IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 28, 2020, accepted August 10, 2020, date of publication August 24, 2020, date of current version September 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3019084

nnAudio: An on-the-Fly GPU Audio to Spectrogram Conversion Toolbox Using 1D Convolutional Neural Networks

KIN WAI CHEUK[1,3], (Graduate Student Member, IEEE), HANS ANDERSON[2], KAT AGRES[3,4], (Member, IEEE), AND DORIEN HERREMANS[1], (Senior Member, IEEE)

[1]Information Systems, Technology and Design, Singapore University of Technology and Design, Singapore 487372
[2]Blue Mangoo Software, Hung Yen 163953, Vietnam
[3]Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore 138632
[4]Yong Siew Toh Conservatory of Music, National University of Singapore, Singapore 117376

Corresponding author: Kin Wai Cheuk (kinwai_cheuk@mymail.sutd.edu.sg)

This work was supported in part by the Singapore University of Technology and Design - Massachusetts Institute of Technology (SUTD-MIT) International Design Center (IDC) Grant no. IDG31800103, in part by Ministry of Education (MOE) Grant no. MOE2018-T2-2-161 and Startup Research Grant no. SRG ISTD 2017 129, and in part by the Singapore International Graduate Award (SINGA) from the Agency for Science, Technology and Research (A*STAR) under Grant no. SING-2018-02-0204.

ABSTRACT In this paper, we present `nnAudio`, a new neural network-based audio processing framework with graphics processing unit (GPU) support that leverages 1D convolutional neural networks to perform time domain to frequency domain conversion. It allows on-the-fly spectrogram extraction due to its fast speed, without the need to store any spectrograms on the disk. Moreover, this approach also allows back-propagation on the waveforms-to-spectrograms transformation layer, and hence, the transformation process can be made trainable, further optimizing the waveform-to-spectrogram transformation for the specific task that the neural network is trained on. All spectrogram implementations scale as Big-O of linear time with respect to the input length. `nnAudio`, however, leverages the compute unified device architecture (CUDA) of 1D convolutional neural network from `PyTorch`, its short-time Fourier transform (STFT), Mel spectrogram, and constant-Q transform (CQT) implementations are an order of magnitude faster than other implementations using only the central processing unit (CPU). We tested our framework on three different machines with NVIDIA GPUs, and our framework significantly reduces the spectrogram extraction time from the order of seconds (using a popular python library `librosa`) to the order of milliseconds, given that the audio recordings are of the same length. When applying nnAudio to variable input audio lengths, an average of 11.5 hours are required to extract 34 spectrogram types with different parameters from the MusicNet dataset using `librosa`. An average of 2.8 hours is required for `nnAudio`, which is still four times faster than `librosa`. Our proposed framework also outperforms existing GPU processing libraries such as `Kapre` and `torchaudio` in terms of processing speed.

INDEX TERMS Convolution, discrete Fourier transform, short time Fourier transform, spectrogram, CQT, constant Q transform, Mel Spectrogram, signal processing, library, PyTorch, GPU.

I. INTRODUCTION

Spectrograms, as time-frequency representations of audio signals, have been used as input for neural network models since the 1980s [1]–[3]. Different types of spectrograms are tailored to different applications. For example, Mel spectrograms and Mel frequency cepstral coefficients (MFCCs) are designed for speech-related applications [4], [5], and the constant-Q transformation is best for music related applications [6], [7]. Despite recent advances in end-to-end learning in the audio domain, such as WaveNet [8] and SampleCNN [9], which make model training on raw audio data possible, many recent publications still use spectrograms as the input to their models for various applications [10]. These applications include speech recognition [11], [12], speech emotion detection [13], speech-to-speech translation [14], speech enhancement [15], voice separation [16], singing voice

The associate editor coordinating the review of this manuscript and approving it for publication was Sotirios Goudos.

conversion [17], music tagging [18], cover detection [19], melody extraction [20], and polyphonic music transcription [21]. One drawback of training an end-to-end model on raw audio data is the longer training time. As pointed out by Lee et al. [11], a model that uses raw audio data as input takes four times longer in terms of training time, and this longer training time only yields slightly better performance compared to a similar model that use spectrograms as their input.

Using spectrograms as input, however, does not come without drawbacks. Each audio recording can be transformed into various spectrograms using different algorithms and parameters. In order to find the audio transformation methods best suited to a specific task, trial and error may be needed. The usual way to conduct these trial and error experiments is to convert audio clips to different frequency domain representations and save each of the representations on the hard disk. After that, the neural networks are trained using each of the different representations and the best performing model is selected. Once the best frequency domain representation has been identified, the transformation parameters, such as window size and number of frequency bins, can be further fine-tuned to obtain an even better result.

Performing a parameter search to obtain the best spectrogram input yields two major problems. First, a considerable amount of hard disk space is required to store different frequency domain representations resulting from the different parameter settings. Given a dataset with 20GB of audio recordings (e.g. MusicNet [22]), the resultant spectrograms can easily occupy up to 1TB of hard disk space if one wants to experiment with different types of spectrograms with different parameters. A detailed case study will be discussed in Section V-A. Second, the audio processing step is usually done separately from the model training. To combine the processing step and model training into one continuous pipeline, on-the-fly spectrogram extraction is needed. The existing methods for time-frequency conversion of audio files, however, are too slow for on-the-fly spectrogram extraction. Most of the above-mentioned applications use `librosa` [23], a popular python audio processing library based on central processing units (CPUs). To use `librosa`together with a neural network model, the spectrograms need to be constantly transferred from a CPU to a graphics processing unit (GPU), since model training is done on GPUs. To make this process more efficient, it would be better to have a library that processes the spectrograms directly on the GPU.

There have been some attempts at implementing methods for GPU-based spectrogram extraction. `Tensorflow` [24] has a `tf.signal` package that performs the Fast Fourier Transform (FFT) and Short-Time Fourier Transform (STFT) on GPUs. There is a high-level API, called `Keras`, for people who want to quickly build a neural network without having to work with `Tensorflow` sessions. `Kapre` [25] is the `Keras` version for GPU-based audio processing. Along similar lines, `PyTorch` [26] has recently developed `torchaudio`, but this tool has not been fully integrated into `PyTorch` at the time of writing this paper. Furthermore, `torchaudio` requires `Libsox` as an extra dependency, and the installation often requires significant trouble-shooting [27]; for example, `torchaudio` is currently not compatible with Windows 10 [28]. Among the three tools, only `Kapre` and `torchaudio` support audio to Mel spectrogram conversion, but none of the existing libraries support constant-Q transform (CQT). Furthermore, only `Kapre` supports neural network-based signal processing since it is the only implementation that supports trainable kernels for time domain to frequency domain transformations. `Kapre`, however, cannot be integrated with the popular machine learning library `PyTorch` due to its `Tensorflow` backend. Despite the GPU support and differentiability, `torchaudio` and `tf.signal` are not neural network-based, meaning that there is no trainable parameter which can be learned or fine-tuned during neural network training. Although `torch-stft`[1] is a native `PyTorch` function without any additional dependency, only STFT is available.

Therefore, to bridge this gap in the field, we introduce a fast, differentiable, and trainable neural network-based audio processing framework called `nnAudio` [29]. To ensure perfect integration with one of the most popular machine learning libraries, we built our spectrogram extraction method using `PyTorch`. This way, our library can be used as a `PyTorch` neural network layer, and all the functionalities available in `PyTorch`, such as data augmentations, can be used together with `nnAudio`. Moreover, our proposed framework includes extended functionalities as compared to other existing libraries, such as calculating Mel spectrograms and constant-Q transforms. More specifically, we use a 1D convolution layer to implement the transformation algorithm, which makes the spectrogram extraction in `nnAudio` a trainable process (see Section V-B). `nnAudio` is hence useful when exploring different input representations for neural network models [30], [31]. Since our proposed framework is based on neural networks, the audio processing can be integrated into the model training as shown in Figure 1(b). That is, there is no need to do audio processing and model training separately, as in the traditional method shown in Figure 1(b). nnAudio enables on-the-fly spectrogram extraction and model training at the same time. In Section IV-A we discuss the improved performance of this method as compared to traditional approaches in Figure 1(a). The library is available online.[2]

A. SUMMARY OF KEY ADVANTAGES

The main contribution of this paper is the development of a GPU-based audio processing framework that is directly integrated into and leverages the power of neural networks. This provides the following benefits:

[1] https://github.com/pseeth/torch-stft

[2] Via PyPI (nnAudio), and `https://github.com/KinWaiCheuk/nnAudio`

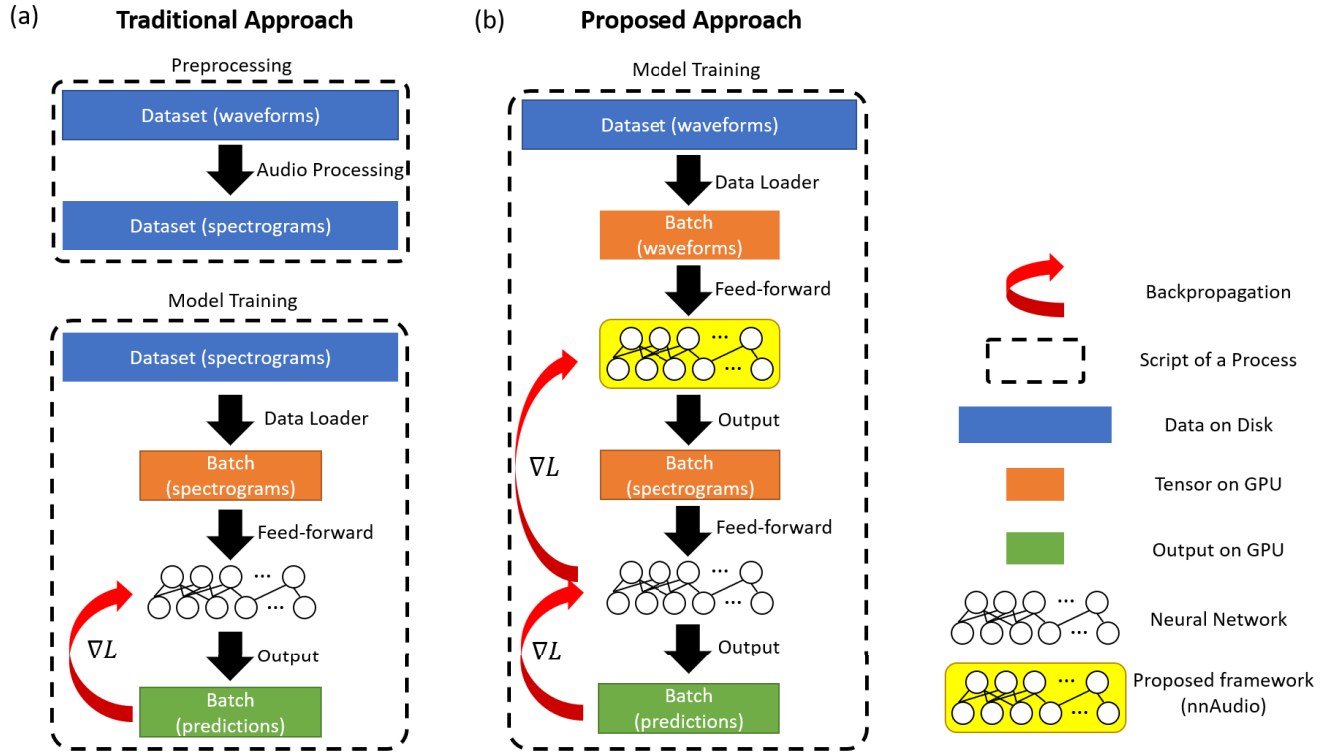

FIGURE 1. A flowchart comparing (a): existing (slow) approach [32]–[40] and (b): our proposed (much faster as shown in Figure 11a) neural network-based audio processing framework (nnAudio). Our proposed neural network is highlighted in yellow. Instead of pre-processing the waveforms, we can now feed-forward waveforms to the neural network directly, and spectrograms can be generated on-the-fly during training. The red arrows indicate how far the backpropagation ∇L may go, this allows the initialized kernels to be fine-tuned during training, resulting in specifically tailored new representations.

1) End-to-end neural network training with an on-the-fly time-frequency conversion layer (i.e. one can directly use raw waveforms as the input to the neural network).
2) Significantly faster processing speed compared to the traditional audio processing approach such as `librosa` [23].
3) CQT algorithms based on neural networks that can be run on GPUs (no neural network-based CQT algorithm that can be run on GPU is available at the time of this writing.)
4) Trainable Fourier, Mel and CQT kernels that can be automatically tailored to the problem at hand.

In the following subsections, we will briefly summarize the mathematics of the Discrete Fourier Transform (DFT). We will then discuss how to initialize a neural network to perform the STFT, Mel spectrogram and constant-Q transform (CQT) in Section-III. In section IV, we compare the speed and output of nnAudio versus a popular python signal processing library, librosa. Finally, we end with potential applications of our proposed library.

II. SIGNAL PROCESSING: A QUICK OVERVIEW

In this section, we will go over the basic transformation methods (DFT) used to convert signals from the time domain to the frequency domain. Readers who have a solid background in signal processing are welcome to skip this section and may continue reading from Section III.

A. DISCRETE FOURIER TRANSFORM (DFT)

When recording audio using any computer or mobile device, the analogue signal is converted to a digital signal before storing the data. Therefore, the audio waveforms consist of discrete data points. The Discrete Fourier Transform can be used to convert this discrete signal from the time domain to the frequency domain. Equation (1) shows the mathematical expression for the discrete Fourier transform [41], where $X[k]$ is the output in the frequency domain; and $x[n]$ is the n^{th} sample of the audio input in the time domain. For real-valued inputs, the frequency domain output $X[k]$ for $k \in [1, N/2]$ is equal to the output $X[k]$ for $k \in [N/2, N-1]$ in reverse order, where N is the window length (which is usually a power of two such as 1,024 and 2,048). To discard this redundant frequency domain information, only the first half of the frequency bins in the frequency domain will be extracted, i.e. $k \in [0, \frac{N}{2}]$. We define the DFT as a split-sum of real and complex components:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cos(2\pi k \frac{n}{N}) - i \sum_{n=0}^{N-1} x[n] \sin(2\pi k \frac{n}{N}). \quad (1)$$

When we use (1) to compute the DFT with a 1D convolutional neural network, we can calculate the real and complex terms separately using real-valued arithmetic.

The frequency k in the DFT is given in terms of normalized frequency (equivalent to cycles per window). The formula to convert the normalized frequency k to the frequency f in units of Hertz (Hz) is given by (2),

$$f = k\frac{s}{N}, \tag{2}$$

where s is the sample rate and N is the FFT window length.

B. DFT FOR ARBITRARY FREQUENCY RANGES

Since k is an integer ranging from zero to half of the window length, the DFT is only capable of resolving a finite number of distinct frequencies. For example, if the sampling rate is 44,100Hz, and the window length is 2,048, then the normalized frequencies for the DFT kernel are $k = [0, 1, 2, \ldots, 1024]$ which corresponds to a DFT kernel with frequencies $f = [0, 21.53, 43.07, \ldots, 22050]$ Hz (using (2)). The frequency resolution under this setting is 21.53Hz. For comparison, the lowest two notes on a piano keyboard are $A0 = 27.5$ Hz and $A\#0 = 29.14$ Hz. With a difference of less than 2 Hz between them, the DFT of 1,024 frequency bins is not sufficient to resolve the correct note. The frequency resolution Δf is given by (3). This resolution can be improved by increasing the window size N, however, increasing the window size results in a decrease in time resolution Δt, as shown in (4). Therefore, we are forced to make a compromise between time and frequency resolution as per (5).

$$\Delta f = \frac{s/2}{N/2} \tag{3}$$

$$\Delta t = \frac{N}{s} \tag{4}$$

$$\Delta f\, \Delta t = 1 \tag{5}$$

The vectors of the DFT transformation matrix are a basis for the set of all complex vectors of length N. This implies that applying the DFT followed by the inverse-DFT results in a perfect reconstruction of the original signal. Invertibility is important for many signal processing applications, but in information retrieval applications such as speech recognition and sound classification, it is not always necessary to use an invertible time-frequency transformation. In such cases we may want to modify the DFT in ways that no longer result in an orthogonal set of basis vectors.

One way to modify the DFT is to change the frequencies of the basis vectors to increase or decrease the number of bins in certain parts of the spectrum. To achieve linear-scale frequency with non-integer multiples of s/N in equation (2) we can replace k with $\sigma(k) = Ak + B$, where A and B are two constants. To find A and B, let f_e and f_s be the ending and starting frequencies of the range we want to analyse, and apply (2) to get (6), where $\mu \in \left[0, \frac{N}{2} + 1\right]$ is the number of bins chosen to be displayed in the spectrogram.

$$\sigma(k) = \frac{(f_e - f_s)N}{\mu s}k + \frac{f_s N}{s} \tag{6}$$

By the same token, we can generate basis vectors for a log-frequency spectrogram by using $\sigma(k) = Be^{Ak}$, resulting in $A = \frac{f_s N}{s}$ and $B = \frac{\ln \frac{f_e}{f_s}}{\mu}$ as shown in (7) below.

$$\sigma(k) = \frac{f_s N}{s}\left(\frac{f_e}{f_s}\right)^{\frac{k}{\mu}} \tag{7}$$

Note that we use the word "basis" informally here. These formulae do not guarantee a linearly-independent set of vectors, so the basis we get from this method may in fact be rank-deficient. When using (7) or (6), (1) becomes (8). This more general time-frequency transform permits us to focus the resolution of our spectrogram in the frequency range where it is most needed. For example, if our starting frequency is $f_s = 50$Hz and the ending frequency is $f_e = 6000$Hz, the linear frequency DFT kernel would have basis vectors with normalized frequency $\sigma(k \in [0, 1024]) = [2.32, 2.59, 2.86, \ldots, 278.10, 278.36]$. This corresponds to the frequency $f = [50, 55.8, 61.6, \ldots, 5988, 5994]$Hz. The frequency resolution has improved from 21.53Hz to 5.8Hz without changing the transform window size.

$$X[k] = \sum_{n=0}^{N-1} x[n]\cos(2\pi\sigma(k)\frac{n}{N}) - i\sum_{n=0}^{N-1} x[n]\sin(2\pi\sigma(k)\frac{n}{N}) \tag{8}$$

Note that this method only changes the spacing between the centres of adjacent frequency bins without affecting the width of the bins themselves. Because each bin represents a range of frequencies in a fixed-width region centred around f as given in (2), we will lose information if we space the bins too far apart.

In the next section, we explain how the DFT in (1) and the variable-resolution DFT in (8) is used to calculate the short-time Fourier transform (STFT) using a convolutional neural network. The frequency scaling factor will be integrated as one of the input features in our neural network-based framework.

III. NEURAL NETWORK-BASED FRAMEWORK

In this section, we will discuss how to calculate the short-time Fourier transform (STFT), Mel spectrogram, and constant-Q transform (CQT) using a 1D convolutional neural network. These are then implemented as a library (nnAudio) in PyTorch[2]. This paper assumes that the readers have a basic understanding of convolutional neural networks. A detailed explanation of CNNs is outside of the scope of this paper. Readers are highly encouraged to consult these papers [42], [43] in order to quickly obtain a grasp of the background knowledge in this area. Readers are also highly encouraged to visit our github page for the details of our implementations[2].

A neural network-based approach means that we encode known audio processing knowledge (the algorithms discussed above) into the neurons of the neural network, so that the neural network behaves in the same way as the original algorithms. The STFT is the fundamental operation for both

Mel spectrogram calculation and CQT. To convert the STFT spectrogram to a Mel spectrogram we simply multiply the spectrogram by a Mel filter bank kernel. Similarly, the computation of the CQT also begins with the STFT, followed by multiplication with a CQT kernel. We begin this section by explaining how we use a convolutional neural network to compute the STFT.

A. SHORT-TIME FOURIER TRANSFORM (STFT)

The Short-time Fourier Transform (STFT), also called the sliding-window DFT, refers to an application of the DFT wherein the signal is cut into short windows before performing the transform rather than performing one large transform on the entire signal [44]. For audio analysis applications, this is the standard way to apply the DFT.

The STFT is usually calculated using the Cooley-tukey Fast Fourier Transform algorithm (FFT), which is preferred because it computes the DFT in $O(N \log N)$ operations, as opposed to $O(N^2)$ for the canonical DFT implementation. However, implementations of the $O(N^2)$ DFT often out-perform the $O(N \log N)$ FFT for small values of N when the underlying platform supports fast vector multiplication. This is especially true when the computation is done in parallel on a GPU. Since neural network libraries typically include fast GPU-optimised convolution functions, we can compute the canonical DFT quickly on those platforms by expressing the vector multiplication in the DFT as a one-dimensional linear convolution operation.

Discrete linear convolution of a kernel h with a signal x is defined as follows,

$$(h * x)[n] = \sum_{m=0}^{M-1} x[n-m]h[m], \tag{9}$$

where M is the length of the kernel h. PyTorch defines a convolution function with a stride argument. The one dimensional convolution of x with h using a stride setting of k, denoted by the symbol $*^k$ is,

$$(h *^k x)[n] = \sum_{m=0}^{M-1} x[kn-m]h[m]. \tag{10}$$

We can use convolution with stride to make fast GPU-based implementations of the short time Fourier transform (STFT). To do this, we take each basis vector of the DFT as the filter kernel h, and compute the convolution with the input signal x once for each basis vector. We set the stride value according to the amount of overlap that we want to have between each DFT window. For example, for zero overlap, we set the stride to N, the length of the DFT; and for 1/2 window overlap, we set the stride to $N/2$.

Note that due to the way convolution is defined in (9) and the way that (10) computes array indices, we need to reverse the order of elements in the DFT basis vectors when creating the convolution kernels. The following expressions are the pair of convolution kernels ($h_{\text{re}}[k, n]$ and $h_{\text{i}}[k, n]$) that represent the real and imaginary components of the k^{th} DFT basis vector respectively,

$$h_{\text{re}}[k, n] = \cos(2\pi k \frac{N-n-1}{N}), \tag{11}$$

$$h_{\text{im}}[k, n] = \sin(2\pi k \frac{N-n-1}{N}). \tag{12}$$

The DFT is usually computed with a function that fades the samples at the edges of each window smoothly down to near zero to avoid the high-frequency artefacts that are introduced by cutting the window abruptly at the edges [45]. Typical examples of DFT window functions include Hann, Hamming, and Blackman types. In a GPU-based DFT implementation using a convolution function with stride (10), we can implement the window smoothing efficiently by multiplying these window function elementwise with the filter kernels h_i and h_r before doing the convolution.

When calculating spectrograms, we typically use the Discrete Fourier Transform of length $N = 2048$ or $N = 4096$, but other values of N are possible. We often cut the DFT windows so that they overlap each other by some amount in order to improve the time resolution. In a signal with T windows, we let X_t be the DFT of the window at index $t \in [0, T-1]$. The time domain representation of the window at index t will be denoted by x_t.

Figure 2 shows the schematic diagram for a neural network-based STFT. There are two main advantages of implementing the STFT using a `PyTorch` 1D convolutional neural network. First, it supports batch processing. Using a neural network-based framework, we can convert a tensor of audio clips to a tensor of spectrograms using tensor operations. Second, the neural network weights can be either fixed or trainable. We will discuss how trainable STFT kernels improve the frequency prediction accuracy in Section V-B.

nnAudio API: The STFT is implemented in nnAudio as the function `Spectrogram.STFT()`, with default arguments: n_fft = 2048, freq_bins = None, hop_length = 512, window = 'hann', freq_scale = 'no', center = True, pad_mode = 'reflect', fmin = 50, fmax = 6000, sr = 22050, trainable = False. This function has an optional argument `freq_scale` which allows the user to choose either a linear or a logarithmic frequency bin scale.

B. MEL SPECTROGRAM

The Mel frequency scale was proposed by Stevens *et al.* in 1937 as an attempt to quantify pitch such that equal differences in Mel-scale pitch correspond to equal differences in perceived pitch, regardless of the frequency in Hertz [46]. In addition to the original Mel scale proposed by Stevens *et al.*, there were several other attempts to obtain a revised version of the Mel scale [47]–[49]. Therefore, there is not a single "right" formula for the Mel scale, as various different formulae coexist in the literature [50]. The traditional frequency to Mel scale conversion is the one mentioned in O'Shaughnessy's book [51], which was implemented in the

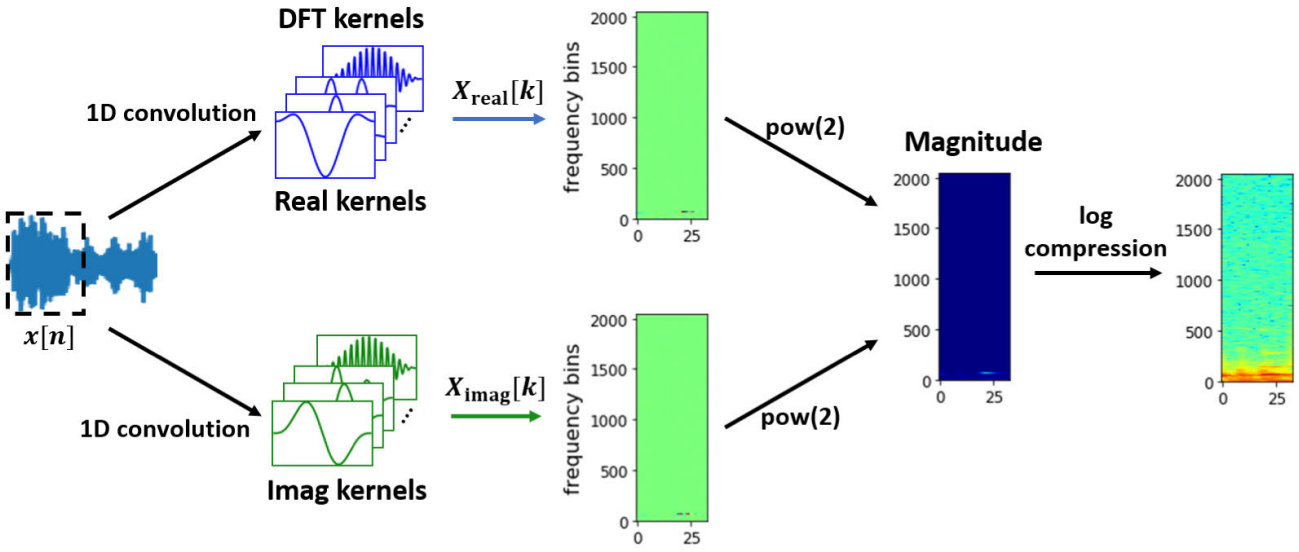

FIGURE 2. An STFT with a sliding window can be achieved by implementing DFT and initializing the 1D convolution kernels as cosine and sine in `PyTorch`. Applying logarithmic compression on the magnitude allows for a better visualization of the spectrogram.

HTK Speech Recognition toolkit [52] as (13)), shown below,

$$m = 2595\log_{10}\left(1 + \frac{f}{700}\right) \tag{13}$$

We refer to this form as 'htk' later on. Equation (14) shows another form that is being used in the Auditory Toolbox for MATLAB [53] and `librosa` (a python audio processing library) [23]. This form is quasi-logarithmic, meaning that the frequency to Mel scale conversion is linear in the low frequency region (usually the breaking point is set to 1,000Hz), and logarithmic in the high frequency region (after the breaking point). The default Mel scale in `librosa` is in the form of (14), but it is possible to change it to the form defined in (13) by setting the `htk` argument to `True`.

$$m = \begin{cases} \frac{3f}{200}, & \text{if } 0\text{Hz} \leq f \leq 1000\text{Hz} \\ \frac{3000}{200} + \frac{27\ln(f/1000)}{\ln 6.4}, & \text{if } f \geq 1000\text{Hz} \end{cases} \tag{14}$$

Once we have the frequency to Mel scale conversion, we can create Mel filter banks (for details on the computation of Mel filter banks, the readers may refer to [5]) that are multiplied to each timestep of the STFT result to obtain a Mel spectrogram [54]. An example of this conversion is shown in Figure 3, which depicts the STFT and Mel-scale spectrograms of a signal that starts with five pure tones at 25Hz, 75Hz, 150Hz, 400Hz, and 450Hz (shown in region A). After 0.25 seconds, three of the tones stop, leaving only 2 tones at 75Hz and 450Hz (shown in region B). After another 0.25 seconds, only the 75Hz tone remains (Region C), and finally, it ends with a single 450Hz tone (Region D). The STFT spectrogram is shown in the left-hand side of Figure 3. In this example, the window size for the STFT is 128 samples, which would generate a spectrogram with 128 frequency bins. The

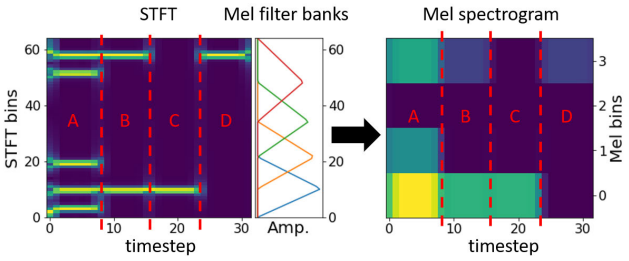

FIGURE 3. Mel spectrogram obtained by combining the STFT result (65 frequency bins) with 4 Mel filter banks.

complete spectrogram contains redundant information due to symmetry, therefore only 65 bins are used in the final STFT result. The hop size for STFT is 32 samples, which equals a quarter of the window size. To obtain a Mel spectrogram with four Mel bins, we need to have four Mel filter banks. The basis functions of a Mel filter bank are triangular in shape and the kernel that converts the raw STFT to the Mel-spectrogram by grouping multiple STFT bins to a single Mel bin.

The exact mapping for the example in Figure 3 is shown in Table 1. There are five frequency components in region A, the three frequency components corresponding to 25 Hz, 75 Hz, and 150 Hz will be mapped to Mel bin 0. Since the Mel filter banks are overlapping with each other, the frequency component 150 Hz will also be mapped to Mel bin 1, while the two high frequency components 400 Hz and 450Hz will only be mapped to Mel bin 3. Each timestep of the STFT is multiplied by the Mel filter banks matrix in the same way to obtain the Mel spectrogram.

nnAudio's implementation of the Mel spectrogram extraction from raw waveforms in `PyTorch` is relatively

TABLE 1. Mapping from frequency bins to Mel bins for the example in Figure 3. The bin indexing starts with 0.

Frequency bins	Corresponding frequencies	Mel bins (htk version)
$1-21$	0Hz to 168.4Hz	0
$11-34$	79.8Hz to 267.3Hz	1
$22-48$	168.4Hz to 377.4Hz	2
$35-64$	267.3Hz to 500Hz	3

straightforward. We obtain the STFT results using the `PyTorch` 1D convolutional neural network described in Section III-A, and then we use Mel filter banks obtained from `librosa`. The values of the Mel filter banks are used to initialize the weights of a single-layer fully-connected neural network. Each time step of the magnitude STFT is fed forward into this fully connected layer initialized with Mel weights. The Mel filter banks only need to be created once when initializing the neural network. These weights can be set as trainable or remain fixed, much like the neural network implementation of STFT as discussed in Section III-A. Figure 3-B shows the schematic diagram of our `PyTorch` implementation of the Mel spectrogram calculation.

nnAudio API nnAudio implements the Mel spectrogram layer as `Spectrogram.MelSpectrogram()`, with default arguments: sr = 22050, n_fft = 2048, n_mels = 128, hop_length = 512, window = 'hann', center = True, pad_mode = 'reflect', htk = False, fmin = 0.0, fmax = None, norm = 1, trainable_mel = False, trainable_STFT = False.

C. CONSTANT-Q TRANSFORM

1) A QUICK OVERVIEW OF THE CONSTANT-Q TRANSFORM (1992 VERSION)

There is a logarithmic relationship between the frequencies of musical pitches: The frequency of a musical pitch doubles for every octave. In order to effectively reflect the relationship between musical pitches on a spectrogram, it is helpful to use a logarithmic frequency scale. One naive solution is to modify the frequencies of the basis functions of the discrete Fourier transform so that the centre frequencies of the bins form a geometric series [55]. There are, however, numerous problems with this approach.

First, it is well-known that the standard DFT basis functions of length N form an orthogonal basis for the space of all complex vectors of length N. The orthogonality of the basis guarantees that the DFT is an energy-preserving transformation. In other words, the magnitude of the transformed output is exactly equal to the magnitude of the input. This is important because it means that we can determine the volume of the input signal simply by looking at the magnitude of the DFT output. If we modify the frequencies of the basis vectors, they become non-orthogonal and therefore the relationship between input and output energy becomes much more complicated.

A second consequence of using unevenly spaced basis vectors in the DFT is that at the upper end of the spectrum, where the vectors are farthest apart, there will be wide gaps between frequency bins. If we insist on using a set of only N vectors as the basis, these gaps are so wide that high frequency tones lying between bins will not be detected at all. The lack of frequency resolution in the high end can be remedied by increasing the number of basis vectors beyond N, but doing so leads to an excessive density on the low frequency end of the spectrum. Since the width of each bin is constant with respect to frequency, this results in significant overlap between bins in the low end. In frequency ranges with significant overlap between bins, the energy shown in the transformed output is exaggerated with respect to the actual energy in the input signal.

The challenges mentioned above are the motivation for the design of the constant-Q transform, first proposed by Brown in 1991 as a modification of the discrete Fourier transform [6] where the window size $N_{k_{cq}}$ scales inversely proportional to the centre frequency of the CQT bin k_{cq} to maintain a fixed number of cycles for sine and cosine within the window. Since the width of each bin is inversely proportional to the length of its basis vector, the width of each CQT frequency bin expands proportionally to the space between bins so that there are no gaps between bins at the upper end of the spectrum and no excessive overlap between bins at the lower end of the spectrum.

In signal processing, the letter Q [56], which stands for quality, indicates the centre frequency divided by the bandwidth of a filter. There are many types of filters for which the term bandwidth is applied and correspondingly there are various different definitions of the bandwidth and of Q. In the context of the CQT, Q is defined to be the number of cycles of oscillation in each basis vector. The corresponding equation for Q is shown in (15), where b is the number of bins per octave. Once Q is known, we can calculate the window size $N_{k_{cq}}$ for each bin k_{cq} by (16). The equation for CQT is very similar to the DFT, with the varying index k replaced by Q and fixed window size N replaced by varying window size $N_{k_{cq}}$ as shown in (17). Despite the fact that constant-Q transform (CQT) uses a similar concept as logarithmic frequency DFT, i.e., both of them have a logarithmic frequency scale, they are not the same. CQT maintains a constant frequency resolution by keeping a constant Q value while the logarithmic frequency STFT has a varying Q. The subtle differences between the CQT and logarithmic frequency scale STFT can be observed in Figure 12, and 13.

$$Q = (2^{\frac{1}{b}} - 1)^{-1} \tag{15}$$

$$N_{k_{cq}} = \text{ceil}\left(\frac{s}{f_{k_{cq}}}\right) Q \tag{16}$$

$$X^{cq}[k_{cq}] = \sum_{n=0}^{N_{k_{cq}}-1} x[n] \cdot e^{-2\pi i Q \frac{n}{N_{k_{cq}}}} \tag{17}$$

2) CQT USING NEURAL NETWORKS

The naive implementation of CQT consists of looping through all of the kernels one by one, and calculating the

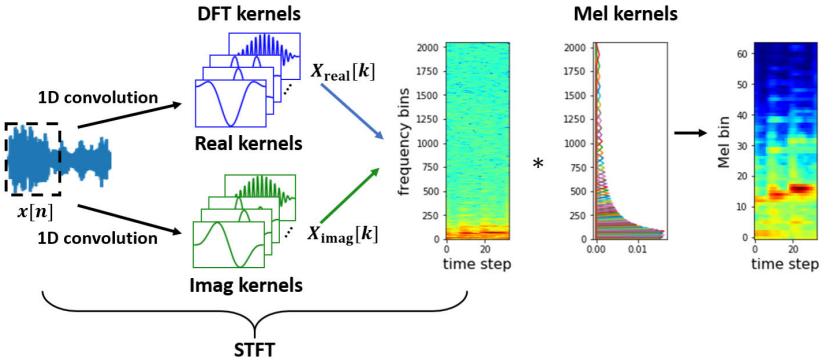

FIGURE 4. nnAudio's neural network-based implementation for Mel spectrograms. The STFT window size is 4,096 and the number of Mel filter banks is 64 in this example.

dot-product between the kernel $e^{-2\pi Q/N_k}$ and the input signal x [6]. This type of implementation, however, is not feasible for our 1D convolution approach. Most neural network frameworks only support a fixed kernel size across different channels for a 1D convolutional neural network. This means that if we have 84 CQT kernels, we would need 84 convolutional networks to include all of the kernels.

Youngberg and Boll [57] first proposed the concept of CQT in 1978. Brown later proposed an efficient way to calculate CQT in 1992 [7]. The trick is to use Parseval's equation [45] as shown in (18), where $a[n]$ and $b[n]$ are arbitrary functions in the time domain, and $A[k]$ and $B[k]$ are the frequency domain versions of $a[n]$ and $b[n]$, respectively. If we define $X[k]$ and $Y[k]$ as the DFT of input $x[n]$ and kernel $e^{-2\pi Q/N_{kcq}}$, respectively, then this approach converts both $x[n]$ and $e^{-2\pi iQ\frac{n}{N_{kcq}}}$ to $X[k]$ and $Y[k]$, respectively, in the frequency domain, and subsequently multiplies them together to get the approximated CQT as shown in (19). It should be noted that both $X[k]$ and $Y[k]$ are matrices containing complex numbers, and N is the longest window size for the CQT kernels, which is equal to the length of the kernel with the lowest frequency. Also, $Y[k]$ is a sparse matrix in this case. Figure 5 shows an example of the CQT kernels in the time domain and frequency domain respectively. The bottom and top kernels correspond to the musical notes A3 (220Hz) and A7 (3520Hz) respectively, with 12 bins per octave and a sampling rate of 8000Hz. There are 60 bins in total. Only the real components for the kernels are shown in Figure 5, but readers should note that $y[n, k_{cq}]$ is a matrix with complex numbers, and each row of $y[n, k_{cq}]$ is transformed to a row of $Y[k, k_{cq}]$ by using the Fast Fourier Transform (FFT). Therefore, the frequency domain CQT are also matrices of complex numbers and the magnitude CQT can be obtained by taking element-wise norm.

$$\sum_{n=0}^{N_k-1} a[n]b[n] = \frac{1}{N}\sum_{k=0}^{N-1} A[k]B[k] \tag{18}$$

$$X^{cq}[k_{cq}] = \sum_{n=0}^{N_{kcq}-1} x[n]\cdot e^{-2\pi iQ\frac{n}{N_{kcq}}} = \frac{1}{N}\sum_{k=0}^{N-1} X[k]Y[k, k_{cq}] \tag{19}$$

Using the definition of CQT from Brown *et al.*, the conversion from the time domain input $x[n]$ to $X[k]$ can be done with a 1D convolutional neural network. The DFT basis vectors will be the kernels for the neural network. Since there is a real part and an imaginary part to the DFT kernels, we need two 1D convolutional neural networks, one network for the real component of the kernels, and another network for the imaginary component. We can perform the DFT using the same procedure described in Section III-A for the STFT. Next, each time step of the STFT result $X[k]$ is multiplied with the same CQT kernels $Y[k, k_{cq}]$. Therefore, the CQT kernels only need to be created once as part of the initialization for the STFT 1D convolutional nerual network. In the end we obtain a CQT matrix $X^{cq}[k_{cq}]$ with real and imaginary parts and the final CQT ouput is calculated using the element-wise magnitude abs$X^{cq}[k_{cq}]$.

Unfortunately there is a major flaw in this approach. If the number of octaves is large and the CQT kernels start at a low frequency, the size of CQT kernels will be huge. For example, if we want to cover 88 notes (from A0 to C8 as the range for a piano) with a sampling rate of 44100Hz and 24 bins per octave, then the longest time domain CQT kernel window size is 54,727, according to (16). When rounding this up to the next power of 2, the window size will be 65,536, assuming that we want the FFT length to be a power of 2. Even though the FFT has not been implemented in nnAudio, we will still follow these recommendations for existing CQT implementations so that we can directly compare them with our implementation. By transforming time domain CQT kernels to frequency domain kernels, we discard half of the kernel length due to symmetry. Therefore, the longest frequency domain CQT kernel has a length of 32,768. With 88 piano keys and 24 bins per octave, the CQT kernels would have a shape of (176, 32768). This also implies that the window

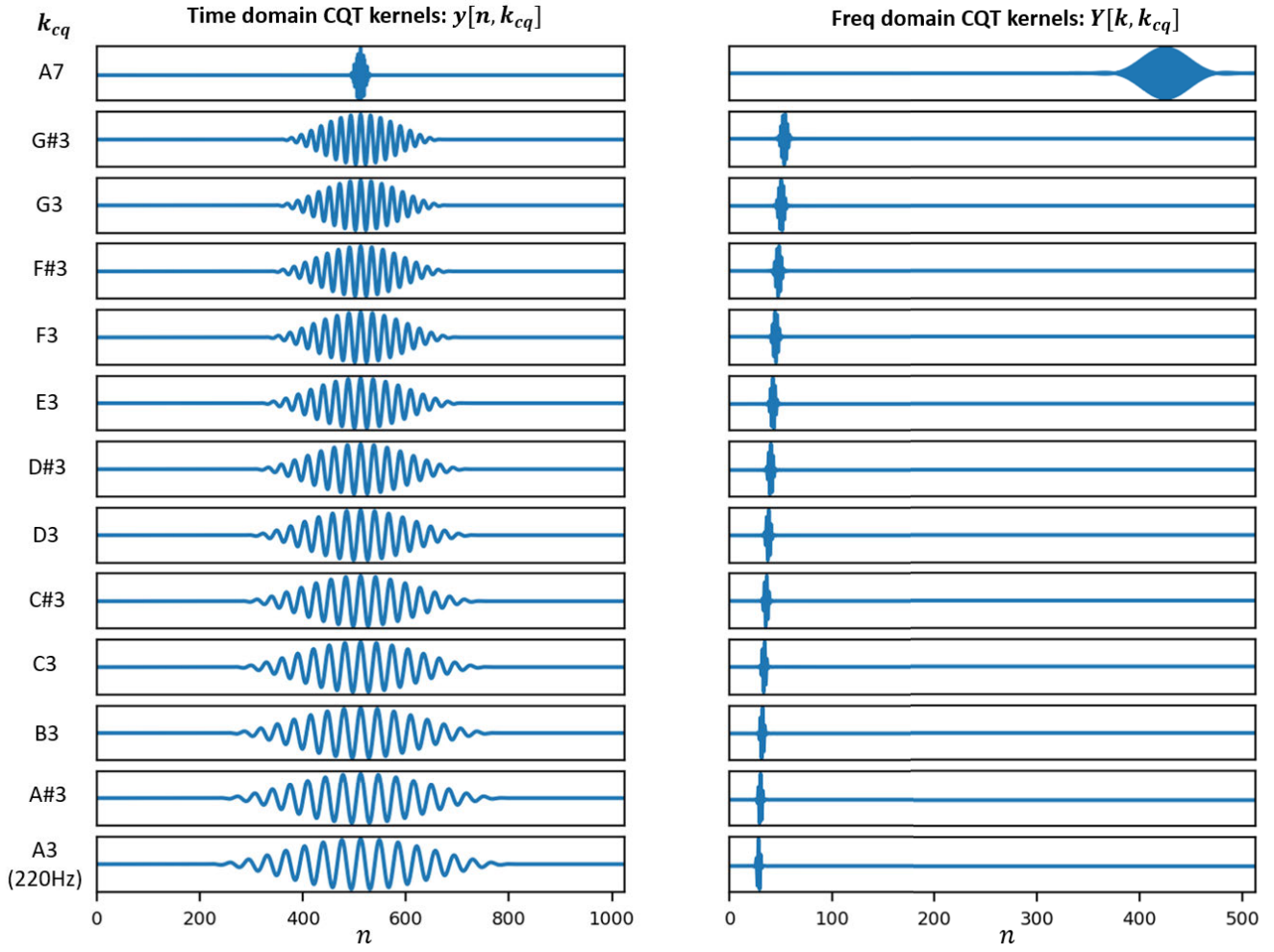

FIGURE 5. An example of CQT kernels whereby the number of bins per octave is set to 12. The x-axis shows the time in digital samples (n). Each CQT kernel has a frequency that corresponds to a musical pitch. Only the real components of $y[n, k_{cq}]$ and $Y[k, k_{cq}]$ are shown here.

size for the STFT would be 32,768, which is extremely long, making this implementation inefficient for huge CQT kernels that have a low frequency. In the following sections, we will discuss how to implement a more efficient version of CQT by using a downsampling method [6].

nnAudio API Despite its inefficiency, we still provide this function for research purposes. It can be executed in nnAudio via the function `Spectrogram.CQT1992`, with default arguments: sr = 22050, hop_length = 512, fmin = 220, fmax = None, n_bins = 84, bins_per_octave = 12, norm = 1, window = 'hann', center = True, pad_mode = 'reflect', device = "cuda:0".

3) DOWNSAMPLING

Next, we will discuss how to do downsampling with a neural network before moving on to the downsampling method used in the computation of the CQT. In order to downsample input audio clips by a factor of two without aliasing, a low pass filter is required so that any frequencies above the downsampled Nyquist frequency will be filtered out first, before performing the actual downsampling. This is performed using a technique called Finite impulse response filtering (FIR). FIR refers to the convolution of an input signal with a filter kernel using the same formula shown earlier in (9). This type of filtering can be implemented efficiently using a convolutional neural network. The definition of FIR is shown in (20), where $x[n - i]$ is the input signal at time step n, b_i is the FIR filter.

To downsample, we first design the low-pass FIR filter kernel using the Window Method [58], which is implemented in `SciPy` as the function `scipy.signal.firwin`. To achieve a steep cutoff at the Nyquist frequency we set the passband of the filter to end at 0.4995 and the stopband to start at 0.5005 times the Nyquist frequency. These values were chosen so as to achieve a steep cutoff. The impulse response and frequency response of our antialiasing filter is shown in Figure 7. This filter is used as the kernel of the downsampling component of our convolutional neural network. An effective antialiasing filter design is important for the CQT implementation, which we explain in the following

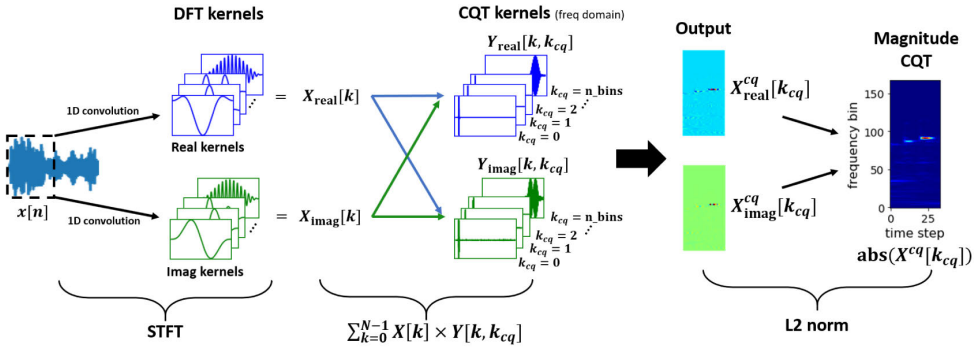

FIGURE 6. nnAudio's implementation of the 1992 version of CQT [7] using a 1D convolutional neural network. The DFT kernels and CQT kernels only need to be initialized once and can be reused.

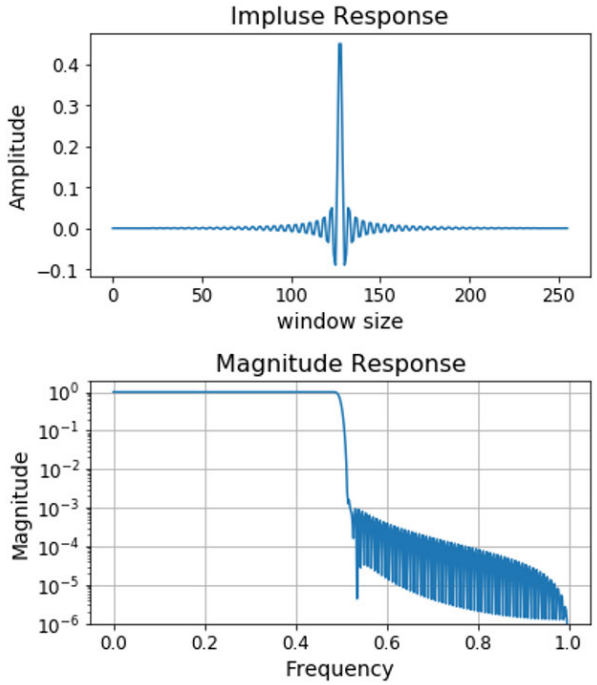

FIGURE 7. Impulse response and magnitude frequency response for the antialiasing filter. This filter forms the kernel for the 1D convolutional neural network that performs downsampling in nnAudio's CQT computation.

section.

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{20}$$

4) CONSTANT-Q TRANSFORM (2010 VERSION)

The constant-Q transform uses basis vectors of varying lengths. The basis kernels for the lowest frequencies are several orders of magnitude longer than the high frequency kernels. Since low frequency audio signals can be accurately represented with lower sample rates, we can compute the lower frequency components of the CQT more efficiently by downsampling the input and using correspondingly shorter filter kernels. This technique is described in detail in [6], [59]. Only one octave of CQT kernels is created using this approach. These CQT kernels usually start from the highest octave due to the short window size as described in (16). By doing so, the computational complexity can be reduced. When applying the CQT kernels (of this highest octave) to the frequency domain input $X[k]$, only the CQT result for the highest octave is obtained. After that, we downsample the input by a factor of two and apply the same CQT kernels to this new input to obtain the CQT result for the next octave down. The same process is repeated until the desired number of octaves is processed. In this approach, the CQT kernels are kept the same while the input audio is being downsampled recursively. By referring to (16), $N_{k_{cq}}$ and Q are constant. When we downsample the audio by a factor of 2, s is reduced by half. In order to keep $N_{k_{cq}}$ and Q constant, $f_{k_{cq}}$ must also be reduced by half. Physically, it means the CQT output obtained by same CQT kernels relative to a downsampled audio with factor 2^{α} is α octave lower than the original audio, where $\alpha \in [1, 2, 3, \ldots]$ is a positive integer. Figure 8 shows the schematic diagram for this implementation. Each downsampled input $x_{\alpha}[n]$ produces the CQT result for one octave. The complete CQT result can then be obtained by appending the results for each of the octaves together.

nnAudio API This algorithm can be executed in nnAudio via the function `Spectrogram.CQT2010`, with default arguments: sr = 22050, hop_length = 512, fmin = 32.70, fmax = None, n_bins = 84, bins_per_octave = 12, norm = True, basis_norm = 1, window = 'hann', pad_mode = 'reflect', earlydownsample = True, device = 'cuda:0'.

5) CQT WITH TIME DOMAIN KERNELS

When Brown and Puckette [7] proposed their more efficient algorithm in 1992, they were facing limitations in computer memory. The time domain CQT kernels form a very large, dense matrix. Storing a matrix like this requires a lot of memory. When converting time domain CQT kernels into frequency domain kernels, the dense matrix becomes a sparse matrix. Storing this sparse matrix using either the compressed

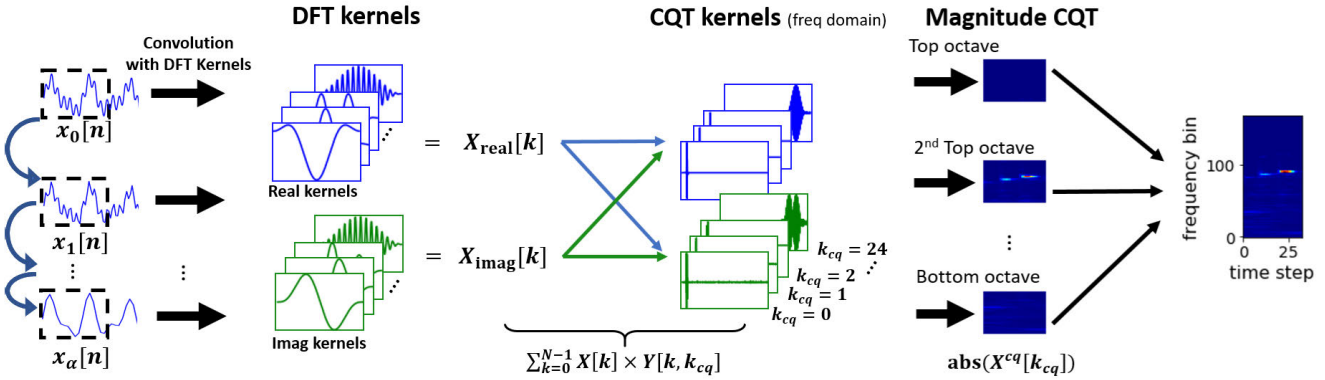

FIGURE 8. Schematic diagram of the 2010 version of CQT [6], [59] using the recursive downsampling method. The kernels only need to be initialized once and can be reused over and over again.

sparse row (CSR) format or the compressed sparse column (CSC) algorithm is more memory efficient than storing a dense matrix. Therefore, by converting the time domain CQT kernels to the frequency domain, the same information is retained, while requiring less memory to store it.

With modern technology, memory is no longer an issue. Thus, it is no longer necessary to convert the time domain CQT kernels to frequency domain kernels. By not doing this conversion, we remove a computationally heavy step, thus improving the CQT computation speed. Both the 1992 version of CQT and the 2010 version of CQT can benefit from this modification. The resulting modified implementation is shown in Figure 9 and 10. The improvement in computational speed is reported as CQT1992v2 and CQT2010v2, respectively, for each algorithm in Figure 11. The differences between the existing CQT algorithm and our proposed modification based on the down-sampling approach are shown in Figure 8 and Figure 10. For the differences between the existing CQT algorithm and our proposed modification based on the Brown and Puckette [7] approach, readers can refer to Figure 6 and Figure 9.

nnAudio API The improved version of CQT1992 is implemented as `Spectrogram.CQT1992v2()` with the default parameters: sr = 22050, hop_length = 512, fmin = 32.70, fmax = None, n_bins = 84, bins_per_octave = 12, norm = 1, window = 'hann', center = True, pad_mode = 'reflect', trainable = False, output_format = 'Magnitude', device = 'cuda:0'.

The improved version of CQT2010 can be executed in nnAudio via the function `Spectrogram.CQT2010v2()` with default parameters: sr = 22050, hop_length = 512, fmin = 32.70, fmax = None, n_bins = 84, bins_per_octave = 12, norm = True, basis_norm = 1, window = 'hann', pad_mode = 'reflect', earlydownsample = True, device = 'cuda:0'.

IV. EXPERIMENTAL RESULTS

In this section, we analyse the speed and the accuracy of the proposed framework, `nnAudio`. We compare our `PyTorch` implementation, nnAudio, with the existing audio processing library `librosa` [23]. More specifically, our STFT implementation is compared to `librosa.stft`, Mel Spectrogram to `librosa.feature.melspectrogram`, and CQT to `librosa.cqt`. We also compare our proposed framework to other existing GPU audio processing libraries such as `Kapre` and `torchaudio`. In the first subsection, we compare the speed required to process 1,770 audio files in .wav format. In the second subsection, we focus on testing the correctness of the resulting spectrograms. In what

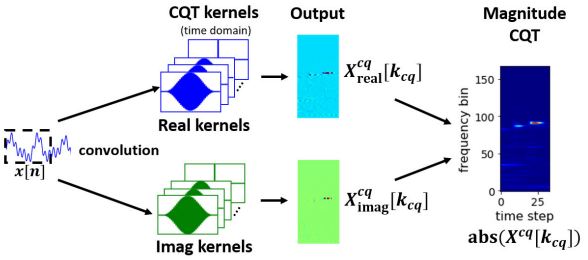

FIGURE 9. A schematic diagram showing our proposed improvement of the CQT1992 algorithm that uses time domain CQT kernels instead of frequency domain kernels, which requires less computational steps compared to the original algorithm as shown in Figure 6.

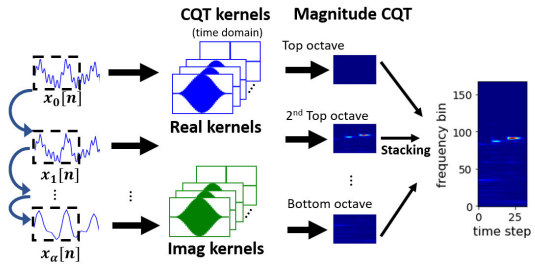

FIGURE 10. A schematic diagram showing our proposed improvement of the CQT2010 algorithm that uses only time domain CQT kernels. Note that the output of the convolution between the audio input and the CQT kernels is still a complex number, even though this is not shown in the figure for simplicity.

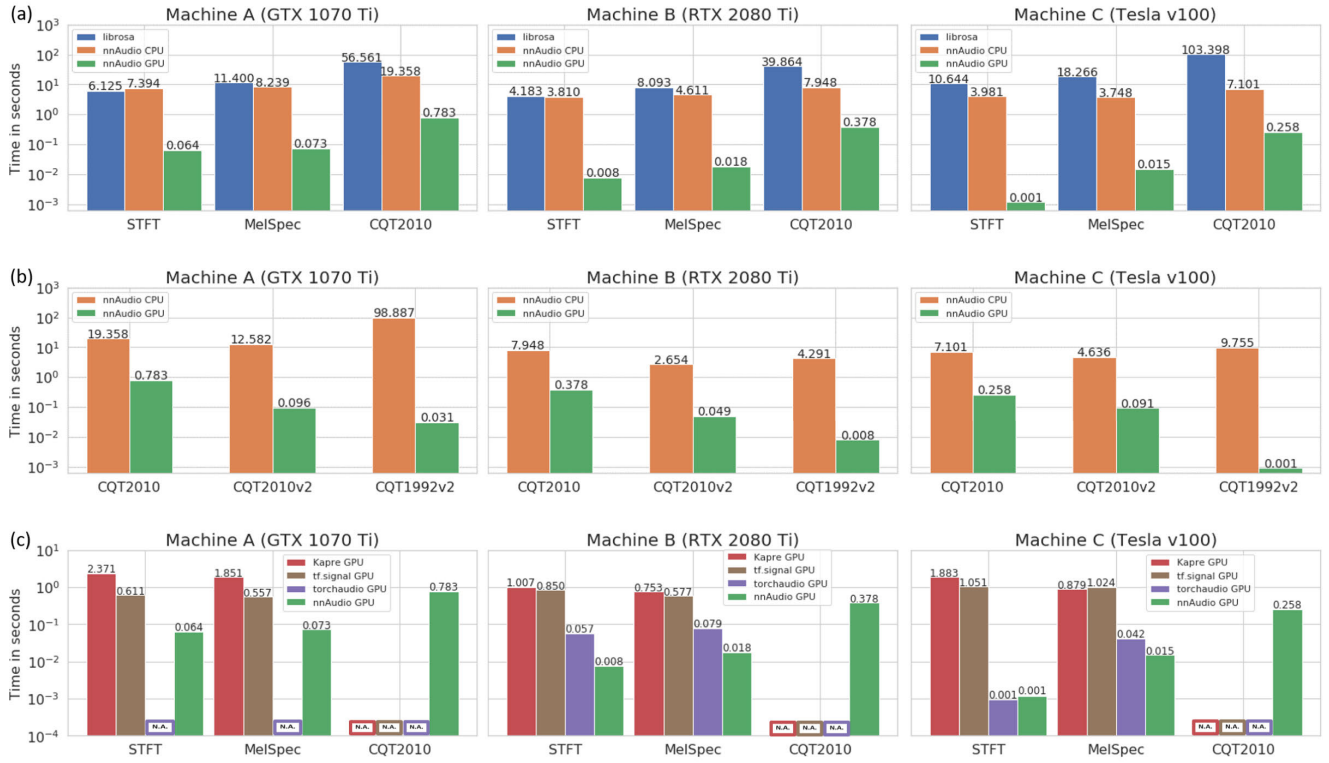

FIGURE 11. (a): Processing times to compute different types of spectrograms with nnAudio GPU, nnAudio CPU, and librosa. (b): Processing times for different versions of CQT. (c): Processing times for different GPU processing libraries: Kapre, tensorflow, torchaudio, and nnAudio.

follows, the different implementations for CQT, namely CQT1992v2 and CQT2010v2, will be discussed individually. These are the implementations that directly use time domain CQT kernels as mentioned in Section III-C5. For the sake of easy reference, the Mel spectrogram will be referred to as MelSpec below.

A. SPEED

1) SETUP

We use the MAPS dataset [60] to benchmark nnAudio. A total of 1,770 .wav files from the `AkPnBcht/UCHO/` folder were used for the benchmark. We discard the first 20,000 samples (which is equivalent to 0.45 seconds under the 44.1kHz sampling rate) from each audio excerpt in order to remove the silence. Each audio excerpt is kept the same length (80,000 samples) by removing the excessive samples in the end. Their final length is equivalent to 1.8 seconds. The audio excerpts are stored as an array with shape $1,770 \times 80,000$. The goal of the speed test is to convert this array of waveforms into an array of spectrograms while maintaining the order of the audio excerpts. We conducted this test on three different machines:

(A) A Windows Desktop with CPU: Intel Core i7-8700 @ 3.20GHz and GeForce GTX 1070 Ti 8Gb GPU
(B) A Linux Desktop with CPU: AMD Ryzen 7 PRO 3700 and 1 GeForce RTX 2080 Ti 11Gb GPU
(C) A DGX station with CPU: Intel Xeon E5-2698 v4 @ 2.20GHz and Tesla v100 32Gb GPU

During the test, we compared the speed of our proposed nnAudio toolkit to one of the popular signal processing libraries, `librosa` [23]. Although `Essentia` is reported to be faster than `librosa` in terms of audio processing speed [5], our experimental results show that `Essentia` is slower than `librosa`. (It takes `Essentia` 30 seconds to finish the STFT task and 180 seconds to finish the CQT task on machine C.) One possible reason is that `Essentia` only supports the versions of STFT and CQT without a moving window. Therefore it can only produce spectrums, not spectrograms. To obtain the spectrograms, we first need to cut the input audio into small audio segments and then apply the CQT or STFT on each of these segments. This is done using extra nested `for` loops, which could cause a slower speed in `Essentia`. On top of that, `Essentia` does not support MelSpec, making a side-by-side comparison to nnAudio impossible. We therefore report the results for `nnAudio` and `librosa` in Figure 11.

Because our task is to transform an array of waveforms to an array of spectrograms (in the same order), `librosa` with multi-process will not work well. The time it takes to finish this task while maintaining the same order for the output array as the original array using multiprocessing is longer than when a plain sequential **for** loop is used. Therefore, the speed test for `librosa` is performed by using a `for`

loop. Furthermore, the performance for `librosa` can be optimized by using caching, but this option is disabled by default. To emulate the situation in which most people use `librosa`, we run the speed test with caching disabled. Even when caching is used, it only reduces the computation time for CQT by around 10 seconds. As for nnAudio, despite the fact that multiple GPUs are available on machine C, only one GPU is used to convert the array of waveforms to the array of spectrograms to ensure a fair comparison with other machines. Since `PyTorch` can also run on a CPU, we will also test this configuration of nnAudio. Finally, the computation time of nnAudio is compared with other GPU-based processing libraries: `Kapre` and `torchaudio`.

2) RESULTS

Figure 11(a) shows the time taken to convert an array of 1,770 waveforms to an array of 1,770 spectrograms using Mel frequency scale, STFT, and CQT on three different machines. It is clear from the figure that our newly proposed method is faster than `librosa` regardless of which machine it is run on. Interestingly, machine A and B (normal desktops) are faster than machine C (DGX station) when running the test on CPU. This is possibly due to the fact that the CPU clock rates for machine A and B are much higher than machine C. Nevertheless, using a GPU reduces the time taken to finish the spectrogram extractions, whereby the performance of the RTX 2080 Ti GPU is similar to the high end Tesla v100 GPU. We should note that, when using `PyTorch` with GPU, extra time is required to transfer the kernels from RAM to GPU memory, which only takes a few seconds. This process can be considered as part of the model initialization. The time required to initialize the models is not included in Figure 11. Table 2 shows the time taken to initialize each neural network model with nnAudio. This time is influenced by the kernel sizes of STFT, MelSpec, and CQT. For STFT, a longer window size (n_fft) results in larger STFT kernels. The same goes for MelSpec and CQT. More time is required to transfer larger kernels to GPU memory. In our experiment, an STFT window size of 4,096 is used for both STFT and MelSpec. For MelSpec, a total of 512 Mel filter banks are used. For the different implementations of CQT, the kernels start at 32.7Hz, which corresponds to the note C1, and 24 bins per octave, covering 176 bins in total. The neural network models used by nnAudio to calculate MelSpec and CQT require operations with multiple kernels (an initial DFT kernel followed by a model-specific kernel), therefore, we break the initialization time down into two steps (columns 2 and 3 in Table 2). Model kernels refer to the convolution kernels specific to each spectrogram type. For MelSpec, the model kernels are the Mel filter banks. For CQT, they consist of both DFT and CQT kernels. The initialization of the kernels of the network only needs to be performed once. As we can observe from Table 2, CQT2010 has a much faster initialization time compared to CQT1992 (5 seconds compared to over 200 seconds). This can be explained as the bottleneck for CQT1992 in the STFT stage. If the starting frequency is too low, the CQT kernels become very long, which in turn causes a huge window size (n_fft) for STFT. In the CQT setting used for the kernel initialization speed test (sampling rate = 44, 100Hz, minimum frequency = 32.7Hz, bins per octaves = 24, and bins = 176), the longest CQT kernel is 46,020, which results in a n_fft of 65,536 (rounding up the to nearest power of two, 2^{16}). To mitigate this problem, a Fast Fourier Transform (FFT) may be used instead of DFT, which will be explored in future research. Another way to prevent this problem would be to use the implementation mentioned in III-C5. Once everything is loaded into the GPU memory, the computation will occur at the speed as shown in Figure 11 (a) and Figure 11 (b). Even when only a CPU is used, nnAudio still outperforms `librosa` and `Essentia` significantly.

As mentioned in Section III-C5, converting the time domain CQT kernels to frequency domain CQT kernels is not necessary if there is enough computing memory. In the experiment, we compare the improvement in computation speed when using the time domain CQT kernels directly. Figure 11 (b) shows how the improved constant-Q Transform (CQT1992v2) and the improved constant-Q Transform with downsampling (CQT2010v2) further improve the computation speed. CQT2010v2 is faster than CQT2010 regardless of whether the CPU or GPU is used. While CQT1992v2 is extremely fast when GPU is used, the CPU version is slower than the CQT2010. Therefore, CQT2010v2 should be used in a computer without GPU, and CQT1992v2 should be used when GPU is available. However, there are subtle differences between the 1992 and 2010 implementation, and under normal circumstances, CQT1992v2 is the best option among all the implementations. The subtle differences between various CQT implementations will be discussed in detail in the following subsection. Nevertheless, to ensure

TABLE 2. The GPU initialization time needed for the two kernels (STFT and model-specific kernel) in each nnAudio neural network model, together with the required memory.

Model	DFT kernels (in s)	Model kernels (in s)	Memory (in MiB)
STFT: n_fft=4096	5.5 ± 0.1	N.A.	1135
MelSpec: n_fft=4096, n_mels=512	5.5 ± 0.04	0.02 ± 0.004	1155
CQT1992: bins_p_oct=24, bins=176	194.9 ± 1.0	5.6 ± 0.08	17505
CQT1992v2: bins_p_oct=24, bins=176	N.A.	4.9 ± 0.08	1157
CQT2010: bins_p_oct=24, bins=176	0.05 ± 0.03	4.8 ± 0.08	1177
CQT2010v2: bins_p_oct=24, bins=176	N.A	4.6 ± 0.05	1089

flexibility, nnAudio provides all implementations discussed above (CQT1992, CQT2010, CQT1992v2, CQT2010v2).

Finally, the speeds of different GPU-based audio processing libraries (Kapre, tensorflow, and torchaudio) are compared with nnAudio on the same task. The results are shown in Figure 11(c). Since torchaudio does not support the Windows operating system, the result is not available for machine A. For STFT, torchaudio is marginally faster than nnAudio on machine C, but nnAudio is always faster than torchaudio on machine B. nnAudio outperforms Kapre on all three machines. For MelSpec, nnAudio is at least three times faster than torchaudio, and at least 40 times faster than Kapre. There is no comparison available for CQT because Kapre and torchaudio do not have this functionality. Kapre is the slowest among the three libraries. This is most likely due to the fact that it directly takes a numpy array as the input which causes slower performance than when a PyTorch tensor is used (like in torchaudio and nnAudio). The speed of tensorflow is slightly faster than Kapre but it is much slower than torchaudio and nnAudio. This result makes sense because Kapre is based on tensorflow, and both torchaudio and nnAudio are based on PyTorch; therefore, the speed of Kapre should be similar to tensorflow whereas the speed of torchaudio should be similar to nnAudio. Although NVIDIA DALI also includes GPU audio to spectrogram processing in its recent releases (since version 0.17.0), its main purpose is for GPU data loading [61], [62]. In other words, unlike the libraries we compared here, NVIDIA DALI is not differentiable (it does not calculate the gradient for the spectrograms). For example, if we have waveforms generated from a neural network, and we want to convert these synthesized waveforms to spectrograms and calculate the loss with respect to ground truth spectrograms, only the libraries we included in Figure 11 support backpropagation all the way back to the synthesized layers. Since one of the main features of nnAudio is to offer differentiable/trainable spectrograms, we do not include NVIDIA DALI (a GPU data loading tool) in our results as this feature is not supported.

B. CONVERSION OUTPUT

1) SETUP

We use librosa as our benchmark to check the correctness of our implementation. The spectrograms produced by our implementation are compared to the librosa output by using the numpy function np.allclose. Four input signals, a linear sine sweep, a logarithmic sine sweep, an impulse tone, and a chromatic scaled played on a piano, are used in this study to determine the model output correctness. The chromatic piano scale is recorded with a piano instrument provided by Garritan Personal Orchestra 5[3] and saved as a .wav file. Because adapting the time domain CQT kernels does not change the output spectrogram, the result for CQT1992 is the same as that for CQT1992v2, and is better than the results for CQT2010 and CQT2010v2. Therefore we will only report the results for the faster and better quality implementation (CQT1992v2) here.

[3]https://www.garritan.com/products/personal-orchestra-5/

2) RESULTS

The results of the accuracy test are shown in Figures 12 and 13. The output magnitudes are displayed in a logarithmic scale so that the subtle differences can be observed easily. When looking at the results, we notice that the STFT results from librosa and nnAudio are very similar to each other with an absolute tolerance of 10^{-2} and a relative tolerance of 10^{-2}. The same can be said for MelSpec, for which the results of both libraries are very similar, with an absolute tolerance of 10^{-3} and a relative tolerance of 10^{-4}. For CQT, the absolute tolerance is 0.8 and the relative tolerance is 2. The CQT output for nnAudio is smoother because we are using the CQT1992v2 approach in our implementation for which downsampling is not required. Figure 14 shows the comparison between our proposed CQT1992v2, our CQT2010v2 and librosa's implementation of CQT (librosa uses the 2010 downsampling algorithm). In the implementation of both librosaand CQT2010v2, the aliasing in the figure is due to downsampling. Although the magnitude of the aliasing is negligible, it is still observable when we use a logarithmic magnitude scale. Further study is required to determine the effects of the aliasing due to downsampling in the neural network models. The CQT1992v2 model, however, is the fastest of all proposed GPU-based CQT implementations (see Figure 11(b)), and its output is the best among the different implementations. Therefore CQT1992v2 should be used, and hence it is set as the default CQT computation algorithm for nnAudio.

V. EXAMPLE APPLICATIONS

In this section, two potential applications of nnAudio will be discussed. First, we will elaborate on using nnAudio to explore different spectrograms as the input for a music transcription model, and discuss how this process can benefit from on-the-fly GPU processing. Second, we will demonstrate that nnAudio allows the STFT kernels to be trained/finetuned, so that a better spectrogram can be obtained.

A. EXPLORING DIFFERENT INPUT REPRESENTATIONS

In this section, we discuss one possible application of this work, namely music transcription [64], [65]. We will show that with nnAudio, one can quickly explore different types of spectrograms as the input for a neural network and easily choose the spectrogram that yields the best transcription accuracy.

Consider the following scenario: we want to do polyphonic music transcription, and we have picked a specific model (fully connected neural network) to tackle this task, but we want to know which input spectrogram representation would yield the best results for this task [66]. In our experiment, a total of four types of spectrograms are explored:

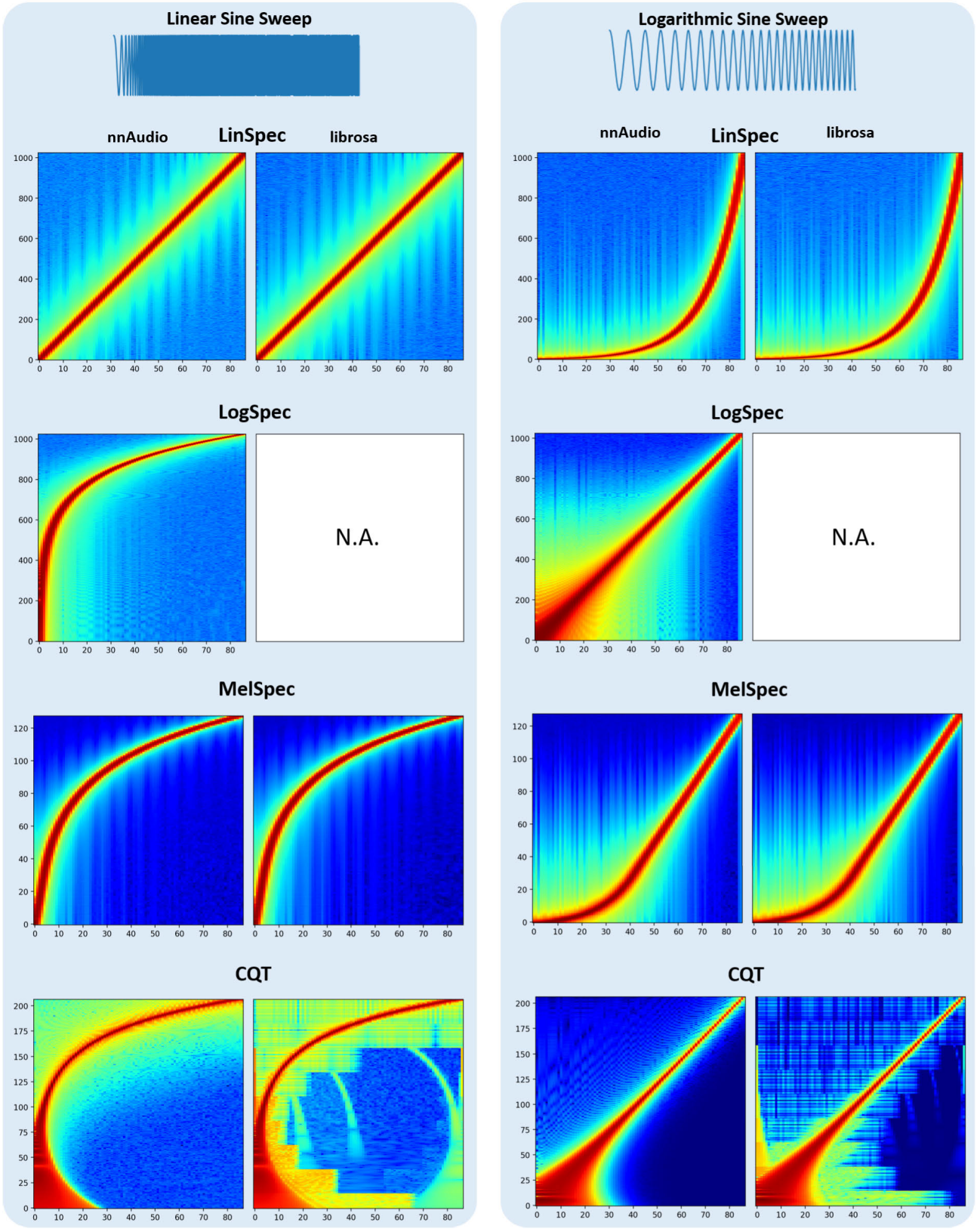

FIGURE 12. Comparing the output of `nnAudio` and `librosa` when converting a linear and logarithmic sine sweep.

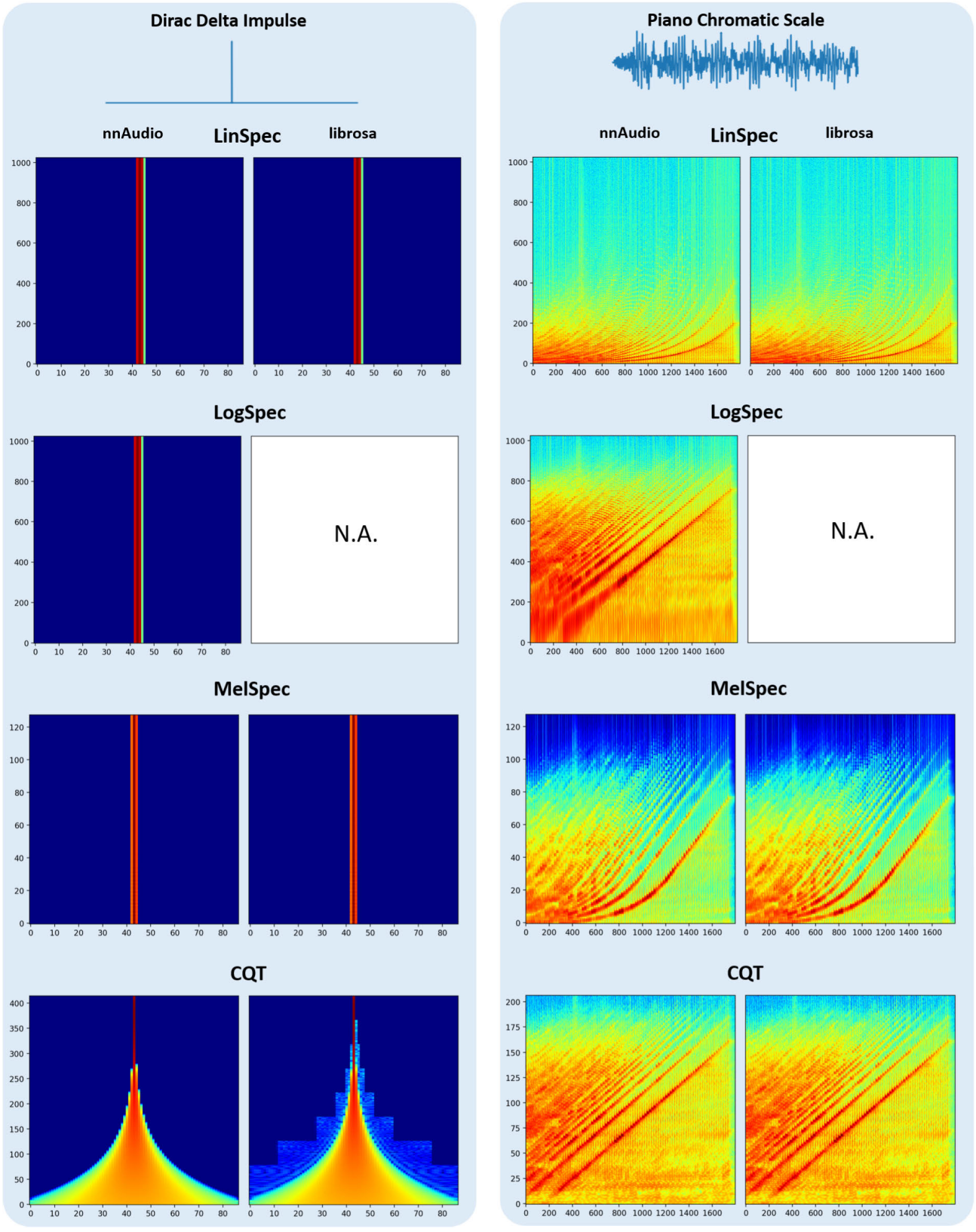

FIGURE 13. Comparing the output of `nnAudio` and `librosa` when converting an impulse tone and a chromatic piano scale.

TABLE 3. Table 3 shows a comparison of the computing time (on the three different machines as before), together with the hard disk space needed to store spectrograms when using traditional data pre-processing with `librosa`on the MusicNet dataset. Different settings for the spectrogram calculation are explored: For CQT, the parameters are (bins, number of bins per octave, hop size); for LinSpec and LogSpec, the parameters are (n_fft, hop size); for Mel spectrograms, the parameters are (n_fft, n_mels, hop size). Note that `librosa`does not provide the spectrogram type of LogSpec, we estimate that the time taken to calculate LogSpec is the same as its LinSpec counterpart. The size occupied by LogSpec is the same as LinSpec, as they have the same output array size given the set parameters. Using nnAudio, we can avoid these long processing times and the huge storage requirements by calculating the spectrograms on-the-fly. The numbers in parentheses in the final column indicate the time taken for nnAudio to finish the same task.

Index	Input type	Parameters	Size GB	Time A min:sec	Time B min:sec	Time C min:sec
1	CQT	(84, 12; 512)	6.6	30:57 *(5:29)*	24:15 *(0:35)*	41:46 *(0:35)*
2	CQT	(84*2, 12*2; 512)	13.1	36:37 *(10:14)*	27:14 *(1:16)*	50:01 *(2:11)*
3	CQT	(84*3, 12*3; 512)	19.7	44:26 *(11:35)*	32:23 *(2:40)*	67:03 *(2:25)*
4	CQT	(84*4, 12*4; 512)	26.2	45:00 *(14:21)*	32:52 *(6:26)*	68:01 *(3:30)*
5	CQT	(84*5, 12*5; 512)	32.8	45:33 *(16:43)*	33:18 *(7:00)*	72:39 *(4:39)*
6	CQT	(84*6, 12*6; 512)	39.3	64:05 *(21:21)*	43:24 *(13:13)*	101:18 *(12:52)*
7	CQT	(84*7, 12*7; 512)	45.9	64:42 *(33:45)*	43:21 *(21:49)*	102:59 *(13:19)*
8	CQT	(84*8, 12*8; 512)	52.4	65:22 *(36:30)*	44:07 *(22:44)*	104:14 *(16:09)*
9	CQT	(84*9, 12*9; 512)	58.9	65:55 *(37:36)*	44:54 *(27:24)*	105:13 *(16:55)*
10	CQT	(84*10, 12*10; 512)	65.5	66:36 *(41:30)*	45:24 *(32:41)*	106:22 *(19:17)*
11	LinSpec	(256, 512)	5.1	1:02 *(0:26)*	0:31 *(0:8)*	1:39 *(0:08)*
12	LinSpec	(512, 512)	10.1	1:33 *(0:28)*	0:51 *(0:08)*	2:33 *(0:08)*
13	LinSpec	(1024, 512)	20	2:39 *(0:37)*	1:31 *(0:08)*	4:16 *(0:08)*
14	LinSpec	(2048, 512)	40	4:56 *(1:07)*	2:54 *(0:15)*	7:34 *(0:13)*
15	LinSpec	(4096, 512)	79.9	9:17 *(2:37)*	5:46 *(0:45)*	15:30 *(0:39)*
16	LogSpec	(256, 512)	5.1	1:02 *(0:26)*	0:31 *(0:08)*	1:39 *(0:08)*
17	LogSpec	(512, 512)	10.1	1:33 *(0:28)*	0:51 *(0:08)*	2:33 *(0:08)*
18	LogSpec	(1024, 512)	20	2:39 *(0:37)*	1:31 *(0:08)*	4:16 *(0:08)*
19	LogSpec	(2048, 512)	40	4:56 *(1:07)*	2:54 *(0:15)*	7:34 *(0:13)*
20	LogSpec	(4096, 512)	79.9	9:17 *(2:37)*	5:46 *(0:46)*	15:30 *(0:39)*
21	MelSpec	(512, 128; 512)	5	1:36 *(0:30)*	0:55 *(0:08)*	2:31 *(0:10)*
22	MelSpec	(512, 256; 512)	10	1:44 *(0:30)*	1:01 *(0:08)*	2:43 *(0:10)*
23	MelSpec	(1024, 128; 512)	5	2:34 *(0:39)*	1:34 *(0:09)*	4:25 *(0:10)*
24	MelSpec	(1024, 256; 512)	10	2:42 *(0:40)*	1:43 *(0:09)*	4:23 *(0:10)*
25	MelSpec	(1024, 512; 512)	20	3:01 *(0:42)*	2:00 *(0:09)*	4:43 *(0:10)*
26	MelSpec	(2048, 128; 512)	5	4:40 *(1:10)*	2:58 *(0:16)*	8:26 *(0:14)*
27	MelSpec	(2048, 256; 512)	10	4:49 *(1:11)*	3:13 *(0:16)*	8:36 *(0:14)*
28	MelSpec	(2048, 512; 512)	20	5:14 *(1:12)*	3:42 *(0:17)*	9:02 *(0:14)*
29	MelSpec	(2048, 1024; 512)	39.9	6:04 *(1:16)*	4:42 *(0:18)*	9:53 *(0:15)*
30	MelSpec	(4096, 128; 512)	5	9:05 *(2:41)*	5:49 *(0:48)*	8:17 *(0:40)*
31	MelSpec	(4096, 256; 512)	10	9:21 *(2:43)*	6:13 *(0:48)*	8:27 *(0:41)*
32	MelSpec	(4096, 512; 512)	20	9:55 *(2:45)*	7:05 *(0:49)*	8:56 *(0:41)*
33	MelSpec	(4096, 1024; 512)	39.9	11:04 *(2:51)*	8:50 *(0:51)*	9:44 *(0:43)*
34	MelSpec	(4096, 2048; 512)	80	13:08 *(3:01)*	12:23 *(0:55)*	10:14 *(0:47)*
			Total space 950.4GB	**Total time** 633:51 *(262:11)*	**Total time** 445:07 *(145:01)*	**Total time** 983:00 *(99.43)*

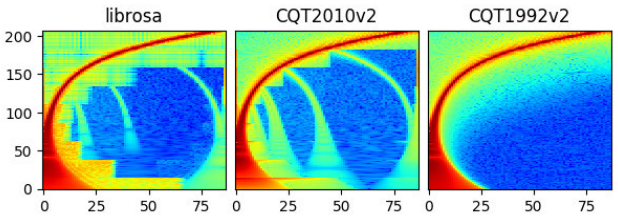

FIGURE 14. A visualisation of the subtle differences between `librosa`, CQT2010v2 and CQT1992v2 implementations using a logarithmic scale. CQT1992v2 yields the best result. A linear sine sweep is used as the input signal.

linear frequency scale spectrogram (LinSpec), logarithmic frequency scale spectrogram (LogSpec), Mel spectrogram (MelSpec), and CQT. In addition, each of these representations will have different parameter settings that we need to consider. For LinSpec and LogSpec, we want to explore five different sizes of Fourier kernels. For MelSpec, we will be exploring four different sizes of Fourier kernels, and for each of these kernels, the number of Mel filter banks will be varied. Finally, for CQT, ten different bins per octave will be examined. This means that there will be a total of 34 different input representations.

If we use MusicNet [22], [63] as our training data, the traditional approach would require the raw waveforms of this dataset to be converted into 34 different spectrograms that are saved to the hard disk. Then we would use a dataloader to load different types of spectrograms into different neural networks to train them one by one (Figure 1(a)). This approach is impractical, as the training set of MusicNet consists of 20GB of waveforms. After processing these waveforms into 34 different input representations (with varying types of spectrograms, window size, etc.), 950.4GB of data would be generated. As shown in Table 3, Obtaining these different representations takes a total of 633, 445 and 983 minutes of

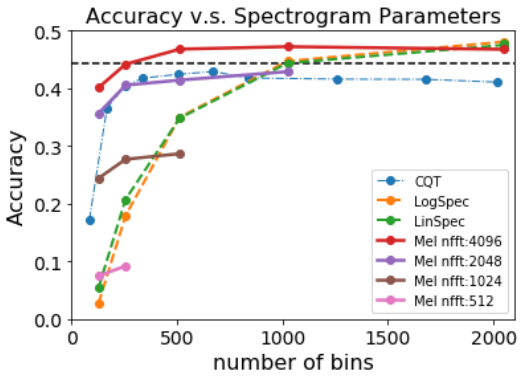

FIGURE 15. Performance of the four different input representations, with different parameters settings, when performing transcription of the audio files in the MusicNet dataset. The dashed black line indicates the transcription accuracy using the same linear model as reported in MusicNet [22], [63].

processing time on machine A, B, and C, respectively (refer to Section IV-A1 for the machine specifications). Even if we delete the spectrograms after training the model to avoid the storage issue, after knowing which type of spectrogram is the best input representation, we still need to obtain that spectrogram again. This traditional approach is tedious.

To alleviate this problem, we use our proposed nnAudio framework to create a neural network capable of converting waveforms to spectrograms on-the-fly (See Section III for how nnAudio works and Figure 1(b) for how nnAudio is different from the traditional approach). Thanks to the fast computation speed obtained by leveraging convolutional neural network on GPUs (Figure 11), we can train 34 different models without the need of saving the spectrograms on the hard disk first. The neural network layer created by nnAudio directly (and quickly) extracts the spectrograms during the model training. By doing so, our dataloader only needs to load the raw waveforms into the neural network. The neural network will then convert the waveforms to spectrograms on-the-fly when training the model (Figure 1(b)). Moreover, as nnAudio extracts spectrograms really fast, it significantly reduces the computation time from 983 minutes to only 99 minutes. Figure 15 shows the transcription accuracy obtained with different input representations. The accuracy is measured by using `mir_eval.multipitch.metrics()`. Because all of these parameters affect the output shape of the spectrograms (number of bins), the results can be plotted as transcription accuracy versus number of frequency bins. It is clear from the image that the input representation and its settings have a big influence on model performance. Using nnAudio, which enables fast comparison of different representations, results in easier to configure and more efficient models.

Next, we want to know if and when (i.e., after how many epochs of training), nnAudio will become slower than the pre-processed approach where the spectrograms are already saved on the hard disk. In this experiment, we will continue using MusicNet as our dataset. Instead of using a simple linear layer as our model, we use a more sophisticated model which consists of two convolutional layers and one bidirectional long short-term memory (LSTM) block. For the on-the-fly approach, the audio files (in .wav format) are loaded, and spectrograms are extracted on-the-fly during training. For the pre-processed approach, the spectrograms are loaded and fed-forward to the neural network (as shown in Figure 1(a)). The experimental results are shown in Figure 16. Here, we also consider the case in which we work on the dataset for the first time. If we take the audio processing time (using librosa) into consideration (dashed blue lines), we can see that the on-the-fly approach using nnAudio (solid green lines) is much faster. However, once the spectrograms are saved on the hard disk, the pre-processed approach (solid blue lines) is faster than nnAudio, since there is no need to repeatedly convert the audio files to spectrograms. As we have discussed before, nnAudio is a useful tool for people to experiment with different spectrograms each with different parameters quickly without any pre-processing. In this setting, nnAudio can drastically reduce the time required for the experimental phase (from the dashed blue lines to the solid green lines). The order of magnitude of how much faster nnAudio can be, compared to the pre-processed approach, depends on the computer configuration, such as whether the CPU is fast enough to load data from RAM to GPU so that the GPU does not stay idle, or whether the GPU is fast enough to handle the data fetched by the CPU. The results for other machines are reported in the supplementary material.

To sum up, nnAudio allows us to integrate audio processing into one of the layers of our neural network model. This layer is responsible for the waveform to spectrogram conversion during the feedforward process. This way, we only need to store the audio clips in the original waveform, without saving extra copies of the dataset for the spectrograms. In addition, nnAudio is also useful when the dataset is so large that it takes tens of hours to convert the data from waveforms to spectrograms. Once the waveforms are ready, they can be loaded batch by batch (when using `PyTorch`) and fed-forward to nnAudio, which then converts batches of waveforms into spectrograms on-the-fly. This saves the user the trouble of processing the original waveforms and saving them as 34 different sets of spectrogram on the hard disk. Yet, it still allows us to perform the same analysis on the results 15. The full details of this experiment are outside of the scope of this paper and may be published in future work.

B. TRAINABLE TRANSFORMATION KERNELS

Because we implement STFT and MelSpec with a 1D convolutional neural network whereby the neuron weights correspond to the Fourier kernels and Mel filter banks, it is possible to further finetune these kernels and filter banks together with the model via gradient descent. This technique is available for all transformations implemented with a neural network, but we will only focus on discussing the STFT and MelSpec in this subsection as an example.

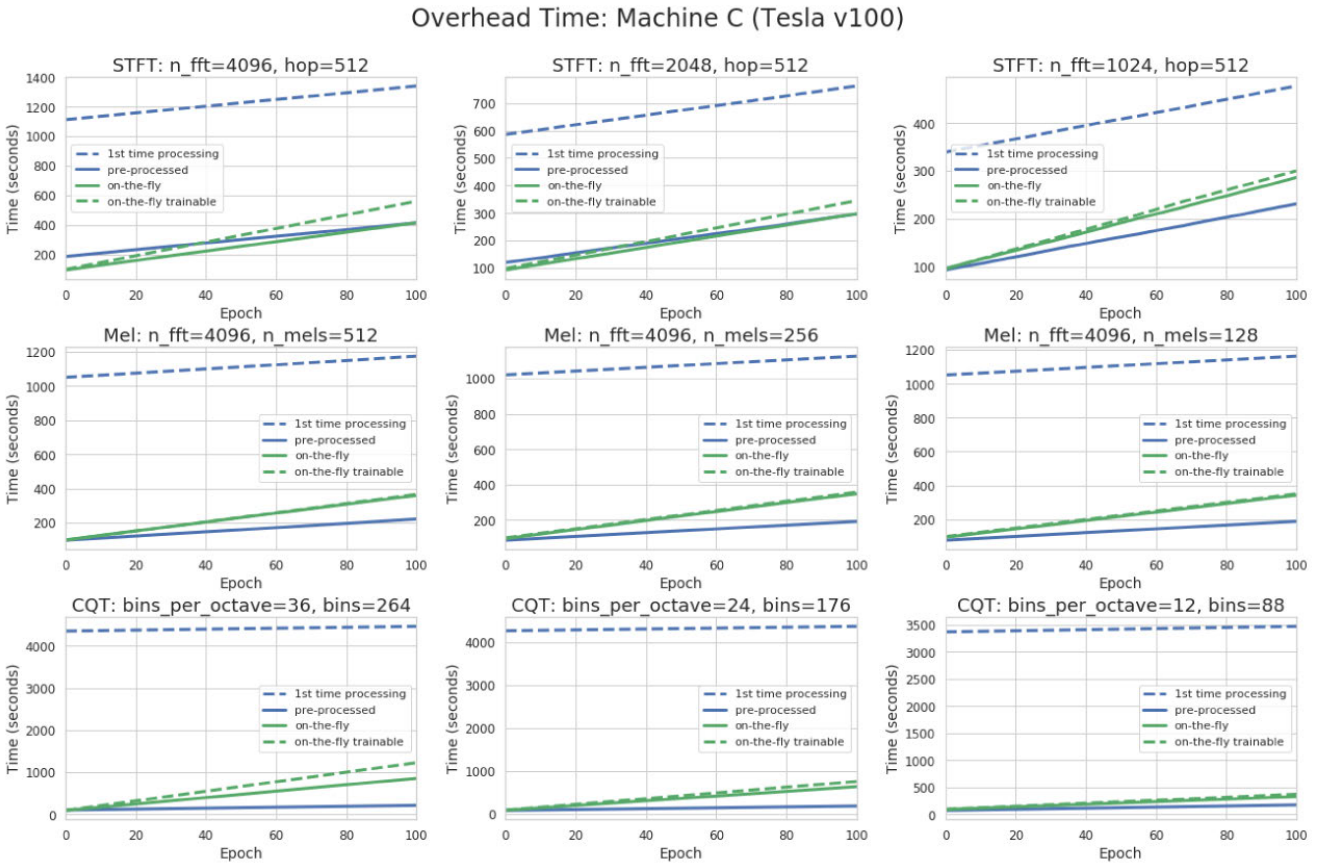

FIGURE 16. Comparisons between on-the-fly audio processing and pre-processed audio in terms of computing time needed to train a neural network model for 100 epochs using `librosa`(dashed blue lines) and nnAudio (green and dashed green lines). The blue lines represent model training using spectrograms as the input; the dashed blue lines include the pre-processing time taken by `librosa`to convert waveforms to spectrograms. The green lines represent model training using waveforms as the input; the dashed green lines represent the model with a trainable waveforms-to-spectrograms layer. This experiment is conducted on machine C; the results for other machines are reported in Figure S1 and Figure S2 of the supplementary material.

Consider the following task: given a pure sine wave, we need to train a model that is able to return the frequency of the signal. To make this task non-trival, the STFT window size is deliberately set to a small number (64), so that the output spectrograms have a very poor frequency resolution. The frequencies for pure sine waves are integers ranging from 200Hz to 22, 050Hz (the Nyquist frequency). In other words, we have only 33 frequency bins to represent the entire audible spectrum from 20 Hz to 20 KHz. To conduct our experiment, we generated 10,925 pure sine waves with different frequencies (between 200 and 22, 050Hz). For each frequency, we generate 10 different pure sine waves with different phases. In total, 109,250 pure sine waves are generated to form our dataset. 80% of these sine waves are used as the training set, and the remaining 20% are used as test set. We explore whether trainable kernels are able to improve the model accuracy. We focus on two models for predicting the frequency of the input sign wave: a fully connected network and a 2D convolutional neural network (CNN). For the fully connected network, we use one layer with one neuron and sigmoid activation. The spectrogram is flattened to a 1D-vector, and used as the input to the model. For CNN, two 2D convolution layers are used, with a kernel size (4×4) for each layer. The final feature maps of the CNN are flattened and fed forward to a fully connected network with one neuron and sigmoid activation. nnAudio is used as the first layer of these models, to convert waveforms to either standard spectrograms, Mel spectrograms, or CQT spectrograms. We set this first layer to be trainable and compare the resulting loss to the same model with this layer set as non-trainable. As can be seen in Figure 17, a trainable transformation layer results in a lower mean square error (MSE) for STFT, MelSpec, and CQT layers and for both the Linear as well as the CNN models.

In order to explain how a trainable STFT, MelSpec, and CQT layer improves the prediction accuracy, we need to study the trained Fourier kernels and Mel filter banks. The first two rows in Figure 18 show the Fourier Basis when the filter bank is $k = 1, 2$. Since the results for the fully connected model are quite similar to the CNN model, we will only report the results for the CNN model here. The column on the left visualizes the original Fourier kernels, and the column on the right visualizes the trained Fourier kernels. Although the overall shape of the trained Fourier kernels is similar to the

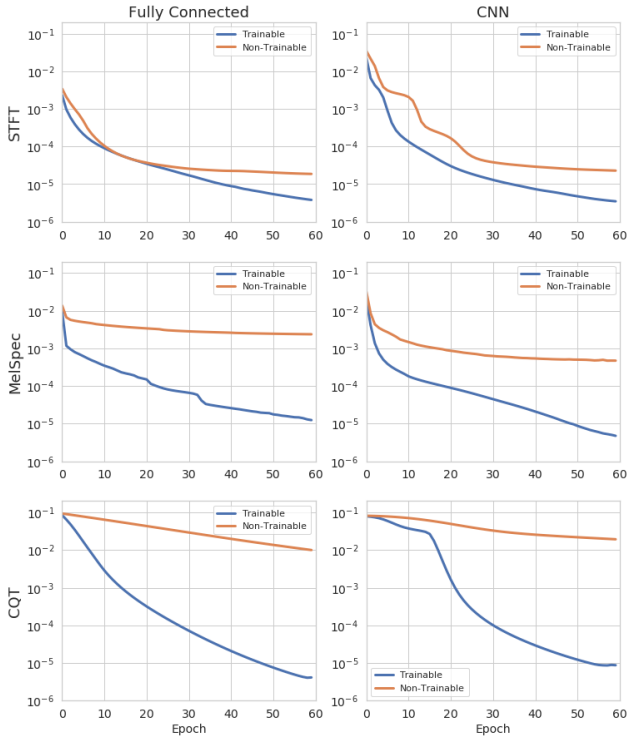

FIGURE 17. The evolution of loss during training for trainable and non-trainable kernels on a frequency prediction task. The models with trainable kernels consistently outperform the models with fixed kernels. The models were trained on 87,400 pure sine waves and evaluated on 21,850 pure sine waves.

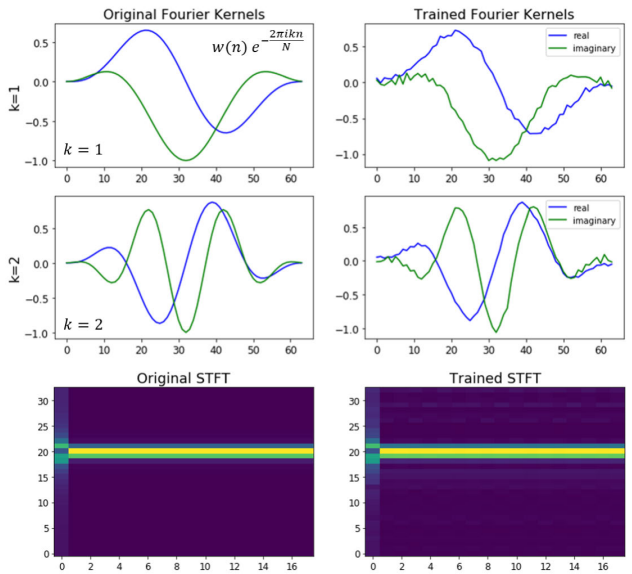

FIGURE 18. The first two rows show the Fourier kernels before and after training (only two of the kernels are shown here), and the third row shows the spectrograms resulting from the original and trained kernels.

original Fourier kernels, it contains some higher frequencies on top of the fundamental frequency for the kernels. These extra frequencies may allow more information to be extracted via STFT. The trained STFT spectrogram is shown in the

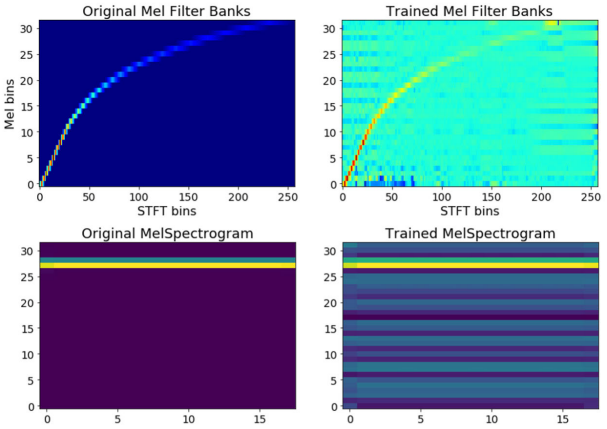

FIGURE 19. The first row shows the complete set of Mel filter banks before and after training. The resulting spectrograms are shown below.

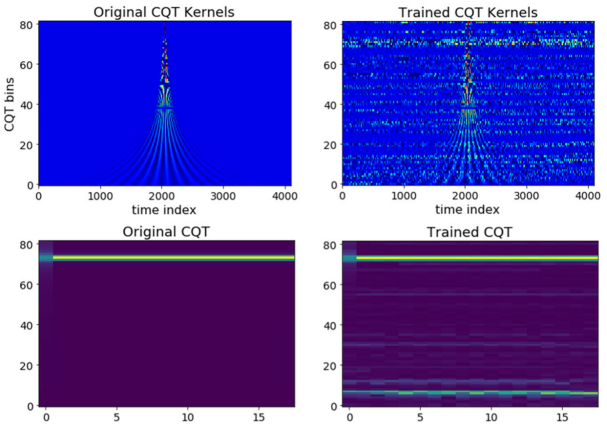

FIGURE 20. The first row shows the complete set of CQT kernels before and after training. Their resulting spectrograms are shown below.

last row of the same figure. It is clear from this figure that it has more overtone-like signals around the fundamental frequency, while the original STFT shows a very clean response for the pure sine wave input. The spectrogram obtained via the trained STFT may be able to provide clues to the neural network about the input frequency of the input signal. The same is true for the trained Mel filter banks and CQT kernels as shown in Figure 19 and 20. By allowing the neural network to further train or finetune the Mel filter banks and CQT kernels, we allow a richer spectrogram to be obtained. This provides the frequency prediction models, regardless of the network architecture, with more information so as to reach a lower MSE loss.

This subsection shows that further training or finetuning the spectrogram transformation layer with nnAudio results in a lower MSE loss. Despite the fact that this analysis uses a simple, artificially generated dataset, it still provides a good example of how a higher-performing end-to-end model can be obtained with a trainable transformation layer. The detailed experimental results are available on the `nnAudio` github repository[2].

VI. CONCLUSION

We have presented a new framework for extracting different types of spectrograms on-the-fly with neural networks. This approach allows one to dynamically train the kernels (including Fourier kernels, Mel filter banks, and CQT kernels) as part of the larger neural network training, specifically adapted to the problem at hand. Our approach has been implemented as the GPU-based library, nnAudio.

Different time domain to frequency domain transformation algorithms such as short-time Fourier transform, Mel spectrograms, and constant-Q transform have been implemented in `PyTorch`, an open-source machine learning library. We leverage the CUDA integration of `PyTorch` that enables fast GPU based audio processing. In our experiments we found that GPU audio processing reduces the time it takes to convert 1,770 waveforms to spectrograms from 10.6 seconds to only 0.001 seconds for the Short-Time Fourier Transform (STFT); from 18.3 seconds to 0.015 seconds for the Mel spectrogram; and from 103.4 seconds to 0.258 seconds for the constant-Q Transform (CQT). These experiments were performed on three different machines: two desktops with GTX 1070 and RTX 2080 Ti respectively, and one DGX station with a Tesla v100 GPU. Although it takes some time (around 5 seconds) to initialize the transformation layer (transferring Fourier kernels from RAM to GPU memory), once everything is ready on the GPU memory, the processing time is in the order of microseconds for a single spectrogram, making the initialization time negligible in the context of training a neural network.

Furthermore, our proposed neural network-based audio processing framework allows for trainable and finetunable Fourier kernels, Mel filter banks, and even CQT kernels. An experiment discussed in Section V-B confirms that trainable kernels result in a better final model on a frequency prediction task compared to non-trainable kernels.

Finally, we present a neural network approach to calculate different versions of the CQT (direct computation, downsampling method, and by removing the frequency domain kernels). To our knowledge, our proposed framework is the first neural network-based audio processing toolbox that supports CQT. When comparing the computation speed of different neural network-based CQT algorithms (Figure 11(b)), we discovered (in Section III-C5) that the CQT algorithm, which uses time domain CQT kernels, performs faster than the commonly used CQT algorithm based on frequency domain kernels [7], [59]. As a result, the CQT computation speed of converting 1,770 waveforms to spectrograms is reduced drastically in our proposed GPU neural network-based framework from 0.258 to only 0.001. When applying nnAudio to a real dataset, MusicNet, it significantly reduces the computation time from 983 minutes to only 99 minutes.

To make our proposed GPU audio processing tool easy to use for other researchers, we have combined all of the algorithms discussed above into a user-friendly PyPI package called nnAudio[2].

ACKNOWLEDGMENT

The authors thank En Yan Koh (Ph.D. student at the Singapore University of Technology and Design (SUTD)) and the SUTD Game Lab for providing the RTX 2080 Ti and GTX 1070 Ti GPUs for our experiments.

REFERENCES

[1] M. J. Palakal and M. J. Zoran, "Feature extraction from speech spectrograms using multi-layered network models," in *Proc. IEEE Int. Workshop Tools Artif. Intell.*, Oct. 1989, pp. 224–230.

[2] K. Hatazaki, Y. Komori, T. Kawabata, and K. Shikano, "Phoneme segmentation using spectrogram reading knowledge," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, May 1989, pp. 393–396.

[3] M. C. Recchione and A. P. Russo, "Feedforward neural network system for the detection and characterization of sonar signals with characteristic spectrogram textures," U.S. Patent 5 502 688, Mar. 26, 1996.

[4] S. Furui, "Speaker-independent isolated word recognition based on emphasized spectral dynamics," in *Proc. ICASSP. IEEE Int. Conf. Acoust., Speech, Signal Process.*, vol. 11, Apr. 1986, pp. 1991–1994.

[5] S. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 28, no. 4, pp. 357–366, Aug. 1980.

[6] J. C. Brown, "Calculation of a constant Q spectral transform," *J. Acoust. Soc. Amer.*, vol. 89, no. 1, pp. 425–434, Jan. 1991.

[7] J. C. Brown and M. S. Puckette, "An efficient algorithm for the calculation of a constant Q transform," *J. Acoust. Soc. Amer.*, vol. 92, no. 5, pp. 2698–2701, Nov. 1992.

[8] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A generative model for raw audio," 2016, *arXiv:1609.03499*. [Online]. Available: http://arxiv.org/abs/1609.03499

[9] T. Kim, J. Lee, and J. Nam, "Comparison and analysis of SampleCNN architectures for audio classification," *IEEE J. Sel. Topics Signal Process.*, vol. 13, no. 2, pp. 285–297, May 2019.

[10] H. Purwins, B. Li, T. Virtanen, J. Schlüter, S.-Y. Chang, and T. Sainath, "Deep learning for audio signal processing," *IEEE J. Sel. Topics Signal Process.*, vol. 13, no. 2, pp. 206–219, May 2019.

[11] J. Lee, J. Park, K. Kim, and J. Nam, "SampleCNN: End-to-end deep convolutional neural networks using very small filters for music classification," *Appl. Sci.*, vol. 8, no. 1, p. 150, Jan. 2018.

[12] C. Kim, S. Kim, K. Kim, M. Kumar, J. Kim, K. Lee, C. Han, A. Garg, E. Kim, M. Shin, S. Singh, L. Heck, and D. Gowda, "End-to-end training of a large vocabulary end-to-end speech recognition system," in *Proc. IEEE Autom. Speech Recognit. Understand. Workshop (ASRU)*, Dec. 2019, pp. 562–569.

[13] J. Zhao, X. Mao, and L. Chen, "Speech emotion recognition using deep 1D & 2D CNN LSTM networks," *Biomed. Signal Process. Control*, vol. 47, pp. 312–323, Jan. 2019.

[14] A. Tjandra, S. Sakti, and S. Nakamura, "Speech-to-speech translation between untranscribed unknown languages," in *Proc. IEEE Autom. Speech Recognit. Understand. Workshop (ASRU)*, Dec. 2019, pp. 593–600.

[15] S. A. Shahriyar, M. A. H. Akhand, N. Siddique, and T. Shimamura, "Speech enhancement using convolutional denoising autoencoder," in *Proc. Int. Conf. Electr., Comput. Commun. Eng. (ECCE)*, Feb. 2019, pp. 1–5.

[16] K. W. E. Lin, B. T. Balamurali, E. Koh, S. Lui, and D. Herremans, "Singing voice separation using a deep convolutional neural network trained by ideal binary mask and cross entropy," *Neural Comput. Appl.*, vol. 32, no. 4, pp. 1037–1050, Feb. 2020.

[17] Y.-J. Luo, C.-C. Hsu, K. Agres, and D. Herremans, "Singing voice conversion with disentangled representations of singer and vocal technique using variational autoencoders," in *Proc. ICASSP-IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2020, pp. 3277–3281.

[18] J. Choi, J. Lee, J. Park, and J. Nam, "Zero-shot learning for audio-based music classification and tagging," in *Proc. 20th Int. Soc. Music Inf. Retr. Conf. (ISMIR)*, A. Flexer, G. Peeters, J. Urbano, and A. Volk, Eds. Delft, The Netherlands: ISMIR, 2019, pp. 67–74.

[19] G. Doras and G. Peeters, "Cover detection using dominant melody embeddings," in *Proc. 20th Int. Soc. Music Inf. Retr. Conf. (ISMIR)*, A. Flexer, G. Peeters, J. Urbano, and A. Volk, Eds. Delft, The Netherlands: ISMIR, 2019, pp. 107–114.

[20] G. Doras, P. Esling, and G. Peeters, "On the use of U-Net for dominant melody estimation in polyphonic music," in *Proc. Int. Workshop Multilayer Music Represent. Process. (MMRP)*, Jan. 2019, pp. 66–70.
[21] R. Kelz and G. Widmer, "Towards interpretable polyphonic transcription with invertible neural networks," in *Proc. 20th Int. Soc. Music Inf. Retr. Conf. (ISMIR)*. Delft, The Netherlands: ISMIR, 2019, pp. 376–383.
[22] J. Thickstun, Z. Harchaoui, and S. M. Kakade, "Learning features of music from scratch," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–14.
[23] B. McFee, C. Raffel, D. Liang, D. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "Librosa: Audio and music signal analysis in Python," in *Proc. 14th Python Sci. Conf.*, vol. 8, 2015, pp. 18–25.
[24] M. Abadi *et al.* (2015). *Tensorflow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: http://tensorflow.org/
[25] K. Choi, D. Joo, and J. Kim, "Kapre: On-GPU audio preprocessing layers for a quick implementation of deep neural network models with keras," 2017, *arXiv:1706.05781*. [Online]. Available: http://arxiv.org/abs/1706.05781
[26] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in Pytorch," in *Proc. NIPS Autodiff Workshop*, 2017, pp. 1–4.
[27] K. Qu. (2019). *Some Problems When Installing Torchaudio on MAC*. Accessed: Sep. 10, 2019. [Online]. Available: https://stackoverflow.com/q/56659166/
[28] L. Ericson. (2019). *How to Install Torch Audio on Windows 10 Conda?* Accessed: Sep. 10, 2019. [Online]. Available: https://stackoverflow.com/q/54872876
[29] K. W. Cheuk, K. Agres, and D. Herremans, "NnAudio: A pytorch audio processing tool using 1D convolution neural networks," in *Proc. ISMIR–Late Breaking Demo*, Delft, The Netherlands, 2019.
[30] B. Balamurali, K. E. Lin, S. Lui, J.-M. Chen, and D. Herremans, "Toward robust audio spoofing detection: A detailed comparison of traditional and learned features," *IEEE Access*, vol. 7, pp. 84229–84241, 2019.
[31] R. Kelz, M. Dorfer, F. Korzeniowski, S. Böck, A. Arzt, and G. Widmer, "On the potential of simple framewise approaches to piano transcription," in *Proc. 17th Int. Soc. Music Inf. Retr. Conf. (ISMIR)*, New York City, NY, USA, 2016, pp. 475–481.
[32] Q. Wang, R. Zhou, and Y. Yan, "A two-stage approach to note-level transcription of a specific piano," *Appl. Sci.*, vol. 7, no. 9, p. 901, Sep. 2017.
[33] P. Yenigalla, A. Kumar, S. Tripathi, C. Singh, S. Kar, and J. Vepa, "Speech emotion recognition using spectrogram and phoneme embedding," in *Proc. Interspeech*, Sep. 2018, pp. 3688–3692.
[34] L. Wyse, "Audio spectrogram representations for processing with convolutional neural networks," 2017, *arXiv:1706.09559*. [Online]. Available: http://arxiv.org/abs/1706.09559
[35] M. Lederle and B. Wilhelm, "Combining high-level features of raw audio waves and mel-spectrograms for audio tagging," 2018, *arXiv:1811.10708*. [Online]. Available: http://arxiv.org/abs/1811.10708
[36] Y. Sakashita and M. Aono, "Acoustic scene classification by ensemble of spectrograms based on adaptive temporal divisions," in *Proc. DCASE Challenge*, 2018, pp. 1–5.
[37] Z. Weiping, Y. Jiantao, X. Xiaotao, L. Xiangtao, and P. Shaohu, "Acoustic scene classification using deep convolutional neural network and multiple spectrograms fusion," in *Proc. Detection Classification Acoust. Scenes Events (DCASE)*, 2017, pp. 1–5.
[38] L. Zhang, I. Saleh, S. Thapaliya, J. Louie, J. Figueroa-Hernandez, and H. Ji, "An empirical evaluation of machine learning approaches for species identification through bioacoustics," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2017, pp. 489–494.
[39] B. Murauer and G. Specht, "Detecting music genre using extreme gradient boosting," in *Proc. Companion Web Conf. Web Conf. (WWW)*, 2018, pp. 1923–1927.
[40] M. Dong, "Convolutional neural network achieves human-level accuracy in music genre classification," 2018, *arXiv:1802.09697*. [Online]. Available: http://arxiv.org/abs/1802.09697
[41] Z. Wang, "Fast algorithms for the discrete W transform and for the discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 32, no. 4, pp. 803–816, Aug. 1984.
[42] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *Handbook Brain Theory Neural Netw.*, vol. 3361, no. 10, p. 1995, 1995.
[43] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[44] S. H. Nawab and T. F. Quatieri, "Short-time Fourier transform," in *Advanced Topics in signal processing*, J. S. Lim and A. V. Oppenheim, Eds. Upper Saddle River, NJ, USA: Prentice-Hall, 1987, pp. 289–337. [Online]. Available: http://dl.acm.org/citation.cfm?id=42739.42745
[45] A. V. Oppenheim and R. W. Schafer, "Fourier transform theorems," in *Discrete-Time Signal Processing*. London, U.K.: Pearson, 1989, p. 60.
[46] J. Volkmann, S. S. Stevens, and E. B. Newman, "A scale for the measurement of the psychological magnitude pitch," *J. Acoust. Soc. Amer.*, vol. 8, no. 3, p. 208, Jan. 1937.
[47] S. S. Stevens and J. Volkmann, "The relation of pitch to frequency: A revised scale," *Amer. J. Psychol.*, vol. 53, no. 3, pp. 329–353, 1940. [Online]. Available: http://www.jstor.org/stable/1417526
[48] G. Fant, "Analys av de svenska konsonantljuden," LM Ericsson Protokoll H/P 1064.194, 1949, p. 139.
[49] W. Koening, "A new frequency scala for acoustic measurements," (in Japanese), *Bell Lab Rec.*, pp. 299–301, 1949. [Online]. Available: https://ci.nii.ac.jp/naid/10009375717/
[50] S. Umesh, L. Cohen, and D. Nelson, "Fitting the Mel scale," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, vol. 1, Mar. 1999, pp. 217–220.
[51] D. O'Shaughnessy, *Speech Communications: Human and Machine*. New York, NY, USA: Wiley, 1987.
[52] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, and V. Valtchev, "The HTK book," *Cambridge Univ. Eng. Dept.*, vol. 3, no. 175, p. 12, 2002.
[53] M. Slaney, "A MATLAB toolbox for auditory modeling work," Interval Res. Corp., Tech. Rep., 1998. [Online]. Available: http://amtoolbox.sourceforge.net/
[54] L. R. Rabiner and R. W. Schafer, *Theory and Applications of Digital Speech Processing*, vol. 64. Upper Saddle River, NJ, USA: Pearson, 2011.
[55] G. V. Haines and A. G. Jones, "Logarithmic Fourier transformation," *Geophys. J. Int.*, vol. 92, no. 1, pp. 171–178, Jan. 1988.
[56] J. O. Smith, III, *Introduction to Digital Filters With Audio Applications* (Center for Computer Research in Music and Acoustics (CCRMA)). Stanford, CA, USA: Stanford Univ., 2007.
[57] J. Youngberg and S. Boll, "Constant-Q signal analysis and synthesis," in *Proc. ICASSP. IEEE Int. Conf. Acoust., Speech, Signal Process.*, vol. 3, Apr. 1978, pp. 375–378.
[58] S. S. Rajput and D. S. Bhadauria, "Implementation of fir filter using efficient window function and its application in filtering a speech signal," *Int. J. Electr., Electron. Mech. Controls*, vol. 1, no. 1, pp. 1–12, 2012.
[59] C. Schörkhuber and A. Klapuri, "Constant-Q transform toolbox for music processing," in *Proc. 7th Sound Music Comput. Conf.*, Barcelona, Spain, 2010, pp. 3–64.
[60] V. Emiya, N. Bertin, B. David, and R. Badeau, "MAPS—A piano database for multipitch estimation and automatic transcription of music," Res. Rep. inria-00544155, 2010, p. 11. [Online]. Available: https://hal.inria.fr/inria-00544155
[61] M. Zolnouri, X. Li, and V. P. Nia, "Importance of data loading pipeline in training deep neural networks," 2020, *arXiv:2005.02130*. [Online]. Available: http://arxiv.org/abs/2005.02130
[62] A. Gayer, Y. Chernyshova, and A. Sheshkus, "Effective real-time augmentation of training dataset for the neural networks learning," *Proc. SPIE*, vol. 11041, Mar. 2019, Art. no. 110411I.
[63] J. Thickstun, Z. Harchaoui, D. P. Foster, and S. M. Kakade, "Invariances and data augmentation for supervised music transcription," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2018, pp. 2241–2245.
[64] E. Benetos, S. Dixon, D. Giannoulis, H. Kirchhoff, and A. Klapuri, "Automatic music transcription: Challenges and future directions," *J. Intell. Inf. Syst.*, vol. 41, no. 3, pp. 407–434, Dec. 2013.
[65] A. Holzapfel and E. Benetos, "Automatic music transcription and ethnomusicology: A user study," in *Proc. 20th Int. Soc. Music Inf. Retr. Conf. (ISMIR)*, A. Flexer, G. Peeters, J. Urbano, and A. Volk, Eds. Delft, The Netherlands: IEEE Press, 2019, pp. 678–684.
[66] K. W. Cheuk, K. Agres, and D. Herremans, "The impact of audio input representations on neural network based music transcription," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Glasgow, U.K., 2020, pp. 1–5.

KIN WAI CHEUK (Graduate Student Member, IEEE) received the B.S. degree in physics and the M.Phil. degree in mechanical engineering from The University of Hong Kong, in 2014 and 2016, respectively. He is currently pursuing the Ph.D. degree with the Singapore University of Technology and Design (SUTD), supervised by Dr. Herremans and affiliated with the Institute of High Performance Computing, Agency for Science Technology and Research (A*STAR). He worked for an IT solution company and had a growing interest in machine learning. He joined Dr. Herremans' Research Group, in 2018, where he started working on music related machine learning projects. His research interests include digital signal processing and music transcription using machine learning. He received the Singapore International Graduate Award (SINGA) for his Ph.D. degree.

HANS ANDERSON received the master's degree in computational mathematics from the University of Minnesota, in 2007, and the Ph.D. degree from the Singapore University of Technology and Design, in 2018. He is currently the Director of Blue Mangoo Software, a company based in Ha Noi that develops musical audio applications for mobile phones and tablets. He is also a Freelance Audio Signal Processing Consultant. He loves designing signal processing algorithms for practical applications and commercial products, with a focus on improving the subjective sound quality of widely-used methods. He is working on eliminating aliasing noise from nonlinear audio effects, such as saturation, compression, and amplifier models.

KAT AGRES (Member, IEEE) received the bachelor's degree in cognitive psychology and cello performance from Carnegie Mellon University, and the Ph.D. degree in psychology in cognitive science from Cornell University, in 2013. She completed two Postdoctoral Research Positions with the School of Electronic Engineering and Computer Science, Queen Mary University of London (QMUL). She is currently an Assistant Professor with the National University of Singapore (NUS). She is also a Research Scientist and a Principal Investigator of the Music Cognition Group, Institute of High Performance Computing (IHPC), which is housed within the Agency for Science, Technology and Research (A*STAR). She has received numerous grants to support her research, including a Fellowship from the National Institute of Health (NIH) and a Training Fellowship from the National Institute of Mental (NIMH). Her research interests include music technology for healthcare and well-being, music perception and cognition, computational creativity, information theoretic modeling of learning and memory, statistical learning, and evaluation of creativity in humans and artificial systems.

DORIEN HERREMANS (Senior Member, IEEE) received the Ph.D. degree in applied economics from the University of Antwerp. She worked with the Centre for Digital Music, Queen Mary University of London. She was a Commercial Engineer in management information systems with the University of Antwerp, in 2005. She worked as a Drupal Consultant and an IT Lecturer with Les Roches University, Bluche, Switzerland. She is currently an Assistant Professor with the Singapore University of Technology and Design, with a joint appointment with the Institute of High Performance Computing, Agency for Science Technology and Research (A*STAR). Her research interests include intersection of machine learning/optimization and digital music/audio. She is a Co-organizer of the First International Workshop ton Deep Learning and Music as part of IJCNN. She was awarded the individual Marie-Curie Fellowship for Experienced Researchers, in 2015. She is also a Guest Editor of *Neural Computing and Applications* (Springer).