportunity to work with if not for this project. It was fascinating to learn more about the audio world and finding out more about things that we interact with every day when we listen to music, such as sample rate and the differences between various audio file formats.

To conclude, this project proved to be an amazing learning opportunity for both of us. We got to implement complex operations, dive deeply into mathematics and built an application using cutting-edge technology. We are satisfied with the project and results we achieved, and consider it a success.

12. Bibliography

[1] SigSep website. https://sigsep.github.io/

[2] Google. *"TensorFlowJS",* 2018. Github repository. https://github.com/tensorflow/tfjs

[3] Koekestra. *"Spleeter implementation by JavaScript"* (in japanese). Website. https://koekestra.com/spleeter_js/

[4] Deezer. *"Spleeter",* 2019. Github repository. https://github.com/deezer/spleeter

[5] Jansson, Andreas; Humphrey, Eric;Montecchio, Nicola; Bittner, Rachel; Kumar, Aparna;Weyde, Tillman. "*SINGING VOICE SEPARATION WITH DEEP U-NET CONVOLUTIONAL NETWORKS*". https://ejhumphrey.com/assets/pdf/jansson2017singing.pdf

[6] Microsoft. "*ONXX.JS"*. Github repository. https://github.com/microsoft/onnxjs

[7] Nockert, Jens. "A *Fast Fourier Transform library for JS".* NPM package https://www.npmjs.com/package/fft

[8] Lysenko, Mikola. *"Short time Fourier transform".* Github repository. https://github.com/mikolalysenko/stft

[9] Brook, Corban. "*Digital Signal Processing for Javascript*". Github repository. https://github.com/corbanbrook/dsp.js

[10] Music Technology Group. *"EssentiaJS".* Github repository. https://github.com/MTG/essentia.js

[11] Google. "*Magenta.js*". Github repository. https://github.com/tensorflow/magenta-js

[12] AudiJS. "*audio -decode*". NPM Package. https://www.npmjs.com/package/audio-decode

[13] NED University, "*EE-373 MACHINE CONTROL SYSTEM - aa*". Image. https://med.neduet.edu.pk/node/131

[14] Scipy. *"Hann Window".* Website https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.hann.html

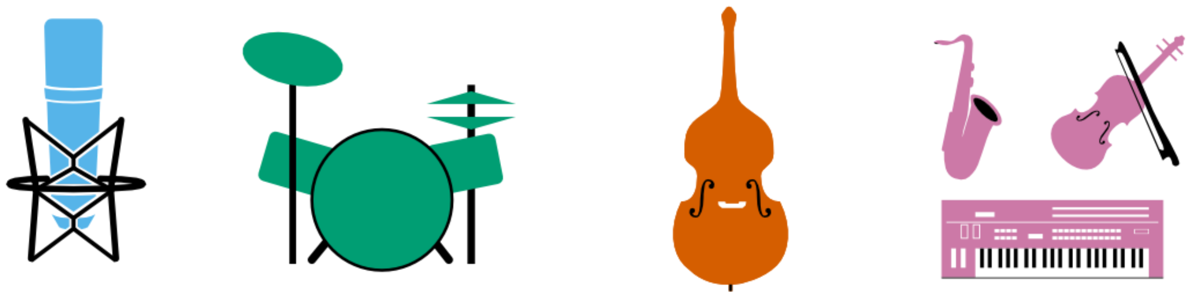
[15] Moussallam, Manuel. "*Releasing Spleeter: Deezer Research source separation engine*", November 4, 2019. Medium Article. https://deezer.io/releasing-spleeter-deezer-r-d-source-separation-engine-2b88985e797e

[16] ftp://ftp.lpp.polytechnique.fr/jeandet/keep/sync/fftw/Window%20function.pdf

[17] Stöter, Fabian. *“open-unmix stem player”.* Github repository. https://github.com/sigsep/stem-player

Appendix

Technical Report



Machine Learning for Audio on the Web

Maria Clara JACINTHO

Delton VAZ

Computer Science and Management, 5th year

University Tutor: Dr. Chouki TIBERMACINE

Project owner: Fabian-Robert STÖTER

2020 - Montpellier

1. Figures

2. Glossary

Term	Definition
DSP	Digital Signal Processing
Fourier Transform	“The Fourier transform decomposes a function of time (a signal) into its constituent frequencies.”[1]
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform - an algorithm that implements the DFT
STFT	Short-time Fourier Transform
ISTFT	Inverse Short-Time Fourier Transform
LSTM	Long short-term memory- “ is an artificial recurrent neural network architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points, but also entire sequences of data.”[2]

[1] https://en.wikipedia.org/wiki/Fourier_transform
[2] https://en.wikipedia.org/wiki/Long_short-term_memory

3. Project Intro

This project has as its objective to transform a neural network model previously written in Python into JavaScript. The model in question is a “Music Source Separation” model, that is, it takes a song and separates its instruments (vocals, drums, bass and others). It was proposed by the SigSep team at the INRIA (*Institut national de recherche en informatique et en Automatique*) and is based on their previous work [1].

Thanks to TensorFlow’s JavaScript port, TFJS [2], the field of machine learning in JavaScript has grown a lot recently. Models created in Python could, in theory, be run on JavaScript, with the exception of a few that used operations not present in TFJS yet. This would allow, for example, the creation of a website where the inference, that is, the prediction by the AI model, is done client-side, eliminating the need for a server.

Since the model by the SigSep team was written in PyTorch [3], which is incompatible with TFJS and has no direct JS equivalent, it was decided in a meeting before the project began that we would initially use another open-source python model, written using TensorFlow: Spleeter [4], by the Deezer research team [5]. In a later date, outside of the scope of this project, the model would be substituted by one of SigSep’s own models.

We were tasked with adapting the pre-processing pipeline, which is similar in both models, from Python into pure JavaScript. Once that was done, we had to either convert the Spleeter model by ourselves or find an already converted version online. Finally, we were to create a website that takes a song, pre-processes it, runs the model on it and outputs the separated sources, playable and downloadable directly on the browser.

In this report, we will analyze the technical aspects of the project. We will start by going over our technology choices, followed by a short introduction into the field of Digital Audio and Fourier Transforms. We will then review the state of the art in the digital signal processing in JavaScript field. Then, we will give a general overview of the project, followed by an in-depth look into each step of the music-processing pipeline.

4. Technology choices

- **JavaScript** - We are using JavaScript since it is the language of the web, even though it is not the language typically chosen for digital signal processing or machine learning applications. Since it is a very popular language, there is a huge community surrounding it, so it is easy to find solutions to bugs. In addition, both students were familiar with JS and had previous experience with it in other projects.

- **TensorFlowJS** (TFJS) - It is a machine learning library developed to be the equivalent of Python's TensorFlow. We decided to use it since it has a converter tool for models developed in Python, eliminating the need to totally re-develop and retrain a model in JS. Furthermore, it also provides some of the digital signal processing operations needed for the project.

- **Node** - We used Node during the development of the project since it was easier to test our program without having to develop an UI. Plus, the Node version of TFJS is much faster than the non-Node one in the IDE, which allowed us to test our program more quickly.

- **Mocha** - We chose Mocha as our testing framework since it has a powerful but simple test-defining syntax. Furthermore, we were already familiar with it from previous projects.

- **Vue** - Vue is a framework for building interfaces and single-page applications. We decided to use it since the project tutor had already developed Vue components that could be reused in this project.

- **Github pages** - It is a simple and free way to host single page applications.

5. Digital Audio Introduction

To be able to perform the unmixing of a song, we first need to understand how to record a sound and store it in a digital system. Sound is an acoustic wave that produces vibrations through a propagation medium. It can be represented by a wave where the x-axis is time and the y-axis is amplitude of the signal, i.e. how loud it is (see Fig. I).

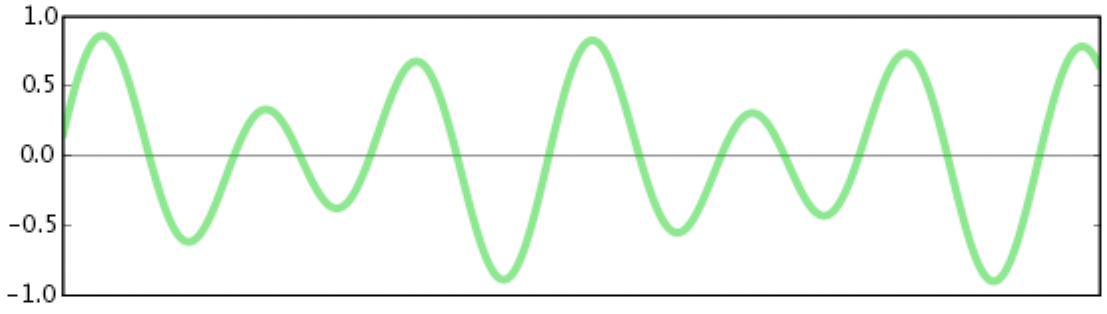


Fig I - An audio wave [6]

If the wave is simple, e.g. a sine wave, you can simply store its mathematical formula (*sin(x)*). When we are dealing with more complex signals, such as a song, that is not possible. What is usually done is discretizing the signal, that is, getting the y-value at a fixed sample rate, thus allowing us to represent a continuous signal in a discrete way. Since we keep the sample rate fixed, we also do not need to record any information about the x-axis (time), since we know that the first sample is at time 0 we can deduce the time of the next sample using the sample rate. The result is an array of values that can be reconstructed into the original wave [7]. There is usually also some sort of compression applied to this data, to save memory.

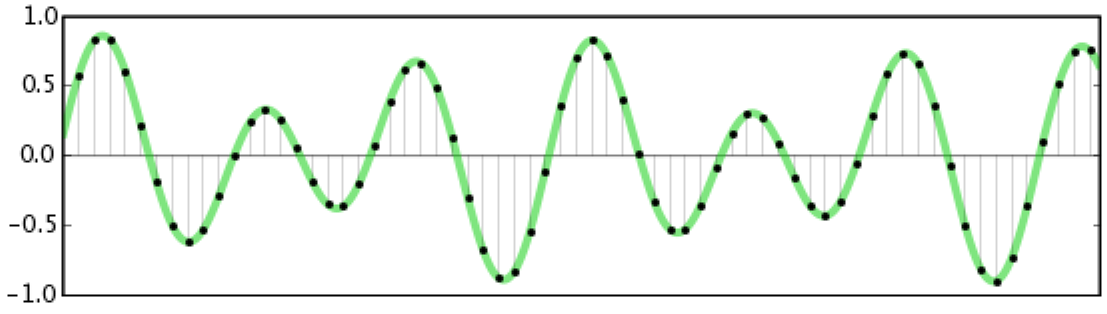


Fig II - An audio wave with the samples marked as black dots [6]

To be able to use an audio file for unmixing, we need to get the raw audio data (the array of values) by using a decoder. In the initial phase of development, we used Node.js, where we used libraries such as **LAME** [8] and **node-wav** [9], both for decoding the audio and encoding it back once we passed it though the model. For the website, we simply used the Web Audio API, that harnesses the browser's own decoders to decode the audio. To encode the audio back into a file, we used a function that manually creates a wav file.

5.1 What is a Fourier Transform?

A Fourier Transform is a function that takes a signal, which is a function of time, and splits it up into the frequencies that compose it. It is a transformation of a function into the **frequency domain**, a complex-valued function of frequency. It still represents the same signal, it is just a different way of looking at it. For example, let us take a signal like in figure III:

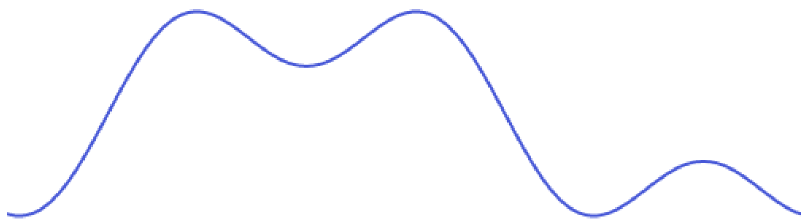


Fig. III - A signal [10]

We can decompose the signal into sine waves. That is to say, if we sum those sine waves, we will get the original signal back:

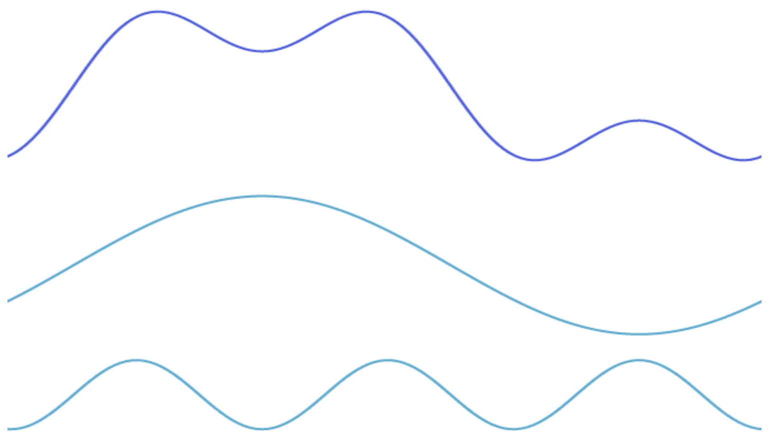


Fig. IV - The signal decomposed [10]

If we have a finite set of points, equally spaced, such as a sampled audio file, we could use the Discrete Fourier Transform (DFT) instead. To calculate a DFT, the algorithm usually used is the Fast Fourier Transform (FFT).

Transforming the signal from the time domain to the frequency domain is interesting for us in this project because it allows us to see which frequencies make up the song and later, with the machine learning model, which frequencies belong to which instrument, allowing us to “unmix” the song.

6. The pipeline - how a song gets unmixed

6.1 Overview of the process

Initially, we take an audio file and decode it into one float array for each audio channel, using a browser’s **AudioContext** or with the **node-lame** audio decoder. We then process each channel individually. Due to the limitations of the browser, we cut the song into chunks and process them individually. Firstly, we perform a Short-time Fourier Transform (STFT) on the signal, which results in a complex Tensor. We then take the results of these operations for both channels and calculate their absolute value (magnitude), which will be the input of the machine learning model and the phase of the mix, to be able to reconstruct the signal later. We then pass the signal through the model, resulting in a filter. This filter, at the moment, is able to isolate the vocals of a song. We are also able to get the accompaniment of the song by simply subtracting the vocals from the original song. We then reconstruct the audio with an Inverse Short-Time Fourier Transform.

6.2 State of the art on digital signal processing in JS

A big part of the project was adapting the existing pre and post-processing pipelines from Python to JavaScript. These processes involve lots of mathematical operators, such as Fourier transforms. In Python, there exist many libraries that handle this type of operation, and even TensorFlow itself is capable of doing them, creating a seamless pipeline.

In JavaScript, it is a different story: the field of digital signal processing (DSP) in JS is relatively new. While there are some libraries that deal with DSP, we struggled to find one that had the operations that we needed, the Inverse Short Time Fourier (ISTFT) in particular. The libraries that exist either did not have the ISTF did not expose it (that is, it is a private function), or simply lacked the documentation necessary to understand how to use it.

Some of the libraries we looked into were:

- ***TFJS*** - Has the STFT but no inverse. There was an issue on the GitHub page to add it, but it was no longer being worked on.
- ***MagentaJS*** [11]- Has both the STFT and ISTFT but they belong to different packages and are not compatible with each other. Both are private functions, and therefore we were not able to use them directly. We ended up reusing some of their ISTFT code with some adaptations.
- ***EssentiaJS*** [12]- Essentia is an open source C++ written library for audio analysis and audio-based retrieval of musical information. The EssentiaJS translates the result that is obtained from the C++ implementation.
- ***Dsp.js*** [13] - does not have STFT/ISTFT
- ***mikolalysenko/stft*** [14] - a simple JS code for both the STFT/ISTFT, using callbacks. Due to the lack of documentation, we were not able to adapt this code to our needs.

We were also heavily inspired by Koekestra, a Japanese developer that also created a website for music source separation using the Spleeter Model. This project had its own DSP library, that was commented on in Japanese. Besides not being an open-source code, it was also very difficult to understand. Therefore, despite being a great inspiration, we did not use any of its code.

Since our project originally uses a model that was built using TensorFlow in Python, we prioritized the TFJS library. The operations in TFJS produce the same result and are compatible with those in Python, so using them would give us the best shot at replicating the original pipeline. There are limitations, of course, such as the lack of some operations (like the ISTFT) or the fact that some operations with complex numbers (such as stack, exp) do not work on the browser, only in Node, so there were adaptations made to the pipeline.

6.3 Short-Time Fourier Transform (STFT)

For this project, we need to understand how the frequencies that compose a song change over time. For that, we need to perform a Short-Time Fourier Transform. Essentially, we apply the DFT to windowed segments of the data, and then display the DFT coefficients as a function of time and frequency [15], as shown in figure V. It is usually represented by a spectogram, which you can see in figure VI.

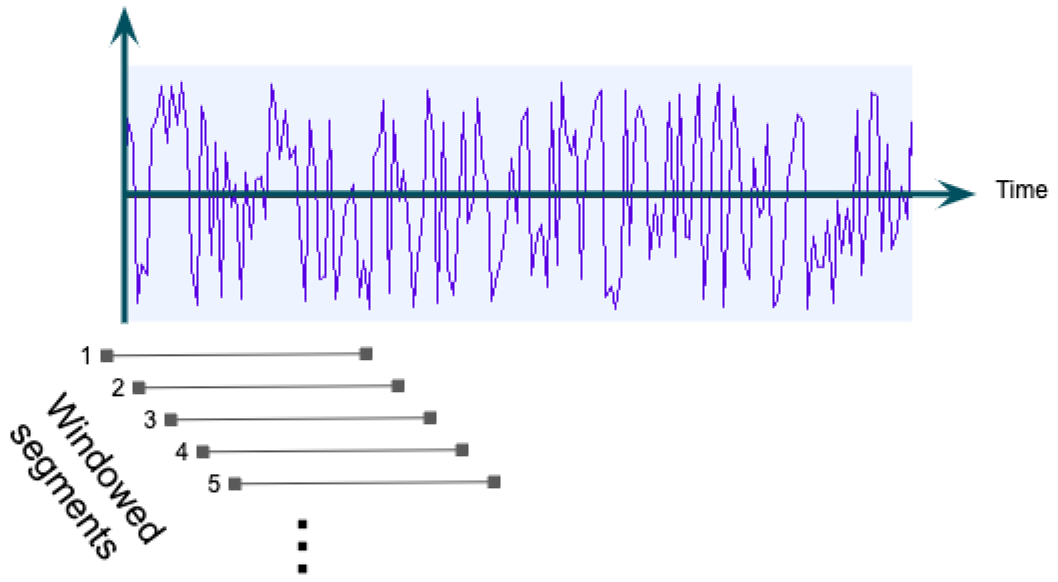


Fig. V - The STFT is essentially the DFT applied in different windowed segments

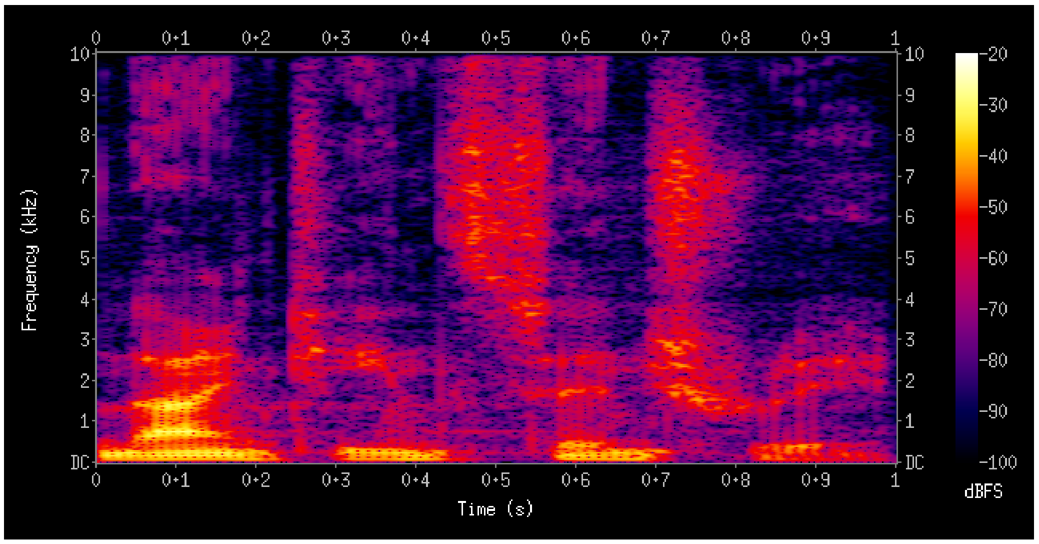


Fig. VI - The spectrogram, showing the variation of the intensity (in dBFS) of a frequency (in Hz) through Time (in seconds) [16]

There are relatively few libraries that provide this operation in JavaScript. We tried many different libraries, such as MagentaJS or the implementation by Mikola Lysenko, but found that most lacked documentation. We decided to use the implementation by TensorFlowJs, since it ensures compatibility with our machine learning model and with the Python implementation, that also uses the TF's STFT. The signature of the function in JavaScript is as follows:

```
tf.signal.stft (signal, frameLength, frameStep, fftLength?, windowFn?)
```

where:

signal (tf.Tensor1D) 1-dimensional real value tensor.

frameLength (number) The window length of samples.

frameStep (number) The number of samples to step.

fftLength (number) The size of the FFT to apply. Optional

windowFn ((length: number) => tf.Tensor1D)

We use:
- **frameLength** = 2048
- **frameStep** = 1024
- **fftLength** = 2048
- **windowFn** = the standard Hann window

The original model in Python uses **frameLength** = 4096, which slows down the computation but increases the frequency resolution. We opted to change this to optimize the programm, since the STFT is already quite a computationally heavy operation for a browser.

The JavaScript version of the STFT also does not have the "**pad_end**" option, to fill the last frame of the signal with zeros to ensure that it has size **frameLength**. We also found that the function creates a "soft beginning" in the song, as shown in figure VII. This problem is caused because the window function was discarding the beginning of the signal. To fix both these issues, we created a function called padSignal, that applies a pad before and after the signal. This pad adjusts the window so no information about the signal is lost.

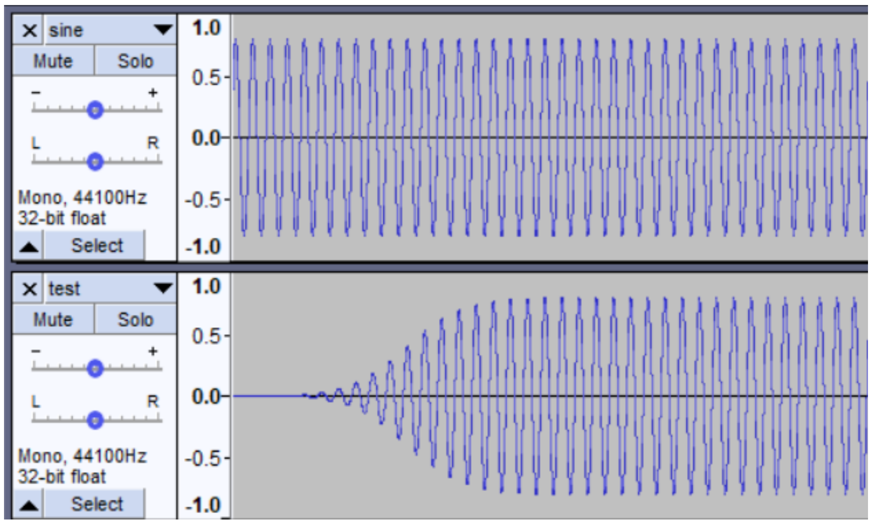


Fig. VII - The original song (top) vs. the song with a soft beginning (bottom)

The result of the STFT operation is a 2D Tensor of size [X][1025], where X varies according to the length of the signal. Before we can input it into the model, however, there are still some operations that need to be done.

6.4 Model

6.4.1 Model conversion

The initial idea was to use Deezer's Spleeter model, since it was developed using TensorFlow, and we could then use the in-build conversion tool from TF to TFJS. Furthermore, there is a website that uses the Spleeter model, made by a Japanese developer.

As we came to discover later, though, it is not possible to simply convert this model to TFJS. When the model was saved, it included the pre and post-processing pipelines, that is, the STFT and the ISTF, and therefore uses operations that are not yet available in TFJS yet (i.e. the ISTFT).

Since the Spleeter model did not work, the SigSep team provided us with a version of their TensorFlow model. We were also unable to convert it to a TFJS model, since it used the Long Short-Term Memory operation, which is also not available on TFJS yet.

We were, however, able to load and use both the Spleeter and the SigSep model with the **tfjs-node** library, directly using the Python saved model instead. This feature was actually released in early January, a testament to how fast the field of machine learning on the web is advancing, and how this project is working with cutting-edge technologies. Unfortunately, since our project involves developing a client-side application, we were not able to continue using this method of loading the model, although it was useful for testing the whole pre- and post-processing pipeline.

We discussed with the SigSep team ways to actually be able to use a model on a client-side application. We decided that the team would train a new model, only with operations available on TFJS. This model is not as accurate as their PyTorch model, but it still performs the sound separation. We decided to move forward with it since building the website is more of a proof-of-concept, and we can easily replace the model with an improved version if the website works well. Furthermore, this new model is more lightweight than the original, making it better suited for the web, since it takes less time to load and to predict.

Once the model was trained, the conversion was simple: we took the SavedModel (.pb) and ran the tfjs-converter script, resulting in a tfjs Graph Model (model.json) and a collection of binary weight files. To load the model, we simply pass the location (URL) of the *model.json*. In the case of our website, the model is located in the *public* folder.

6.4.3 Model input

The model input is not simply the output of the STFT. Up until this point, each channel was being processed separately. Now, we will combine them again to create the model input. We then need to reshape the results so they match the model's input, which is a 4D tensor of shape [256][1][2][1025], where:

- 256 frames
- 1 sample/batch
- 2 channels
- 1025 frequencies

Sometimes, the result of the STFT does not have exactly 256 frames. It usually occurs at the very end of the song, if the length of the song (in samples) is not perfectly divisible by 2048 (frame length). Since our model needs exactly 256 frames, if the current chunk has less than that, we add frames to it, as shown below, where FRAMES = 256 and FREQUENCIES = 1025 and resultSTFT is a 2D tensor, with shape [X][1025]:

```
let number_of_frames = resultSTFT[0].shape[0]

if(number_of_frames < FRAMES){
    let fillZeros = FRAMES - number_of_frames
    let pad =tf.zeros([fillZeros,FREQUENCIES], 'complex64')
    resultSTFT[0] = tf.concat([resultSTFT[0], pad])
    resultSTFT[1] = tf.concat([resultSTFT[1], pad])
}
```

There are two main components needed for the model: the magnitude and the phase of the STFT. The magnitude is used for the model prediction and the phase is used to reconstruct the signal. This calculation is transforming a complex number (or, in our case, tensor) represented in the **cartesian coordinates** $z = a + ib$ into **polar coordinates**.

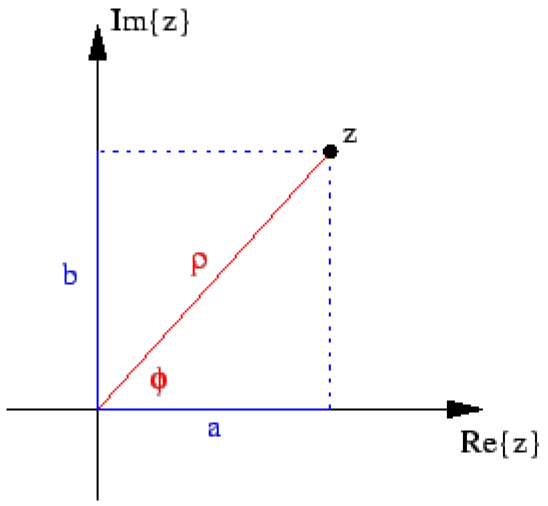


Fig. VIII - A complex number represented in cartesian and polar coordinates [17]

To find the magnitude ρ and the phase Φ we can use the following mathematical operations:

$$\rho = \sqrt{a^2 + b^2}$$
$$tan(\phi) = \frac{b}{a}$$

We can easily calculate the magnitude by taking using TFJS's *tf.abs* operation. To achieve the input shape, we first calculate the absolute of each channel separately, which results in a [256][1025] tensor. We then stack these results on top of each other, and transpose it, to make a [256][1025][2] tensor. Finally, to create a dimension of size 1, we use the *expandDims* operation, thus achieving the correct input shape.

```
 let absChannel0 = tf.abs(res0)
 let absChannel1 = tf.abs(res1)

 const model_input = tf.stack([absChannel0, absChannel1]).transpose([1, 0, 
2]).expandDims(1)
```

For the phase, it is a bit more complicated, since we are not able to use the *tf.stack* operation on complex numbers. We first need to transform them into float tensors, perform the same stacking and reshaping operation described above, and then turn the result back into a complex tensor. We are then able to calculate the angle of the phase *Φ* by using the arctangent operation, as shown in the code bellow:

```
    let chan0R = tf.real(res0)
    let chan0I = tf.imag(res0)
    let chan1R = tf.real(res1)
    let chan1I = tf.imag(res1)

    let chanR = tf.stack([chan0R.arraySync(),
chan1R.arraySync()]).transpose([1, 0, 2]).expandDims(1)
    let chanI = tf.stack([chan0I.arraySync(),
chan1I.arraySync()]).transpose([1, 0, 2]).expandDims(1)

    let mix_stft = tf.complex(chanR, chanI)

    let mix_angle = tf.atan2(tf.imag(mix_stft), tf.real(mix_stft))
```

6.4.4 Model Prediction

Now that we have the correct input shape, we are able to use the model. But what exactly is the **output** of the model? Simply put, the output is a **filter** that, when applied to the song, filters out all of the frequencies that do not belong to a given instrument. In our case, our lightweight model only gives us a filter for the vocals, but we can still get the accompaniment track by subtracting the vocals from the song. We will not go into the details of the model's architecture or inner workings, but if you want more information you can read the original paper by the SigSep team [1]. In this section we will take a closer look at all these processes.

Firstly, to get the output of the model, we simply need to call *model.predict.* We wrap this in a *tf.tidy* statement, to ensure that all tensors left over in the computation are disposed of, and not kept on memory:

```
    const output = tf.tidy(() => {
        return model.predict(input["model_input"]);
    })
```

The result is a tensor with the same shape as the input: [256][1][2][1025]. To reconstruct the song, we will do the inverse operation from before, essentially going from the polar coordinates to cartesian coordinates. The formula is:

$$Z = \rho e^{i\phi}$$

Unfortunately, we are not able to do the $e^{i\phi}$ operation, since the TFJS's *tf.exp* operation does not work with complex numbers on browsers. We have to use Euler's formula instead:

$$e^{i\phi} = cos(\phi) + i\,sin(\phi)$$

And therefore we have:

```
let estimate = tf.tidy(() => {
        return tf.mul(
            tf.complex(output, tf.zeros([FRAMES, N_BATCHES, N_CHANNELS,
FREQUENCIES])),
            tf.complex(tf.cos(input["mix_angle"]),tf.sin(input["mix_angle"]))
        //tf.exp(tf.complex(tf.zeros([FRAMES, N_BATCHES, N_CHANNELS,
FREQUENCES]), input["mix_angle"]))
        )
    })
```

To get the background track (without vocals), we do the same operation, but instead of using simply *output*, we use:

```
(input["model_input"].sub(output))
```

These operations result in complex tensors that can be passed through the Inverse Short-Time Fourier Transform and then saved as an audio file.

6.5 Inverse short-time Fourier Transform (ISTFT)

The Inverse Short-Time Fourier Transform was one of the most difficult parts of the project. This operation is not yet implemented in the TFJS library, and neither in any other digital signal processing libraries in JavaScript. Most implementations we found manually implemented the windowing function and some even implemented the inverse discrete fourier transform.

The implementation we found that works best with our preprocessing pipeline was from the MagentaJS library. This library, also maintained by Google, implemented the JavaScript-equivalent of the ISTFT from TensorFlow, but without using any TFJS operations.

Unfortunately, this ISTFT operation was an internal function and not exported, so we could not use it directly. Besides, it has many fixed values that were incompatible with our use-case. Our solution was to base ourselves upon this function but adapt it to our needs.

The basic idea of the ISTFT is to be the inverse of the STFT, that is, to perform the Inverse Discrete Fourier Transform using a specific windowing function in order to recover the original signal.

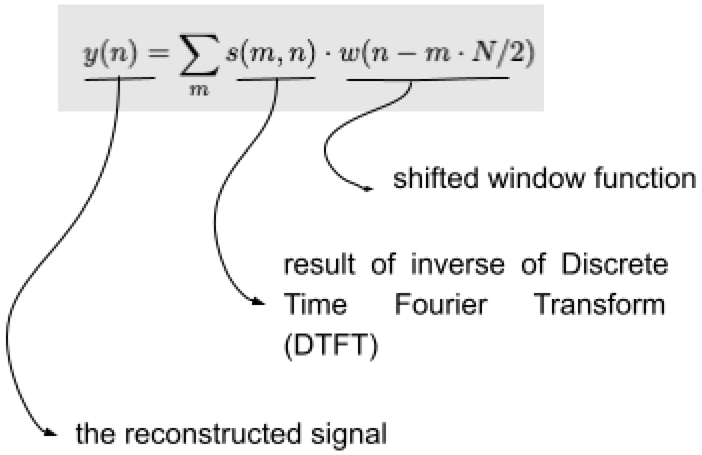

$y(n) = \sum_m s(m,n) \cdot w(n - m \cdot N/2)$
shifted window function
result of inverse of Discrete Time Fourier Transform (DTFT)
the reconstructed signal

Fig X - Sum of overlapping blocks to obtain the final signal [18]

To allow inversion of an STFT via the inverse STFT, it is necessary that the signal windowing obeys the constraint of Constant OverLap Add (COLA) conditions. This ensures that every point in the input data is equally adjusted, avoiding aliasing and allowing full reconstruction.

MagentaJS implementation uses a Node library called FFT [19] to compute the IFFT on a single frame and has a COLA [20] normalization factor equals to 75% using a Hann windowing function. It does not fit in our case since the Hann windowing function for normalization used by the model provided by the SigSep team has a factor equals to 100%.

Another problem that we encountered is the fact that Magenta uses interleaved complex representation e.g. if we have a complex array such as [a+ib, c+id], MagentaJS represents it as [a,b,c,d]. This is not compatible with our STFT implementation or our model, both of which use TFJS *complex64* type to represent complex numbers.

We decided that, instead of continuing using the Node FFT library, we would substitute it for a TFJS function. TensorFlowJS implements the Inverse Real Fast Fourier transform (IRFFT) which returns only the real part of the Inverse Fast Fourier Transformed. We only need the real part of the answer since the imaginary part will always be equal to 0 anyway, since we are rebuilding a signal of Real values. With this it is possible to develop ISTFT using only native functions of TensorFlowJs.

Firstly we calculate the expected size of the output signal, that is, the sum of all the m parts from the equation above, and allocate a memory space for the output:

```
const expectedSignalLen = nFft + hopLength * (nFrames - 1);
const istftResult = new Float32Array(expectedSignalLen);
```

We then the execute TensorFlow's IRFFT:

```
let irfftTF = complex.irfft()
```

Adjust the normalization to the condition of 2096 FFT divided by the size of the hop which is 1024. In this case the, as said before, the Hann windowing function for normalization is 1 but we left it as a dynamic parameter to cover the possibility of future changes in the model inputs:

```
let normalizationFactor = tf.mul(ifftWindowTF, tf.tensor(winFactor));
```

The window function for the signal reconstruction is then applied. The arraySync() method returns an array with the values instead of a tensor object:

```
let res = tf.mul(irfftTF, normalizationFactor).arraySync();
```

And finally we overlap and add [21] the blocks to obtain the final signal y(n):

```
for(let i = 0; i < nFrames; i++) {
       let sample = i * hopLength;
       let yTmp = res[i];
       yTmp = add(yTmp, istftResult.slice(sample, sample + nFft));
       istftResult.set(yTmp, sample);
   }
```

The result of this function is a float array. We do not process the whole song in one go, but instead cut it into chunks. After each chunk is processed it is added to a result array. This array will later be transformed back into audio files.

6.7 Audio Encoding

To transform the float array that is outputted from the ISTFT back into an audio file, we need to encode it. During the initial phase, developed with Node, we used the *wavefile* library to create a wav file. For the browsers, we opted for building the wav file manually, that is, manually setting the headers that identify the file as a wav, as it appears to be one of the fastest solutions for audio encoding. We chose to output a wav file because it eliminates the need for compression (at the expense of a larger file) and to maintain consistency with the Python model, that also outputs this format.

7. Site

To be able to put the whole pipeline that we have developed, we created a simple Vue website. We chose to use Vue because the SigSep team had already developed components that could be reused on our site. We settled for a simple, minimalistic UI, with a focus on creating a component that could be embedded on other websites. In this section, we will give a brief overview of Vue and go over some parts of the code.

7. 1 The Vue framework

Vue is a JavaScript framework for building UI and applications. It implements a model-viewmodel-view architecture, which separates the development of the UI from the business logic.

We develop a single page application, with a drag-and-drop file picker, *vue-dropzone* [22]. We decided to limit our files to 50 MB, as we found that files larger than that caused the application to crash. Once the file is chosen, it gets loaded into an audio component, the standard HTML5 *audio* tag. The user is then able to click the “process song” button. Once the song is finished, the dropzone, player and button are hidden and the player is shown. The user is then able to play either the mixed song, only the vocals or only the accompagnement. The user is also able to download a wav file of the isolated vocals or accompagnement.

To link our work with the machine learning pipeline and the website it was very simple: we exported the functions we wanted to use and then simply linked our script file in the Vue view file.

To perform the source separation, firstly we decode the audio file. For that, we use the Web Audio API’s Audio context. We then save the float arrays (one for each channel) into a global variable *aud.src,* as seen in the code below. This method is triggered by the "vdropzone-complete" event emitted by the vue-dropzone component.

```
function readFile(file){
    const fileReader = new FileReader()
    fileReader.onerror = function(){ console.log("Error when reading the file")
}
    fileReader.onload = function(file){
        decodeFile(file.name, fileReader.result)
    }
    fileReader.readAsArrayBuffer(file)
}

function decodeFile(fileName, arrBuffer){
    const audioContext = new AudioContext({"sampleRate":SAMPLE_RATE})
    audioContext.decodeAudioData(arrBuffer,
        function(data){
            const source = audioContext.createBufferSource()
            source.buffer = data
            if(source.buffer.sampleRate != SAMPLE_RATE ||
source.buffer.numberOfChannels != 2){
                alert("Sorry, we can only process songs with a 44100 sample
rate and 2 channels")
                throw new Error('Cannot process song')
            }
            aud.src = [source.buffer.getChannelData(0),
source.buffer.getChannelData(1)]
            return
        }
    , () => {console.log("Error on decoding audio context")})
}
```

Once the song is finished loading, the "Process song" button is enabled. Once the user clicks it, it triggers the processSong method. It receives the location (URL) of the TFJS GraphModel (model.json), and executes the whole pipeline: STFT, model prediction, ISTFT and wav creation. It returns a dictionary with the wav file blob and the stem name (either "vocals" or accompagnement, as seen in the code below:

```
async processSong(){
      this.$refs.processButton.loading = true
      modelProcess(this.publicPath).then((result) =>
        {
          this.shouldRenderSong = false
          this.shouldRenderDropzone = false
          this.shouldRenderPlayer = true
          this.combKey = Math.ceil(Math.random() * 10000)
          let trackstoload = []
          for (let stem of result.stems) {
            trackstoload.push(
                { 'name': stem.name,
                  'customClass': stem.name,
                  'solo': false,
                  'mute': false,
                  'src': stem.data
              })
          }
          this.tracklist = trackstoload
          }
      )
```

We hosted the website on Github Pages, since the SigSep team's website was already hosted that way, in addition to being free and very easy to use.

7.2 Sharing Platform

Initially, we had planned to create a platform that would allow users to share the results of their separations. While we did not manage to begin the development of the sharing platform, we did some research about how we would go about implementing it:

- The user will save the files resulting from the unmixing in their Google Drive or Dropbox, and make them public. We could use the APIs of these services to automatically add the files in the user's account instead of having to manually upload them.
- They will then take the URLs of these files and enter them in the sharing platform, which will build the player that allows the user to mix the song right on the browser
- The platform will create a short link back to this player
- The platform can use Firebase to store which shortened URL belongs to which set of file URLs

Since the user will be the one hosting the unmixed files, it also eliminates any problems we might have with copyright infringement, as we would not be liable for those files.

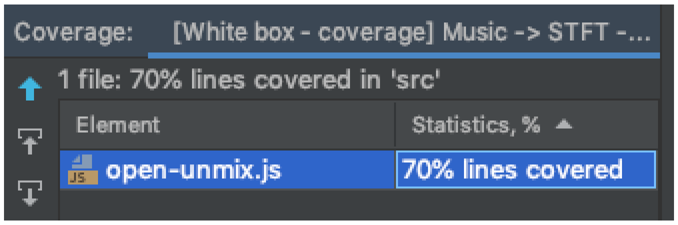
8. Tests

The specific characteristics of digital signal processing algorithms make them a great challenge to be tested [23]. The time-dependent nature adds a new and difficult element to testing since then it is easy to make implementation errors in these algorithms due to the intrinsic complexity.

Initially, the Mocha framework was used to perform the tests using Node as the basis. The tests aimed to ensure that the error returned by the functions and reconstructions of the signals were less than 10e-6.

The tests were divided into:

- Functional
    - **White-Box** [24]:
      We also used a coverage test (white box) to check the different conditions in which the code can be found. With the help of IDE WebStorm, we obtained 70% coverage. The 30% missing are from functions that were only used for downloading the files and other browser-specific code. Since the tests were performed using Mocha-Node it was not possible to cover the remaining 30%.



Coverage: [White box - coverage] Music -> STFT -...

1 file: 70% lines covered in 'src'

Element	Statistics, %
open-unmix.js	70% lines covered

Fig. X - We achieved 70% of code coverage

-
    - **Integration Test**
      The integration test was to check how the code behaved when all parts were being used together. For example
        1. execute the STFT;
        2. run the model;
        3. execute ISTFT to reconstruct the signal.

      We also performed a test to verify the correctness of the results. For example, it is expected that a signal that passes through the STFT-ISTFT functions will be returned perfectly reconstructed. To check if that is indeed the case, we used the mean square error function, available on TFJS and commonly used in machine learning to check the correctness of the predictions. As mentioned above, the error assumed should be less than 10e-6 and an example is found below:

```
describe('[Correctness Test] Signal -> STFT -> ISTFT -> Signal', function()
{
   let power = 15 //2^15
   it('should return an error lower than 10e-6 when using random signal',
function() {
       //Generate random array
       signal = tf.randomNormal([Math.pow(2,power)]).arraySync();
   });

   it('should return an error lower than 10e-6 when using sine signal',
function() {
       //Generate a sine wave
       signal = generateSineWave(power)
   });

   afterEach(function(){
       let originalArray = signal;
       //Perform STFT
       signal = code.preProcessing(signal, specParams)
       //Perform ISTFT
       let ISTFTResult = code.istft(signal, specParams, 1.0);
       let res = tf.losses.meanSquaredError(
           originalArray,
           ISTFTResult
       assert.ok(res.arraySync() < 10e-6)
   })
});
```

- Non-Functional
    - Performance Testing and Volume Testing

      For the performance test, the Mocha tool was used together with the Node version of the TFJS. The volume tests were performed in order to have a clearer idea of how the code behaves during the processing of small or large input signals, which can be a continuous signal or totally discontinued signal.
      The performance test, in addition, was performed in a common browser (Chrome) since TensorFlowJs uses the GPU (WebGL) to perform some calculations. In the end, we used the performance test to check what would be the real behavior in case of memory overload and the browser itself and

how to treat it if necessary so that the project developed does not cause damage to the user's device.

The tests were performed on the device described below:

MacBook Air (13-inch, 2017) - 120Gb SSD
***Operational System -** macOS Mojave 10.14.6*
***Processor -** 1,8 Ghz Intel Core I5*
***Memory** - 8GB 1600 Mhz DDR3*
***Graphic Card -** Intel HD Graphics 6000 1536 MB*
***Browser -** Google Chrome v.80.0.3987.87 64 bits*

9. Conclusion

This project was an incredible opportunity for us. It allowed us to discover an area that we would probably have no contact with otherwise. We learn a lot about digital signal processing, machine learning, JavaScript, Vue, math and audio. It was not always easy but it was definitely very rewarding.

The most unexpected aspect of the project was the math. We always know to expect some math from machine learning related projects, but we underestimated the DSP part. Initially we expected to find libraries that had the operations we needed already implemented, but we were very surprised to discover that it was not the case. It ended up being like solving a puzzle: finding different pieces that fit together well to achieve the desired result. It forced us to get more into the technical details of each operation, since we had to ensure the compatibility between each function.

We also found out that the human ear is an incredible organ, capable of detecting the smallest click or cut in a song. While we had ways to objectively measure the results of our pipeline, it was often enough to listen to the resulting audio file to find out if the process had worked correctly. Even when we were getting what is supposed to be a statistically great result, e.g. having a mean square error of less than 10^{-6}, we could still hear a click in the audio file. We faced many difficulties with adapting certain functions to our pipeline, but getting it *almost correct* was not good enough, so we spent a lot of time making minor adjustments to the whole program to make sure everything was perfect. We could not have done this without the help of the project tutor, Fabian Stöter, who was always available at the Slack channels to help us with any questions that we had and to explain (more than one time) some of the principles of DSP.

Finally, we noted that the technologies involved in the project are changing and evolving rapidly. For example, most of the DSP operations that we used (from the TFJS library) were implemented within the last 6 months. It is hard to guess the kinds of improvements that will be made in the near future, since new functionalities are being added to the TensorFlowJS everyday. We hope that soon the operations we needed during the project, such as the ISTF or the LSTM Cell for the model, will be added soon.

In conclusion, we are satisfied with the project, since we achieved the main objective: to create a working machine learning application for music separation on the web. We hope that our project helps the audio processing on the web community. We hope to be able to continue working with the SigSep team to make the sharing platform a reality. Overall, we consider the project a success.

10. Bibliography

[1] Stöter, Fabian; Uhlich, Stefan; Liutkus, Antoine; Mitsufuji, Yuki.*"Open-Unmix - A Reference Implementation for Music Source Separation"*, 08/09/2019. https://joss.theoj.org/papers/10.21105/joss.01667

[2] Google. *"TensorFlowJS",* 2018. Github repository. https://github.com/tensorflow/tfjs

[3] SigSep. *"Open Unimix PyTorch",* 2019. Github repository. https://github.com/sigsep/open-unmix-pytorch

[4] Deezer. *"Spleeter",* 2019. Github repository. https://github.com/deezer/spleeter

[5] Jansson, Andreas; Humphrey, Eric;Montecchio, Nicola; Bittner, Rachel; Kumar, Aparna;Weyde, Tillman. "*SINGING VOICE SEPARATION WITH DEEP U-NET CONVOLUTIONAL NETWORKS*". https://ejhumphrey.com/assets/pdf/jansson2017singing.pdf

[6] Audacity. *"Digital Audio Fundamentals".* Website. https://manual.audacityteam.org/man/digital_audio.html

[7] Furneaux, Mark. "*Digital Audio: The Basics*". April 10, 2017. Youtube Video. https://www.youtube.com/watch?v=9z5HOZ4C1KY

[8] Rajlich, Nathan. "*Node-Lame*". NPM Package. https://www.npmjs.com/package/lame

[9] Gal, Andreas. "High performance WAV file decoder and encoder for node". Github repository. https://github.com/andreasgal/node-wav

[10] Swanson, Jez. *"An Interactive Introduction to Fourier Transforms".* Website. http://www.jezzamon.com/fourier/

[11] Google. "*Magenta.js*". Github repository. https://github.com/tensorflow/magenta-js

[12] Music Technology Group. *"EssentiaJS".* Github repository. https://github.com/MTG/essentia.js

[13] Brook, Corban. "*Digital Signal Processing for Javascript*". Github repository. https://github.com/corbanbrook/dsp.js

[14] Lysenko, Mikola. *"Short time Fourier transform".* Github repository. https://github.com/mikolalysenko/stft

[15] Müller, Meinard. *“Fundamentals of Music Processing”.* PowerPoint presentation. https://www.audiolabs-erlangen.de/content/05-fau/professor/00-mueller/04-bookFMP/02-slides/Mueller_FMP_Chapter2.pdf

[16] Aquegg. *“Spectrogram of the spoken words "nineteenth century"* https://en.wikipedia.org/wiki/Spectrogram#/media/File:Spectrogram-19thC.png

[17] Riley, Lewis. “Review of Complex Numbers”. Website. http://webpages.ursinus.edu/lriley/ref/complex/node1.html

[18] Selesnick, Ivan. *“Short-Time Fourier Transform and Its Inverse”,* April 14, 2009. http://eeweb.poly.edu/iselesni/EL713/STFT/stft_inverse.pdf

[19] Nockert, Jens. “A *Fast Fourier Transform library for JS”.* NPM package https://www.npmjs.com/package/fft

[20] Smith III, Julius. *“Mathematical definition of the STFT”.* https://ccrma.stanford.edu/~jos/sasp/Mathematical_Definition_STFT.html#19930

[21] Shirnewar, G.S. *“Overlap add”,* September 1, 2014. Youtube video. https://www.youtube.com/watch?v=FPzZj30hPY4

[22] Winsemius, Rowan. “*Vue-Dropzone*”. Github repository. https://github.com/rowanwins/vue-dropzone

[23] Oshana, Rob. “*Testing and Debugging DSP Systems, Part 6*”, March 26, 2007. EE Times. https://www.eetimes.com/testing-and-debugging-dsp-systems-part-6/

[24] Software Testing Help. “*Types Of Software Testing: Different Testing Types With Details*”. November 10, 2019. https://www.softwaretestinghelp.com/types-of-software-testing/

APRENDIZADO DE MÁQUINAS PARA ÁUDIO NA *WEB*

Maria Clara Machry Jacintho[1]
Professor Dr. Marcelo Oliveira Johann[2]

RESUMO

O objetivo deste projeto consiste em desenvolver uma aplicação *Web* para permitir a separação das fontes de áudio de uma música diretamente no navegador. Na aplicação foi utilizado um modelo de aprendizado de máquinas pré-treinado combinado com as técnicas de Processamento Digital de Sinais (*Digital Signal Processor* (DSP) para realizar essa separação. O desenvolvimento consistiu em implementar manualmente, empregando-se a linguagem de programação do *JavaScript*, as operações de DSP necessárias que até então não existiam nas bibliotecas *online*; converter o modelo *Machine Learning* (ML); e criar um *website* para disponibilizar a aplicação na *Web*.

Palavras-chave: Aprendizado de Máquinas. Áudio na *Web*. Separação das Fontes de Áudio. Processamento Digital de Sinais

ABSTRACT

The objective of this project is to develop a Web application to allow the separation of the audio sources of a song directly in the browser. The application used a pre-trained machine learning model combined with Digital Signal Processing (DSP) techniques to perform this separation. The development consisted of implementing, using the JavaScript programming language, the necessary DSP operations that until then did not exist in online libraries; convert the Machine Learning (ML) model; and create a website to make the application available on the Web.

Keywords: Machine Learning. Audio on the Web. Separation of Audio Sources. Digital Signal Processing.

1 INTRODUÇÃO

O Projeto aqui apresentado foi desenvolvido entre os meses de Dezembro de 2019 a Fevereiro de 2020 como um Trabalho de Conclusão do Curso direcionado ao curso de Informática e Gestão (*Informatique et Gestion*) pela Universidade de Polytech Montpellier da França pelos acadêmicos, Delton de Andrade Vaz e Maria Clara Machry Jacintho. O Projeto foi sugerido e coordenado por Fabian-Robert Stöter, à época pesquisador do Instituto Nacional de Pesquisa em Ciências da Computação e Automação (INRIA) da França, em conjunto com o grupo que trabalhava com a separação de sinais.

[1] Aluna do Curso de Ciências da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS). *E-mail*: mcmachry@gmail.com.
[2] Professor orientador. *E-mail*: johann@inf.ufrgs.br.

O principal objetivo do Projeto consiste no desenvolvimento de uma ferramenta para separar os instrumentos de uma música diretamente em um navegador de *internet*, ou seja, permitir ao usuário extrair apenas, por exemplo, o som do violão ou os vocais de uma peça, sem a necessidade de qualquer tipo de servidor para o respectivo processamento. Esse tipo de extração, popularmente conhecida como ‘desmixagem’, é realizada por um modelo de aprendizado de máquinas que executa as inferências diretamente no navegador.

Neste sentido, os pesquisadores do INRIA desenvolveram um modelo de separação de fontes musicais com a linguagem *Python*[3], utilizando a *framework*[4] de Inteligência Artificial (IA) *PyTorch* (STÖTER, *et al*., 2019). Continuando essa linha de pesquisa, a equipe que foi denominada de SigSep[5] procurou trazer essa funcionalidade para a *Web*[6], cuja inspiração foi o trabalho desenvolvido pela empresa japonesa, KoeKestra (KOEKESTRA, 2019), que criou um *site* permitindo ao usuário ’desmixar’ uma canção diretamente no navegador, utilizando um modelo de aprendizagem da máquina denominado *Spleeter* (HENNEQUIN, 2020), que por sua vez, foi criado pela empresa francesa Deezer.

Para realizar a ‘desmixagem’ de uma música, a peça deve passar por uma *pipeline*[7] que pode ser dividida em três etapas: pré-processamento; inferência; e; pós-processamento. As etapas de pré-processamento e de pós-processamento envolvem o Processamento Digital de Sinais (*Digital Signal Processing* (DSP)), envolvendo as operações das Transformadas de *Fourier*. Já, a etapa de inferência consiste em utilizar um modelo de IA pré-treinado e desenvolvido em *Python*, que por sua vez, deve ser convertido para a linguagem de programação *JavaScript*[8].

Até o momento existem poucas bibliotecas disponíveis de código para DSP na linguagem *JavaScript*. Em função disso, durante o desenvolvimento do Projeto, não foi possível encontrar uma biblioteca que implementasse todas as funções necessárias, como a Transformada de *Fourier* de Tempo Curto (*Short Time* Fourier *Transform* (STFT)) e a Transformada de *Fourier* de Tempo Curto Inversa (*Inverse Short Time* Fourier *Transform*

[3] A linguagem *Python*. foi criada em 1991 e apresenta características que “[...] possibilitam escrever o mesmo requisito em menos linhas de código que o necessário em outras linguagens de programação e além de adotado na construção de soluções *Web*, também é muito utilizado em aplicações que lidam com processamento de texto, *machine learning* e recomendação de conteúdo” (DEVMEDIA, 2021).

[4] *Framework*: “[...] é um pacote de códigos prontos que podem ser utilizados no desenvolvimento de sites. A proposta de uso dessa ferramenta é aplicar funcionalidades, comandos e estruturas já prontas para garantir qualidade no projeto e produtividade” (SOUZA, 2019, p. 1).

[5] Equipe SigSep: Equipe do INRIA que está se especializando no tópico de separação de sinais, incluindo o orientador do Projeto original, o pesquisador Fabian-Robert Stöter.

[6] *World Wide Web* também conhecida como *WWW* ou simplesmente *Web*.

[7] *Pipeline*: processo com várias etapas.

[8] *JavaScript*: linguagem de programação muito utilizada no desenvolvimento *Web*.

(ISTFT)). Foi necessário, portanto, implantar essas operações em *JavaScript*, utilizando-se as implementações em outras linguagens como referência de modelo.

Assim sendo, neste Trabalho de Conclusão de Curso é desenvolvido um resumo estendido do Projeto citado anteriormente, dividindo-o em nove capítulos. No primeiro é descrita uma breve introdução sobre as pretensões do Projeto. No segundo capítulo desenvolve-se a fundamentação teórica abordando os conceitos de áudio; de processamento digital de sinais com a Transformada de *Fourier*; e o aprendizado de máquinas.

No terceiro capítulo é realizada uma exploração de trabalhos relacionados ao tema com uma revisão dos detalhes de implementação para o Projeto em questão. No quarto capítulo consta a metodologia utilizada; e no quinto são demonstrados os testes realizados. No sexto capítulo demonstra-se a plataforma de compartilhamento; no sétimo o trabalho futuro pretendido; e no oitavo está descrita a conclusão obtida com a realização do estudo. Na parte final estão listadas todas as referências utilizadas.

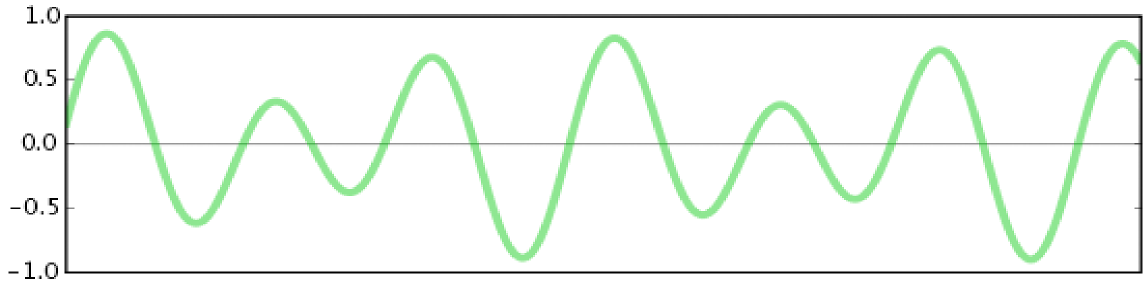
2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é desenvolvida a fundamentação teórica com a finalidade de oferecer um embasamento para a realização do estudo, contemplando a caracterização do áudio e da Transformada de Fourier.

2.1 ÁUDIO

Um som pode ser caracterizado como uma onda longitudinal que produz as vibrações por intermédio de um meio de propagação, em geral, o ar, e que provoca uma sensação auditiva em uma pessoa (KINSLER; FREY, 1962). O áudio é representado pelo gráfico ilustrado na Figura 1, onde o 'eixo x' é o tempo; enquanto que o 'eixo y' é a amplitude do sinal. No caso de ondas acústicas, a amplitude se refere a quão alto é o som.

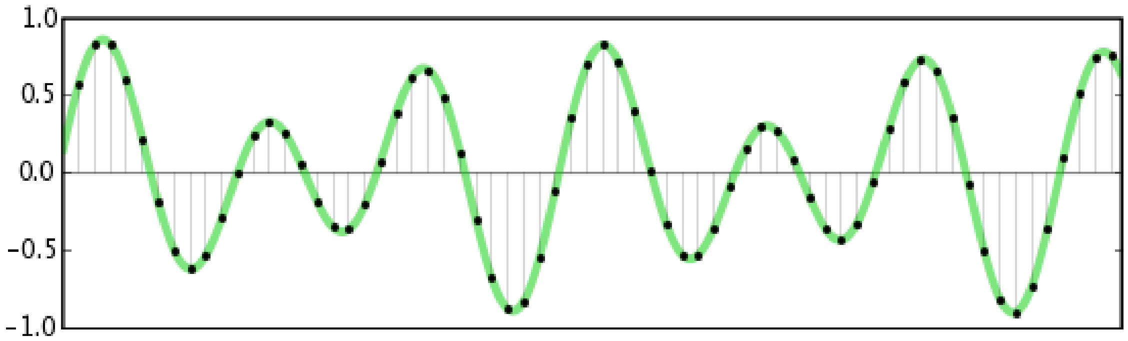
Figura 1 – Uma onda de áudio



Fonte: *Audacity* (2020).

Armazenar essa onda de maneira digital é um processo complicado. Se o sinal for simples, como por exemplo, uma onda senoidal, é possível guardar apenas a sua fórmula matemática, sen(x). Se o sinal for mais complexo, como uma canção, isso não é viável. Nesses casos, o que se faz é discretizar o sinal, isto é, obter o valor do 'eixo y' a uma taxa de amostragem fixa, como ilustra a Figura 2, permitindo representar um sinal contínuo de forma discreta.

Figura 2 – Uma onda de áudio com as amostras marcadas como pontos negros



Fonte: *Audacity* (2020).

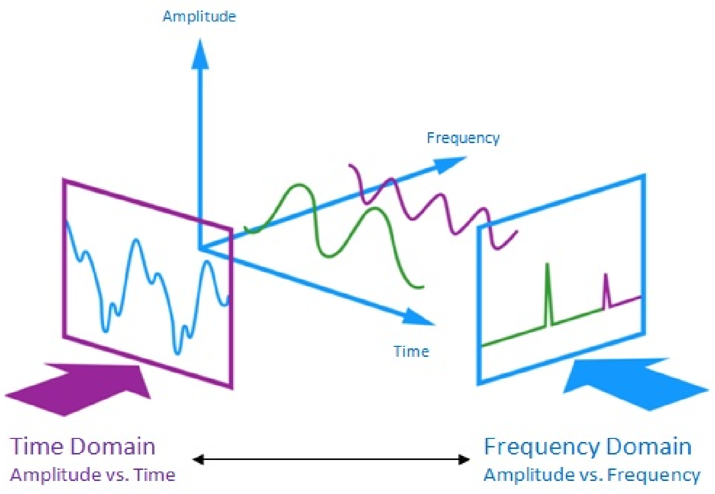
Acrescenta-se que, como a taxa de amostragem é fixa, também não é necessário gravar nenhuma informação sobre o 'eixo x' (tempo), já que a primeira amostragem é no tempo 0 (zero), é possível deduzir o tempo da próxima amostra utilizando a taxa de amostragem. O resultado é um conjunto de valores que podem ser reconstruídos para formar a onda original. Normalmente, também há algum tipo de compressão aplicada a estes dados, para economizar memória.

2.2 TRANSFORMADA DE *FOURIER*

Uma Transformada de *Fourier* compreende uma 'transformação' ou um mapeamento que toma um sinal (função do tempo) e o transforma em uma 'função complexa de frequência'. O sinal representado ainda é o mesmo, sendo a transformada apenas uma maneira diferente de olhar para ele, como pode ser observado na Figura 3.

Cabe destacar que os termos da Figura 3 estão escritos na língua inglesa, o que remete a necessidade de traduzi-las para o português para uma melhor compreensão como segue: *amplitude* (amplitude); *frequency* (frequência); t*ime* (tempo); *time domain* (domínio do tempo); e *frequency domain* (domínio de frequência).

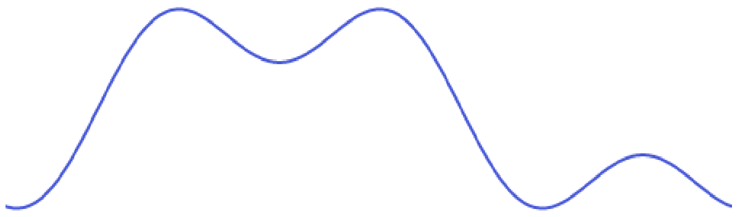
Figura 3 – Domínio do tempo e da frequência



Fonte: *Mechanical Engineering Department* (MED)[9] (2018).

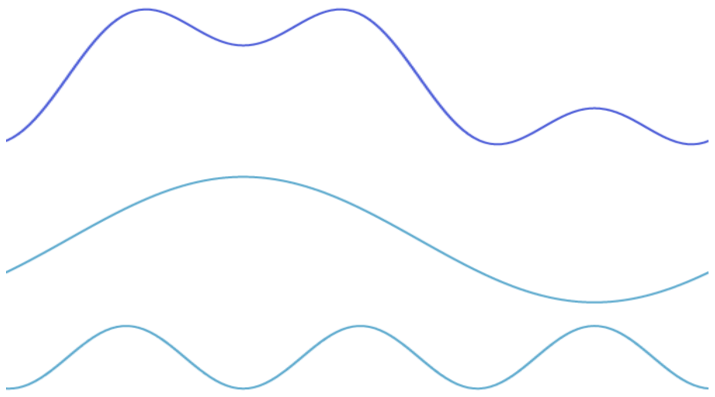
Ao considerar o sinal apresentado na Figura 4, como exemplo, é possível decompô-lo em ondas senoidais, como representado na Figura 5. Ao somar essas ondas, tem-se como resultado final o sinal original de volta.

Figura 4 – Um sinal



Fonte: Swanson (2019).

Figura 5 – Sinal decomposto



Fonte: Swanson (2019).

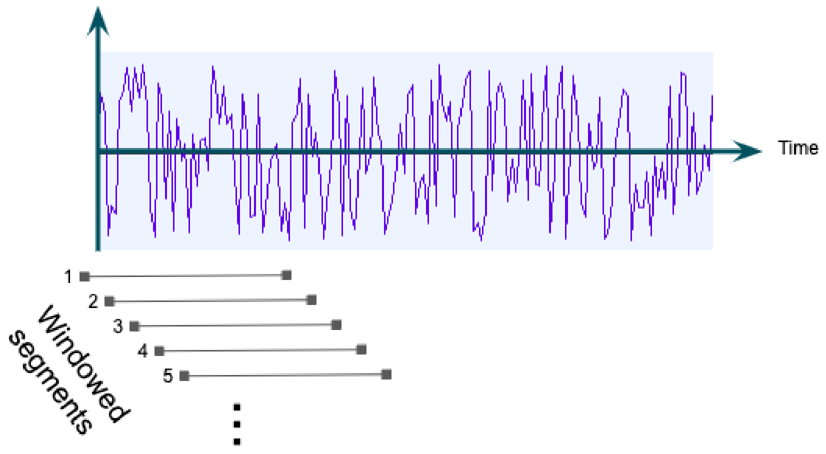
[9] *Mechanical Engineering Department* (MED): Departamento de Engenharia Mecânica.

Assim sendo, se houver, em vez de uma função contínua, um conjunto finito de pontos igualmente espaçados, como no caso de áudio digital, pode-se utilizar a Transformada Discreta de *Fourier* (*Discrete Fourier Transform* (DFT)) em seu lugar. Para calcular uma DFT, o algoritmo normalmente utilizado é a Transformada Rápida de Fourier (*Fast* Fourier *Transform* (FFT)).

2.2.1 STFT e ISTFT

Para compreender como as frequências que compõem um sinal mudam com o tempo, não basta aplicar uma Transformada de *Fourier*. É preciso primeiro partir o sinal em segmentos, aplicando uma função de janela, e então aplicar a Transformada de *Fourier* sobre eles, como pode ser visualizado na Figura 6. Essa operação é conhecida como a Transformada de *Fourier* de Tempo Curto (STFT) (COHEN, 1995); e o seu resultado é um espectrograma, que permite acompanhar a evolução das frequências durante o tempo.

Figura 6 – STFT: essencialmente a DFT aplicada em diferentes segmentos janelados[10]



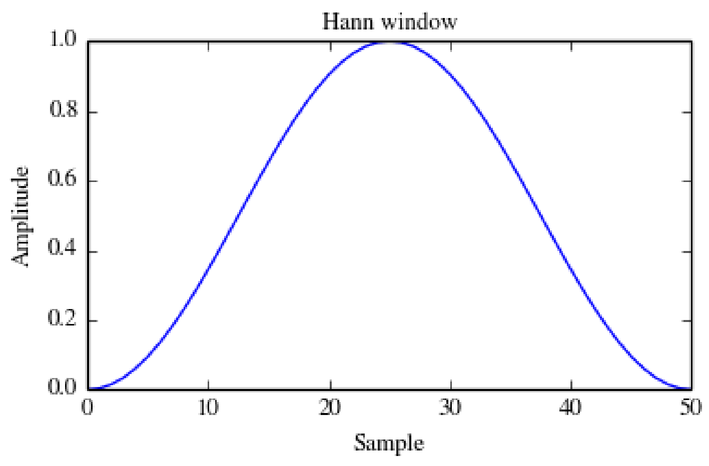
Fonte: Elaboração própria (2020).

Desse modo, a função de janela utilizada deve ser nula em todos os pontos, menos dentro de um determinado intervalo. Uma das funções janelas mais populares utilizadas tem sido a janela de Hann[11], que pode ser observada na Figura 7.

[10] *Time*: tempo. *Windowed segments*: segmentos em janela.

[11] Janela de Hann: compreende um cone formado que utiliza um cosseno elevado ou seno-quadrado com extremidades que tocam zero. A janela de Hann nomeada em homenagem a Julius van Hann, um meteorologista austríaco. É também conhecido como *Cosine Bell.* A maioria das referências referentes à janela de Hann é oriunda da literatura de processamento de sinal, onde ela é empregada como uma das muitas funções de janela para os valores de suavização. É também conhecido como 'apodização' (que significa 'retirar o pé', ou seja, suavizar descontinuidades no início e no final do sinal amostrado) ou função de redução gradual (SCIPY, 2014).

Figura 7 – Janela Hann



Fonte: Scipy (2014).

A Transformada *Fourier* de Tempo Curto Inversa (*Inverse Short Time Fourier Transform* (ISTFT)) tem a finalidade de transformar um espectrograma de volta em um sinal (COHEN, 1995). Para tal, é aplicada a Transformada de *Fourier* de Tempo Curto Inversa e é utilizado o método de *Overlap-And-Add* (OLA)[12], que reconstrói o sinal.

3 CONTEXTO E ESTUDO DA ARTE

Neste capítulo é apresentado o contexto e o estudo da arte sobre a temática de separação de fontes de áudio; e em relação à inteligência artificial em *JavaScript* (JS) e o processamento digital de sinais em JS.

3.1 SEPARAÇÃO DE FONTES DE ÁUDIO

Nesta seção são contemplados os trabalhos prévios; em seguida aborda-se a caraterização do modelo *Open Unmix*; e para complementar é apresentado o modelo *Open Source* de separação de fontes *Spleeter*.

3.1.1 Trabalhos Prévios

A separação de fontes de áudio pode ser implementada utilizando-se apenas as técnicas de processamento digital de sinais. De fato, uma das primeiras implementações *open source* (código aberto) para a separação de fontes, foi a *openBlissart*[13], disponibilizada no ano

[12] *Overlap-And-Add* (OLA): Operação de convolução para recriar um sinal janelado (SMITH, 1999).
[13] *OpenBlissart*: biblioteca de 'desmixagem', termo em inglês não possui tradução para o português.

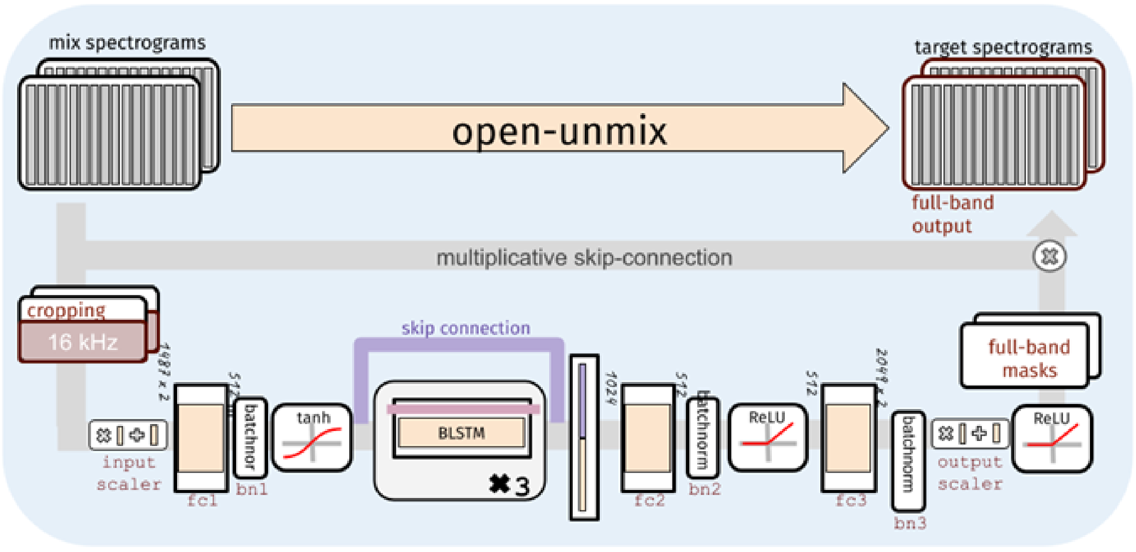
de 2011, e emprega a Fatoração de Matriz Não-Negativa (*Non-negative Matrix Factorization* (NFM))[14] (WENINGER., 2011). Além disso, existem também as aplicações que utilizam tanto o DSP quanto os modelos de aprendizado de máquinas, como por exemplo, a biblioteca *untwist*[15] (ROMA, 2016).

Cabe salientar que um dos problemas do campo de separação de áudio é a falta de uma implementação de referência (STÖTER, *et al*., 2019). É, portanto, muito complexo comparar bibliotecas diferentes, já que cada uma delas mede a sua performance (desempenho) contra as referências diferentes. Outro problema é o fato que as pessoas com pouca experiência no campo de áudio, mas interessadas em desenvolver os algoritmos de aprendizado de máquinas para a área, acabam tendo muitas dificuldades para aprender a parte teórica de processamento digital de sinais.

3.1.2 *Open Unmix*

O *Open Unmix* é um modelo de separação de música criado pelos pesquisadores: Fabian-Robert Stöter, Stefan Ulrich, Antoine Liutkus e Yuki Mitsufuji (STÖTER, *et al*., 2019), cuja arquitetura está representada na Figura 8.

Figura 8 – Arquitetura do modelo *Open Unmix*



Fonte: Stöter *et al.* (2019).

[14] Fatoração de Matriz Não-Negativa (*Non-negative Matrix Factorization* (NFM)): "Técnica de redução de dimensionalidade e análise de dados que produz uma representação em partes, não negativa e esparsa para os dados de entrada não negativos" (DHILLON; SRA, 2005, p. 1).
[15] Biblioteca *untwist:* biblioteca de 'desmixagem', termo em inglês não possui tradução para o português.

A motivação por trás do trabalho foi desenvolver um modelo *Open Source*, extensível e 'hackeável'[16], que apresentasse performance comparável aos modelos do estado da arte, porém que se configurasse como um modelo simples de se entender. Isso foi realizado graças ao fato que 'as etapas de pré-processamento e pós-processamento; carregamento de dados; treinamento e; os modelos do código estavam isolados e foram fáceis de substituir', permitindo às pessoas com maior conhecimento de aprendizado de máquinas se direcionarem apenas ao desenvolvimento de novos modelos e de arquiteturas em vez de terem que lidar com as etapas de processamento de sinais digitais.

Neste sentido, o Projeto é constituído de vários modelos diferentes, cada um treinado para separar uma fonte específica, como exemplos, as vocais, a bateria, dentre outros. Cada modelo possui uma rede *Long Short Term Memory* (LSTM)[17] bidirecional com três camadas de profundidade, permitindo que o modelo seja treinado com sinais de qualquer tamanho.

Assim, recebe como entrada o espectrograma de magnitude e apresenta como saída uma máscara, que, ao ser multiplicada com o espectrograma original, possibilita o filtro das frequências que não pertencem à fonte específica para a qual o modelo foi treinado. Os modelos foram desenvolvidos com a *Framework PyTorch*[18] e treinados utilizando-se o conjunto de dados (*Dataset*) MUSDB18, que contém mais de 150 canções e fontes separadas oferecendo domínio público.

3.1.3 ***Spleeter***

Outro modelo *Open Source* de separação de fontes compreende o *Spleeter* desenvolvido pela empresa francesa Deezer (HENNEQUIN, 2020). A arquitetura do referido modelo utiliza *U-nets* que é uma rede neural convolucional (*Convolutional Neural Network* (CNN))[19], codificadora e decodificadora (*encoder and decoder*), com *skip connections* (pular conexões), e foi implementado utilizando a *Framework TensorFlow*.

Como empresa de *streaming*[20] de música, a Deezer disponibiliza um catálogo amplo de músicas, o qual foi empregado para treinar o modelo aqui proposto, resultando em inferências mais precisas e ágeis se comparado ao modelo *Open Unmix*. Esse modelo,

[16] *Hackeável*: termo com a conotação de modificável e/ou extensível.
[17] *Long Short Term Memory*: Memória Longa de Curto Prazo.
[18] *Framework PyTorch*: Biblioteca de aprendizado de máquinas em Python, desenvolvida pelo *Facebook*. (PYTORCH, 2021).
[19] Rede Neural Convolucional (*Convolutional Neural Network* (CNN)): arquitetura de redes neurais, muito utilizada no campo de aprendizado de máquinas para o reconhecimento de padrões (BENGIO; LECUN, 2007).
[20] *Streaming*: transmissão.

originalmente criado em linguagem *Python*, foi posteriormente adaptado para a linguagem de programação *JavaScript* pela empresa Japonesa, KoeKestra (KOEKESTRA, 2019).

3.2 INTELIGÊNCIA ARTIFICIAL EM *JAVASCRIPT*

No ano de 2015, a empresa *Google* desenvolveu a *Framework TensorFlow*, para *Python*, sendo uma das mais utilizadas atualmente para o desenvolvimento de modelos de Inteligência Artificial (*Machine Learning* – LM[21]). Como forma de se aproximar à criação e ao treinamento de modelos da *Web*, foi criado o *TensorFlowJS* (TFJS) (GOOGLE, 2018a), que compreende uma versão *JavaScript* da *Framework* original. Além dos métodos para criar os modelos, o TFJS contém diversas operações que são úteis no pré-processamento e no pós-processamento de dados.

3.3 PROCESSAMENTO DIGITAL DE SINAIS EM JS

Como o estudo necessitava da utilização de certas operações de Processamento Digital de Sinais, foi realizada uma pesquisa das bibliotecas *online* já existentes que pudessem ser utilizadas no Projeto. Essas operações, quando utilizadas em alguma aplicação *Web*, são geralmente implementadas em outras linguagens e rodam em algum servidor, pois são computacionalmente muito intensas. Há, portanto, poucas bibliotecas *online* na linguagem *JavaScript* (JS) que possam rodar em um *browser* (navegador) que as implementem.

Dentre as bibliotecas *online* que foram encontradas, tem-se a FFT.js que possui um método de Transformada Rápida de *Fourier* (*Fast Fourier Transform* (FFT)) e a sua inversa, que podem ser utilizadas para o desenvolvimento das operações necessárias para o Projeto, a STFT e a ISTFT, já conceituadas no item 2.2.1.

A segunda biblioteca *online* encontrada foi uma implementação realizada por Mikola Lysenko, que utiliza a FFT.js e implementa tanto a STFT quanto a ISTFT, porém elas estavam ligadas e não seria possível utilizar o resultado da STFT para a inferência do modelo do Projeto (LYSENKO, 2013). Além dessas, foi encontrada apenas uma biblioteca *online* que implementasse essas operações, a Magenta, porém nessa implementação a ISTFT foi implementada em uma classe privada e era incompatível com a STFT pretendida para o Projeto. Verificou-se também que, outras bibliotecas *online* não implementam tais operações. Os resultados estão resumidos no Quadro 1, considerando-se como período de pesquisa o início do ano de 2020.

[21] *Machine Learning*: Aprendizado de Máquina.

Quadro 1 – Bibliotecas STFT/ISTFT para *JavaScript* no início de 2020

Implementação	FFT	STFT	IFFT	ISTFT	Observações
FFT.js	Sim	Não	Sim	Não	Nockert (2012).
mikolalysenko/stft	Sim*	Sim	Sim*	Sim†	Sem documentação; Utiliza FFT.JS; † possui STFT e ISTFT integradas uma a outra, não sendo possível o processamento entre as duas partes; Lysenko (2013).
Dsp.js	Sim	Não	Sim	Não	Brook (2010).
Essentia.js	Não	Não	Não	Não	O *status* desta biblioteca, de acordo com seu GitHub ReadMe: “Em desenvolvimento, não otimizado e uso por sua própria conta e risco”. As funções que precisaria para o Projeto também foram comentadas. MTG[22] (2019).
TFJS	Sim	Sim	Sim	Não	Google (2018a).
Magenta	Sim*	Sim†	Sim*	Sim†	Utiliza FFT.js; †STFT é incompatível com ISTFT; Google (2018b).

Fonte: Elaboração própria (2020).

4 METODOLOGIA

O programa do Projeto apresenta a seguinte estrutura: uma *pipeline* de pré-processamento; a inferência; e o pós-processamento. No pré-processamento, um arquivo de áudio é decodificado em um *buffer*[23], no qual a STFT é aplicada. O resultado é então transformado de coordenadas cartesianas para coordenadas polares, abrangendo a fase e a magnitude do sinal.

Por sua vez, a magnitude é em seguida utilizada como a entrada do modelo; Enquanto que, a saída do modelo é combinada com a fase, e então passa por uma Transformada Inversa de Tempo Curto (ISTFT), reconstruindo o sinal. Para finalizar, o sinal é remontado em arquivo de áudio.

[22] *MUSIC TECHNOLOGY GROUP* (MTG).

[23] *Buffer*: região da memória para o armazenamento temporário de dados (termo em inglês não possui tradução para o português).

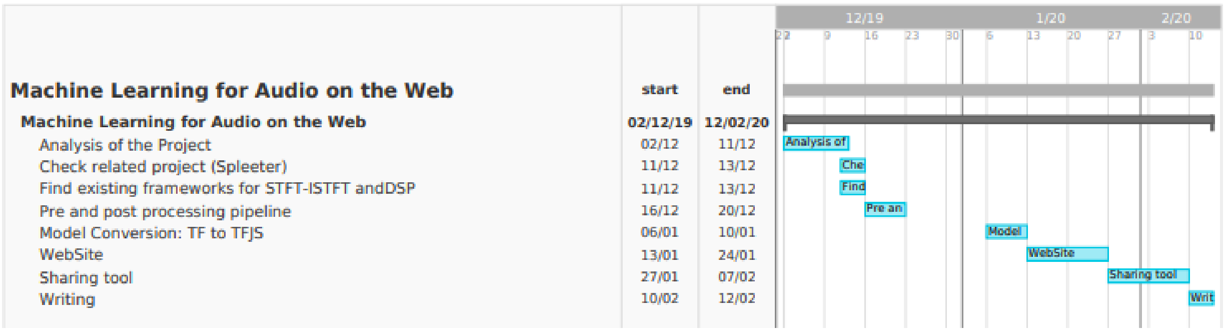
4.1 ORGANIZAÇÃO DO PROJETO

O Projeto foi desenvolvido pelos alunos, Maria Clara Machry Jacintho e Delton de Andrade Vaz[24], como Trabalho de Conclusão do Curso para o curso de Informática e Gestão, pela Universidade Polytech Montpellier, localizada na França. O projeto teve duração de três meses, período durante o qual os referidos alunos se dedicaram exclusivamente ao Projeto.

Os pesquisadores tiveram acesso a uma sala de laboratório na Universidade Polytech Montpellier, na França, onde puderam implementar o estudo. Inicialmente, foram realizadas diversas reuniões entre os alunos e o precursor do Projeto, no caso, o Dr. Fabian-Robert Stöter e, também, com o orientador do Projeto, o professor Chouki Tibermacine da referida Universidade.

As reuniões serviram como base para a criação do diagrama de Gantt, disponível na Figura 9, como planejamento para o Projeto. A ideia para o gerenciamento do Projeto era seguir a metodologia cascata, concluindo inteiramente uma etapa para depois iniciar a próxima. A estimativa de duração de cada etapa foi realizada por meio de discussões com os orientadores e mediante experiência própria dos participantes.

Figura 9 – O diagrama de Gantt original



Machine Learning for Audio on the Web	start	end
Machine Learning for Audio on the Web	**02/12/19**	**12/02/20**
Analysis of the Project	02/12	11/12
Check related project (Spleeter)	11/12	13/12
Find existing frameworks for STFT-ISTFT andDSP	11/12	13/12
Pre and post processing pipeline	16/12	20/12
Model Conversion: TF to TFJS	06/01	10/01
WebSite	13/01	24/01
Sharing tool	27/01	07/02
Writing	10/02	12/02

Fonte: Elaboração própria (2020).

Salienta-se que, além das reuniões iniciais, foram realizadas também as reuniões regulares durante o desenvolvimento do Projeto, com periodicidade de uma vez a cada duas semanas. Os acadêmicos também mantiveram o contato com os orientadores por intermédio da ferramenta *Slack*[25], onde era possível enviar mensagens para ambos orientadores em caso de dúvidas. Graças ao fato de poderem se dedicar exclusivamente ao Projeto, os alunos trabalhavam em torno de seis horas diárias na execução do Projeto.

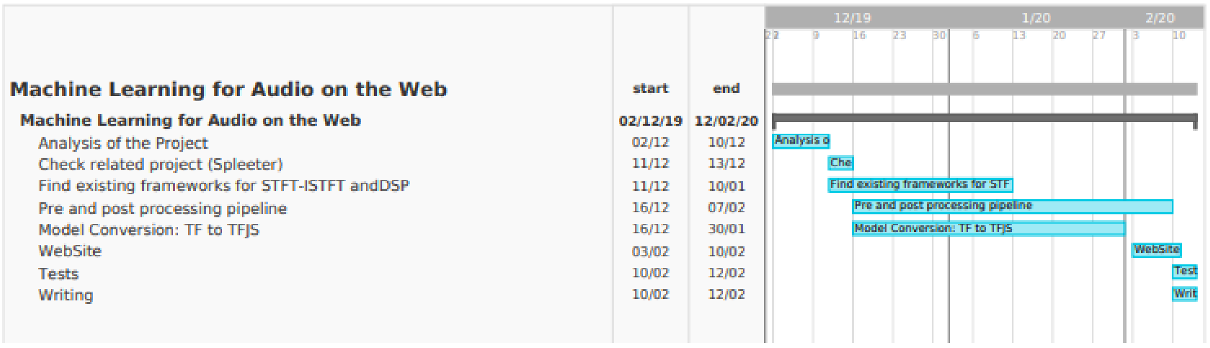
[24] O acadêmico, Delton de Andrade Vaz, pertence ao Curso de Engenharia da Computação, pela Universidade Federal do Rio Grande do Sul (UFRGS) e, concordou em disponibilizar o Projeto para este estudo, sendo que está desenvolvendo outro tema em seu Trabalho de Conclusão de Curso deste semestre.

[25] Ferramenta *Slack*: plataforma para troca de mensagens (SLACK, 2021).

A maior parte do desenvolvimento ocorreu na forma de programação em dupla (*pair-programming*). Como o trabalho envolve muitos conhecimentos matemáticos e, também, específicos do campo de processamento digital de sinais, foi necessário muita pesquisa e aprendizado por parte dos alunos. Dessa forma, a programação em dupla garantia que nenhum dos acadêmicos ficasse sem compreender alguma parte da implementação, o que aumentava a qualidade do Projeto.

Durante a execução do Projeto, os participantes começaram a utilizar os métodos ágeis, integrando rapidamente as diversas funcionalidades, independentes do Projeto. Inicialmente, o tempo necessário para a realização de algumas partes do Projeto foi subestimado, notavelmente o tempo de desenvolvimento das *pipelines* de pré-processamento e de pós-processamento. O planejamento inicial contava com a existência de alguma biblioteca *online* com implementações de STFT e de ISTFT, porém foi necessário implementá-las desde o seu início demandando mais tempo que o previsto. O diagrama de Gantt para o desenvolvimento do Projeto teve que ser alterado e se configurou como mostra a Figura 10.

Figura 10 – Diagrama de Gantt atualizado



Machine Learning for Audio on the Web	start	end
Machine Learning for Audio on the Web	**02/12/19**	**12/02/20**
Analysis of the Project	02/12	10/12
Check related project (Spleeter)	11/12	13/12
Find existing frameworks for STFT-ISTFT andDSP	11/12	10/01
Pre and post processing pipeline	16/12	07/02
Model Conversion: TF to TFJS	16/12	30/01
WebSite	03/02	10/02
Tests	10/02	12/02
Writing	10/02	12/02

12/19
1/20
2/20
2
9
16
23
30
6
13
20
27
3
10
Analysis o
Che
Find existing frameworks for STF
Pre and post processing pipeline
Model Conversion: TF to TFJS
WebSite
Test
Writ

Fonte: Elaboração própria (2020).

4.2 IMPLEMENTAÇÃO DO PROJETO

A implementação do Projeto iniciou com a etapa do pré-processamento; em seguida foi escolhido o modelo a ser utilizado; apresentada a Transformada de *Fourier* escolhida; e na sequência foi desenvolvida o *Website*; cujas etapas estão detalhadas nos próximos itens.

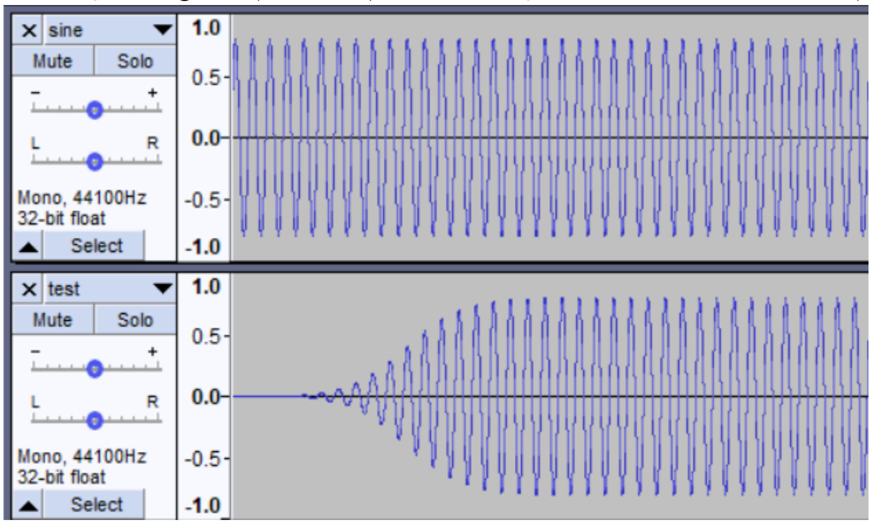
4.2.1 Pré-processamento

O início da *pipeline* envolve a decodificação do áudio para um *buffer*. A seguir, é aplicada uma operação para realizar o *padding* (preenchimento) desse *buffer*, para garantir que o sinal apresente um tamanho divisível pela janela da STFT. Finalmente, a STFT é aplicada ao sinal, que é em seguida convertido em coordenadas cartesianas para as coordenadas polares, e assim fica pronto para o modelo pretendido.

A STFT escolhida foi a implementação do *TensorFlowJS*, explicado pelo fato de ser compatível com o modelo de aprendizado de máquinas e tornar o programa mais coerente com a implementação *Python*, pois utiliza também a STFT do *TensorFlow*.

Essa implementação cria um 'início suave' na canção, como mostrado na Figura 11, porque a função de janela descarta o início do sinal e também não apresenta uma opção para o preenchimento do último segmento do sinal. Para corrigir estes dois problemas criou-se uma função que adiciona zeros antes e depois do sinal, ajustando a janela para que nenhuma informação sobre o sinal fosse perdida.

Figura 11 – Canção original (em cima) *versus* canção com um início suave (em baixo)

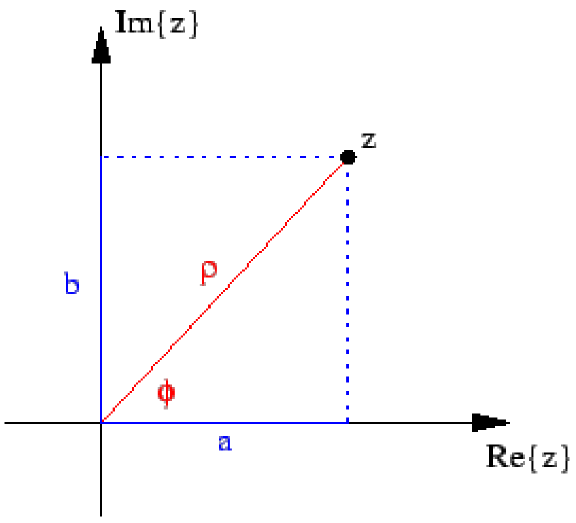


Fonte: Elaboração própria (2020).

O resultado da operação STFT é um Tensor 2D de tamanho N x *FrameLength*[26], onde, para esse Projeto, N é igual a 100 e o *FrameLength* é 2.049. Antes de passar para o modelo propriamente dito, foi preciso primeiro transformar o resultado da STFT de coordenadas cartesianas em coordenadas polares, ou seja, a magnitude e a fase, como ilustrado na Figura 12.

Figura 12 – Um número complexo representado em coordenadas cartesianas e polares

[26] *Framelength*: tamanho do quadro, opção utilizada na definição da operação STFT.



Fonte: Riley (2004)

Embora a obtenção da magnitude seja simples, para se conseguir a fase é um pouco mais difícil, pois envolve as operações que, em navegadores com *WebGL*[27] habilitado, não suportam os números complexos. Por isso, os tensores de números complexos se caracterizam em suas partes como real e imaginária, gerando dois tensores reais, aplicando-se as operações necessárias e, posteriormente, reconstruindo-se o tensor de número complexo.

É necessário efetuar a operação inversa com a saída do modelo, ou seja, ir de coordenadas polares de volta para as coordenadas cartesianas. A equação para a sua obtenção é representada por $Z = \rho e^{i\varphi}$; onde Z equivale ao número resultante, ρ é a magnitude e φé a fase. Infelizmente, isso causa o mesmo problema de antes, onde não é possível realizar a operação $e^{i\varphi}$ com os números complexos na linguagem de programação *JavaScript* no navegador. Para contornar tal situação, foi utilizada a equação de Euler, $e^{i\phi} = cos(\phi) + isin(\phi)$. O cálculo efetuado resulta em um conjunto complexo que pode ser passado por intermédio da Transformada de *Fourier* de Tempo Curto Inverso (ISTFT).

4.2.2 Modelo escolhido

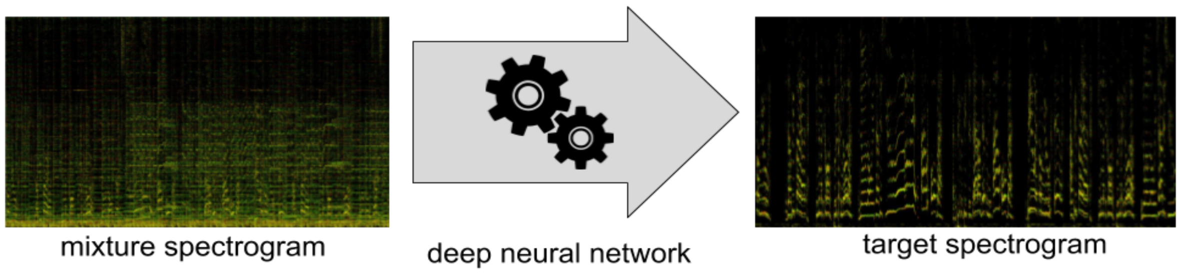
Inicialmente, o modelo de IA a ser utilizado seria o *Spleeter*, da empresa Deezer, desenvolvido originalmente em *Python*, e em seguida seria convertido para *JavaScript* por meio da ferramenta de conversão do *Tensor FlowJS* (TFJS). Entretanto, não foi possível converter esse modelo, pois ele utiliza as operações que não estão disponíveis no TFJS, como

[27] *WebGL*: *Application Programming Interface* (API) (Interface de Programação de Aplicativos) para a utilização de aceleramento de *Graphics Processing Uni* (GPUs) ((Unidade de Processamento Gráfico) na *Web* (KHRONOS, 2021).

ocorre na ISTFT. Neste sentido, a outra opção de modelo, desenvolvido pela equipe SigSep utilizando o *Tensor Flow* 1.5 foi testada, mas como também utilizava as operações ainda não implementadas no TFJS, tais como as células de *Long Short-Term Memory*, não foi possível realizar a conversão.

A solução encontrada foi o desenvolvimento de um novo modelo, que só utilizasse as operações disponíveis na TFJS, assim substituindo as camadas LSTM por convoluções. Esse modelo apresenta um comprometimento entre a precisão e a complexidade computacional, gerando resultados menos precisos se comparados aos modelos originais, mas sendo caracterizado como mais leve, o que permite um funcionamento mais ágil no navegador. A ideia por trás do modelo é de agir como um filtro que remova as frequências que não pertencem ao instrumento alvo, como pode ser identificado na Figura 13.

Figura 13 – Representação visual do modelo[28]



Fonte: Elaboração própria (2020).

O modelo utilizado gera apenas os vocais, o que também auxilia a poupar muita complexidade computacional. Ainda é possível obter o acompanhamento da música ao subtrair os vocais gerados da canção original.

4.2.3 ISTFT

A Transformada Inversa de *Fourier* de Tempo Curto (ISTFT) compreendeu uma das partes consideradas mais difíceis do Projeto. Isto ocorreu pelo fato que os acadêmicos não tinham a experiência necessária no campo de processamento digital de sinais, e foi necessário muito estudo para poder implementar essa operação em *JavaScript*.

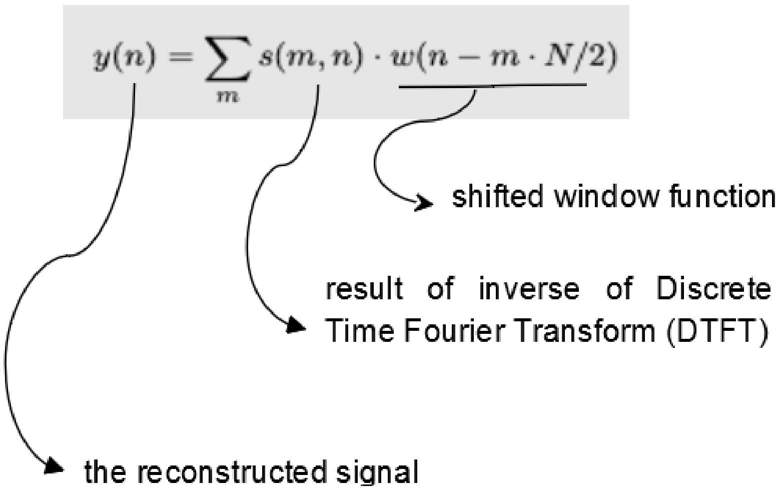
Como mencionado anteriormente, no item de trabalhos prévios (3.1.1), foram testadas diversas implementações diferentes dessa operação, e a que melhor se adaptaria à

[28] *Mixture spectogram* (espectrograma de mistura)*; deep neural network* (rede neural profunda); *target spectogram* (espectrograma alvo).

pipeline de pré-processamento seria a configuração da biblioteca *MagentaJS*. Excepcionalmente, esta operação do ISTFT não funcionou quando empregado ao Projeto, por apresentar diversos valores fixos e por se caracterizar como uma função não exportada pela biblioteca, além de ser incompatível com a implementação da STFT da própria biblioteca *MagentaJS*. Desse modo, a função foi utilizada como um ponto de partida para uma solução original, mais adaptada às necessidades do Projeto.

A ideia básica da ISTFT então é representar o inverso da STFT, ou seja, executar a Transformada Discreta Inversa de *Fourier* utilizando uma função de janela específica para recuperar o sinal original, como ilustrado na Figura 14.

Figura 14 – Soma de blocos sobrepostos para obter o sinal final[29]


$y(n) = \sum_m s(m,n) \cdot w(n - m \cdot N/2)$
shifted window function
result of inverse of Discrete Time Fourier Transform (DTFT)
the reconstructed signal

Fonte: Selesnick (2009).

Na época em que o trabalho foi desenvolvido, a biblioteca *TensorFlowJS* não possuía a Transformada Rápida de *Fourier* Inversa (IFFT), apenas a Transformada Rápida de *Fourier* Inversa Real (IRFFT). Essa função, dada uma entrada complexa, retorna a parte real do resultado da IFFT. Como o Projeto trabalha apenas com os sinais de valores reais, isto é, uma música, a parte imaginária da IFFT será sempre 0 (zero), já que se pretende reconstruir o sinal original. Isso possibilitou o uso dessa operação no Projeto, permitindo assim o desenvolvimento da ISTFT apenas com as funções da TFJS.

A entrada da função é um tensor complexo, de tamanho N x *Framelength*, como o *output* da STFT. Primeiro, a função IRFFT da biblioteca TFJS é aplicada nesse tensor. Em seguida, ajusta-se a função de janela de acordo com o fator de normalização desejado. Para

[29] *Shifted window function*: função de janela deslocada. *Result of inverse of discrete*: resultado do inverso do discreto. *The reconstructes signal*: o sinal reconstruído.

esse Projeto, o fator desejado é sempre 1, para que a janela seja compatível com os parâmetros utilizados na STFT, permitindo assim a reconstrução do sinal. O fator foi deixado parametrizável para que essa implementação de ISTFT fosse reutilizável fora do contexto do Projeto. Uma vez normalizada, a função janela foi aplicada no resultado da *IRFFT* como mostrado no Quadro 2.

Quadro 2 – Função janela aplicada no resultado *IRFFT*

```
let irfftTF = complex.irfft()

// Adjust normalization for 2096/1024 (factor of 1.0 with stft/istft)
// used by the model
let normalizationFactor = tf.mul(ifftWindowTF, tf.tensor(winFactor));

// Apply window
let res = tf.mul(irfftTF, normalizationFactor).arraySync();
```

Fonte: Elaboração própria (2020).

Desse modo, o resultado passa então pelo processo de ‘sobreposição e soma’ (*overlap-add,* em inglês), onde os ‘n’ segmentos do vetor são somados, reconstruindo assim o sinal. O resultado desta função é uma matriz de ponto flutuante, como indicada no Quadro 3.

Quadro 3 – Matriz e ponto flutuante

```
// Overlap and add
   for(let i = 0; i < nFrames; i++){
      let sample = i * hopLength;
      let yTmp = res[i];
      yTmp = add(yTmp, istftResult.slice(sample, sample + nFft));
      istftResult.set(yTmp, sample);
   }

return istftResult
```

Fonte: Elaboração própria (2020).

Acrescenta-se que, uma música não é processada de uma vez só – ao invés disso, ela é cortada em pedaços, os quais por sua vez são processados individualmente, e então adicionados a uma matriz de resultados. Uma vez que todos os segmentos tenham sido processados, essa matriz é transformada novamente em um arquivo de áudio WAV[30].

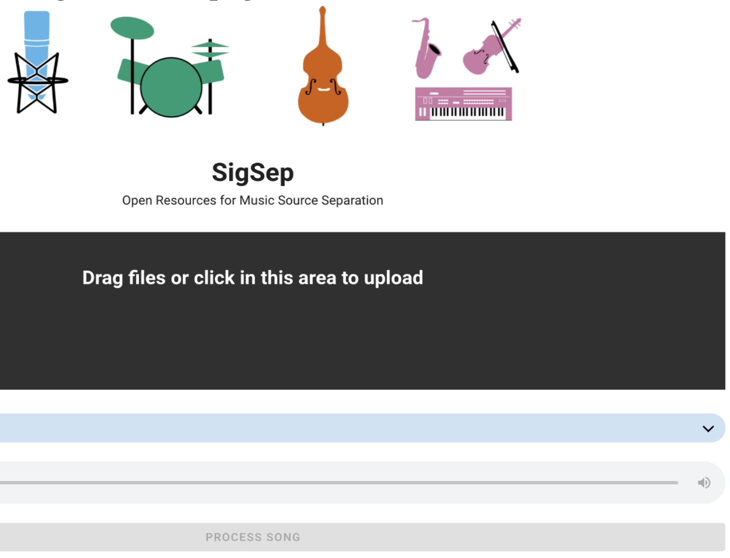
[30] *WAV*: tipo de arquivo de áudio sem compressão (FLEISCHMAN, 1998).

4.2.4 *Website*

Para que a aplicação pudesse rodar em navegadores, foi necessário desenvolver um *Website*. Para isso, foi escolhido a *Framework JavaScript Vue*, que facilita o processo de desenvolvimento de uma Interface de Usuário (IU). Essa *Framework* implementa uma arquitetura *model-viewmodel-view*[31], que separa o desenvolvimento da IU da lógica do programa. Ela foi escolhida, pois o tutor do Projeto já havia desenvolvido um componente capaz de tocar e mixar as múltiplas faixas de áudio utilizando tal *Framework*. O *site* já está disponível em <https://sigsep.github.io/open-unmix-js/>.

O *site* desenvolvido consiste em um seletor de arquivos de arrastar e soltar (*drag-and-drop*) *vue-dropzone*[32] (WINSEMIUS, 2016), como pode ser observado na Figura 15.

Figura 15 – A página inicial do *site*


SigSep
Open Resources for Music Source Separation
Drag files or click in this area to upload
Select model
0:00 / 0:00
PROCESS SONG

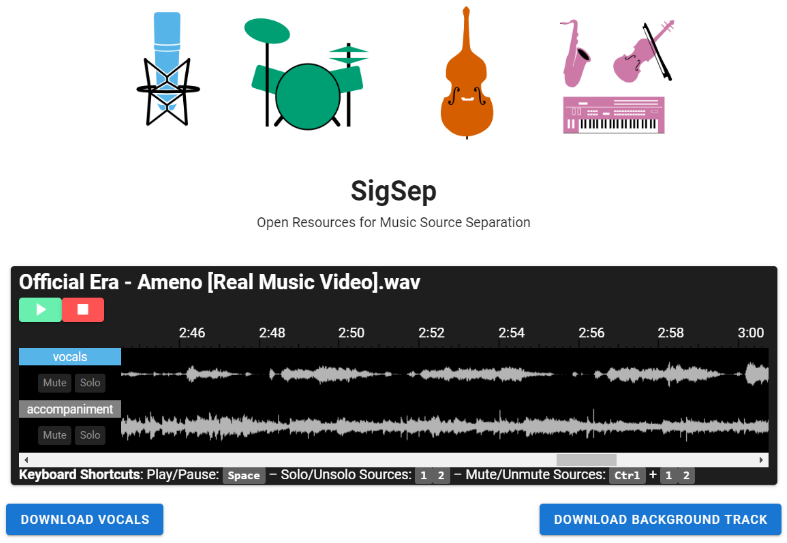
Fonte: Elaboração própria (2020).

Uma vez selecionado, o arquivo é carregado em memória e colocado em uma *tag* de áudio HTML5[33] padrão, para que o usuário possa ouvir a música selecionada. O limite de tamanho atual é de 50 MB[34], pois os arquivos maiores tendem a travar a aplicação.

[31] *Model-viewmodel-view*: padrão de arquitetura de páginas *Web* onde os componentes visuais são separados da lógica de negócios.
[32] *Vue-dropzone*: componente utilizado para o *upload* de arquivos no *site* (termo em inglês não possui tradução para o português).
[33] HTML5: Última versão do *HyperText Markup Language* (HTML) que contém alguns componentes utilizados para a construção do *site* (*MDN WEB DOCS*, 2021a;b).
[34] *MB: Megabyte*.

O usuário pode então selecionar um modelo de aprendizado de máquinas. Essa funcionalidade foi desenvolvida depois do Projeto original, e permite que um usuário teste diferentes modelos e selecione aquele mais apropriado para o seu caso de uso. Ele então pode clicar no botão *Process Song*[35] para começar o processamento da música, e poderá durar entre alguns segundos até alguns minutos, dependendo do tamanho da faixa musical. Uma vez pronta a 'desmixagem', as faixas musicais são carregadas no tocador, onde o usuário pode tocar apenas os vocais, ou apenas o acompanhamento ou ambos ao mesmo tempo, como demonstrado na Figura 16. Ele também pode baixar um arquivo *WAV* dos vocais isolados ou do acompanhamento, para em seguida compartilhar os resultados com outras pessoas.

Figura 16 – Resultados da 'desmixagem' no tocador


SigSep
Open Resources for Music Source Separation
Official Era - Ameno [Real Music Video].wav
2:46
2:48
2:50
2:52
2:54
2:56
2:58
3:00
vocals
Mute
Solo
accompaniment
Mute
Solo
Keyboard Shortcuts: Play/Pause: Space – Solo/Unsolo Sources: 1 2 – Mute/Unmute Sources: Ctrl + 1 2
DOWNLOAD VOCALS
DOWNLOAD BACKGROUND TRACK

Fonte: Elaboração própria (2020).

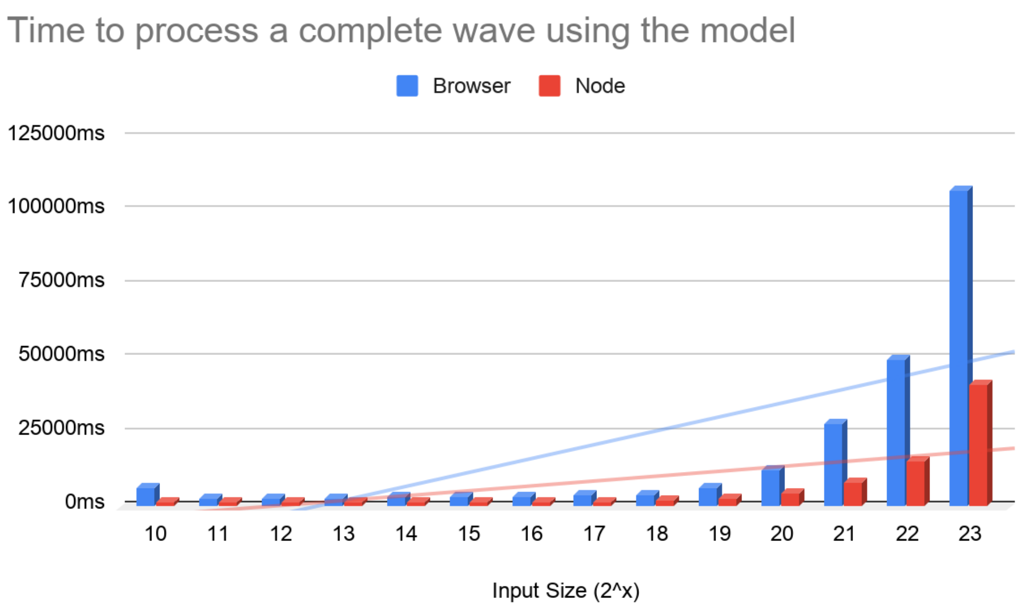
5 TESTES DO PROJETO

Para garantir que as operações desenvolvidas estivessem corretas, foram executados os testes de integração que verificam o comportamento das diferentes partes do programa quando operam juntas. Espera-se, por exemplo, que um sinal passe pela STFT e pela ISTFT e seja perfeitamente reconstruído. Para tanto, foi desenvolvido um teste que gerasse um sinal aleatório, aplicando-se a STFT e a ISTFT e calculando-se o erro quadrático médio em relação ao sinal original. O resultado foi um erro menor do que 10^6.

[35] *Process Song*: Processar Música.

Para medir o desempenho do programa, foram rodados alguns testes da *pipeline* inteira em dois ambientes diferentes, no *Node* e em um *browser* (navegador do *Google Chrome*), com *inputs*[36] de tamanhos diferentes, medindo o tempo levado para o processamento. Os resultados estão representados na Figura 17.

Figura 17 – Diferença no tempo de execução, navegador *versus* Node



Fonte: Elaboração própria (2020).

É possível perceber na Figura 17 que, o tempo de processamento em ambos ambientes cresce exponencialmente de acordo com a entrada, porém esse crescimento é menos pronunciado no ambiente *Node*[37]. Isso provavelmente se deve ao fato que, nesse ambiente, a *pipeline* pode rodar em paralelo, enquanto que no *browser* ela está limitada a apenas um *thread* (fio).

Pode-se notar também que, o primeiro teste (2^{10}) leva mais tempo que os testes a seguir, porque inclui o tempo que leva para que o modelo carregue, o que só é realizado uma vez para cada instância do programa.

6 PLATAFORMA DE COMPARTILHAMENTO

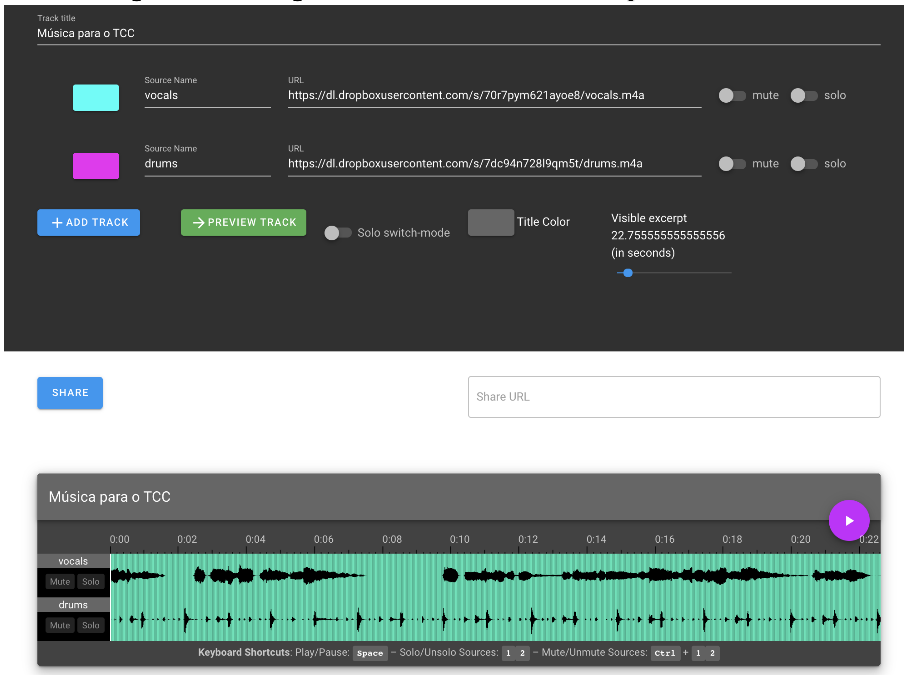
Com a conclusão do trabalho original, o Projeto continua em desenvolvimento. Um dos objetivos iniciais era a construção de uma plataforma onde os usuários pudessem fazer o

[36] *Inputs*: entradas.
[37] *Node*: ambiente de execução para aplicações desenvolvidas na linguagem *JavaScript* (NODEJS, 2021).

upload[38] das faixas 'desmixadas' e fosse então gerado um tocador compartilhável, como o utilizado no *site* de 'desmixagem'. Foi decidido que, em vez de suportar o *upload* direto de arquivos, seria mais conveniente utilizar uma *URL*[39] que aponta para o arquivo em algum serviço de compartilhamento, como o *Dropbox*[40]. Dessa forma, os pesquisadores evitam custos desnecessários com o armazenamento desses arquivos e possíveis problemas legais se um usuário realizar o *upload* de uma faixa que possui direitos autorais (*copyright*).

O *site* desenvolvido no Projeto apresenta uma arquitetura simples – um *front-end Vue*, como o do *site* inicial, e um banco de dados *NoSQL*[41], o *Firebase*. O *Firebase* é uma plataforma da *Google* que contém diversas soluções de banco de dados e de monitoramento para as aplicações da *Web*. Para guardar as informações das faixas a serem compartilhadas, foi utilizado o *Firestore*, que é um banco de dados *NoSQL*, de fácil configuração e de baixo custo. Desse modo, o *site* consiste em uma página para a criação de um reprodutor de música compartilhado, onde o usuário pode configurar o título, a cor e o nome das faixas, dentre outras configurações de aparência, como pode ser visualizado na Figura 18.

Figura 18 – Página inicial do *site* de compartilhamento



Track title
Música para o TCC

	Source Name	URL		
🟦	vocals	https://dl.dropboxusercontent.com/s/70r7pym621ayoe8/vocals.m4a	mute	solo
🟪	drums	https://dl.dropboxusercontent.com/s/7dc94n728l9qm5t/drums.m4a	mute	solo

Solo switch-mode

Title Color

Visible excerpt
22.755555555555556
(in seconds)

Share URL

Música para o TCC

Keyboard Shortcuts: Play/Pause: `Space` – Solo/Unsolo Sources: `1` `2` – Mute/Unmute Sources: `Ctrl` + `1` `2`

Fonte: Elaboração própria (2020).

[38] *Uploaad*: envio.
[39] *URL: Uniform Resource Locator*: localizador padrão de recursos.
[40] *Dropbox:* plataforma de armazenamento e de compartilhamento de arquivos (DROPBOX, 2021).
[41] *NoSQL*: banco de dados não relacionais.

Uma vez que o usuário esteja satisfeito com o tocador, ele poderá compartilhá-lo por intermédio do botão *Share* (compartilhar). Assim sendo, o programa envia então os dados deste reprodutor de música para o *Firestore*, que por sua vez, salva em um documento com os seguintes campos:

title: *string*: título do tocador;

titleColor: *string*: cor do título;

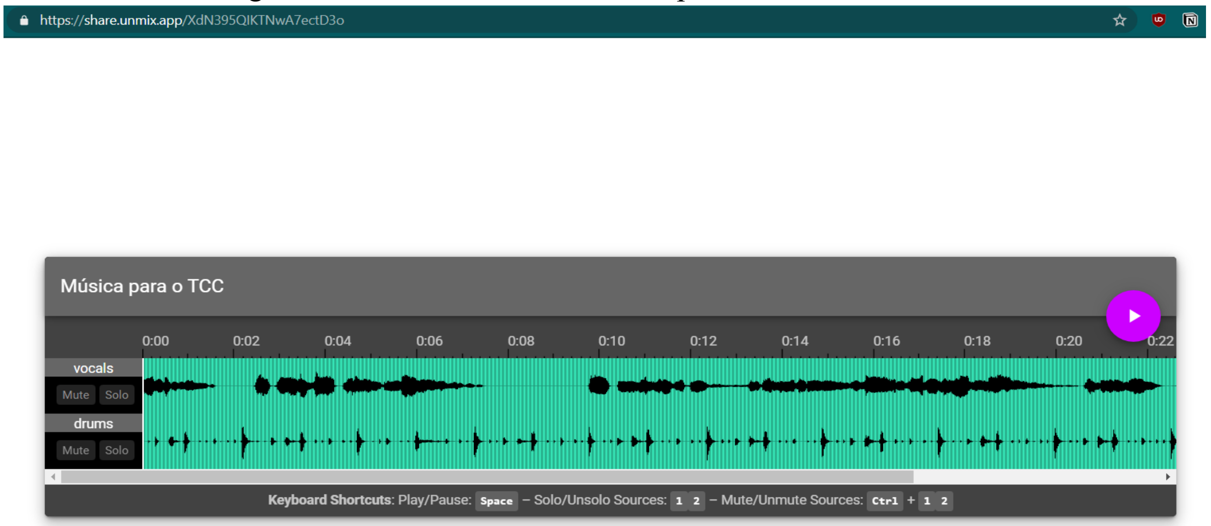
zoom: *int*: ampliação das ondas exibidas no tocador;

streams: lista de faixas, onde cada faixa possui:

- *name*: *string*: nome da faixa;
- *customClass*: *string*: dados sobre a aparência da faixa;
- *solo/mute*: *bool*: se a faixa começa a tocar mutada (sem som) ou em solo;
- *src*: *string*: *URL* do faixa.

Uma vez que os dados tenham sido salvos no banco de dados, o *Firestore* retorna o identificador único do documento, que é utilizado para criar a *URL* única a qual identifica aquele tocador. Quando um usuário entra no *site* a partir de uma *URL* que aponta para um tocador específico, a aplicação extrai esse identificador único da *URL* e a utiliza para consultar o *Firestore*, o qual retorna o documento com as configurações para montar o tocador, como mostrado na Figura 19. O *site* já está disponível para o acesso dos usuários em <https://share.unmix.app>.

Figura 19 – Acesso a um tocador específico através da *URL*


https://share.unmix.app/XdN395QIKTNwA7ectD3o
Música para o TCC
0:00
0:02
0:04
0:06
0:08
0:10
0:12
0:14
0:16
0:18
0:20
0:22
vocals
Mute
Solo
drums
Mute
Solo
Keyboard Shortcuts: Play/Pause: Space – Solo/Unsolo Sources: 1 2 – Mute/Unmute Sources: Ctrl + 1 2

Fonte: Elaboração própria (2020).

7 TRABALHO FUTURO

O Projeto aqui apresentado continua em desenvolvimento, e possui algumas melhorias ainda planejadas. No início de 2021, os pesquisadores escreveram um artigo que será submetido como demonstração do Projeto para a *Web Audio Conference* (WAC) 2021, que ocorrerá *online* entre os dias 05 e 07 de julho daquele ano. O referido artigo foi aprovado, e, portanto será apresentado pelos pesquisadores virtualmente.

Acrescenta-se a isso que, estão planejadas melhorias na performance da aplicação, explorando as soluções como *Web Workers*[42] e *Web Assembly*[43]. Finalmente, uma vez que o projeto seja apresentado na WAC, no mês de julho de 2021, será lançado um pacote *N*ode *Package Manager* (NPM)[44] contendo as funções de DSP e de carregamento de modelos, para que outros desenvolvedores possam utilizá-las em seus projetos futuros.

8 CONCLUSÃO

O desenvolvimento do Projeto com certeza foi muito desafiador, pois, o que no início parecia ser fácil, como a *pipeline* das etapas de pré-processamento e de pós-processamento acabou se tornado a parte mais complicada do Projeto. Apesar das dificuldades, foi sem dúvida um Projeto muito recompensador, pois permitiu o aprendizado sobre um campo que não é muito explorado na área de graduação, no caso, o processamento de áudio.

Acrescenta-se que, a parte mais complexa foi, sem dúvida, o desenvolvimento da função de Transformada de *Fourier* de Tempo Curto Inversa, que envolveu muita pesquisa e aprendizado do domínio de processamento digital de sinais. Trabalhar com áudio no geral é desafiador, pois o ouvido humano consegue detectar sons muito sutis, então qualquer erro na função acabava gerando distorções audíveis no produto final.

Apesar de o trabalho original ter sido concluído há mais de um ano, os pesquisadores continuam trabalhando ativamente no Projeto, pois ainda se configura como uma fonte de desafios interessantes de programação. Sem dúvida, mesmo após esse trabalho o Projeto irá continuar, sempre com o objetivo de trazer o estado da arte da 'desmixagem' de músicas para o maior público possível.

[42] *Web workers*: compreende o objeto que permite a execução de código em *threads* separados do *thread* principal no navegador (MDN WEB DOCS, 2021c).

[43] *Web assembly*: Alvo de compilação portátil para aplicações *Web* (*WEBASSEMBLY*, 2021).

[44] *N*ode *Package Manager* (NPM): gerenciador de pacotes que é utilizado para a instalação de bibliotecas no ambiente de execução Node (NPM, 2021).

9 REFERÊNCIAS

AUDACITY. ***Digital audio fundamentals***. Publicado em: 2020. Disponível em: <https://manual.audacityteam.org/man/digital_audio.html>. Acesso em: 02 maio 2021.

BENGIO Y.. LECUN, Yann. ***Convolutional networks for images, speech, and time-series***. Publicado em: 07 nov. 2007. Disponível em: <https://www.researchgate.net/publication/2453996_Convolutional_Networks_for_Images_Speech_and_Time-Series> Acesso em: 02 maio 2021.

BROOK, Corban. ***Digital signal processing for javascript***. Publicado em: 2010. Disponível em: <https://github.com/corbanbrook/dsp.js>. Acesso em: 10 jan. 2020.

COHEN, Leon. ***Time-frequency analysis***. *Upper Saddle River*, New Jersey, Estados Unidos: Prentice Hall PTR, 1995.

DEVMEDIA. **Guia completo de *Python***. Disponível em: <https://www.devmedia.com.br/guia/python/37024>. Acesso em: 02 maio 2021.

DHILLON, Inderjit S.; SRA, Suvrit. ***Generalized Nonnegative Matrix Approximations with Bregman Divergences***. *Department of Computer Sciences: The Univ. of Texas at Austin Austin, TX 78712*, 2005. .Disponível em: <https://papers.nips.cc/paper/2005/file/d58e2f077670f4de9cd79 63c85 7f2534-Paper.pdf>. Acesso em: 02 maio 2021.

DROPBOX. **Mantenha a vida organizada e o fluxo do trabalho**: tudo em um só lugar. Disponível em: <https://www.dropbox.com/>. Acesso em: 26 abr. 2021.

FLEISCHMAN, Eric. ***Wave and AVI codec registries***. Publicado em: jun. 1998. Disponível em: <https://datatracker.ietf.org/doc/html/rfc2361>. Acesso em: 26 abr. 2021.

GOOGLE.. ***TensorFlowJS***. Publicado em: 2018a. Disponível em: <https://github.com/tensorflow/tfjs> Acesso em: 25 jan. 2020.

GOOGLE. ***Magenta.js***. 2018b. Disponível em: <https://github.com/tensorflow/ magenta-js>. Acesso em: 02 maio 2021.

HENNEQUIN, R Romain; KHLIL, Anis; VOITURET, Felix; MOUSSALLAM, Manuel. *Spleeter*: *a fast and efficient music source separation tool with pre-trained models*. ***The Journal of Open Source Software***, v. 5, n. 50, p. 1-4, 2020. Disponível em: <https://www.researchgate.net/publication/342429039_Spleeter_a_fast_and_efficient_music_source_separation_tool_with_pre-trained_models>. Acesso em: 02 maio 2021.

KINSLER, Lawrence E.; FREY, Austin R. ***Fundamentals of acoustics***. 2 ed. New York: John Wiley & Sons, In. 1962.

KOEKESTRA. ***Spleeter implementation by JavaScript*, 2019?** Publicado em: 2019. Disponível em: <https://koekestra.com/spleeter_js/>. Acesso em: 10 jan. 2020.

KHRONOS. ***Khronos Releases Final WebGL 1.0 Specification***. Disponível em: <https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification>. Acesso em: 02 maio 2021.

LYSENKO, Mikola. ***Short time Fourier transform***. Publicado em: 2013. Disponível em: <https://github.com/mikolalysenko/stft>. Acesso em: 10 jan. 2020.

MDN WEB DOCS. **HTML**: *Hyper TExt Markup Language*. Publicado em: 14 abr. 2021a. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTML>. Acesso em: 02 maio 2021.

______. **HTML$_5$**. Publicado em: 03 fev. 2021b. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>. Acesso em: 02 maio 2021.

______. ***Web Workers* API**. Publicado em: 06 maio 2021c. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API>. Acesso em: 08 maio 2021.

MECHANICAL ENGINEERING DEPARTMENT – MED. **EL-305 *Instrumentation & Control***. Publicado em: 2018. Disponível em: <https://med.neduet.edu.pk/node/131>. Acesso em: 02 dez. 2019.

MUSIC TECHNOLOGY GROUP – MTG. ***EssentiaJS***. Publicado em: 2019. Disponível em: <https:// github.com/MTG/essentia.js>. Acesso em: 25 jan.2020.

NODE PACKAGE MANAGER – NPM. ***Build amazing tbnhigs***. Disponível em: <https://www.npmjs.com/>. Acesso em: 02 maio 2021.

NODEJS. ***Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine***. Disponível em: <https://nodejs.org/en/>. Acesso em: 26 abr. 2021

NOCKERT, Jens. ***A fast Fourier transform library for JS***. Publicado em: 2012. Disponível em: <https://www.npmjs.com/package/fft>. Acesso em: 02 dez. 2019.

PYTORCH. ***From reserach to production***: *key features & capabilities*. Disponível em: <https://pytorch.org/>. Acesso em: 26 abr. 2021.

RILEY, Lewis A.. ***Review of complex numbers***, Publicado em: 2004. Disponível em: <http://webpages.ursinus.edu/lriley/ref/complex/node1.html>. Acesso em: 07 fev. 2020.

ROMA, Gerard; GRAIS, Emad M.; SIMPSON, Andrew J. R.; SOBIERAJ, Iwona; PLUMBLEY, Mark D.. ***Untwist***: *a new toolbox for audio source separation*. In: *Extended abstracts for the Late-Breaking Demo Session of the* 17th *International Society for Music Information Retrieval Conference* (ISMIR), p. 1-4, 2016. Disponível em: <https://pure.hud.ac.uk/en/publications/untwist-a-new-toolbox-for-audio-source-separation>. Acesso em: 21 de jan. 2020.

SCIPY. ***Scipy.signal.hann***. Publicado em: 11 maio 2014. Disponível em: <https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.hann.html>. Acesso em: 02 dez. 2019.

SELESNICK, Ivan W.. ***Short-time Fourier transform and its inverse***. Publicado em: 14 abr. 2009. Disponível em: <https://eeweb.engineering.nyu.edu/iselesni/EL713/STFT/stft_inverse.pdf>. Acesso em: 26 abr. 2021.

SLACK. **Plataforma para troca de mensagens**. Disponível em: <https://slack.com/intl/en-br/>. Acesso em: 02 maio 2021.

SMITH, Steven W.. ***The scientist and engineer´s guide to digital signal processing***: chapter 18: FFT convolution: the overlap-add method, Califórnia: Califórnia Tecnhical Pub, 1999. Disponível em: <http://www.dspguide. com/CH18.PDF>. Acesso em: 02 maio 2021.

SOUZA, Ivan. ***Framework***: descubra o que é, para que serve e por que você precisa de um para o seu *site*. Publicado em: 05 dez. 2019. Disponível em: <https://rockcontent. com/br/blog/ framework/>. Acesso em: 02 maio 2021.

STÖTER, Fabian-Robert; ULRICH, Stefan; LIUTKUS, Antoine; MITSUFUJI, Yuki *Open-Un*mix*: a reference implementation for music source separation*. ***The Journal of Open Source Software***, v. 4, n. 41, p. 1-6, 2019. Disponível em: <https://www.researchgate. net/publication/335688695_...>. Acesso em: 21 jan. 2020.

SWANSON, Jez. ***An interactive introduction to Fourier Transforms***. Publicado em: 10 set. 2019. Disponível em: <https://www.jezzamon.com/fourier/>. Acesso em: 25 jan. 2020.

WEBASSEMBLY. **Definição de *Webassembly***. Disponível em: <https://webassembly.org/>. Acesso em: 02 maio 2021.

WENIGER, Felix; LEHMANN, Alexander; SCHULLER, Björn. ***OpenBliSSART***: *design and evaluation of a research toolkit for blind source separation in audio recognition tasks*. In: *International Conference on Acoustics* (IEEE) and *Acoustics, Speech and Signal Processing* (ICASSP), 2011. Disponível em: <https://ieeexplore.ieee.org/document/5946809/figures# figures>. Acesso em: 21 jan. 2020.

WINSEMIUS, Rowan. ***Vue-Dropzone***. Publicado em: 2016. Disponível em: <https://github. com/rowanwins/vue-dropzone>. Acesso em: 21 jan. 2020.