# conTorchionist: A flexible nomadic library for exploring machine listening/learning in multiple platforms, languages, and time-contexts

**José Henrique Padovani**[1*], **Vinícius Cesar de Oliveira**[2†]

[1]ECrIS/CPMC – Escola de Música – Universidade Federal de Minas Gerais (UFMG)
Av. Pres. Antônio Carlos, 6627 - Pampulha, Belo Horizonte - MG / Brazil

[2]NICS/Instituto de Artes – Universidade Estadual de Campinas (UNICAMP)
R. da Reitoria, 165 - Cidade Universitária, Campinas - SP / Brazil

jhp@ufmg.br, oviniciuscesar@gmail.com

***Abstract.*** *The concept of 'machine listening' has profoundly shaped the development of interactive music systems, evolving from symbolic MIDI processing to a wide range of audio analysis techniques. While the recent proliferation of neural networks has greatly expanded these capabilities, it has also introduced complex processes that often function as opaque 'black boxes', limiting creative exploration. To address this, we present 'conTorchionist': a flexible, nomadic library designed to foster a more transparent approach to machine listening and learning. Built upon libtorch/PyTorch, it features a single shared core with dedicated interfaces for diverse environments, including Pure Data, Max, SuperCollider, and Python. This architecture unifies real-time and non-real-time workflows, leveraging libtorch's strengths in GPU-accelerated audio analysis, signal processing, and neural network inference. Ultimately, conTorchionist provides an adaptable toolset that empowers researchers and artists to seamlessly bridge different platforms, languages, and time-contexts in their creative and investigative work.*

## 1. Introduction

A number of advances in computational technologies were driven by biological and physiological metaphors, leading to widespread concepts and techniques, of which *neural networks*, *genetic algorithms*, *machine learning*, *computer vision*, and *artificial intelligence* are just a few examples. Among these, one of the most relevant in the context of music and sound computing is *machine listening* — a concept that, even in the early days of *interactive music systems* (IMS) primarily based on MIDI data processing, was already used to structure interactive pieces. Unlike traditional *live-electronics* setups, structured through the manipulation of audio signals in real-time, interactivity in those systems came to be structured by stablishing two main component categories or roles: *listening/sensing* on one side, and *performing/acting/composing* on the other. This perception-action duality can be identified in several early works and discussions, where *interaction* emerged as a new paradigm for musical practices. These practices strongly relied on computers and digital technologies to structure machine listening processes capable of identifying musical elements and structures by simulating human-like auditory and cognitive abilities [1, 2, 3, 4, 5, 6, 7, 8].

In parallel with the development of interactive music systems, the broader idea of *machine listening* has been explored in various fields, such as Automatic Speech Recognition (ASR), Music Information Retrieval (MIR), and Computational Auditory Scene Analysis (CASA). In these areas — in which the concept of a "listening machine" arose independently of the early IMS practices —, this paradigm can be related to different processes for analyzing, processing, and interpreting audio signals in ways that mimic human auditory perception across diverse contexts [9, 10, 11, 12, 13, 14, 15].

This context, alongside advancements in machine learning, has driven a significant expansion of the techniques and applications associated with *machine listening* in computer music and sound computing. This has resulted in diverse tools, libraries, and frameworks that enable the creative, investigative, and commercial use of *machine listening* techniques in various contexts, such as music production, recommendation systems, and interactive applications. Many of these tools rely on neural networks and other artificial intelligence resources, dramatically expanding what may be done with them.

However, few tools are available for use across multiple platforms, GPU/CPU architectures, and programming languages. This limitation hinders applying the same algorithms and techniques in different contexts, such as real-time audio processing in interactive music systems — like Pure Data (PD), Max, and SuperCollider (SC) — or non-real-time audio analysis in Python. Consequently, it restricts workflows and exploratory practices that could benefit from processing large numbers of sound files to train a model, for example[1]. Furthermore, although some of these tools provide neural processing capabilities across multiple operating systems, a disparity is notable: while audio processing units (UGens, externals, classes, etc.) are diverse and abundant, enabling highly flexible audio processing chains, the same is not true for neural network and machine learning algorithms, especially within computer music environments. Typically, the artist has very limited control over or access to the logic of model building or its internal mechanisms, which can restrict creative exploration and experimentation, leading to more alienated relationships

*Supported by CNPq and FAPEMIG.
†Supported by CAPES.

[1]A remarkable and inspiring exception is FluCoMa, which allows for cross-platform use in Pure Data, Max, SuperCollider, and CLI, with community-built wrappers also available for Python and an extension for the Reaper DAW, ReaCoMa.

between the artist and the technical tool, that fall into the category of *black box*.

In this context, we present **conTorchionist**, a library, in its early development stage, that aims to provide a flexible and adaptable toolkit for researchers and artists, enabling exploratory and creative practices in music creation and research involving *machine listening*, *machine learning*, and *neural networks*. The library is based on *libtorch*, the C++ counterpart of the well-known *PyTorch* library, and is designed as a single shared core with interfaces for diverse environments, including Pure Data, Max, SuperCollider, and Python. It integrates machine listening and machine learning for both real-time and non-real-time applications. As it is built on *libtorch*, *conTorchionist* leverages the strengths of that library in three key areas: 1) *machine learning*, by enabling the training and execution of models created in *PyTorch*; 2) audio analysis / processing/ synthesis, by providing techniques such as Fourier transforms, spectrograms, melspectrograms, and the like; and 3) tensor-based computation, by facilitating efficient processing on various GPU hardware architectures, such as NVIDIA/CUDA and Apple/MPS.

The structure of this paper is as follows: in the next section, we contextualize our work by discussing existing tools and libraries for machine listening and machine learning in music creation environments. The third section provides an overview of the *conTorchionist* library's architecture, highlighting some key ideas that are guiding its development. Section 4 describes several components of the library at its current stage of implementation, including objects for signal analysis, feature extraction, and machine learning. The final section outlines our conclusions and plans for future work, discussing the importance of developing adaptable tools for music creation and research involving machine learning resources.

## 2. Related work

A number of libraries, packages, and extensions have been developed to support machine listening and learning for audio analysis, processing, and synthesis in various contexts. Some of these tools are specifically designed for music creation environments, such as Pure Data, Max, and SuperCollider, while others are more general-purpose libraries intended for use in Python or other programming languages.

We will briefly outline the native and the most popular extensions available in these environments to contextualize some choices regarding the *conTorchionist* design[2].

### 2.1. Built-in tools

In Pure Data, the most well-known objects and extensions for retrieving pitch-related features and detecting musical events are probably the objects `[env~]` (dB RMS analyzer), `[bonk~]` (spectral onset detector), `[fiddle~]` (pitch and partial detection), `[pique]` (spectral frame peak analysis), and `[sigmund~]` (combining pitch/peak detection and tracking, among other things), by Miller Puckette[16][3]. These objects—some of which also have Max versions—provide basic functionalities for detecting features related to pitch, onset, and dynamics: the most traditional parameters used to describe sound in Western music.

Native Max objects for performing audio analysis and processing include `[retune~]` (pitch shifting), `[loudness~]` (loudness analysis, in LUFS), `[peakamp~]` (peak amplitude detection), and `[meter~]` / `[levelmeter~]` (RMS amplitude detection). As in Pure Data, a number of third-party objects are also available, and it is possible to use objects in these environments to deal with arrays/buffers and FFT/RFFT processes to construct more complex audio analysis and processing chains. Max also comes with `[waveform~]` and `[spectroscope~]`, which are useful for visualizing audio signals and their frequency content, respectively.

SuperCollider includes many UGens and classes for audio analysis and spectral processing. In the "Analysis" section of the documentation, 17 UGens and their respective classes are listed, which are self-descriptive: `AmpComp`, `AmpCompA`, `Amplitude`, `BeatTrack`, `BeatTrack2`, `DetectSilence`, `Keytrack`, `Loudness`, `MFCC`, `Onsets`, `Peak`,`PeakFollower`, `Pitch`, `RunningSum`, `SendPeakRMS`, `Slope`, and `ZeroCrossing`. SuperCollider also ships with many FFT-related UGens, including common feature extraction processes such as `SpecCentroid` and `SpecFlatness` [17].

As a general-purpose language, Python does not natively include any audio-related built-in modules or packages.

### 2.2. Extensions, plugins/quarks, and packages

In addition to the built-in tools, many extensions, plugins, and packages are available for Pure Data, Max, SuperCollider, and Python. As an exhaustive list of these tools is beyond the scope of this paper, we will focus on some of the most relevant and widely used libraries and extensions that are not only popular but also serve as references and inspiration for the design of *conTorchionist*. As some libraries and extensions are available for multiple environments, we will not organize this section by environment but rather by the name of those tools.

**`timbreID`** `timbreID` is one of the most well-known libraries for audio feature extraction in Pure Data, developed by William Brent [18, 14]. It provides many objects for extracting a wide range of audio features, including spectral and temporal features. It also includes the `[timbreID]` object, which allows for comparing vectors of features and incoming audio signals, enabling diverse processes such as

[2]This section references the following major versions: Pure Data 0.55, Max 9, SuperCollider 3.13, and Python 3.12.

[3]While some of these objects are technically extensions and do not belong to the core source code, they are usually distributed with the *vanilla* version of Pure Data, which is the de facto standard version.

classification, clustering, and similarity resynthesis (concatenative synthesis).

**Zsa.Descriptors** Zsa.Descriptors is a library for real-time audio descriptor analysis available for Max, developed by Mikhail Malt and Emmanuel Jourdan. The library provides a wide range of descriptors, including amplitude-based features, spectral analysis tools, energy and dynamics descriptors, and statistical measures.

**SCMIR** SCMIR is a library for Musical Information Retrieval (MIR) in SuperCollider developed by Nick Collins [19]. It provides a range of tools for analyzing and processing audio signals, including features for beat tracking, onset detection, and spectral analysis. SCMIR is structured into SuperCollider Classes, UGens, and a set of five additional executables (gmm, hmm, NeuralNet, noveltycurve, similaritymatrix) related to Gaussian Mixture Models, Hidden Markov Models, Neural Networks, k-Nearest Neighbors, Self-Organizing Maps, and various Markov model variants. Collins is also the author of a set of other tools for SuperCollider that, although not directly related to MIR, enable machine listening processes such as source separation through non-negative matrix factorization (NMF), polyphonic pitch tracking, constant Q pitch tracking, and auditory modeling through filter banks.

**FluCoMa (Fluid Corpus Manipulation)** FluCoMa is a multi-environment toolset for analyzing and processing audio in Max, Pure Data, SuperCollider, and the command-line interface (CLI), developed by the FluCoMa team and led by Pierre Alexandre Tremblay[20, 21, 22, 23]. It is the only library that provides a complete framework for audio analysis, feature extraction, and manipulation in multiple environments, allowing for the use of machine learning techniques in real-time and non-real-time audio processing. FluCoMa's structure consists of a core C++ library that leverages the CMake concept of *interface libraries* to build the source code for each algorithm during the configuration phase of the compilation process. The library provides a wide range of audio analysis and feature extraction tools, including spectral, temporal, and machine learning techniques. It has a strong focus on community building, with forums, documentation, and tutorials available to help users get started. Although the project does not provide a Python interface, James Bradbury has developed a Python wrapper and a REAPER extension for FluCoMa, allowing users to use the CLI version of the library in Python and REAPER, respectively [24].

**Librosa** Librosa is arguably the most popular Python library for audio and music analysis using Music Information Retrieval (MIR) techniques [25]. It provides a wide range of tools for audio feature extraction, including spectral analysis, temporal analysis, and machine learning techniques. Librosa is based on the SciPy and NumPy packages, and it allows for extracting a wide range of audio features, including spectral features (e.g., Mel-frequency cepstral coefficients, chroma features), temporal features (e.g., onset detection, beat tracking), and statistical features (e.g., zero-crossing rate, spectral centroid). It also provides tools for audio visualization and manipulation, making it a powerful and accessible tool for audio analysis and processing in Python.

**Essentia** Essentia is a C++ library with Python bindings for audio analysis and processing, developed by the Music Technology Group at Universitat Pompeu Fabra [26]. Like Librosa, Essentia provides a wide range of tools for audio feature extraction, including spectral analysis, temporal analysis, and machine learning techniques. While Librosa, written entirely in Python, focuses on general-purpose audio analysis, Essentia, written in C++ with Python bindings, prioritizes efficiency and scalability. Essentia is designed for both real-time and non-real-time audio processing. Its comprehensive toolkit for visualization and manipulation further establishes it as a powerful and versatile option for audio processing in Python.

**PyTorch / libtorch / torchaudio** PyTorch / libtorch is a comprehensive ecosystem for machine learning that also offers tools for audio analysis and processing through torchaudio. PyTorch is arguably the most popular machine learning library in Python, providing a flexible and efficient framework for building and training neural networks [27]. libtorch extends this functionality to C++, allowing for the deployment of models in high-performance applications and the integration of machine learning capabilities into C++ applications. torchaudio serves as the official audio library, providing essential tools for audio data processing, including I/O functionalities, a wide array of signal processing transformations (such as spectrograms and Mel-frequency cepstral coefficients), and utilities for working with audio datasets. While the range of audio processing tools in torchaudio is not as extensive as in dedicated libraries like Librosa or Essentia, they take advantage of the underlying PyTorch framework for efficient computation using tensors and multiple backends, including CPU and GPU. This integration allows for seamless interoperability between audio processing and machine learning tasks, enabling the development of end-to-end pipelines that can handle both audio data and machine learning models [28, 29].

**nn_tilde** nn_tilde is a Max and Pure Data external developed by Antoine Caillon and Philippe Esling for exploring neural networks through RAVE (Realtime Audio Variational AutoEncoder) models built using PyTorch. The nn_tilde library provides a single object, nn~, as an interface between Max/PD real-time signal processing and the libtorch C++ interface for neural networks. Gianluca Elia has ported nn_tilde to SuperCollider as nn.ar, allowing for its use in that environment as well[4]. As nn~ and nn.ar rely on libtorch, both provide options for GPU acceleration[5] [30, 31].

[4] [https://github.com/elgiano/nn.ar](https://github.com/elgiano/nn.ar)

[5] The support for GPU acceleration in PD was implemented by Benjamin Wesch and integrated into the base code of nn_tilde in version 1.6.0.

## 3. Overview of the *conTorchionist* architecture

The *conTorchionist* library's design is strongly influenced by previous tools and is intended to facilitate the integration of machine listening/learning techniques and processes into exploratory, open-ended creative and investigative practices. Briefly, it aims:

1. to work on different operating systems (Windows, macOS, and Linux) and architectures (ARM and Intel), allowing for computation on both CPU and GPU. GPU support includes NVIDIA/CUDA on Linux and Windows, and MPS (Metal Performance Shaders) on Apple Silicon devices.[6]
2. to use the same core algorithms and techniques in different environments, allowing for cross-platform workflows, such as using the same model in different operating systems and architectures, or using the same audio analysis process in different contexts (e.g., real-time audio processing in Pure Data, Max, and SuperCollider, or non-real-time audio analysis in Python),
3. to provide various objects, classes, libraries, and binaries that can be assembled to create different types of audio analysis, processing, and synthesis chains, allowing for a more flexible and exploratory approach to machine listening/learning techniques in different contexts,
4. to be highly configurable in terms of sound analysis, enabling the use of the library alongside other libraries and tools, such as `librosa` and `Essentia` in Python, or `FluCoMa` in Pure Data, Max, and SuperCollider,
5. to provide tools to use machine learning models in different contexts, such as non-real-time for audio analysis and corpus manipulation, or real-time for interactive music systems, allowing for the practical exploration of machine learning techniques in music creation and research.

The architecture is structured as follows:

- **Core library**: A C++ library that encapsulates the functionalities of *libtorch*, providing a unified interface to different environments and languages. The core library is designed to be platform/API agnostic, allowing for the use of the same core algorithms and techniques in different environments, such as Pure Data, Max, SuperCollider, and Python. The core library is divided into two main parts: the *Processors* and the *Utils*. The *Processors* are responsible for the core functionalities of the library, such as audio analysis, feature extraction, and machine learning. The *Utils* provide common functionalities that are used by the Processors, such as device management, audio unit conversion, and tensor manipulation.
- **Wrappers**: Language/environment-specific bindings that provide common functionalities, such as device selection, argument parsing, and a common interface with the core library. Each wrapper has its own structure to encapsulate the core library's functionalities idiomatically for the target environment/language, allowing for easier and more homogeneous use. For instance, the Pure Data wrapper provides a set of utility functions to parse arguments, manage CPU/GPU devices, and handle type conversions.

In its current development stage, the focus is on ensuring the core *Processors* function as expected within core tests and the Pure Data wrapper. The ultimate goal, however, is to have the core *Processors* operational in all wrappers — what was first implemented, as proof-of-concept, with the `RMSOverlapProcessor`.

## 4. Processors and objects

Currently, the *conTorchionist* library provides basic spectral analysis tools. The effort, in the current stage, is to make these core sinal processing processors have a high customizability and flexibility, comparing its results with other libraries to allow its intercompatibility with existing audio toolsets, specially `librosa` and `FluCoMa`. While this set of Processors is still small, it is designed to be extensible and modular, allowing for the addition of new processors enabling other feature extraction and audio analysis techniques — such as chroma features, and others — as well as audio manipulation processes.

### 4.1. Audio and Machine Listening Processors

The core of the conTorchionist library features a set of modular signal processing processors, each designed to provide real-time-capable audio analysis and transformation functionalities. These processors are implemented in C++ and exposed to environments such as Pure Data and Max through dedicated wrapper objects.

The `RFFTProcessor` and `IRFFTProcessor` classes provide real and inverse real Fast Fourier Transform operations, respectively. These processors support configurable windowing (Hann, Hamming, Blackman, Blackman-Harris etc.), diverse magnitude and power normalization modes, and output formats (complex, magnitude-phase, power-phase, dB-phase). Internally these classes use *libtorch*'s implementation of the RFFT/IRFFT algorithms, ensuring high performance and compatibility with the rest of the library. In the current Pure Data and Max wrappers, these processors are exposed as the `[torch.rfft~]` and `[torch.irfft~]` objects, respectively.

The `SpectrogramProcessor` builds upon the `RFFTProcessor` and on an auxiliary `CircularBuffer`. It supports multiple output units (magnitude, power, dB) and can be configured for different overlap/hop sizes and window lengths. The processor is designed to efficiently handle streaming audio independently of the host environment block/buffer size management,

[6] Other devices provided by *libtorch*, such as XLA (Accelerated Linear Algebra) and ROCm (Radeon Open Compute), are expected to be supported in the future, as they are already available in *libtorch* and *PyTorch*.

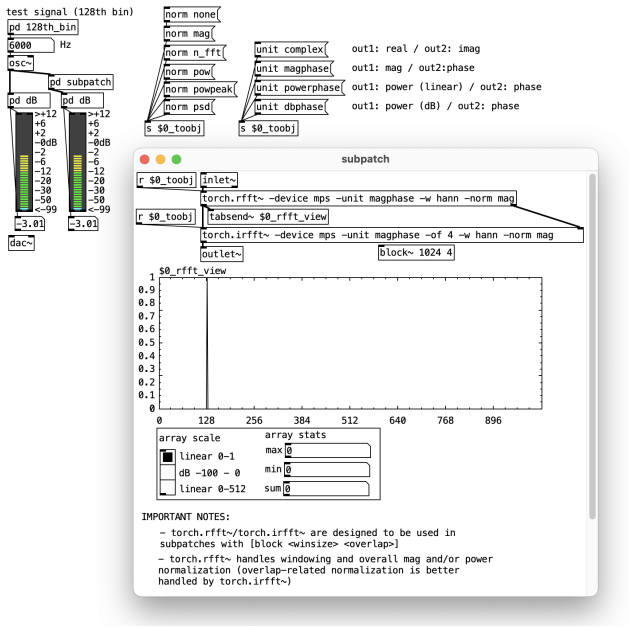


**Figure 1: Pure Data objects `[torch.rfft~]` and `[torch.irfft~]`, which use the `RFFTProcessor` to perform RFFT/IRFFT.**

allowing for real-time spectrogram computation. In the current Pure Data and Max wrappers, this processor is exposed as the `[torch.spectrogram~]` object.

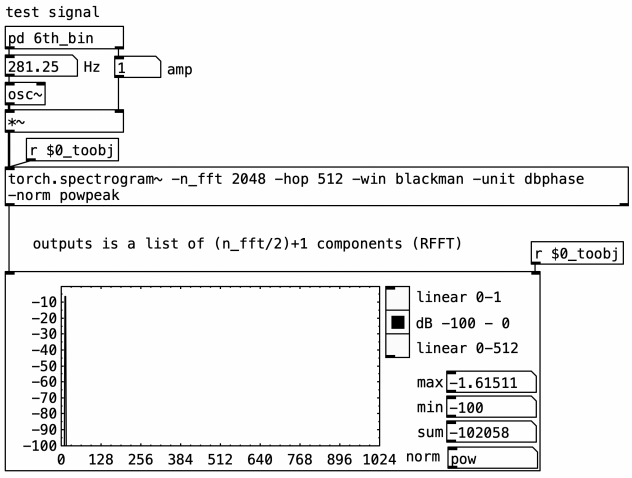


**Figure 2: Pure Data object `[torch.spectrogram~]`, based on the `SpectrogramProcessor`.**

The `MelSpectrogramProcessor` extends the spectrogram analysis by mapping the frequency axis to the mel scale. It supports multiple mel scale formulas and filterbank normalization (Slaney, HTK, none). The processor leverages the core's unit conversion utilities for accurate Hz<–>mel transformations, and can operate on any supported device. The mel filterbank is constructed dynamically based on user parameters. In the current Pure Data wrapper, this processor is exposed as the `[torch.melspectrogram~]` object.

The `MFCCProcessor` implements real-time

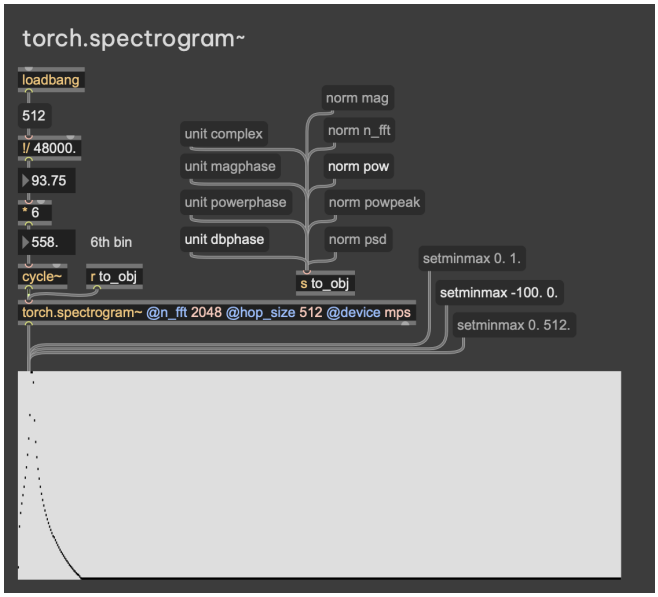


**Figure 3: The object `[torch.spectrogram~]` in Max.**

Mel-frequency cepstral coefficients (MFCC) extraction. It builds upon the `MelSpectrogramProcessor` to compute MFCCs by applying a discrete cosine transform (DCT) to the mel-spectrogram. As the other processors, it supports various configurations, including FFT size, hop length, number of mel filters, MFCC coefficient count, and normalization schemes. Also, it is possible to choose the device on which the computations will be performed, just like in the other processors. In the current Pure Data wrapper, this processor is used to implement the `[torch.mfcc~]` object .

The `RMSOverlapProcessor` implements efficient root-mean-square (RMS) energy analysis, using an overlap-add approach to smooth the RMS output. This processor is particularly useful for envelope following and dynamic analysis in real-time contexts. In the current Pure Data and Max wrappers, this processor is exposed as the `[torch.rms_overlap~]` object. In the current SuperCollider wrapper, it is exposed as the `RMSOverlap` class/UGen. In the current Python wrapper, it is exposed as the `RMSOverlap` class.

All audio processors are designed to be device-agnostic, with seamless support for CPU, CUDA, and MPS backends. The conversion between tensors and input/output formats (e.g., audio buffers, lists) is handled by the wrapper objects, ensuring that the processors work internally by making all the required tensor computation in the selected device.

### 4.2. Neural Network Processors

The machine learning and neural network-related objects are organized into two main workflows that reflect the design philosophy of *conTorchionist*: (1) running complex TorchScript models, built and pre-trained in Python, for real-time inference; and (2) building, training, and interactively exploring neural networks directly within the musical

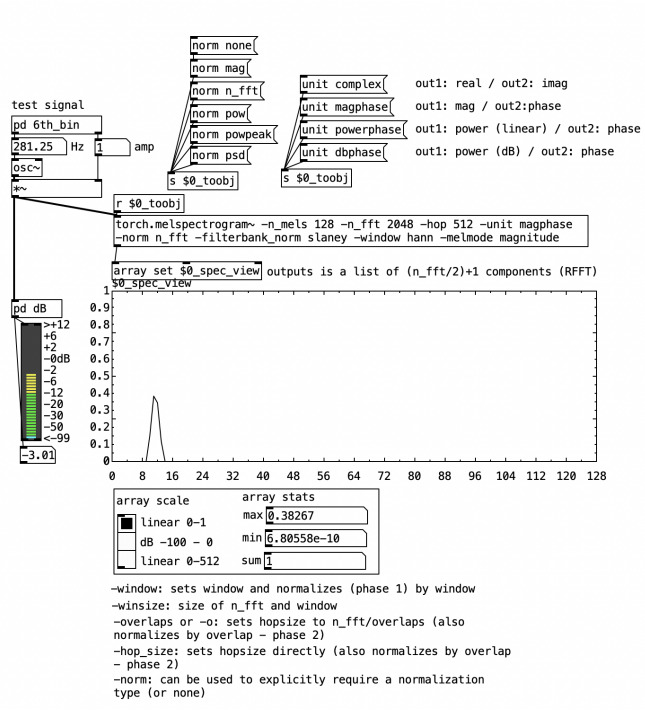


**Figure 4: Pure Data object `[torch.melspectrogram~]`, which use the `MelSpectrogramProcessor` to perform mel spectrogram analysis.**

computing environment. This organization allows for the dynamic integration of offline processes with real-time interactions.

The `TorchTSProcessor`, its related `[torch.ts]` object, and the `TorchTSProcessorSignal` with its related `[torch.ts~]` object enable the real-time loading and execution of *TorchScript* models. Briefly, the processor and the wrapper objects, allow to load and run pre-trained models built in Python using *PyTorch*. This approach, in conjunction with the future implementation of non-real time routines in Python that use the audio processors described in Section 4.1, will enable the use of diverse machine learning models for diverse tasks, such as audio classification, feature extraction, audio processing, and synthesis.

In Pure Data, the `[torch.ts]` object loads and runs *TorchScript* (.ts) model, and is designed to process symbolic or control data — such as parameter lists comprising sound and musical features, or MIDI messages. The `[torch.ts~]` object is the signal processing equivalent of `[torch.ts]`. This object is designed to load a *TorchScript* model and use it as an engine for real-time signal processing. It has multi-channel support, compatible with Pure Data's `[snake~ in]` and `[snake~ out]` objects[7]. Multi-channel mode can be set up via the `-mc` flag; the object is then created with one inlet and one outlet compatible with Pure Data's multi-channel objects.

[7]These objects implement Pure Data logic for dealing with multi-channel audio signals, and were introduced in version 0.54

One of the key features of `[torch.ts~]` is its dual processing mode, enabling both synchronous and asynchronous operation. The synchronous mode sends the input signal directly to the model for processing, without using intermediate buffers or multi-threading. In contrast, the asynchronous mode (configured using the `-async` flag) enables the creation of an internal circular buffer (implemented in the `CircularBuffer` class) with a specific size by parsing the flag `-buffersize` followed by the size value. This circular buffer accumulates successive blocks of the input signal and processes them together. Each time the circular buffer is completely filled, the model processing routine is called in a separate thread, and the entire buffer is processed. The processor output is copied to a second circular buffer, and the sequential audio vectors are then sent to the wrapper objects' outlet. This mode of operation allows for running models that require greater processing capacity separately from the audio thread, thereby reducing processing cost.[8]

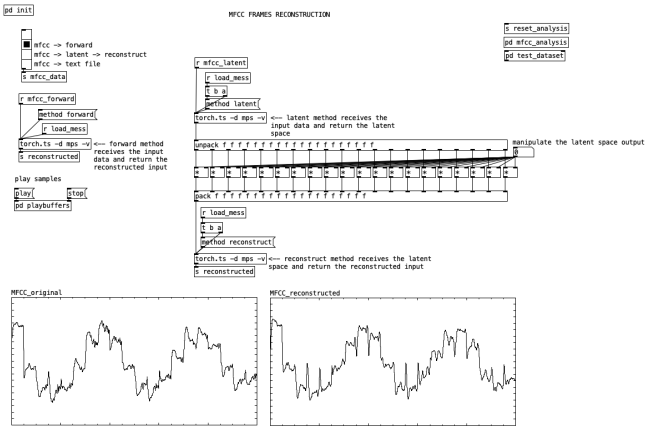


**Figure 5: Pure Data object `[torch.ts]`, which use the `TorchTSProcessor` to load an autoenconder *TorchScript* model and perform the reconstruction of MFCC frames**

The `SequentialProcessor` was designed to adapt the functionality of *PyTorch's* `torch.nn.Sequential` class, acting as a container that executes a sequence of layers in their order of addition. It facilitates the creation of models directly within the music computing environment, transforming it into an exploratory space for building and training simple neural networks, prototyping processes, and applying them directly and pragmatically. Each `SequentialProcessor` instance creates an individual named container, with a designated loss function, optimizer, and weight initialization method provided as arguments and flags. Once configured, instances of other compatible processors can be added to this container. Consequently, the constructed networks can be trained and can process input data layer by layer in

[8]It is worth noting that, just like in the `nn_tilde` /`[nn~]` and `NN.ar()` architectures, mentioned in section 2, `TorchTs` and the related wrapper objects do not make any assumptions about the internal behavior of the models they load. All inference logic, methods, internal states, expected formats, and processing strategies are built inside the *TorchScript* model itself.

sequential order. In the current Pure Data wrapper, this processor is exposed as the `[torch.sequential]` object.

The `Ls2TensorProcessor` was designed to create tensor datasets from incoming features — such as those produced by `MFCCProcessor` —, thereby bridging audio analysis and neural network processing. It implements an internal buffer that accumulates float data and lists of floats, which are then used to create a tensor of a specific shape. Similar to `SequentialProcessor`, each instance of `Ls2TensorProcessor` has a unique associated name, which allows other processors, such as `SequentialProcessor`, to dynamically access its internal tensor for machine learning tasks, including training, inference, or data manipulation. In the current Pure Data wrapper, `Ls2TensorProcessor` is exposed as `[torch.ls2tensor]` object.

The `LinearProcessor`, `MHAProcessor`, `ActivationProcessor`, and `ReshapeProcessor` implement linear layers, multi-head attention layers, activation functions, and reshape methods, respectively. Each of these processors has configurable parameters, including layer sizes (e.g., input, output, embeddings, heads, and batch) and methods (e.g., reshape methods, activation functions, and bias settings). They can be used to process input individually and, when compatible with the `SequentialProcessor`, function as modular building blocks that can be added to a sequential container as layers. In the current Pure Data wrapper, these processors are exposed as the `[torch.linear]`, `[torch.mha]`, `[torch.activation]`, and `[torch.reshape]` objects.

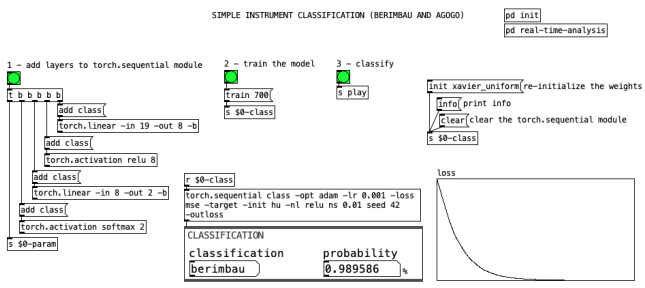


**Figure 6: A Pure Data patch showing an example of building a simple classification model by adding `[torch.linear]` and `[torch.activation]` objects to an instance of `[torch.sequential]`.**

## 5. Conclusion and Future Work

This paper presented *conTorchionist*, a flexible library for exploring machine listening and learning algorithms in musical creation across various platforms, languages, and application contexts. The main philosophy behind the *conTorchionist* library is to provide a flexible tool that addresses the existing gaps in integrating machine listening and learning algorithms into the practice of interactive music creation.

As we have shown, concept of a "nomadic" and flexible tool, enabling the use of the core processors in different environments, languages, architectures, operational systems, and time-contexts. This approach ensures interoperability, allowing the same algorithms and logic to be applied consistently in different artistic and investigative situations such as interactive music, computer-assisted composition/analysis, empirical musicology, etc.

While the *conTorchionist* library is in its early stages of development and many Processors and wrappers are still to be developed/improved, we believe that its flexibility and distribution under an open-source license (LGPLv3) will encourage its use and adaptation in various contexts, hopefully gathering contributions from the community[9].

## 6. Acknowledgments

The authors would like to thank FAPEMIG, CAPES and CNPq for their financial support. Also, we would like to thank the developers of the open-source libraries and tools mentioned in this paper, as they were not only a source of inspiration but also a reference for the design of `conTorchionist`.

## References

[1] Robert Rowe. Machine listening and composing with cypher. *Computer Music Journal*, pages 43–63, 1992.

[2] Robert Rowe, Brad Garton, Peter Desain, Henkjan Honing, Roger Dannenberg, Dean Jacobs, Stephen Travis Pope, Miller Puckette, Cort Lippe, Zack Settel, and George Lewis. Editor's Notes: Putting Max in Perspective. *Computer Music Journal*, 17(2):3–11, 1993.

[3] George E. Lewis. Interacting with latter-day musical automata. *Contemporary Music Review*, 18(3):99–112, January 1999.

[4] George E. Lewis. Too Many Notes: Computers, Complexity and Culture in "Voyager". *Leonardo Music Journal*, 10:33–39, 2000.

[5] Nicholas M. Collins. *Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems*. PhD thesis, Faculty of Music, University of Cambridge, Cambridge, 2006.

[6] Nick Collins. Musical robots and listening machines. In Nick Collins and Julio Escrivan, editors, *The Cambridge Companion to Electronic Music*, pages 171–184. Cambridge University Press, 1ª edição edition, December 2007.

[7] Nick Collins. *Machine Listening in SuperCollider*, pages 439–461. MIT Press, Cambridge, Mass, 2011.

[8] Tristan Jehan. *Creating Music by Listening*. PhD thesis, Massachusetts Institute of Technology, School of Architecture and Planning . . . , 2005.

[9] J. Stephen Downie. Music information retrieval. *Annual review of information science and technology*, 37(1):295–340, 2003.

[10] Geoffroy Peeters. A large set of audio features for sound description (similarity and classification) in the CUIDADO project. pages 1–25, 2004.

[11] Bozena Kostek. *Perception-Based Data Processing in Acoustics: Applications to Music Information Retrieval and*

[9] The *conTorchionist* library is available at `https://ecris.cc` and `https://github.com/ecrisufmg/contorchionist`

*Psychophysiology of Hearing*. Number v. 3 in Studies in Computational Intelligence. Springer, Berlin ; New York, 2005.

[12] Anssi Klapuri and Manuel Davy. *Signal Processing Methods for Music Transcription*. Springer, New York, 2006.

[13] Shipra J. Arora and Rishi Pal Singh. Automatic speech recognition: A review. *International Journal of Computer Applications*, 60(9), 2012.

[14] Guy J Brown and DeLiang Wang. *Computational Auditory Scene Analysis: Principles, Algorithms, and Applications*. 2015.

[15] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach*. Signals and Communication Technology. Springer London, London, 2015.

[16] Miller Puckette. Pure data: Recent progress. In *Proceedings of the Third Intercollege Computer Music Festival*, pages 1–4. Citeseer, 1997.

[17] Supercollider. SuperCollider Documentation. https://docs.supercollider.online/Browse.html#UGens%3EAnalysis, 2025.

[18] William Brent. Cepstral analysis tools for percussive timbre identification. In *Proceedings of the 3rd International Pure Data Convention*, pages 1–7, São Paulo, 2009.

[19] Nick Collins. SCMIR: A SuperCollider Music Information Retrieval Library. *International Computer Music Conference Proceedings*, 2011, 2011.

[20] Pierre Alexandre Tremblay, Owen Green, Gerard Roma, and Alexander Harker. From collections to corpora: Exploring sounds through fluid decomposition. In *International Computer Music Conference and New York City Electroacoustic Music Festival*, pages 223–228. International Computer Music Association, 2019.

[21] Pierre Alexandre Tremblay, Gerard Roma, and Owen Green. Enabling programmatic data mining as musicking: The fluid corpus manipulation toolkit. *Computer Music Journal*, 45(2):9–23, 2021.

[22] Ted Moore, James Bradbury, Pierre Alexandre Tremblay, and Owen Green. Making Machine Learning Musical: Reflections on a Year of Teaching FluCoMa. *Journal SEAMUS*, 32, 2021.

[23] Pierre Alexandre Tremblay, Owen Green, Gerard Roma, James Bradbury, Ted Moore, Jacob Hart, and Alex Harker. Fluid corpus manipulation toolbox. 2022.

[24] James Bradbury. *Harnessing Content-Aware Programs for Computer-Aided Composition in a Studio-Based Workflow*. PhD thesis, University of Huddersfield, 2021.

[25] Brian McFee, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. Librosa: Audio and music signal analysis in python. *SciPy*, 2015:18–24, 2015.

[26] Dmitry Bogdanov, N Wack, Emilia Gómez, Sankalp Gulati, Perfecto Herrera, Oscar Mayor, Gerard Roma, Justin Salamon, Jose Zapata, and Xavier Serra. *ESSENTIA: An Audio Analysis Library for Music Information Retrieval*. November 2013.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 721, pages 8026–8037. Curran Associates Inc., Red Hook, NY, USA, December 2019.

[28] Jeff Hwang, Moto Hira, Caroline Chen, Xiaohui Zhang, Zhaoheng Ni, Guangzhi Sun, Pingchuan Ma, Ruizhe Huang, Vineel Pratap, and Yuekai Zhang. TorchAudio 2.1: Advancing speech recognition, self-supervised learning, and audio processing components for PyTorch. In *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 1–9. IEEE, 2023.

[29] Yao-Yuan Yang, Moto Hira, Zhaoheng Ni, Artyom Astafurov, Caroline Chen, Christian Puhrsch, David Pollack, Dmitriy Genzel, Donny Greenberg, and Edward Z. Yang. Torchaudio: Building blocks for audio and speech processing. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6982–6986. IEEE, 2022.

[30] Antoine Caillon and Philippe Esling. RAVE: A variational autoencoder for fast and high-quality neural audio synthesis, December 2021.

[31] Antoine Caillon and Philippe Esling. Streamable Neural Audio Synthesis With Non-Causal Convolutions, April 2022.