Music demixing with the sliced Constant-Q Transform

Sevag Hanssian

Music Technology Area
Department of Music Research
Schulich School of Music
McGill University, Montreal, Canada

August 2022

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Arts.

Acknowledgement

I would like to thank my supervisor, Professor Ichiro Fijunaga, for providing guidance throughout my master's degree and thesis. His rigor and attention to detail have made me a better researcher. Next, I owe thanks to my labmates at the Distributed Digital Music Archives & Libraries Lab (DDMAL), who were always available to help; Néstor Nápoles López, Timothy de Reuse, Emily Hopkins, and all the others. In the McGill Music Technology department, Professor Philippe Depalle helped me hone my signal processing knowledge, and Professor Gary Scavone's emphasis on creativity allowed me to do my best work. I would also like to thank my family and close friends for their unconditional support and encouragement.

Finally, I am grateful to the open-source community for music source separation research, including the SigSep organization[1] on GitHub, which provides high-quality code to accompany academic papers, and the 2021 Music Demixing challenge,[2] which encouraged open-source contributions and fostered a collaborative environment.

1. https://github.com/sigsep
2. https://mdx-workshop.github.io/

Abstract

Music source separation is the task of extracting an estimate of one or more isolated sources or instruments (for example, drums or vocals) from musical audio. The task of music demixing or unmixing considers the case where the musical audio is separated into an estimate of all of its constituent sources that can be summed back to the original mixture. Models for music demixing that use the Short-Time Fourier Transform (STFT) as their representation of music signals are popular and have achieved success in recent years. However, the fixed time-frequency resolution of the STFT, arising from the time-frequency uncertainty principle, requires a tradeoff in time-frequency resolution that can significantly affect music demixing results. The sliced Constant-Q Transform (sliCQT) is a time-frequency transform with varying time-frequency resolution that avoids the time-frequency tradeoff of the STFT. The model proposed by this thesis replaces the STFT with the sliCQT in a recent model for music demixing, to investigate the impact on the results.

Résumé

La séparation de sources musicales consiste à extraire d'un enregistrement audio une ou plusieurs sources ou instruments de musique (tels que des percussions ou de la voix). Plus spécifiquement, la tâche de démixage correspond au cas où l'enregistrement audio est séparé en toutes ses sources constitutives, qui peuvent être re-mixées pour restituer l'ensemble original. Les modèles de démixage de musique qui se fondent sur la transformée de Fourier à court-terme (TFCT) pour représenter les signaux sonores sont populaires et ont été utilisées avec succès au cours des dernières années. Toutefois, le compromis entre résolution temporelle et résolution fréquentielle préfixé dans la TFCT, résultant du principe d'incertitude temps-fréquence, peut affecter considérablement les résultats du démixage audio. La transformée en tranche à facteur de qualité constant (sliCQT) est une transformée temps-fréquence pour laquelle le compromis entre résolutions temporelle et fréquentielle varie selon la région spectrale considérée, ce qui permet une plus grande flexibilité d'approche qu'avec la TFCT. Le modèle proposé dans cette thèse remplace la TFCT par la sliCQT dans un modèle de démixage audio et évalue l'impact de ce choix sur les résultats.

Contents

List of Figures

List of Code Listings

List of Tables

1 Introduction

The study of acoustic signals forms the core of many fields including sonar, seismology, audio, music, and speech. Acoustic signals are functions of time, representing the evolving amplitude of the sound pressure wave (Rabiner and Schafer 2010). Another way to characterize acoustic signals is by their frequency components, which can be computed from the Fourier Transform (Rabiner and Schafer 2010).

The Fourier Transform represents the frequency components of an acoustic signal by a sum of infinite sinusoids, but it does not describe how these frequencies evolve with time (Kutz 2013). Conversely, many important signals such as speech and music contain frequency components that change with time (Gabor 1946). For these signals, joint time-frequency analysis is required.

One method for performing time-frequency analysis is to multiply the signal being studied by finite, consecutive windows of a short duration, and taking the Fourier Transform of each windowed section. This is also referred to as the Short-Time Fourier Transform (STFT) or spectrogram (Rabiner and Schafer 2010). Rabiner and Schafer (2010) describe the resulting spectral analysis as an important technique in analyzing and understanding audio that has been in use since the 1930s.

Music source separation is the task of extracting an estimate of one or more isolated sources or instruments (for example, drums or vocals) from musical audio (Cano et al. 2018; Liu and Li 2009). The task of music demixing or unmixing considers the case where the musical audio is separated into an estimate of all of its constituent sources that can be summed back to the original mixture.[1] Music demixing systems are commonly designed to split mixed songs into four sources: vocals, drums, bass, and other (guitar, piano, etc.), following the example of Western pop music used in mixing and demixing datasets (Liutkus et al. 2017; Bittner et al. 2014; Rafii et al. 2017, 2019).

The STFT is a popular tool used by various music demixing systems throughout the years (Cano et al. 2018; Liu and Li 2009; Stöter, Liutkus, and Ito 2018; Fitzgerald 2010; Stöter et al. 2019; Grais and Plumbley 2017; Grais, Zhao, and Plumbley 2021). However, the STFT is subject to a fixed and bounded time-frequency resolution, such that one cannot have maximal resolution in both time and frequency, and can trade them off by changing the duration and the type of the window (Gabor 1946; Kutz 2013).

1. Thanks to Fabian-Robert Stöter for the discussion on the difference between music source separation, demixing, and unmixing.

The tradeoff in time-frequency resolution is an important consideration in music demixing (Simpson 2015; Kavalerov et al. 2019). Systems have been proposed that use multiple STFTs (Fitzgerald and Gainza 2010; Driedger, Müller, and Disch 2014) or that replace the STFT with different time-frequency transforms (Fitzgerald and Gainza 2010; Shi, Ziqiang et al. 2019; Burred and Sikora 2006), to improve results through the manipulation of time-frequency resolution.

1.1 Motivation

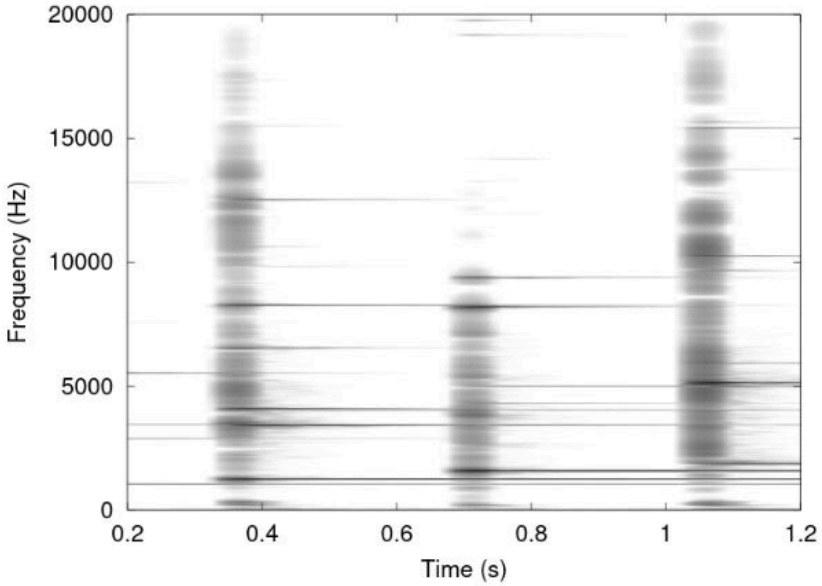
Musical signals have characteristics that lead to conflicting requirements in the STFT. According to Dörfler (2002), music needs to be analyzed with long-duration windows in the low-frequency region for a high frequency resolution, because the bass notes lay the harmonic basis of a song. Conversely, the high-frequency region contains broadband sounds and components of transients, which are useful for timbre identification and rhythm; these transients have fast attacks and decays and need to be analyzed with short-duration windows (Dörfler 2002). Schörkhuber, Klapuri, and Sontacchi similarly state that

> a well known disadvantage of the STFT is the rigid time-frequency resolution trade-off providing a constant absolute frequency resolution throughout the entire range of audible frequencies. In contrast to this we know that due to both musical and auditory aspects frequency resolution is preferred that increases from high to low frequencies (and vice versa for time resolution) (Schörkhuber, Klapuri, and Sontacchi 2012, 1).

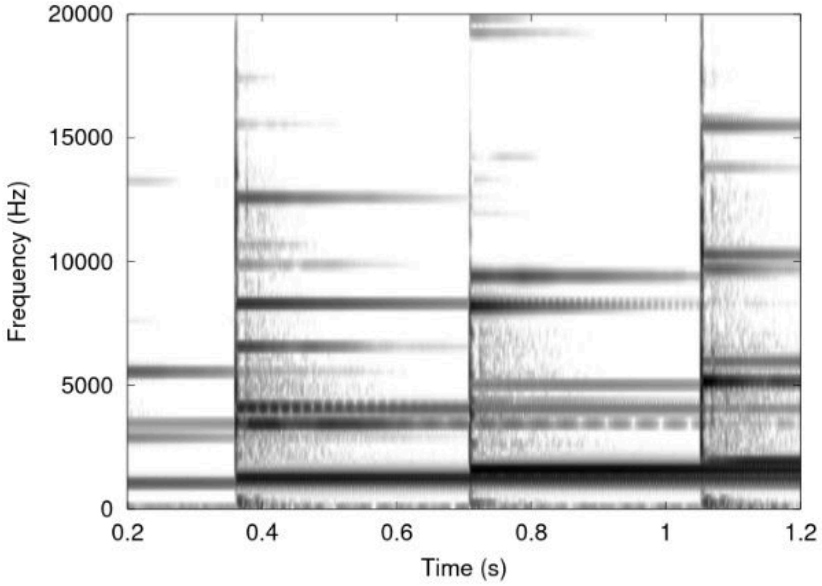
Figure 1 shows a musical glockenspiel signal analyzed with STFTs using different window sizes. Note how in Figure 1(a), due to the high frequency resolution, the temporal events are blurry or smeared such that the times of note onsets are unclear. The inverse case is shown in Figure 1(b), which contains sharp localization of note onsets from the high time resolution, but blurry or smeared frequency components.

Rubinstein, Bruckstein, and Elad (2010) note that the first use of the STFT was by Gabor (1946) to analyze a speech signal, which used fixed-size Gaussian windows. This was called the Gabor transform, and is now considered a special case of the STFT (Rubinstein, Bruckstein, and Elad 2010).

The Constant-Q Transform (CQT) was originally proposed by Brown (1991) to analyze musical signals with a logarithmic frequency scale that matched the notes of a musical pitch scale. The resulting transform used long-duration windows in the low frequency regions and short-duration windows in the high frequency regions. The visual comparison of the CQT



(a) Wide window STFT (93ms).

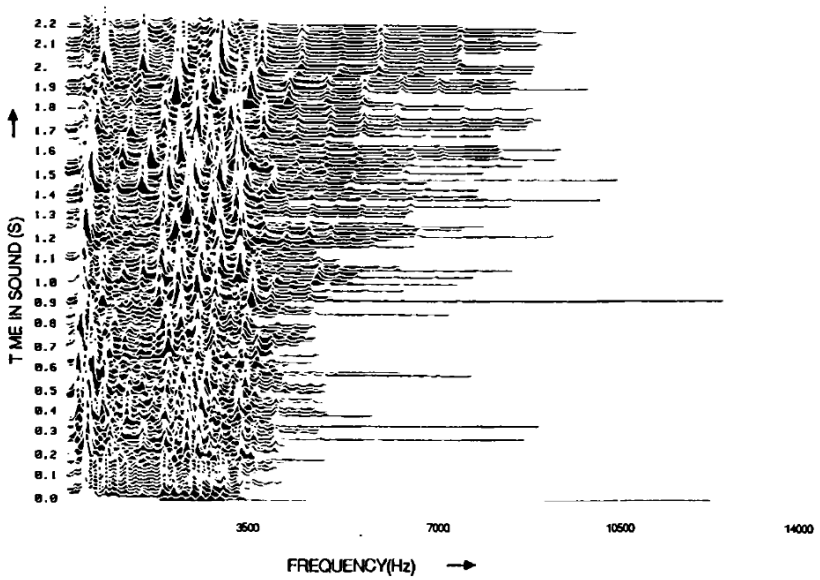


(b) Narrow window STFT (6ms).

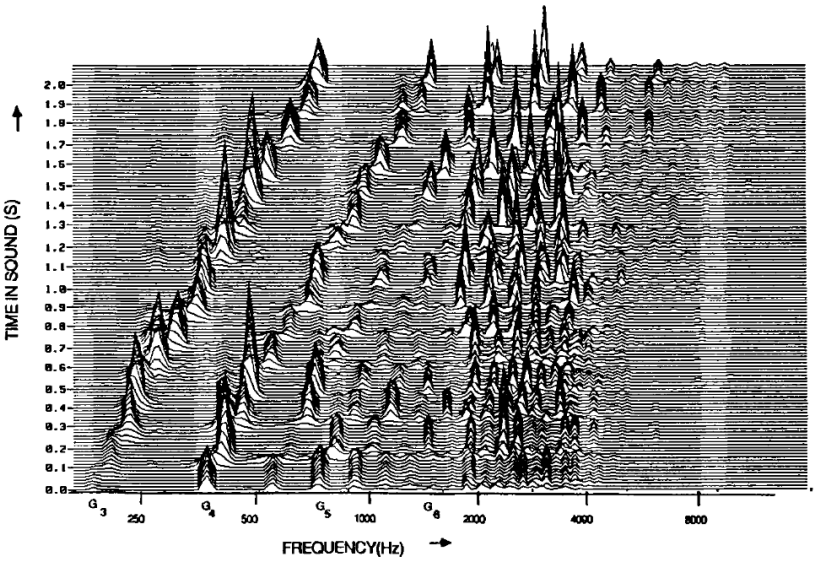
Figure 1: Different window size STFT spectrograms of a glockenspiel signal (Jaillet, Balazs, and Dörfler 2009, 1).

and the Fourier Transform is shown in Figure 2. From the use of short windows in the high frequency regions, the CQT demonstrates good time resolution (Schörkhuber, Klapuri, and Sontacchi 2012) in the region of transients and broadband signals, which are important for timbre and instrument identification (McAdams 2019; Siedenburg 2019).

Building on the Gabor transform (or STFT), and using the CQT as the motivating appli-



(a) Linear-frequency Fourier transform.



(b) Constant-Q transform.

Figure 2: Violin playing the diatonic scale, $G_3(196\text{Hz}) - G_5(784\text{Hz})$ (Brown 1991, 430).

cation, Balazs et al. (2011) proposed the Nonstationary Gabor transform (NSGT), a time-frequency transform with varying time-frequency resolution and perfect inverse. The NSGT is computed by varying the size of the window on which the Fourier Transform is taken. Holighaus et al. (2013) introduced a realtime variant of the NSGT, called the *sliced Constant-Q transform* (sliCQT). The NSGT spectrogram of the musical glockenspiel signal is shown in Figure 3, with minimal blurriness (or good resolution) in both time and frequency, which is an improvement over the STFT spectrogram seen in Figure 1.

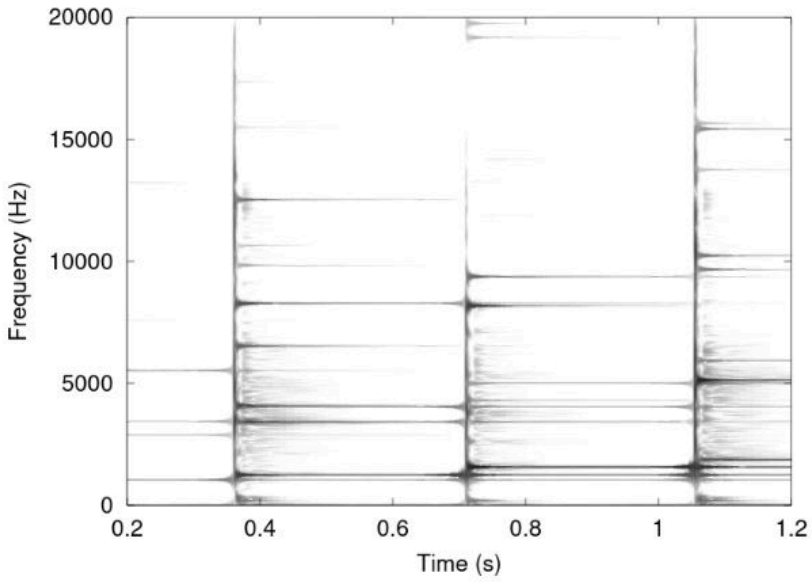


Figure 3: NSGT spectrogram of a glockenspiel signal, varying window (6–93ms) (Jaillet, Balazs, and Dörfler 2009, 4).

1.2 Thesis objectives and related work

STFT-based music demixing techniques may be limited by the fixed time-frequency resolution of the STFT. The objective of this thesis is to replace the STFT in a music demixing model with the sliCQT, which has a varying time-frequency resolution. Open-Unmix (Stöter et al. 2019) is a state-of-the-art deep neural network for music demixing based on the STFT, with an open-source reference implementation.[2] The goal of this thesis is to adapt Open-Unmix to create a usable music demixing software application based on the sliCQT.

Several music demixing papers (Fitzgerald 2010; Driedger, Müller, and Disch 2014; Simpson 2015; Kavalerov et al. 2019) describe the use of different window sizes of the STFT to improve source separation performance. By contrast, using the NSGT instead of the STFT may be of interest, as it is a single transform which might replace the need for multiple STFTs.

Other music demixing systems have been proposed that use the CQT instead of the STFT (Fitzgerald and Gainza 2010; Shi, Ziqiang et al. 2019; Burred and Sikora 2006). However, these papers used implementations of the CQT which lacked a stable inverse (Hu et al. 2014), and for which different approximate inversion schemes have been proposed (Schörkhuber and Klapuri 2010; Fitzgerald, Cranitch, and Cychowski 2006). The NSGT is a generalization of the CQT that can be used to implement a CQT with perfect inverse, or CQ-NSGT (Velasco et al. 2011; Schörkhuber et al. 2014).

2. https://github.com/sigsep/open-unmix-pytorch

The NSGT and sliCQT can also use any frequency scale that is monotonically increasing, for example, musical or psychoacoustic scales. Additionally, the NSGT and sliCQT have open-source reference implementations in MATLAB[3] and Python.[4]

1.3 Contribution and results

My first contribution in this thesis is the adaptation of the reference Python NSGT/sliCQT library[4] to use PyTorch,[5] a deep learning framework with graphical processing unit (GPU) support. This allows the NSGT and sliCQT to be computed on the GPU, which, due to its capabilities for parallel computing, is the preferred device on which to train deep neural networks (Paszke et al. 2019). The resulting library[6] allows future researchers to use the NSGT and sliCQT in any GPU-based machine learning or deep learning model.

My second contribution in this thesis is the adaptation of Open-Unmix, a deep neural network for music demixing[7] (Stöter et al. 2019), to replace the STFT with the sliCQT. Ideas were also incorporated from CrossNet-Open-Unmix (Sawata et al. 2021), which is a variant of Open-Unmix, and the Convolutional Denoising Autoencoder (Grais and Plumbley 2017), a different deep neural network for music demixing. The final result, named xumx-sliCQ,[8] is a neural network that performs the task of music demixing using the sliCQT.

1.4 Outline

This thesis is organized as follows. In Chapter 2, I will cover the background of acoustic signals, frequency analysis, and the important transforms in this thesis including the Fourier Transform, STFT, CQT, NSGT, and sliCQT. I will also present an overview of machine learning for audio and music signals, an introduction to software and Python code concepts, and a survey of music source separation and music demixing. In Chapter 3, I will describe the methodology for the two objectives of the thesis, which are the PyTorch adaptation of the NSGT/sliCQT library, and the replacement of the STFT with the sliCQT inside Open-Unmix. In Chapter 4, I will show and discuss the experimental results, and in Chapter 5 I will conclude the thesis.

3. http://ltfat.org/doc/filterbank/cqt_code.html
4. https://github.com/grrrr/nsgt
5. https://pytorch.org
6. https://github.com/sevagh/nsgt
7. https://github.com/sigsep/open-unmix-pytorch
8. https://github.com/sevagh/xumx-sliCQ/tree/main

2 Background

In this thesis, my starting point is Open-Unmix (Stöter et al. 2019), a deep neural network for music demixing written in the Python programming language. Open-Unmix uses the Short-Time Fourier Transform (STFT) as its representation of musical signals, which is subject to the limitations of the time-frequency uncertainty principle, or time-frequency tradeoff. I will be exploring the sliced Constant-Q Transform (sliCQT) as a replacement of the STFT, and my hypothesis is that using the sliCQT instead of the STFT will improve the music demixing performance of Open-Unmix due to the varying time-frequency resolution of the sliCQT that is more suitable for musical or auditory analysis.

In this chapter, I will introduce the theoretical background necessary to understand my goal and hypothesis.

In Section 2.1, I will give an overview of acoustic signals, including the time-domain waveform and discrete-time signals in digital systems.

In Section 2.2, I will introduce the frequency domain, which is alternative representation of acoustic signals. I will start by describing frequency analysis, the Fourier transform, and joint time-frequency analysis, leading to the ubiquitous Short-Time Fourier Transform (STFT). Continuing from the STFT, I will describe nonuniform time-frequency transforms that were designed for music analysis, including the Constant-Q Transform (CQT), the Nonstationary Gabor Transform (NSGT), and the sliced Constant-Q Transform (sliCQT).

In Section 2.3, I will give an overview of machine learning and deep learning, and how these methods are typically applied to audio or musical applications, commonly in the form of convolutional neural networks (CNN) or recurrent neural networks (RNN).

In Section 2.4, I will give an overview of important concepts in software and code relevant to this thesis, including Python and Git version control.

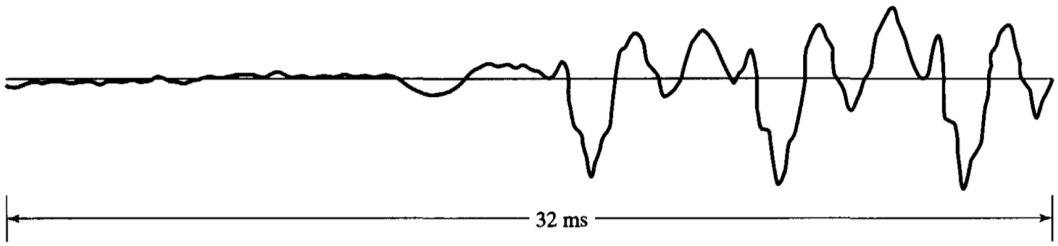
In Section 2.5, I will describe the tasks of music source separation and music demixing. I will provide a definition of the tasks, their motivations and uses, a survey of historical approaches to computational audio source separation, and trends in STFT-based music source separation leading to the current state of the art.

2.1 Acoustic signals and the time-domain waveform

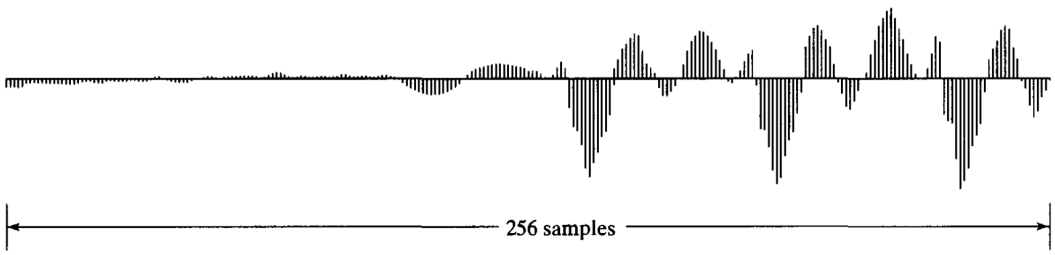
Oppenheim, Schafer, and Buck define a signal as a function of one or more variables which "conveys information about the state or behavior of a physical system" (Oppenheim, Schafer,

and Buck 1999, 8). Some examples of signals are a 2-dimensional image, which is a brightness function of two spatial variables, and a speech signal, which is a function of time. Moore describes the more specific case of a sound or audio signal as being the description of the vibration of an object and how it impresses this vibration upon the "surrounding medium (usually air) as a pattern of changes in pressure" (Moore 2013, 2).

According to Rabiner and Schafer (2010), the waveform is a natural and mathematically convenient representation of a signal. For acoustic signals, the waveform represents the continuously varying pattern of the signal as a function of the continuous variable t, which represents time. A continuous-time signal x, also called an analog signal, is denoted by $x_a(t)$. For digital processing, continuous-time signals need to be sampled periodically to form a sequence of numbers (Oppenheim, Schafer, and Buck 1999). The resultant domain is called discrete time, and a discrete-time signal is denoted by $x[n]$. A continuous-time signal and its discrete representation are shown in Figure 4.



(a) Continuous-time waveform.



(b) Discrete-time waveform.

Figure 4: A continuous-time signal and its discrete-time representation sampled with $T = 125\mu s$ (Oppenheim, Schafer, and Buck 1999, 10).

Continuous-time signals are converted into discrete-time representations by two processes: sampling and quantization. Points on the continuous time axis of the waveform are *sampled* periodically, and the amplitude values of the waveform at these sampled points are *quantized* to find the closest digital number (Rabiner and Schafer 2010). Time sampling is controlled by the sampling period T seconds, or the sampling rate $F_s = {}^1/_T$ Hz, which define the periodicity of the sampling. Amplitude quantization is controlled by the number of quantization levels 2^B, where B is the number of bits per sample of the representation.

First, continuous time needs to be sampled into the discrete time domain. The relationship of a discrete-time signal $x[n]$ of a continuous-time signal $x_a(t)$ is defined by $x[n] = x_a(nT)$, where n is an integer and $T = {}^1/_{F_s}$ is the sampling period. The Nyquist-Shannon sampling theorem (Oppenheim, Schafer, and Buck 1999), described independently by Nyquist (1928) and Shannon (1948), states that the maximum frequency of a signal that can be represented by a sampling rate F_s is $F_{\text{nyq}} = {}^{F_s}/_2$, which is also called the Nyquist rate or Nyquist frequency.

Aliasing is one of the pitfalls of choosing an inappropriate sampling rate for the frequencies present in the signal under observation (McClellan, Schafer, and Yoder 2003). The diagram in Figure 5 shows the phenomenon of aliasing due to undersampling, which occurs when a time-domain signal is undersampled, i.e., $F_{\text{sig}} > F_{\text{nyq}}$, and the continuous-time signal cannot be reconstructed accurately.

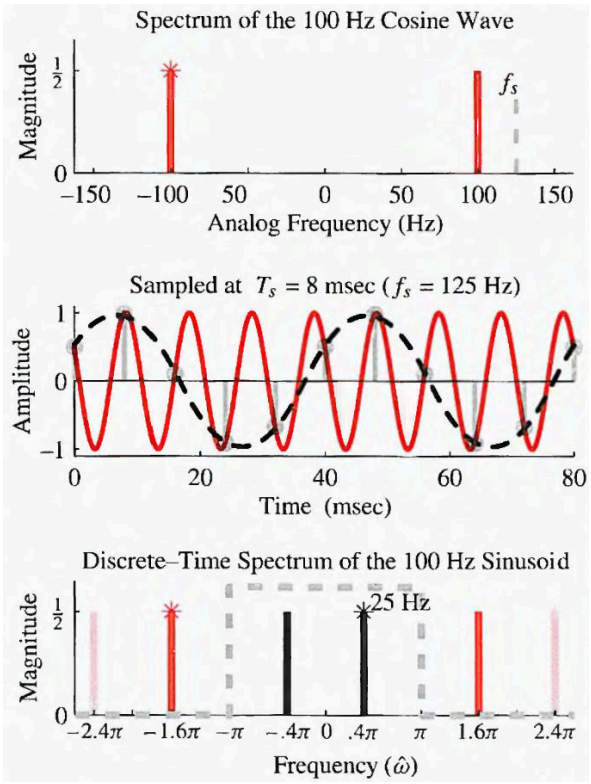
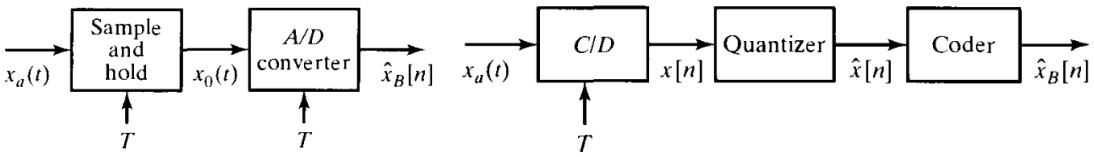


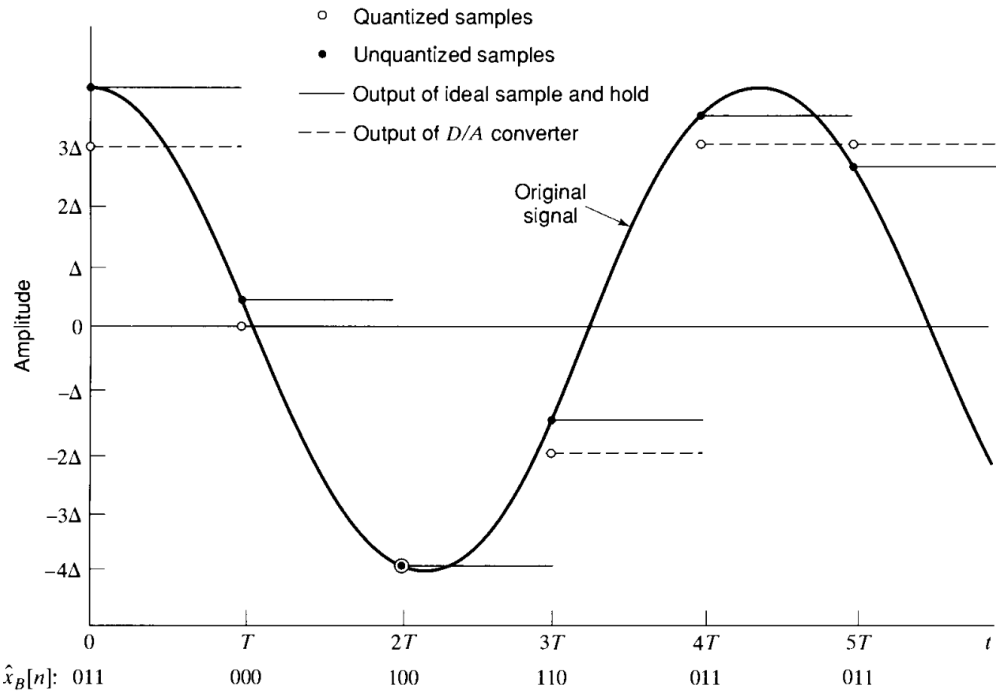
Figure 5: An undersampled cosine wave (red) and the resulting incorrect reconstruction (black) (McClellan, Schafer, and Yoder 2003, 82).

Second, the continuous amplitude needs to be quantized. Stated by Oppenheim, Schafer, and Buck, "the quantizer is a nonlinear system whose purpose is to transform the input sample $x[n]$ into one of a finite set of prescribed values" (Oppenheim, Schafer, and Buck 1999, 190). The second variable that relates to the quantization process is the number of

quantization levels. The quantization operation is represented as $\hat{x}[n] = Q(x[n])$, where $\hat{x}[n]$ is the quantized sample. Quantization levels can be defined to be uniform (evenly spaced) or nonuniform, and in essence the sample values are rounded to the nearest quantization level 2^B, recalling that B is the number of bits per sample in the representation (Oppenheim, Schafer, and Buck 1999). An A/D or analog-to-digital converter (ADC) circuit and its operation on a waveform is shown in Figure 6.



(a) Analog-to-digital converter (ADC). (b) ADC details: sampler and quantizer.



(c) Waveform results of the ADC process.

Figure 6: An ADC converter circuit showing the time sampling and amplitude quantizing operations (Oppenheim, Schafer, and Buck 1999, 188, 190, 192).

2.2 Transforms of acoustic signals

Moore states that:

> [a]lthough all sounds can be specified by their variation in pressure with time, it is often more convenient, and more meaningful, to specify them in a different way

> when the sounds are complex. This method is based on a theorem by Fourier, who proved that almost any complex waveform can be analyzed, or broken down, into a series of sinusoids with specific frequencies, amplitudes, and phases. This is done using a mathematical procedure called the Fourier Transform (Moore 2013, 4).

Throughout this section, the description of the Fourier Transform of acoustic signals and the evolution of further transforms that build on it will be covered in detail.

Also, throughout this section and the rest of this chapter, an example waveform will be used for illustrative purposes. This is the glockenspiel waveform,[9] which is used in various audio signal processing papers on the topic of time-frequency (Dörfler 2002; Balazs et al. 2011; Jaillet, Balazs, and Dörfler 2009; Jaillet and Torrésani 2007; Velasco et al. 2011; Siedenburg and Doerfler 2011), because the glockenspiel contains both tonal and transient properties, which have conflicting needs for time and frequency resolution in their analysis. Figure 7 shows the discrete-time waveform of the glockenspiel signal. Throughout this section, demonstrations of each transform will use this same glockenspiel signal as the input $x[n]$. The signal has a total duration of 5.94 seconds, and is sampled with a rate of 44,100 Hz.

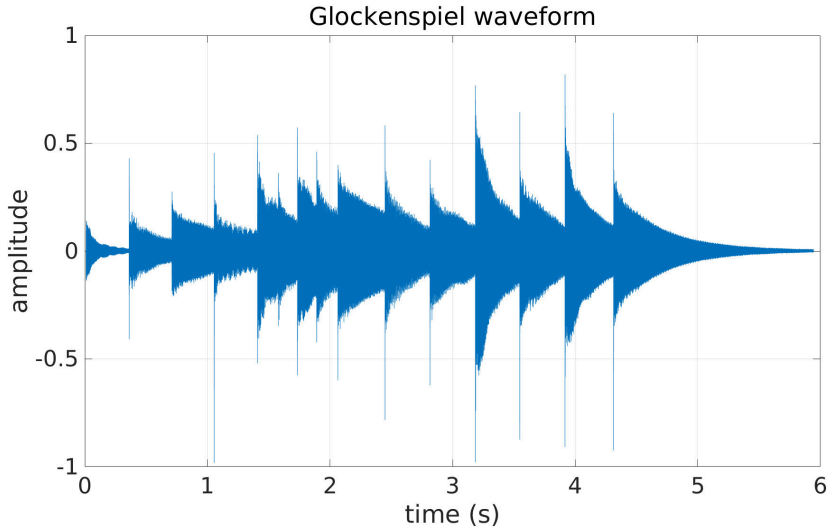
2.2.1 Frequency analysis and the Fourier Transform

The Fourier Transform originated as an integral transform in mathematics, which are a class of "useful tools for solving problems involving certain types of partial differential equations (PDE), mainly when their solutions on the corresponding domains of definition are difficult to deal with" (Domínguez 2016, 54). The Fourier Transform was originally introduced by Joseph Fourier in his earlier papers (Fourier 1807, 1811), and fully expanded and collected in his seminal work on heat (Fourier 1822). The connection of the Fourier Transform to music is described by Lostanlen, Andén, and Lagrange, who state that:

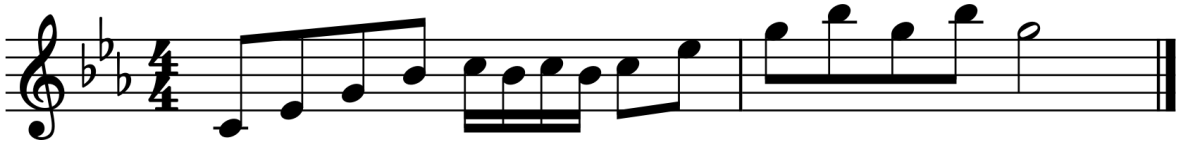
> [b]eyond the scope of thermal conduction, Joseph Fourier's treatise on the Analytical Theory of Heat (1822) profoundly altered our understanding of acoustic waves. It posits that any function of unit period can be decomposed into a sum of sinusoids, whose respective contribution represents some essential property of the underlying periodic phenomenon. In acoustics, such a decomposition reveals the resonant modes of a freely vibrating string (Lostanlen, Andén, and Lagrange 2019, 461).

The continuous-time Fourier Transform (CTFT) of a time-domain acoustic waveform is

9. https://ltfat.github.io/doc/signals/gspi.html



(a) Glockenspiel waveform.



(b) Glockenspiel notes.

Figure 7: Glockenspiel waveform in (a) playing a C minor pentatonic melody with 15 strikes. The struck notes are $C, Eb, G, Bb, C, Eb, G, Bb$, shown in (b).

defined by equation (1), and its inverse is defined by equation (2) (McClellan, Schafer, and Yoder 2003, 308):

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt \tag{1}$$

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}d\omega \tag{2}$$

McClellan, Schafer, and Yoder (2003) refer to $X(\omega)$ as the frequency-domain representation of the signal $x(t)$. Equation (2) defines the signal $x(t)$ in terms of a sum of infinitely many complex-exponential signals with $X(\omega)$ controlling the amplitude and phases of these signals. The continuous-time Fourier Transform provides a one-to-one mapping of the time domain to the frequency domain; it is a complex-valued function of ω, which is the variable that represents the angular frequency in radians. The Fourier Transform can be expressed in the rectangular form in equation (3) or polar form in equation (4) (Oppenheim, Schafer, and

Buck 1999, 49):

$$X(e^{j\omega}) = X_{\text{real}}(e^{j\omega}) + jX_{\text{imag}}(e^{j\omega}) \tag{3}$$

$$X(e^{j\omega}) = |X(e^{j\omega})|e^{j\angle X(e^{j\omega})} \tag{4}$$

The quantities $|X(e^{j\omega})|$ and $\angle X(e^{j\omega})$ are referred to as the magnitude and phase respectively. The Fourier Transform is also referred to as the spectrum, while its magnitude and phase are the magnitude and phase spectra, respectively (Oppenheim, Schafer, and Buck 1999).

In contrast to the CTFT, which maps a continuous-time signal $x(t)$ to a continuous frequency $X(\omega)$, the discrete-time Fourier Transform (DTFT) maps a discrete-time signal $x[n]$ to a continuous frequency $X(\omega)$. The DTFT is derived by sampling the CTFT in time, and is defined by equation (5) (Rabiner and Schafer 2010, 289):

$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{5}$$

Finally, the discrete Fourier Transform (DFT) maps a discrete-time signal $x[n]$ to a discrete frequency $X[k]$. The DFT is derived from the DTFT by evaluating (5) at a discrete set of equally spaced frequencies $\omega = (2\pi/N)k$, where $k = 0, 1, ..., N-1$, and considering it for a finite-length sequence $x[n]$ which is non-zero only within the interval $0 \leq n \leq L-1$. The DFT is defined by equation (6) (McClellan, Schafer, and Yoder 2003, 393):

$$X[k] = \sum_{n=0}^{L-1} x[n]e^{-j(\frac{2\pi}{N})kn} \tag{6}$$

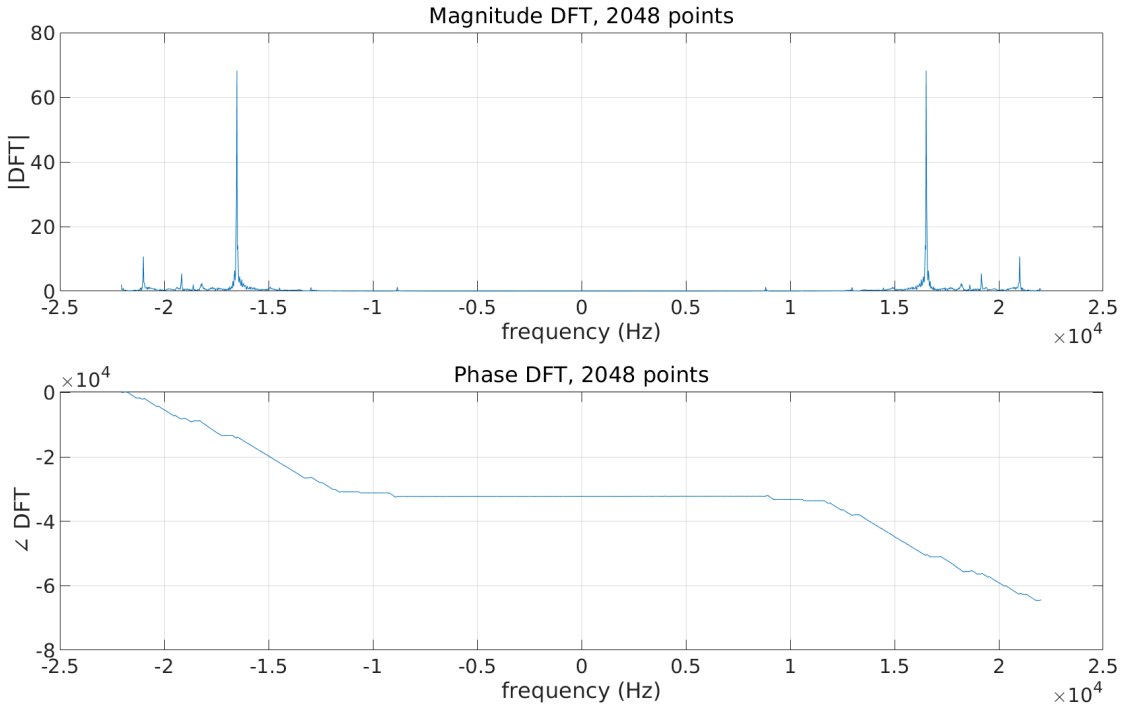
The number of points, or samples, of the DFT, determines the frequency resolution, also called *df* or Δf, which is the frequency spacing between each point in the resulting spectrum (Oppenheim, Schafer, and Buck 1999). The frequency resolution of an N-point DFT is $df = F_s/N$, where F_s is the sampling rate of the input signal. An illustration of the magnitude and phase spectra of the DFT are shown in Figure 8, using two different lengths of DFT to show the difference in the low and high frequency resolutions.

For a real-valued input signal $x[n]$, the spectrum has the conjugate symmetry property, i.e., the magnitudes of the DFT coefficients are mirrored symmetrically around the center frequency, while the phases of the DFT coefficients are mirrored anti-symmetrically around the center frequency (McClellan, Schafer, and Yoder 2003). This center bin corresponds to

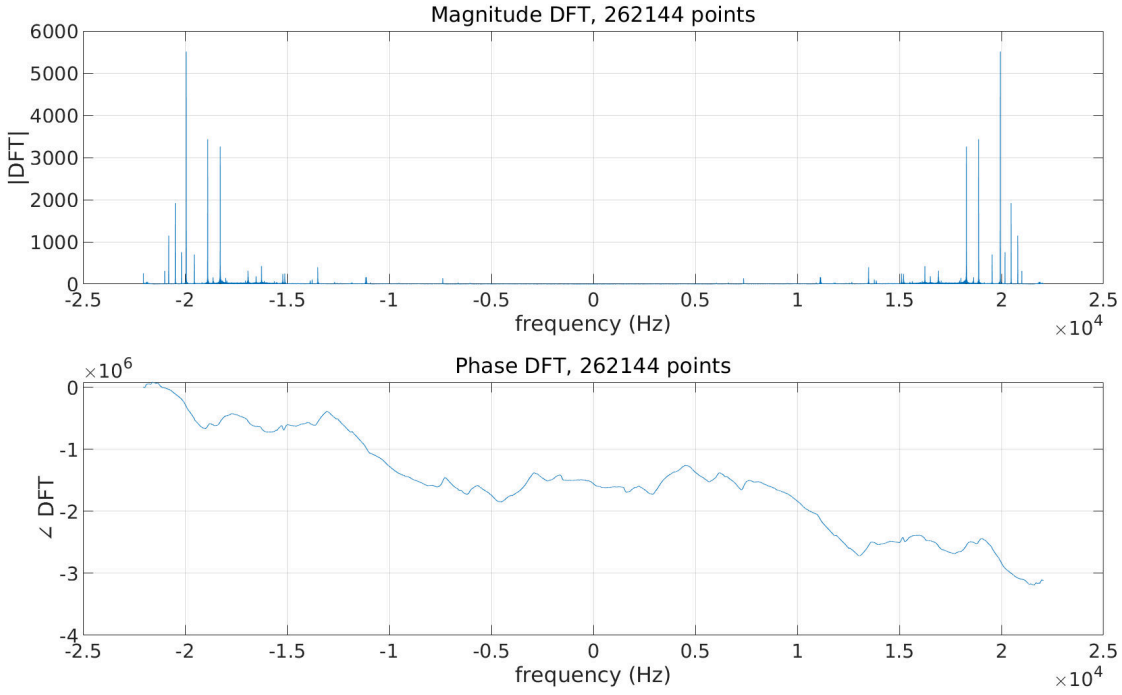
the Nyquist frequency $F_s/2$ where F_s is the sampling rate of the signal. For a given DFT of N points, the points of the output spectrum between 0–$N/2+1$ correspond to 0–$F_s/2$ Hz, and the points $N/2+1$–N correspond to the same frequency bins in reverse order, i.e., $F_2/2$–0 Hz. In practice, therefore, only the first $N/2+1$ points of the N-point DFT of a real signal are useful.

The CTFT is continuous in both time and frequency, and the DTFT is discrete in time and continuous in frequency. Signals need to be transformed from the continuous to the discrete domain via sampling to be processed digitally or computationally, which was covered previously in Section 2.1. Therefore, the DFT, which is discrete in both time and frequency, is used in digital computations (McClellan, Schafer, and Yoder 2003).

According to Skiena (2008), algorithms in computer science are often described by their time complexity using a hypothetical computer where simple operations take one time step. The "Big-O" notation, or $O(N)$, provides the upper bound of algorithm running time in relation to number of input elements. In Oppenheim, Schafer, and Buck (1999, Chapter 9) it is described that in its original formulation, the algorithmic complexity of the DFT is $O(N^2)$, or in other words, the running time of the algorithm grows proportionally the size of the input signal squared (Skiena 2008). Starting from legendary mathematician Carl Friedrich Gauss in 1805 (Heideman, Johnson, and Burrus 1985), and reaching its most famous formulation published by Cooley and Tukey (1965), a family of efficient algorithms for the computation of the DFT by computing a series of smaller DFTs, known collectively as the Fast Fourier Transform (FFT), reduced this computation time to $O(N \log N)$. This resulted in the FFT becoming one of the most important algorithms of the 20th century (Dongarra and Sullivan 2000).



(a) 2,048-point DFT, $df = 21.533Hz$.



(b) 262,144-point DFT, $df = 0.168Hz$.

Figure 8: DFT of the Glockenspiel waveform, using a low frequency resolution with 2,048 points in (a), and a high frequency resolution with 262,144 points in (b). Note the more detailed frequency components shown in the higher frequency resolution transform in (b).

2.2.2 Joint time-frequency analysis with the Gabor Transform and Short-time Fourier Transform (STFT)

Continuing from the discussion of the DFT in the previous Section 2.2.1, Oppenheim, Schafer, and Buck state that "often, in practical applications of sinusoidal signal models, the signal properties (amplitudes, frequencies, and phases) will change with time. For example, nonstationary signal models of this type are required to describe radar, sonar, speech, and data communication signals. A single DFT estimate is not sufficient to describe such signals..." (Oppenheim, Schafer, and Buck 1999, 714).

Dennis Gabor's seminal signal processing paper, *The Theory of Communication*, introduced significant and far-reaching concepts in the time and frequency analysis of acoustic signals (Gabor 1946). Gabor quotes famed American telecommunication engineer John Carson (Brittain 1996) to describe the limitations of the Fourier Transform:

> [t]he foregoing solutions [of the Fourier Transform], though unquestionably mathematically correct, are somewhat difficult to reconcile with our physical intuitions and our physical concepts of such variable frequency mechanisms as, for instance, the siren (Gabor 1946, 431).

According to Korpel, "Gabor came to the conclusion that the difficulty lay in our mutually exclusive formulations of time analysis and frequency analysis ... he suggested a new method of analyzing signals in which time and frequency play symmetrical parts" (Korpel 1982, 3624).

Gabor derived the principal of time-frequency uncertainty from the Heisenberg uncertainty principle in quantum physics, which states that the more precisely the position of an electron is determined, the less precisely the momentum is known, and conversely (Heisenberg, 1927, cited in Hall, 2006). Gabor states that "although we can carry out the analysis [of the acoustic signal] with any degree of accuracy in the time direction or frequency direction, we cannot carry it out simultaneously in both beyond a certain limit" (Gabor 1946, 432). This is referred to as the time-frequency uncertainty principle or the Gabor limit. Gabor defines the unit of time-frequency information $\Delta t \Delta f$ as the *logon*, where Δt and Δf are defined as "the uncertainties inherent in the definition of the epoch t and frequency f of an oscillation" (Gabor 1946, 432).

In order to demonstrate the mutually exclusive formulations of time and frequency, which leads to Gabor's time-frequency uncertainty principle, Figure 9(a) shows the unit impulse contrasted with the DC-component DFT. The DFT spectrum of a "sinusoid of zero frequency" (McClellan, Schafer, and Yoder 2003, 13) has only one nonzero value in the fre-

quency domain at the 0 Hz frequency component (direct current, or DC), but has an infinite extent in the time domain. The unit impulse is the "simplest [time-domain] sequence because it has only one nonzero value, which occurs at n = 0" (McClellan, Schafer, and Yoder 2003, 107), but has an infinite extent in the frequency domain.

Another way of visualizing the tradeoff of time and frequency is shown in Figure 9(b), where the frequency, or periodicity, of the sine wave is more apparent over longer periods of time Δt, but the detail of the individual time-domain sample values become lost.

The result of the time-frequency uncertainty principle is a consequence of how the Fourier Transform is used to swap between the mutually exclusive domains of time and frequency. Several psychacoustic studies have shown that humans can exhibit better time-frequency resolution than the Gabor limit, indicating that there are processes involved in the perception of sounds that cannot be explained by the Fourier Transform alone. Moore describes one of these experiments:

> It is concluded that models based on a place (spectral) analysis should be subject to a limitation of the type $\Delta f \cdot d \geq$ constant, where Δf is the frequency difference limen for a tone pulse of duration d. [...] It was found that at short durations the product of Δf and d was about one order of magnitude smaller than the minimum predicted [...] (Moore 1973, 610).

More recently, according to Oppenheim and Magnasco:

> [w]e have conducted the first direct psychoacoustical test of the Fourier uncertainty principle in human hearing, by measuring simultaneous temporal and frequency discrimination. Our data indicate that human subjects often beat the bound prescribed by the uncertainty theorem, by factors in excess of 10 (Oppenheim and Magnasco 2012, 4).

Oppenheim and Magnasco goes on to state that "most sound analysis and processing tools today continue to use models based on spectral theories... [w]e believe it is time to revisit this issue" (Oppenheim and Magnasco 2012, 4).

When performing joint time-frequency analysis, it is preferable to minimize time-frequency uncertainty, or to set the *logon* ($\Delta t \Delta f$) to its lowest possible value. Gabor asks:

> What is the shape of the signal for which the product $\Delta t \Delta f$ actually assumes the smallest possible value? [... it is] the modulation product of a harmonic oscillation of any frequency with a pulse of the form of the probability function (Gabor 1946, 435).

Gabor performed joint time-frequency analysis by multiplying overlapping, temporally consecutive portions of the input signal with shifted copies of the Gaussian window function (i.e., the probability function), and by taking the Fourier Transform of the windowed segments of the signal. The Gabor transform $G(f)$ of a discrete-time signal $x(n)$ is described by equations (7):

$$\mathbf{G}(\mathbf{f}) = [G_1(f), G_2(f), ..., G_k(f)] \tag{7}$$
$$G_m(f) = \sum_{n=-\infty}^{\infty} x(n)g(n - \beta m)e^{-j2\pi\alpha fn}$$

where $g(\cdot)$ is a Gaussian low-pass window function localized at 0, $G_m(f)$ is the DFT of the signal centered around time βm, and α and β control the time and frequency resolution of the transform (Rubinstein, Bruckstein, and Elad 2010).

The STFT, or Short-Time Fourier Transform, has been described independently from Gabor's work (Allen and Rabiner 1977), but additional research in the 1980s (Rubinstein, Bruckstein, and Elad 2010) led to the STFT being formalized and described as a special case of the Gabor transform, in recognition of Gabor's pioneering work. The STFT $X(f)$ of a discrete-time signal $x(n)$ is described by equations (8):

$$\mathbf{X}(\mathbf{f}) = [X_1(f), X_2(f), ..., X_k(f)] \tag{8}$$
$$X_m(f) = \sum_{n=-\infty}^{\infty} x(n)g(n - am)e^{-j2\pi fn}$$

where $g(\cdot)$ are the time-shifted, localized windows, $X_m(f)$ is the DFT of the audio signal centered about time am, and a is the hop size between successive time-shifts of the window (Rubinstein, Bruckstein, and Elad 2010). Note how similar equations (7) and (8) are, which is expected since the original Gabor transform is the STFT with a Gaussian window. In practice, the STFT allows the use of different windows and overlap sizes (Heinzel, Rüdiger, and Schilling 2002), as long as the constant overlap-add (COLA) constraint is respected (Griffin and Lim 1984). Figure 10 shows how a windowed Fourier Transform (i.e., Gabor transform or STFT) is performed on a waveform.

Figure 11 shows different sizes of *logon* in the time-frequency plane, and how the Gabor transform and the STFT with higher frequency or higher time resolution appear on the time-frequency plane.

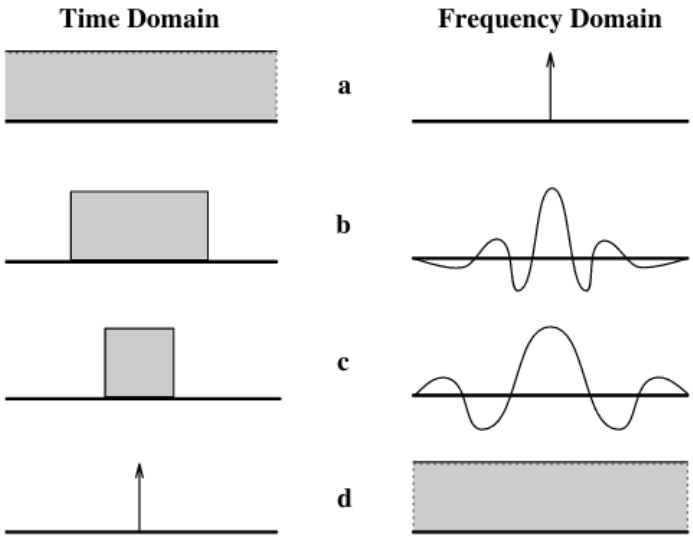
The STFT shown in equation (8) outputs a continuous frequency $X(f)$ from a discrete-time

signal $x(n)$. To use the STFT in digital computations, it must be sampled in frequency, in the same way that was shown for the DFT in equation (6). The frequency-sampled STFT has the same number of output frequency bins as the DFT. Its output is a matrix of columns where each column contains the DFT coefficients from one window of the input signal. The rows represent the frequency bins and the columns represent the time windows or frames. Section 2.2.1 described that half of the coefficients of an N-point DFT for a real-valued input signal $x[n]$ were redundant. The frequency-sampled STFT has the same behavior, such that it outputs $(N/2 + 1)$ non-redundant frequency bins. The frequency spacing between each bin is $df = F_s/N$, and the frequency in Hz corresponding to each bin is f (Hz) = bin $\times F_s/N$. The frequency bins 0 to $(N/2 + 1)$ therefore correspond to the frequency range of 0 to $(F_s/2)$ Hz. Figure 12 shows the rectangular matrix of coefficients which is the output of the STFT, and the frequency bins and frequencies in Hz corresponding to the rows of the matrix.

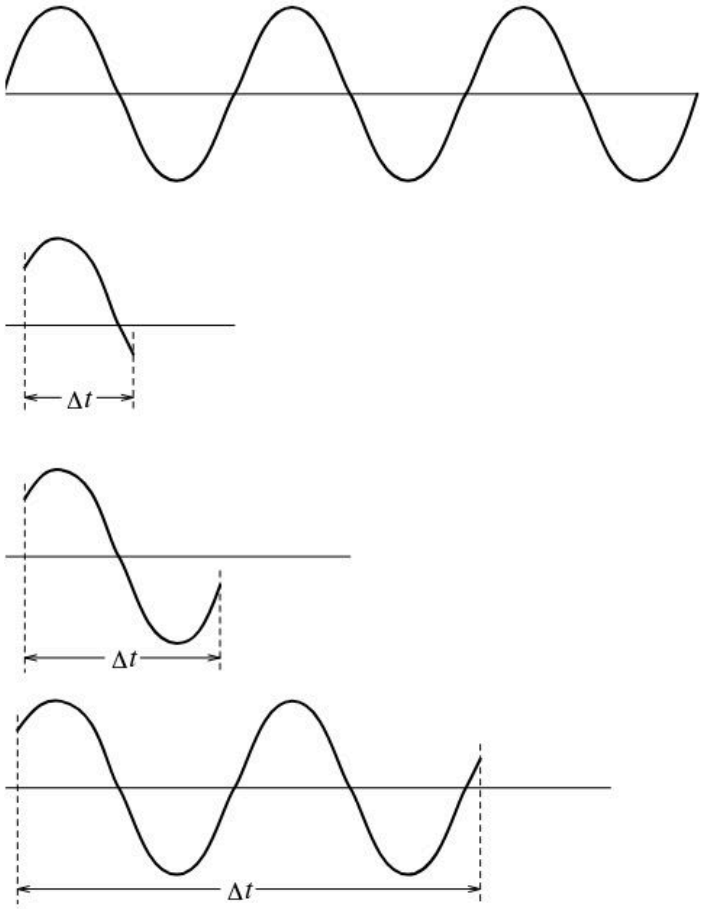
A key characteristic of the STFT is its fixed time-frequency resolution across the entire frequency spectrum. To change the time-frequency resolution, the window size must be changed. Figure 14 shows the STFT with the Hamming window and Figure 15 shows the Gabor transform, which is the STFT with the Gaussian window, intending to show that the STFT and Gabor transform are practically identical to each other. Three different window sizes are shown for each: 128 samples, 2,048 samples, and 16,384 samples, which at the sample rate of the glockenspiel signal (44,100 Hz) represent $128/44,100$ = 2.9 ms, 46.44 ms, and 371.52 ms respectively. The different window sizes show a visual demonstration of the time-frequency tradeoff. The Hamming window was chosen because it is the default window in the MATLAB `spectrogram` function.[10] The Gaussian and Hamming windows are shown together in Figure 13.

The time-frequency tradeoff of the STFT is a result of taking the Fourier transform of fixed-size windows applied to the input signal. In upcoming sections, I will show time-frequency transforms that use variable-size windows, resulting in a time-frequency resolution that varies across the frequency spectrum. These transforms are also called nonuniform, to distinguish from the uniform or fixed time-frequency resolution of the STFT. Section 2.2.3 introduces the Constant-Q Transform (CQT), a time-frequency transform designed for music analysis. Section 2.2.4 and Section 2.2.5 show the Nonstationary Gabor Transform (NSGT) and sliced Constant-Q Transform (sliCQT), which are time-frequency transforms with varying time-frequency resolution motivated by the CQT. Section 2.2.6 will show how nonuniform time-frequency transforms differ from the uniform STFT.

10. https://www.mathworks.com/help/signal/ref/spectrogram.html



(a) Mutually exclusive formulations of time and frequency by two extremes, the unit impulse (bottom left) and the 0 Hz cosine DFT spectrum (top right).



(b) The periodicity of the sine wave is more apparent with a longer Δt, at the expense of temporally localized samples.

Figure 9: Time-frequency tradeoff intuitions (Maclennan 1994, 103, 106).

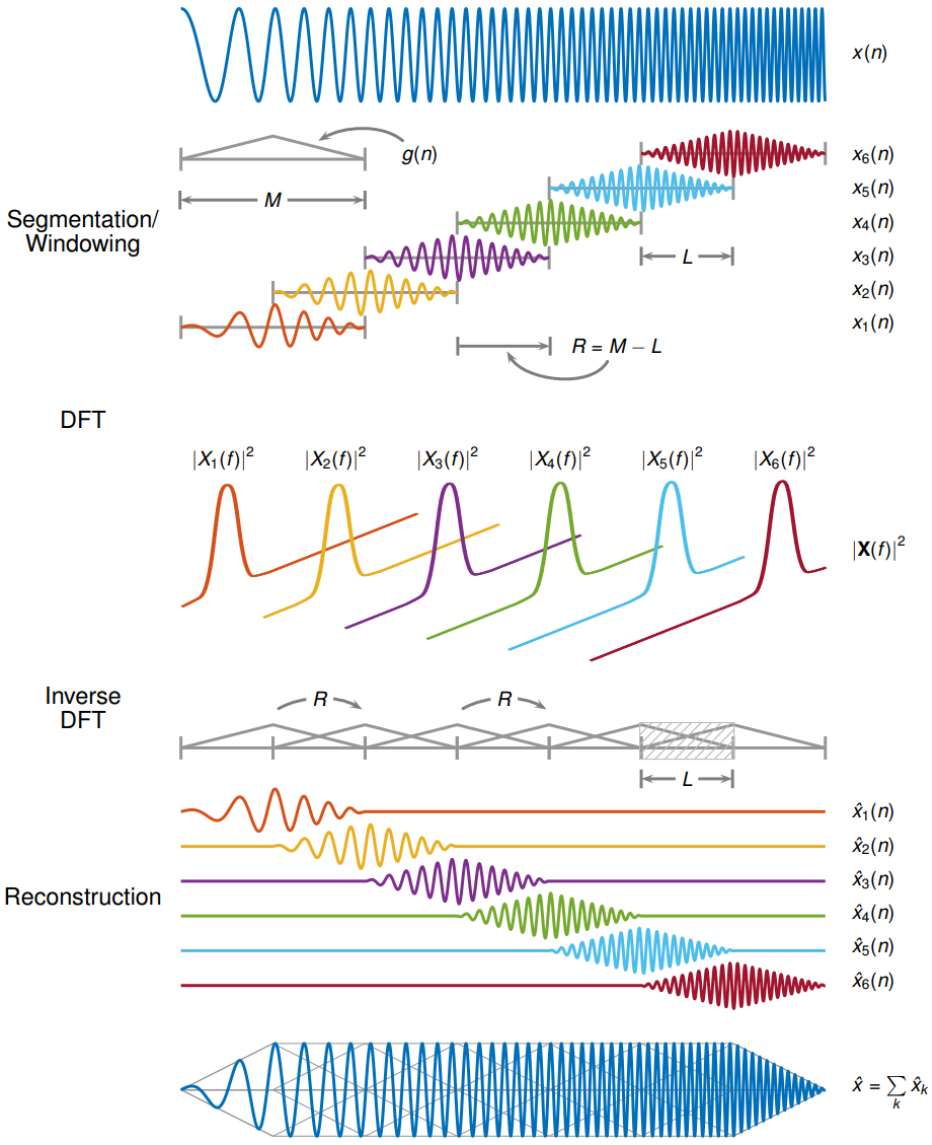
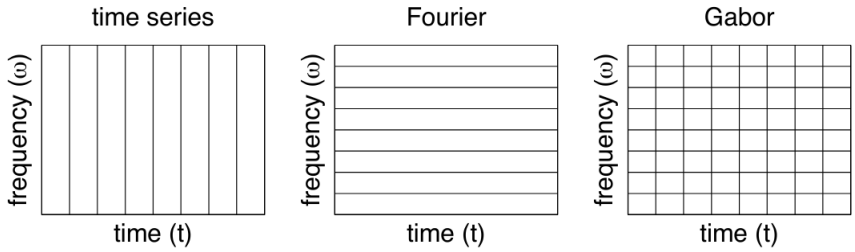
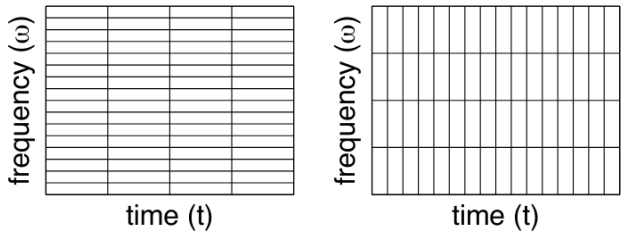


Figure 10: Forward and inverse STFT or Gabor transform.*

*. https://www.mathworks.com/help/signal/ref/iscola.html



(a) Pure time domain, pure frequency domain, and Gabor's time-frequency tiles of $\Delta t \Delta f = 1$.



(b) High frequency resolution vs. high time resolution.

Figure 11: Different tiling of the time-frequency plane (Kutz 2013, 326, 327).

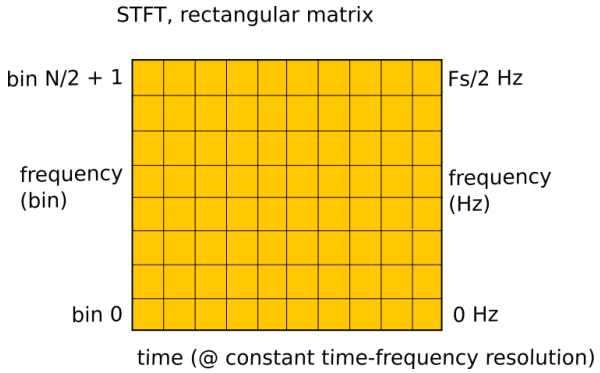


Figure 12: Rectangular matrix output of the frequency-sampled STFT.

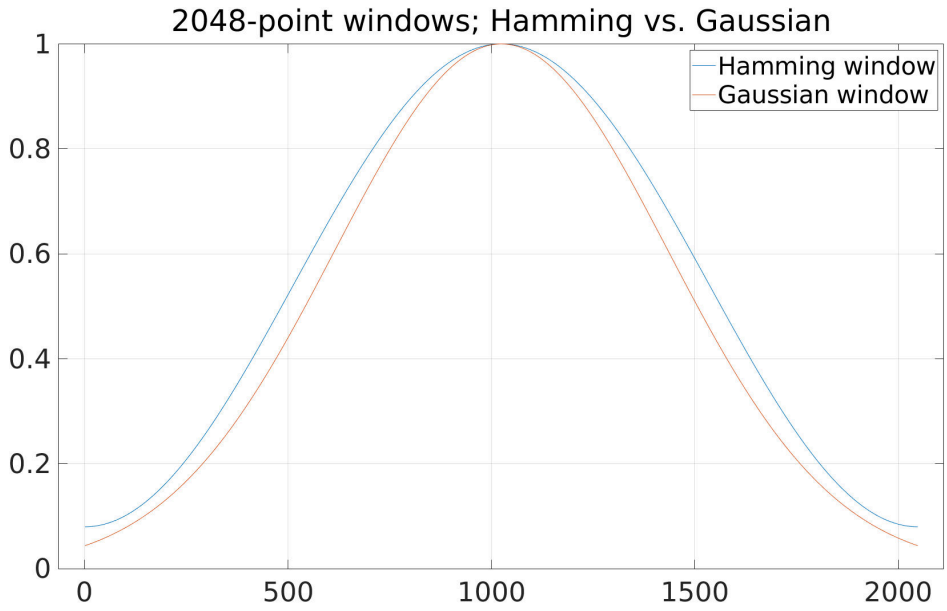
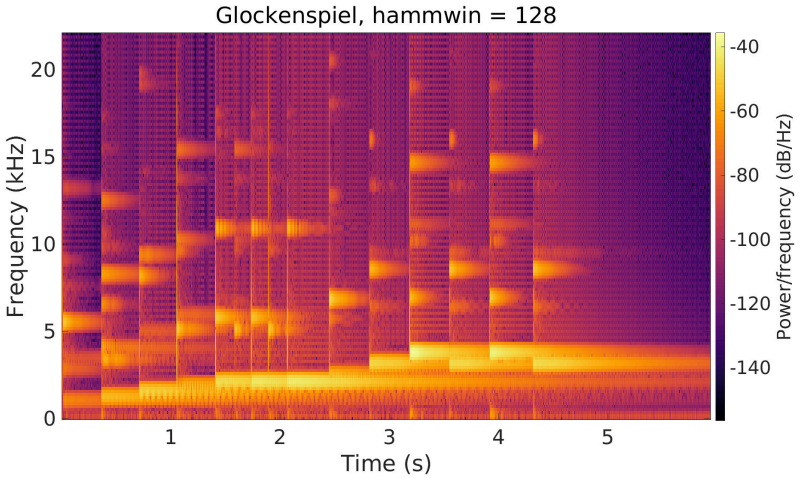
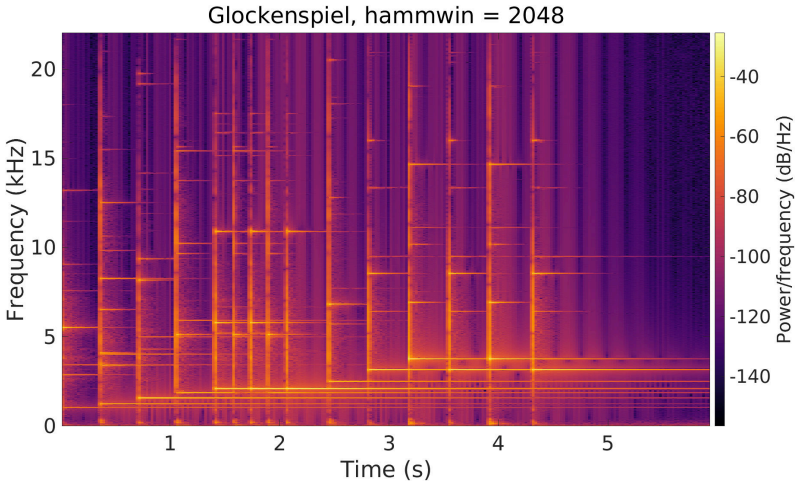


Figure 13: 2,048-sample Hamming and Gaussian windows. The Gaussian window is truncated and uses a width factor of 2.5.*

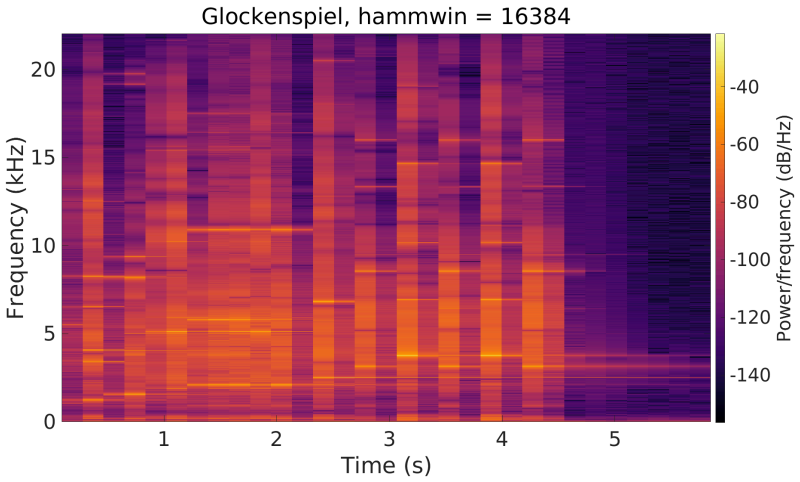
*. https://www.mathworks.com/help/signal/ref/gausswin.html



(a) STFT, 128-sample Hamming.

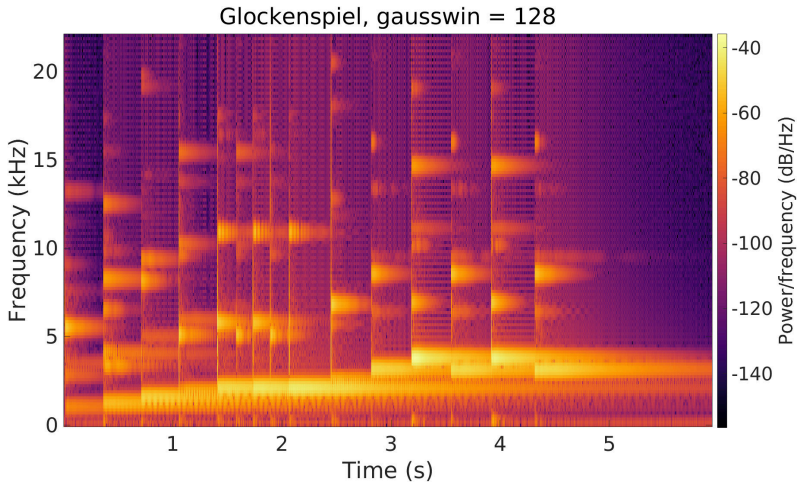


(b) STFT, 2,048-sample Hamming.

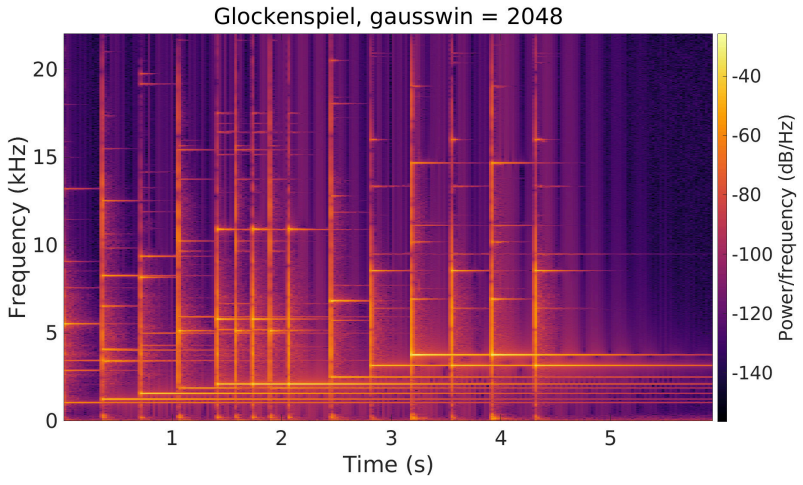


(c) STFT, 16,384-sample Hamming.

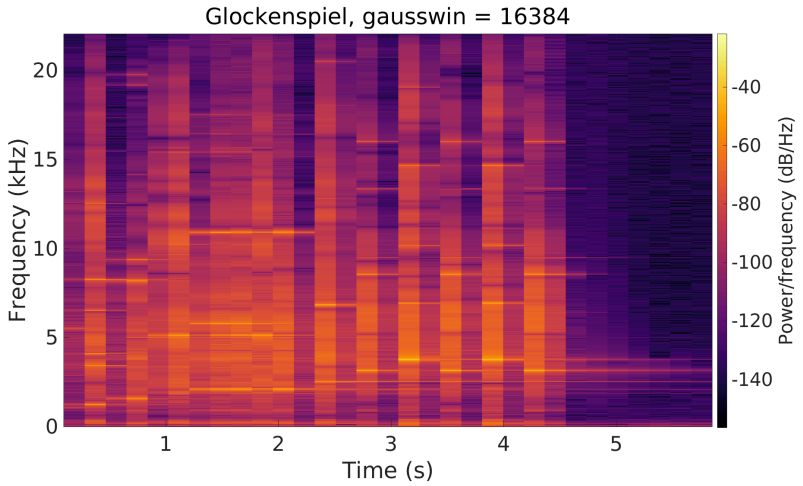
Figure 14: Magnitude spectrograms with the Hamming window STFT, shown in order of the smallest to largest window size. The horizontal lines (i.e., frequency components) are becoming sharper from the increasing frequency resolution, while the vertical lines (i.e., temporal events or note onsets) are becoming blurrier from the decreasing time resolution.



(a) Gabor transform, 128-sample Gaussian.



(b) Gabor transform, 2,048-sample Gaussian.



(c) Gabor transform, 16,384-sample Gaussian.

Figure 15: Magnitude spectrograms with the Gabor transform, i.e., the STFT with a Gaussian window. Note that they look identical to the STFT with a Hamming window in Figure 14. The different window functions are compared in Figure 13.

2.2.3 Constant-Q Transform (CQT)

Schörkhuber, Klapuri, and Sontacchi stated that the problem with the STFT is its "rigid time-frequency resolution trade-off providing a constant absolute frequency resolution throughout the entire range of audible frequencies" (Schörkhuber, Klapuri, and Sontacchi 2012, 1). As described in Section 1.1, music should be analyzed with a high frequency resolution in the low-frequency region, and with a high time resolution in the high-frequency region (Dörfler 2002; Schörkhuber, Klapuri, and Sontacchi 2012).

The Constant-Q Transform (CQT) was proposed by Brown (1991) and Brown and Puckette (1992) to analyze musical signals with a logarithmic frequency scale that matched the notes of a musical pitch scale. A demonstration of a violin analyzed with the DFT and CQT was shown in Figure 2.

The name "Constant-Q" refers to the constant ratio of the frequency being analyzed to the frequency resolution of analysis, or $f/\delta f = Q$, such that the frequency resolution increases with the center frequency to maintain the Q factor. Q comes from the term "quality factor," used to characterize resonant structures in physics (Christopoulos et al. 2019); Moore also uses Q to describe the spectral bandwidth of the human auditory system (Moore 2013).

The CQT uses long-duration windows in the low frequency regions and short-duration windows in the high frequency regions, resulting in good time resolution for transients (Schörkhuber, Klapuri, and Sontacchi 2012). Figure 16 shows some properties of the linearly-spaced DFT spectrum compared to the CQT, and the different window sizes used for each frequency bin of interest. Equations (9) describe how the different windows $W[n, k]$ are applied to the signal:

$$\begin{aligned} \text{window length} = N[k] &= \frac{f_s}{f_k} Q \\ W[k, n] &= \rho + (1 - \rho) \cos\left(\frac{2\pi n}{N[k]}\right) \end{aligned} \tag{9}$$

The parameter ρ in the window function $W[k, n]$ in equation (9) should be set to 0.5 for a Hann window or $25/46$ for a Hamming window.

This first implementation of the CQT was not designed to be invertible, until Schörkhuber and Klapuri (2010) introduced an algorithm for an approximate inverse of the CQT back to the time-domain waveform, with an inversion error of $\approx 10^{-3}$. The approximate inverse for the CQT was also approached differently by Fitzgerald, Cranitch, and Cychowski (2006).

Schörkhuber and Klapuri describe the CQT as "a time-frequency representation where the frequency bins are geometrically spaced and the Q-factors (ratios of the center frequencies to bandwidths) of all bins are equal" (Schörkhuber and Klapuri 2010, 1). Frequency bins that are geometrically spaced are also logarithmically spaced with a base of two (Diniz et al. 2006). The bins-per-octave setting of the CQT is related to the Constant-Q ratio by the formula $Q = 2^{1/\text{bins}}$. For example, for 12 bins-per-octave, this results in a Q factor of 1.059, which is the well-known "12th root of two" based on the Western chromatic scale with equal temperament (Wilkerson 2014; Jacoby et al. 2019).

Velasco et al. (2011) implemented the CQT with the Nonstationary Gabor Transform (NSGT) (Balazs et al. 2011), which allowed for perfect reconstruction for the first time. This is also called the CQ-NSGT, or Constant-Q Nonstationary Gabor Transform.

Holighaus et al. (2013) followed up with a realtime algorithm, the *sliCQ* or "sliced Constant-Q" transform (sliCQT), which can process the input signal in fixed-size slices, as opposed to operating on the entire input signal like the NSGT. Finally, Schörkhuber et al. (2014) introduced the Variable-Q scale and improved the phase of the transform. From Velasco et al. (2011), the total frequency bins of the CQ-NSGT or sliCQT can be derived from the bins-per-octave, and the minimum and maximum frequencies of the desired frequency scale, shown in equation (10):

$$K = [B \log_2 (\frac{\xi_{\text{max}}}{\xi_{\text{min}}}) + 1] \tag{10}$$

where K is the total bins of the CQT, B is the bins-per-octave, and $\xi_{\text{min,max}}$ are the minimum and maximum frequencies. For example, for $B = 12$ bins-per-octave, $\xi_{\text{min}} = 83$ Hz and $\xi_{\text{max}} = 22{,}050$ Hz, the result is $K \approx 97$ total frequency bins.

In the landscape of music analysis libraries, Essentia,[11] LTFAT,[12] and the MATLAB Wavelet Toolbox[13] have converged on the CQ-NSGT (Velasco et al. 2011; Holighaus et al. 2013; Schörkhuber et al. 2014). However, librosa[14] still uses the older implementation with the approximate reconstruction (Schörkhuber and Klapuri 2010). Finally, Velasco et al. (2011) have released an open-source reference Python implementation[15] of the CQ-NSGT and the sliCQT.

11. https://essentia.upf.edu/
12. https://ltfat.org/
13. https://www.mathworks.com/help/wavelet/ref/cqt.html
14. https://librosa.org/
15. https://github.com/grrrr/nsgt

Several examples of magnitude spectrograms of the glockenspiel signal are generated using the STFT, shown in Figure 17, and the CQT, shown in Figure 18. The spectrograms were generated from the standard STFT spectrogram in MATLAB[16] and the CQ-NSGT implementation of the CQT from the MATLAB Wavelet Toolbox.[17] Note how the CQT spectrograms show more spectral information than the STFT spectrograms at all frequency resolutions.

As mentioned throughout this section, the best implementation choice for the CQT is the NSGT, which allows for a perfect inverse transform. Section 2.2.4 will introduce and describe the general NSGT, and Section 2.2.5 will introduce and describe the sliCQT, which is a realtime variant of the NSGT. The NSGT and sliCQT allow the analysis of audio with an arbitrary nonuniform frequency scale not limited to the Constant-Q scale, which will give me more freedom in this thesis to find a frequency scale that performs best for music source separation applications.

16. https://www.mathworks.com/help/signal/ref/spectrogram.html

17. The CQT is included in the MATLAB Wavelet Toolbox because the CQT is considered to be a type of wavelet transform (Schörkhuber and Klapuri 2010). Wavelet transforms are an important topic in time-frequency analysis, and have many uses in audio and musical applications. However, they are beyond the scope of this thesis.

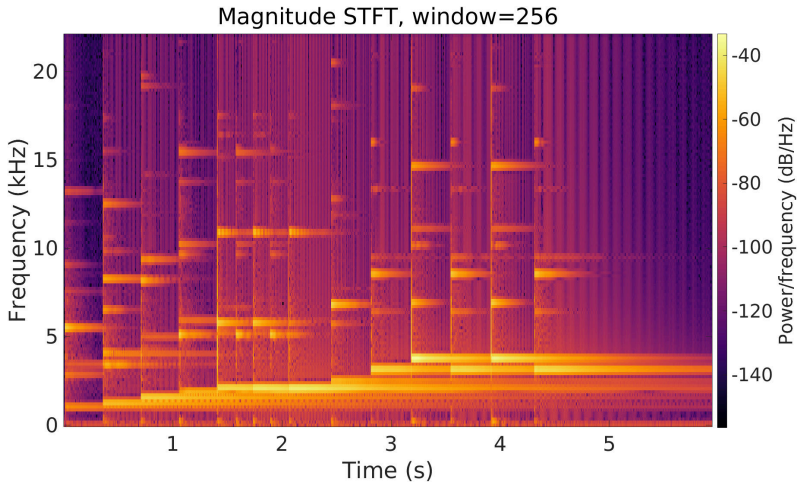
	Constant Q	DFT
Frequency	$(2^{1/24})^k \cdot f\min$ exponential in k	$k\,\Delta f$ linear in k
Window	variable $= \mathrm{N[k]} = \frac{SR \cdot Q}{f_k}$	constant $= N$
Resolution		
Δf	variable $= f_k/Q$	constant $=$ SR/N
$\frac{f_k}{\Delta f_k}$	constant $= Q$	variable $= k$
Cycles in Window	constant $= Q$	variable $=$ k

(a) Properties of DFT, CQT.

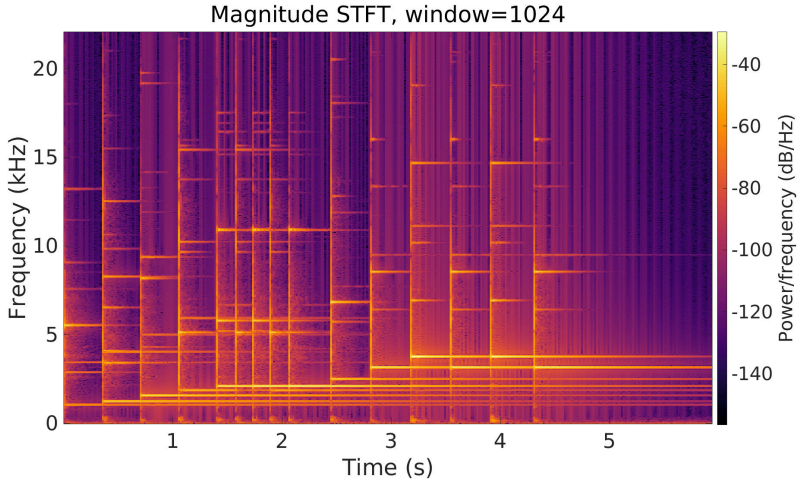
Channel	Midinote	Frequency (Hz)	Window (Samples)	(ms)
0	53	175	6231	195
6	56	208	5239	164
12	59	247	4406	138
18	62	294	3705	116
24	65	349	3115	97
30	68	415	2619	82
36	71	494	2203	69
42	74	587	1852	58
48	77	699	1557	49
54	80	831	1309	41
60	83	988	1101	34
66	86	1175	926	29
72	89	1398	778	24
78	92	1664	1308	41
84	95	1978	1100	34
90	98	2350	926	29
96	101	2797	778	24
102	104	3327	654	20
108	107	3956	550	17
114	110	4710	462	14
120	113	5608	388	12
126	116	6675	326	10
132	119	7942	274	9
138	122	9461	230	7
144	125	11 216	194	6
150	128	13 432	162	5

(b) Window sizes for computing the CQT.

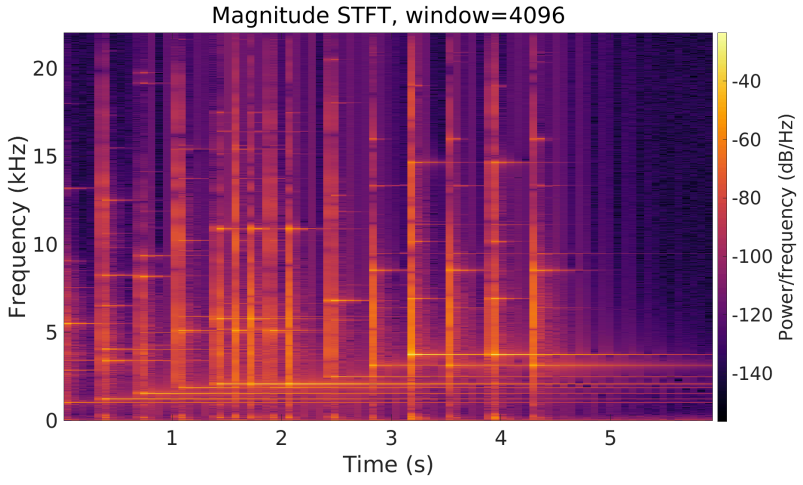
Figure 16: Various aspects of the original CQT (Brown 1991, 427, 428).



(a) STFT spectrogram, window size 256.

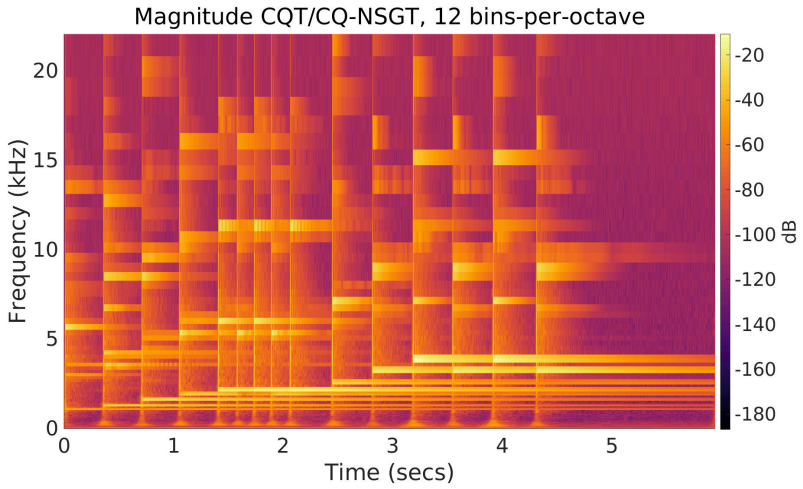


(b) STFT spectrogram, window size 1,024.

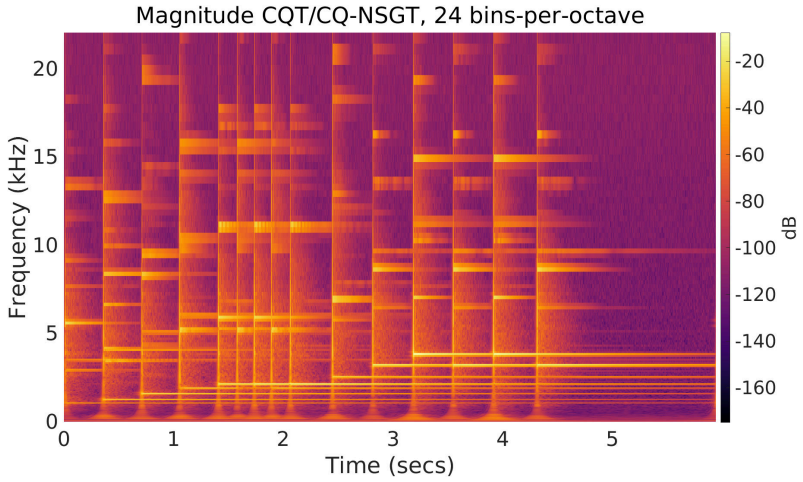


(c) STFT spectrogram, window size 4,096.

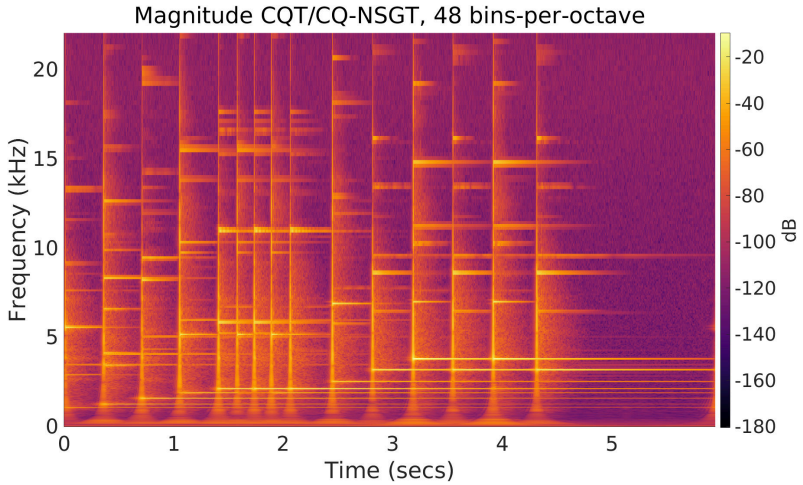
Figure 17: STFT magnitude spectrograms of the glockenspiel signal. Three different window sizes are shown in increasing order, demonstrating an increasing frequency resolution (sharper horizontal lines) and decreasing time resolution (blurrier vertical lines).



(a) CQT spectrogram, 12 bins-per-octave.



(b) CQT spectrogram, 24 bins-per-octave.



(c) CQT spectrogram, 48 bins-per-octave.

Figure 18: CQT magnitude spectrograms of the glockenspiel signal. Three different bins-per-octave are shown in increasing order. The time and frequency resolution is varied within a single spectrogram, unlike the fixed time-frequency resolution of each spectrogram in Figure 17. Using more bins-per-octave increases the highest frequency resolution and decreases the lowest time resolution.

2.2.4 Nonstationary Gabor Transform (NSGT)

Dörfler (2002) in their dissertation analyzed music with multiple Gabor dictionaries (i.e., STFTs). Figure 19 shows the time-frequency tradeoff of the short and long window STFT analyses of the glockenspiel signal for its transient and tonal characteristics.

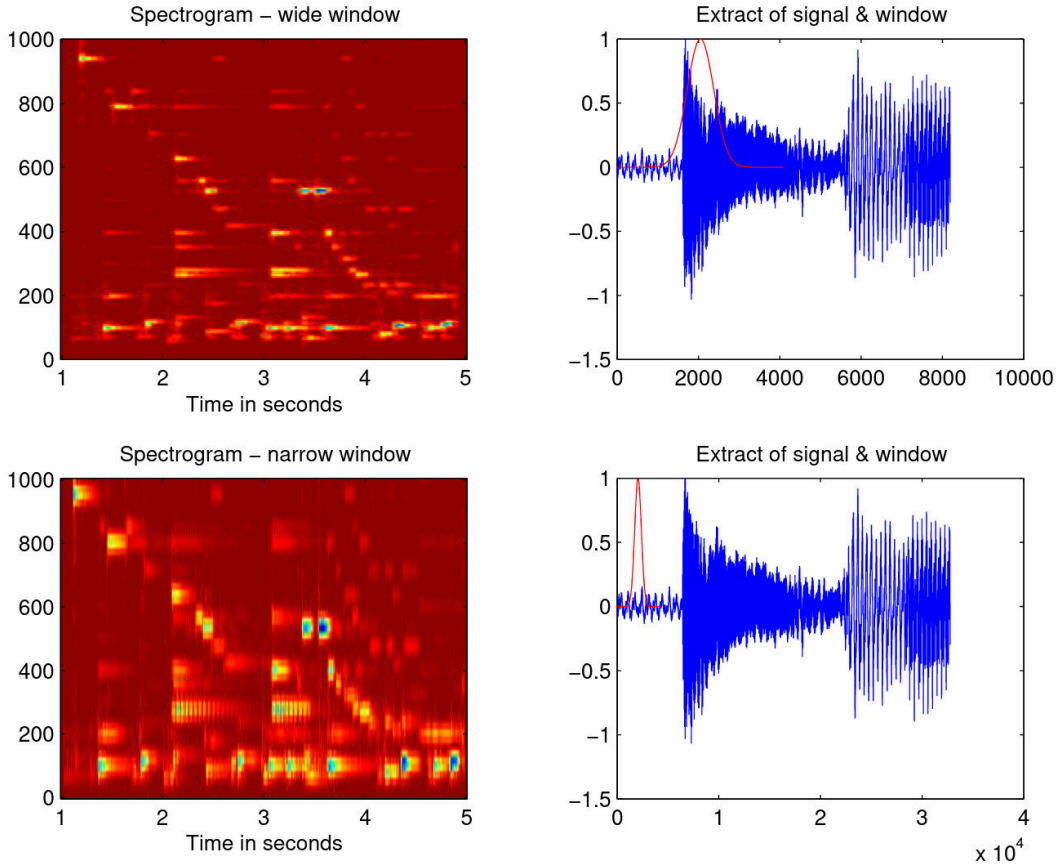


Figure 19: Time-frequency tradeoff for a glockenspiel signal (Dörfler 2002, 20).

As summarized by Liuni et al., "[t]he definition of multiple Gabor frames, which is comprehensively treated in [Dörfler 2002], provides Gabor frames with analysis techniques with multiple resolutions... The nonstationary Gabor frames [...] are a further development; [...] they provide for a class of FFT-based algorithms [...] together with perfect reconstruction formulas" (Liuni et al. 2013, 2). In other words, a single Nonstationary Gabor transform can replace the need for multiple distinct Gabor transforms (or STFTs).

Rubinstein, Bruckstein, and Elad describe the shift from the term *transform*, (e.g., STFT), to *dictionary*, stating that works by Mallat and Zhang 1993 and Chen, Donoho, and Saunders 2001 began a "fundamental move from transforms to dictionaries for [...] signal representation" (Rubinstein, Bruckstein, and Elad 2010, 1049). Accordingly, an important outcome

of this terminology change was the "idea that a signal was allowed to have more than one description in the representation domain, and that selecting the best one depended on the task" (Rubinstein, Bruckstein, and Elad 2010, 1049).

Using multiple transforms, such as using two Gabor transforms (or STFTs) with a small and large window for the transient and tonal properties of music, is also called an *overcomplete* dictionary (Rubinstein, Bruckstein, and Elad 2010), where there is a lot of redundancy in the transform domain.

The advantage of the CQT over such approaches is that it contains these desirable transform properties in one single transform. The NSGT is a time-frequency transform with varying time-frequency resolution, whose motivating application is the CQT (Jaillet, Balazs, and Dörfler 2009; Balazs et al. 2011). It is constructed from frame theory (Balazs et al. 2017), a mathematical technique for computing redundant, stable ways of representing a signal (Kovačević and Chebira 2008). The NSGT can use nonuniform time and frequency spacing in its time-frequency analysis of a signal, and it has a perfect inverse operation with practically no reconstruction error.

The construction of nonstationary Gabor frames relies on three properties of the windows and time-frequency shift parameters used (Balazs et al. 2011, 2):

- "The signal f of interest is localized at time- (or frequency-) positions n by means of multiplication with a compactly supported (or limited bandwidth, respectively) window function g_n"
- "The Fourier Transform is applied on the localized pieces $f \cdot g_n$. The resulting spectra are sampled densely enough in order to perfectly re-construct $f \cdot g_n$ from these samples"
- "Adjacent windows overlap to avoid loss of information. At the same time, unnecessary overlap is undesirable. We assume that $0 < A \leq \sum_{n \in \mathbb{Z}} |g_n(t)|^2 \leq B < \infty$, a.e. (almost everywhere), for some positive A and B"

These requirements lead to invertibility of the frame operator and therefore to perfect reconstruction. Balazs et al. continues on to say that

> [m]oreover, the frame operator is diagonal and its inversion is straightforward. Further, the canonical dual frame has the same structure as the original one. Because of these pleasant consequences following from the three above-mentioned requirements, the frames satisfying all of them will be called painless nonstationary Gabor frames and we refer to this situation as the painless case (Balazs et al. 2011, 1482)

The derivation of the NSGT starts from the definition of the Gabor transform from Section 2.2.2. In the standard Gabor transform, the same window function (also called the Gabor atom or Gabor function) is shifted in time to cover entire signal (Liuni et al. 2013), described by equation (11):

$$g_{m,n}(t) = g(t - na)e^{2\pi jmbt} \tag{11}$$

As stated by Liuni et al., "[w]e will indicate such a frame as *stationary*, since the window used for time-frequency shifts does not change and the time-frequency shifts form a lattice of $a \times b$" (Liuni et al. 2013, 3). Figure 20 shows the resulting uniform $a \times b$ tiling of the time-frequency plane.

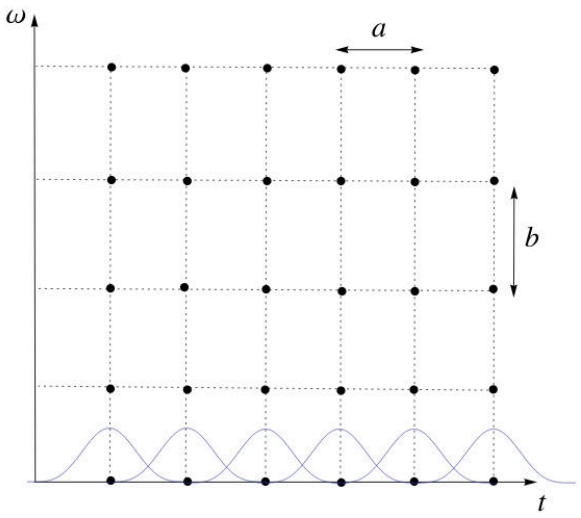


Figure 20: Uniform time-frequency resolution of the stationary Gabor transform (Liuni et al. 2013, 3).

The definition of the multiple Gabor systems of Dörfler (2002) continues from the stationary Gabor transform. Given the stationary Gabor atom, where a and b are the time-frequency shift parameters and $f(t)$ is the reconstructed input signal, the Gabor transform is described by equations (12):

$$\begin{aligned} g_{m,n}(t) &= g(t - na)e^{j2\pi mbt}, m, n \in \mathbb{Z} \\ f(t) &= \sum_{m,n\in\mathbb{Z}} c_{m,n}g_{m,n}(t) \end{aligned} \tag{12}$$

An overcomplete system that uses R distinct STFTs (or Gabor transforms) with different window sizes, where a_r and b_r are the time-frequency shift parameters for each window size,

can be described with equations (13):

$$g^r_{m,n}(t) = g(t - na_r)e^{j2\pi mb_r t}, m, n \in \mathbb{Z}$$
$$f(t) = \sum_{r=0}^{R-1} \sum_{m,n\in\mathbb{Z}} c^r_{m,n} g^r_{m,n}(t) \tag{13}$$

For a resolution that changes with time, the Nonstationary Gabor atom is chosen from a set of functions $\{g_n\}$ and a fixed frequency sampling step b_n, shown in equation (14):

$$g_{m,n}(t) = g_n(t)e^{j2\pi mb_n t}, m, n \in \mathbb{Z} \tag{14}$$

Balazs et al. describe the system in equation (14) further:

> [...] the functions $\{g_n\}$ are well-localized and centered around time-points a_n. This is similar to the standard Gabor scheme [...] with the possibility to vary the window g_n for each position a_n. Thus, sampling of the time-frequency plane is done on a grid which is irregular over time, but regular over frequency at each temporal position (Balazs et al. 2011, 1485).

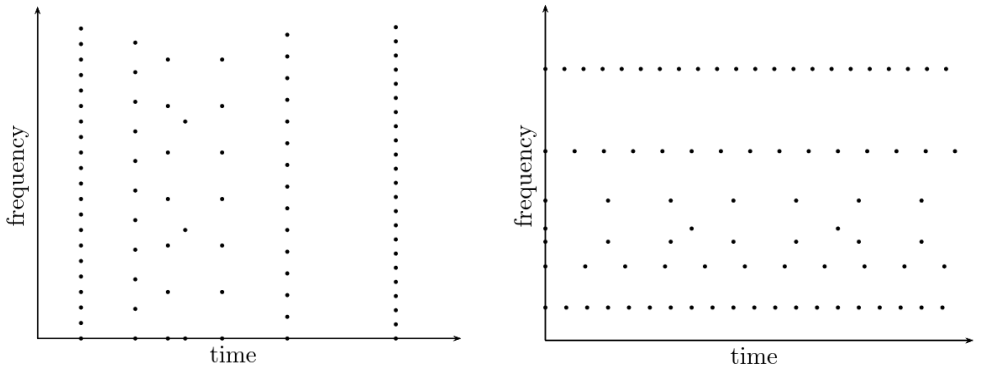
For a resolution that changes with frequency, the Nonstationary Gabor atom is chosen from a family of functions $\{h_m\}$, which are "well-localized band-pass functions with center frequency b_n" and a fixed time sampling step a_m, shown in equation (15) (Balazs et al. 2011, 1486):

$$h_{m,n}(t) = h_m(t - na_m), m, n \in \mathbb{Z} \tag{15}$$

Figure 21 show both the cases of the varying resolution by time and by frequency in terms of sampled points on the time-frequency grid.

The NSGT coefficients for a signal of length L are given by an FFT of length M_n for each window g_n. For each window, there are L window operations and $O(M_n \cdot \log(M_n))$ FFT operations. The overall algorithmic complexity of the NSGT is therefore $O(N \cdot (M \log(M)))$ for N windows, where $M = \max(M_n)$.

I believe that the general NSGT is more interesting to study than the CQ-NSGT (Constant-Q NSGT), as it can be constructed with nonuniform frequency scales besides the Constant-Q scale, such as the psychoacoustic mel and Bark scales (Rabiner and Schafer 2010), or the Variable-Q scale, which combines the Constant-Q scale with the psychoacoustic concept of the equivalent rectangular bandwidth (ERB) (Schörkhuber et al. 2014; Huang, Dong, and Li



(a) Gabor atoms that change with time.

(b) Gabor atoms that change with frequency.

Figure 21: Varying time-frequency sampling of the Nonstationary Gabor transform (Balazs et al. 2011, 1485, 1487).

2015). In essence the NSGT can be thought of as a filterbank (Schörkhuber et al. 2014).

Throughout the rest of this thesis, the NSGT and sliCQT implementation used will be the open-source, reference Python library.[18] The reference library contains two example scales: the Constant-Q/logarithmic scale, and the mel psychoacoustic scale. Figure 22 demonstrates the NSGT with the Constant-Q scale, and Figure 23 demonstrates the NSGT with the mel scale. For both figures, a low and high frequency resolution transform using 100 and 500 total frequency bins, respectively, is generated, using a frequency range of 20–22,050 Hz. Details of the frequency scales used to generate the NSGT spectrograms are shown in Table 1.

In addition to the Constant-Q and mel scales included in the reference library, I will describe additional frequency scales that may be interesting for music or audio analysis.

The Variable-Q scale (Schörkhuber et al. 2014; Huang, Dong, and Li 2015) is the same as the Constant-Q scale, except with a small fixed frequency offset, denoted by gamma or γ (Hz), added to each frequency bin. Schörkhuber et al. provide the motivation for the Variable-Q scale:

> ... [The] CQT has several advantages over STFT when analysing music signals. However, one considerable practical drawback is the fact that the analysis/synthesis atoms get very long towards lower frequencies. This is unreasonable both from a perceptual viewpoint and from a musical viewpoint. Auditory filters in the human auditory system are approximately Constant-Q only for frequen-

18. https://github.com/grrrr/nsgt

> cies above 500 Hz and smoothly approach a constant bandwidth towards lower frequencies. Accordingly, music signals generally do not contain closely spaced pitches at low frequencies, thus the Q-factors (relative frequency resolution) can safely be reduced towards lower frequencies, which in turn improves the time resolution (Schörkhuber et al. 2014, 5).

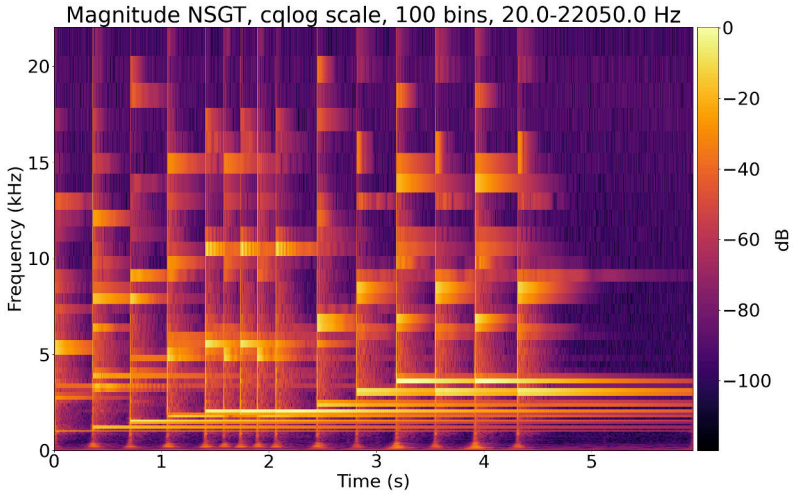
Both Schörkhuber et al. (2014) and Huang, Dong, and Li (2015) cite the ERBlet transform (Necciari et al. 2013) as a psychoacoustic transform, which in turn motivates the Variable-Q Transform. Schörkhuber et al. (2014) and Huang, Dong, and Li (2015) provide the following equations (16) for computing the bandwidth B_k of the frequency bin (or filter channel) k where b is the bins-per-octave or bpo:

$$B_k = \sigma f_k + \gamma, \sigma = 2^{\frac{1}{b}} - 2^{\frac{1}{b}} \tag{16}$$

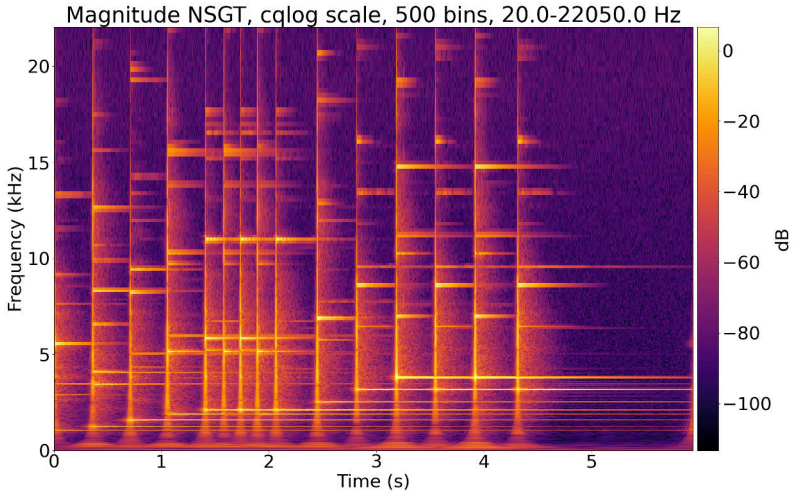
The effect is to widen the bandwidths of the windows at the low frequency bins significantly, since the offset is comparable in its order of magnitude to the lower frequency bandwidths. Schörkhuber et al. (2014) show some examples with $\gamma = [0, 3, 6.6, 10, 30]$ Hz. As the center frequency increases, the effect of the small offset becomes negligible. This results in a widening of the Q-factors in the low frequency bins, but it becomes Constant-Q in the high frequency bins.

Finally, the Bark psychoacoustic scale is introduced to complement the included mel scale, because the Bark scale has been used with success in music source separation (Litvin and Cohen 2011) and in percussion instrument classification (Herrera, Dehamel, and Gouyon 2003). In this thesis, I use the formula shown in equation (17) to convert between Bark and Hz frequencies (Voran 2008):

$$f_{\text{Bark}} = 6 \cdot arcsinh\Big(\frac{f_{\text{Hz}}}{600}\Big), f_{\text{Hz}} = 600 \cdot sinh\Big(\frac{f_{\text{Bark}}}{6}\Big) \tag{17}$$

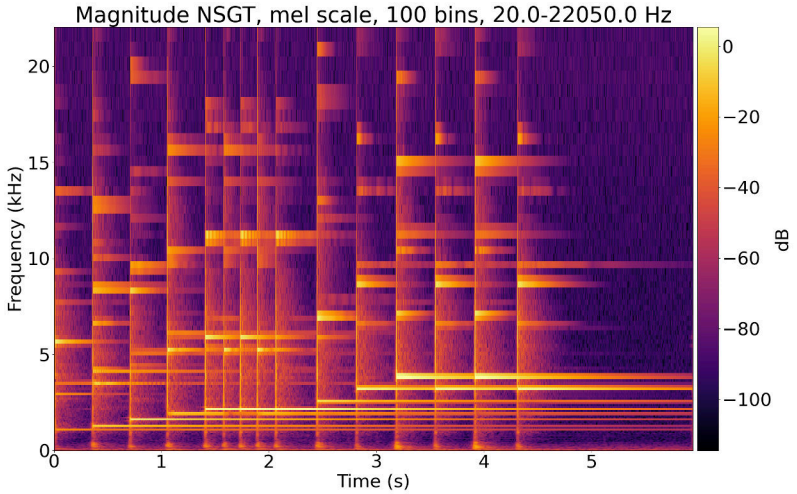


(a) NSGT, Constant-Q scale, 100 bins.

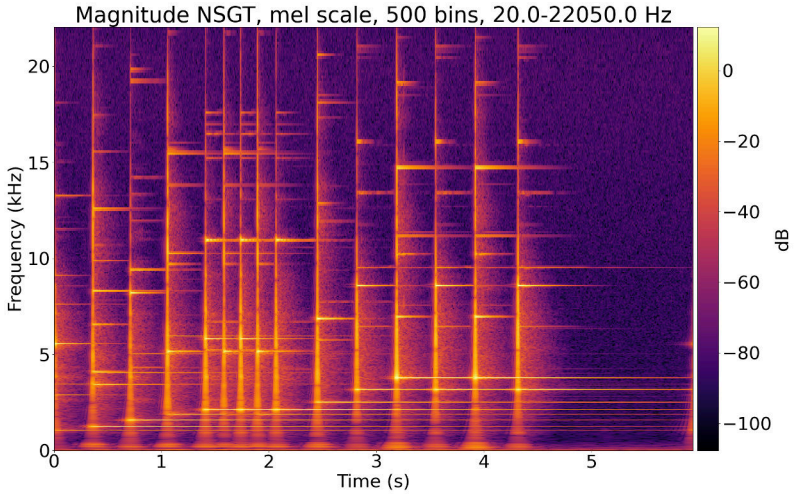


(b) NSGT, Constant-Q scale, 500 bins.

Figure 22: Constant-Q NSGT spectrograms of the glockenspiel signal.



(a) NSGT, mel scale, 100 bins.



(b) NSGT, mel scale, 500 bins.

Figure 23: mel-scale NSGT spectrograms of the glockenspiel signal.

Table 1: Different frequencies and Q-factors for various NSGT scales.

Scale	Bins	First five frequency bins
Constant-Q	100	20.00, 21.47, 23.04, 24.73, 26.54
mel	100	20.00, 45.56, 72.02, 99.42, 127.80
Constant-Q	500	20.00, 20.28, 20.57, 20.86, 21.16
mel	500	20.00, 25.00, 30.03, 35.10, 40.21

Scale	Bins	Last five frequency bins
Constant-Q	100	16,614.38, 17,832.63, 19,140.20, 20,543.64, 22,050.00
mel	100	19,087.44, 19,789.79, 20,517.07, 21,270.17, 22,050.00
Constant-Q	500	20,845.91, 21,140.62, 21,439.50, 21,742.61, 22,050.00
mel	500	21,428.92, 21,582.58, 21,737.31, 21,893.11, 22,050.00

Scale	Bins	First five Q-factors
Constant-Q	100	7.06, 7.06, 7.06, 7.06, 7.06
mel	100	0.40, 0.88, 1.34, 1.78, 2.21
Constant-Q	500	35.62, 35.62, 35.62, 35.62, 35.62
mel	500	2.01, 2.49, 2.97, 3.45, 3.92

Scale	Bins	Last five Q-factors
Constant-Q	100	7.06, 7.06, 7.06, 7.06, 7.06
mel	100	13.83, 13.85, 13.86, 13.88, 13.89
Constant-Q	500	35.62, 35.62, 35.62, 35.62, 35.62
mel	500	69.97, 69.98, 70.00, 70.02, 70.03

2.2.5 sliCQ Transform (sliCQT)

The NSGT processes the entire input signal at once. In cases where the input signal must be processed in fixed-size chunks, such as realtime streaming, the sliCQ Transform (sliCQT, or sliced Constant-Q Transform) was created (Velasco et al. 2011; Holighaus et al. 2013). The slicing operation is shown in Figure 24.

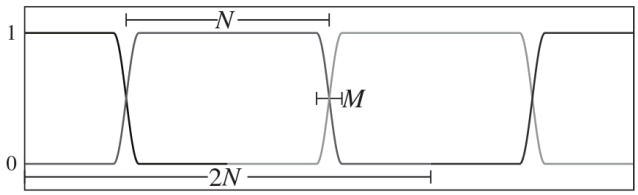


Figure 24: Slicing the input signal with 50% overlapping Tukey windows. N is the slice length and M is the transition area (Holighaus et al. 2013).

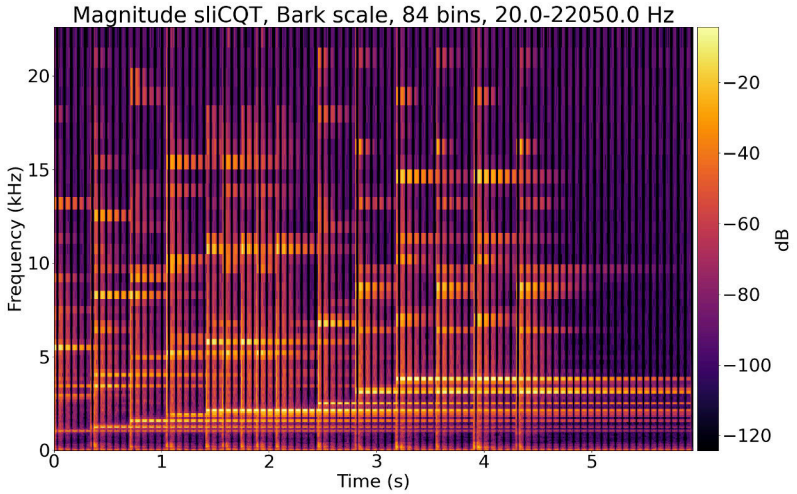
Additionally, the "slicing windows are symmetrically zero-padded to length $2N$, reducing time-aliasing significantly" (Holighaus et al. 2013, 10), with the effect that adjacent slices need to be 50% overlap-added with each other to create the final spectrogram. There is no inverse operation for the 50% overlap-add provided in the paper. Figure 25 demonstrates this characteristic of the sliCQT.

Note the terminology of the sliCQT, which uses slice length instead of the window length of the STFT, and transition length or transition area instead of the overlap or hop length of the STFT. Slice and transition are used to distinguish the larger slice-wise operation of the sliCQT from the local windowing and overlapping done *within* each slice to achieve the desired time-frequency resolution. The real overlap between slices can vary up to a maximum of the transition area.

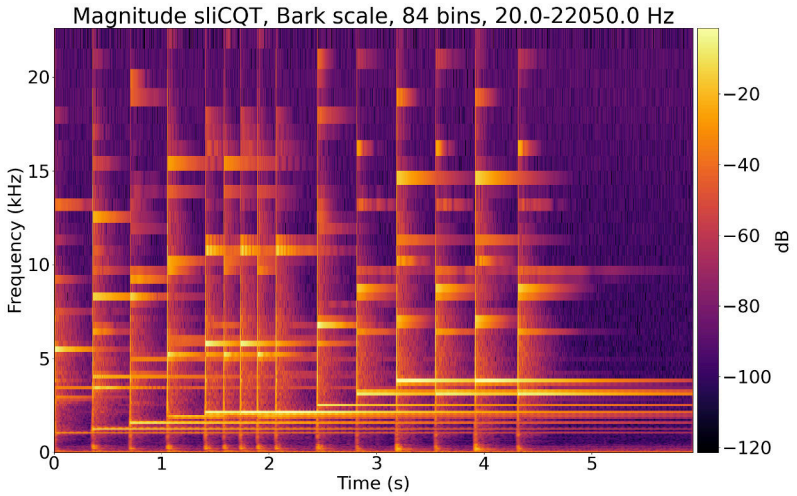
2.2.6 Ragged time-frequency transforms

Schörkhuber and Klapuri state that the consequence of nonuniform frequency analysis, such as that seen in the CQT or NSGT, leads to:

> ... a data structure that is more difficult to work with than the time-frequency matrix (spectrogram) obtained by using Short-Time Fourier transform in successive time frames. The last problem is due to the fact that in CQT, the time resolution varies for different frequency bins, in effect meaning that the "sampling" of different frequency bins is not synchronized (Schörkhuber and Klapuri 2010, 1).



(a) Adjacent slices placed side-by-side without overlap-adding.



(b) Adjacent slices with 50%-overlap-add.

Figure 25: sliCQT spectrograms demonstrating the necessary slice 50% overlap-add due to the symmetric zero-padding of each slice.

This same statement also applies to the NSGT, where different frequency bins have a different time resolution. An illustration is shown in Figure 26. The shape of the transform can be referred to as an irregular or ragged matrix.[19]

Note that to produce a spectrogram like the NSGT in Figure 26, the slices returned by the sliCQT must be overlap-added as described in Section 2.2.5 and shown in Figure 25. The NSGT outputs the spectrogram directly, without any further modifications needed.

19. https://xlinux.nist.gov/dads/HTML/raggedmatrix.html

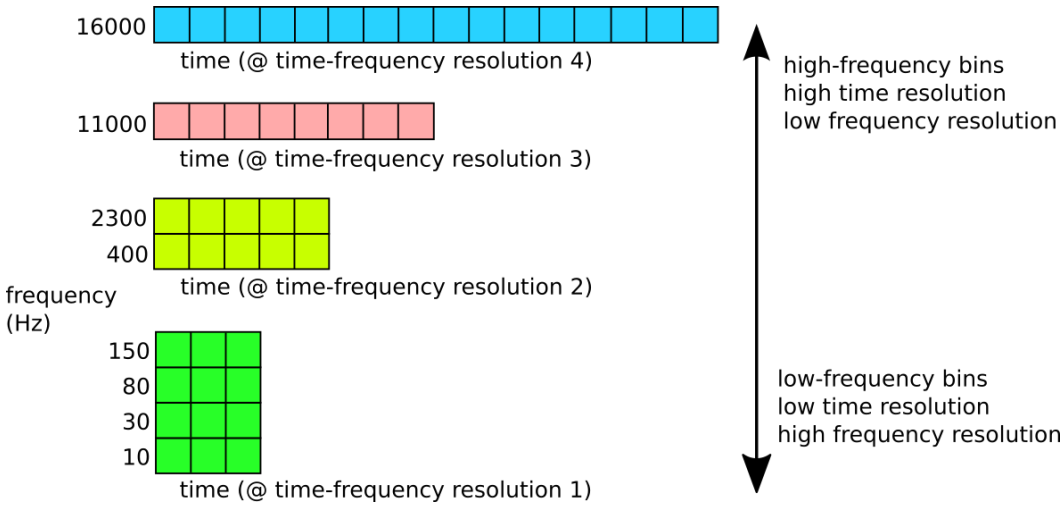


Figure 26: Illustration of the ragged NSGT, where frequency bins are grouped by their time resolution. The green-colored matrix represents the frequency bins analyzed with the highest frequency resolution and lowest time resolution, resulting in frequencies spaced close together and the lowest number of temporal frames. Moving upwards, the yellow, pink, and blue matrices have a decreasing frequency resolution and increasing time resolution, resulting in an increasing spacing between frequencies and an increasing number of temporal frames.

2.3 Machine learning and deep learning for music signals

Machine learning (ML) is a technique for modeling complex, unknown systems from data (Bastanlar and Ozuysal 2014, 105):

> In many scientific disciplines, the primary objective is to model the relationship between a set of observable quantities (inputs) and another set of variables that are related to these (outputs). [...] Machine learning provides techniques that can automatically build a computational model of these complex relationships by processing the available data and maximizing a problem dependent performance criterion.

According to Beysolow II, deep learning (DL) "is the subfield of machine learning that is devoted to building algorithms that explain and learn a high and low level of abstractions of data that traditional machine learning algorithms often cannot" (Beysolow II 2017, 1). A characteristic of deep networks is that they have many hidden layers, named so "because we do not necessarily see what the inputs and outputs of these neurons are explicitly beyond knowing they are the output of the preceding layer," (Beysolow II 2017, 2) to model more complex relationships between the input and output. Figure 27 shows a deep network with hidden layers.

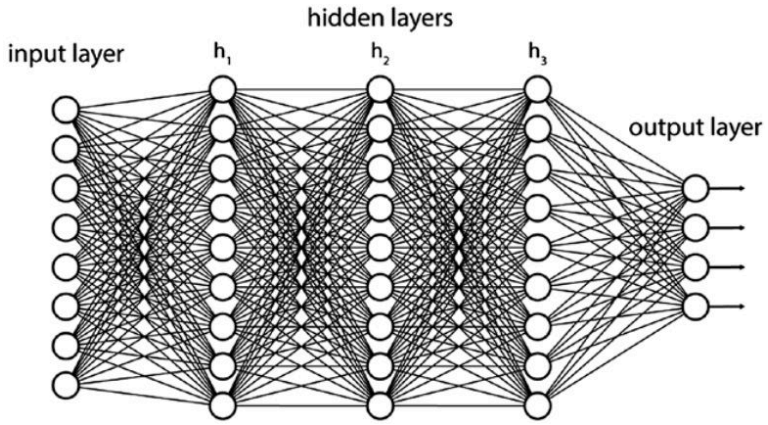


Figure 27: Deep neural network with hidden layers (Beysolow II 2017, 2).

Beysolow II states that "most machine learning algorithms as they exist now focus on function optimization, and the solutions yielded do not always explain the underlying trends within the data nor give the inferential power that artificial intelligence was trying to get close to" (Beysolow II 2017, 1). However, machine learning methods have achieved success in many fields including natural language processing (Johri et al. 2021), computer vision (Esposito and Malerba 2010), and audio (Purwins et al. 2019), indicating that data-driven

approaches are useful, despite falling short of general artificial intelligence (Iman, Arabnia, and Branchinst 2020).

Optimization techniques have been used in statistical, approximate, or stochastic signal processing (Pereyra et al. 2016; Gray and Davisson 2004) for decades (Mattingley and Boyd 2010). Some of the classic examples include basis pursuit and matching pursuit (Mallat and Zhang 1993; Chen, Donoho, and Saunders 2001). Signal processing techniques and machine and deep learning are compatible with one another and can be used together (Rehr and Gerkmann 2019, 2016). According to Bianco et al., signal processing and machine learning should both be considered in the study of acoustic signals, shown in Figure 28:

> Whereas physical models are reliant on rules, which are updated by physical evidence (data), machine learning (ML) is purely data-driven. By augmenting ML methods with physical models to obtain hybrid models, a synergy of the strengths of physical intuition and data-driven insights can be obtained (Bianco et al. 2019, 3591).

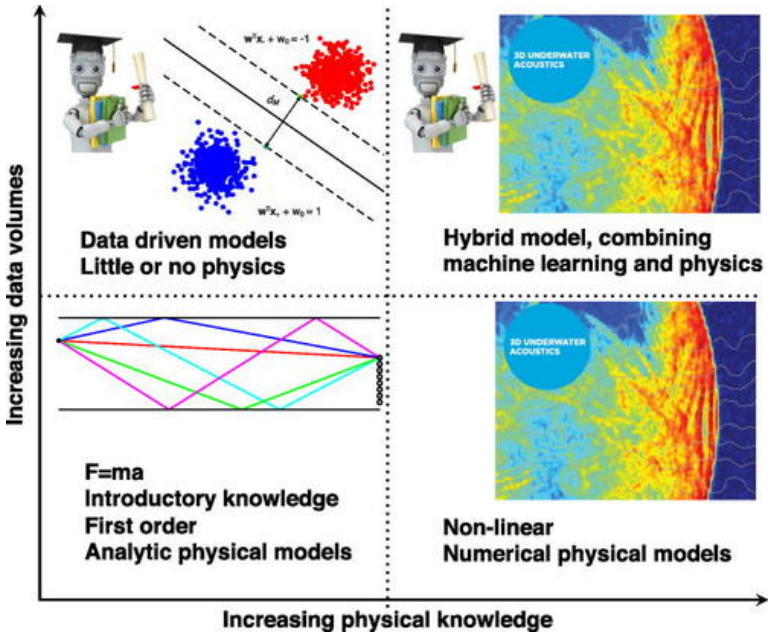


Figure 28: Synergy of physical models and machine learning methods (Bianco et al. 2019, 3591).

In machine learning, the input data is typically split into a training set and a test set (Bastanlar and Ozuysal 2014). The model makes predictions with its initial parameters using the training input. It measures how correct its prediction was by using a loss function to compare its predicted output to the training (or ground truth) output. Contemporary machine learning is dependent on the field of optimization (Boyd and Vandenberghe 2004;

Gambella, Ghaddar, and Naoum-Sawaya 2020; Le et al. 2011) for the loss function and parameter updates (Ketkar 2017). The performance of the network is measured by its ability to generalize on the test set, which was data not seen during training. There is sometimes a third set called the validation set, which is also unseen during training and typically used to perform hyperparameter tuning (Xu and Goodacre 2018). Hyperparameters are the parameters of a machine learning model that are defined by the user (Beysolow II 2017). The user tries to maximize the performance of the model on the validation set by modifying the hyperparameters.

Music signals are a form of acoustic signal, and the domain of music signal processing is inseparable from the larger field of signal processing (Müller et al. 2011). It follows that ML and DL techniques can be extended to music signal processing applications. Purwins et al. (2019) describe contemporary deep learning architectures that are used for audio applications. They state that models from the field of computer vision, originally designed for 2D images where pixels are related spatially, may not exactly fit audio waveforms, where amplitude values are related temporally. However, convolutional neural networks, recurrent neural networks, and sequence-to-sequence models are three popular types of network architecture that have been adapted with varying levels of success to the audio and music domains (Purwins et al. 2019).

A new music source separation model, which is the result of this thesis, will be presented in Chapter 3. It will have two variants, one based on a convolutional neural network (CNN) architecture, and one based on a recurrent neural network (RNN) architecture. To provide the necessary background, I will give an overview of convolutional neural networks (CNN) for audio and music applications in Section 2.3.1, I will give an overview of recurrent neural networks (RNN) for audio and music applications in Section 2.3.2.

2.3.1 Convolutional neural networks (CNN)

Convolutional neural networks (CNN) "are based on convolving their input with learnable kernels" (Purwins et al. 2019, 3). These are useful for both images and audio because of their ability to exploit spatial or temporal correlation in data (Khan et al. 2020). Figure 29 shows how a learnable convolution kernel slides over the pixels of the input image.

Figure 30 shows how a convolutional and transpose convolutional or deconvolutional layer are the inverse operation of each other.

The network architecture and feature map for an audio CNN is shown in Figure 31, where the extracted feature map is visualized in the spectral domain.

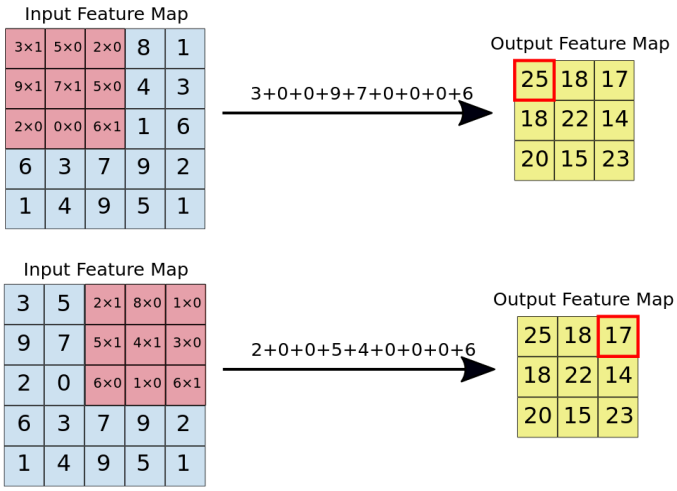


Figure 29: 3×3 convolution kernel sliding over patches of input pixels.*

*. https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks

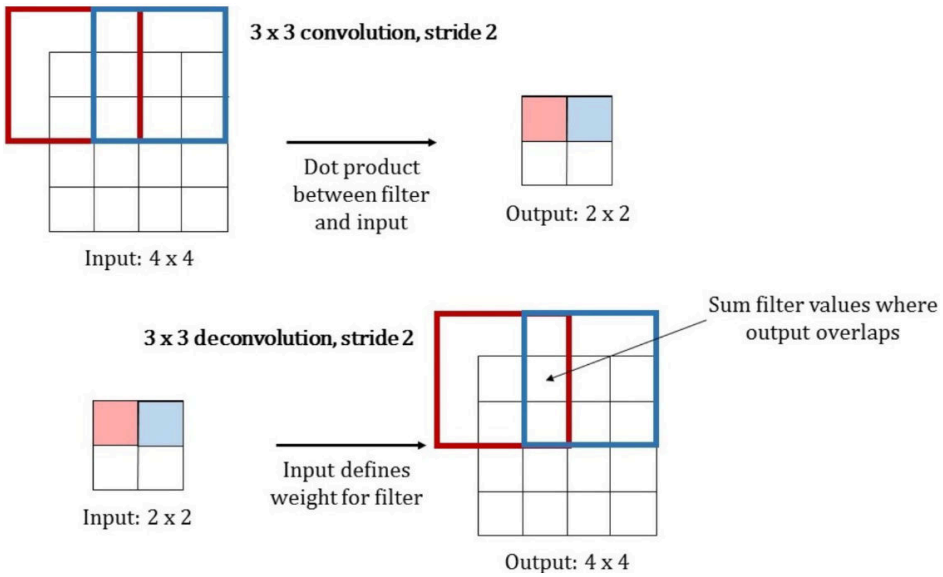
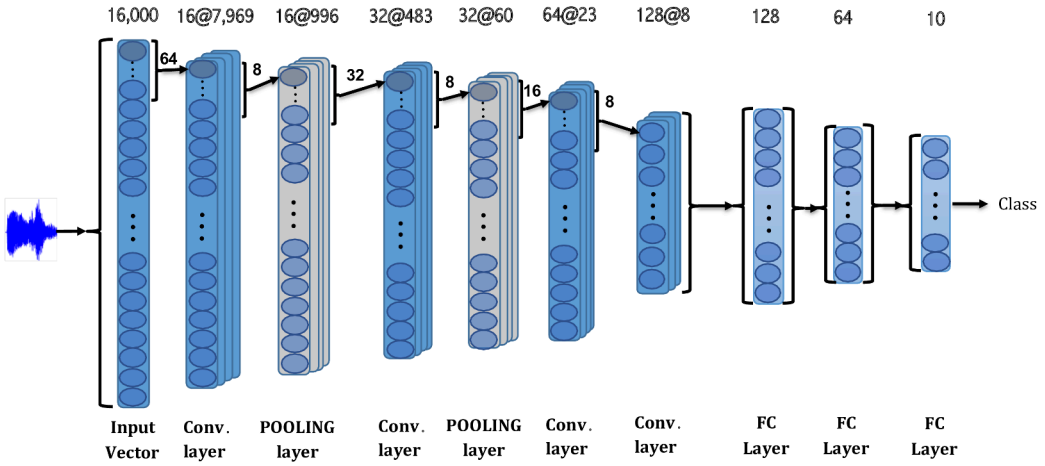


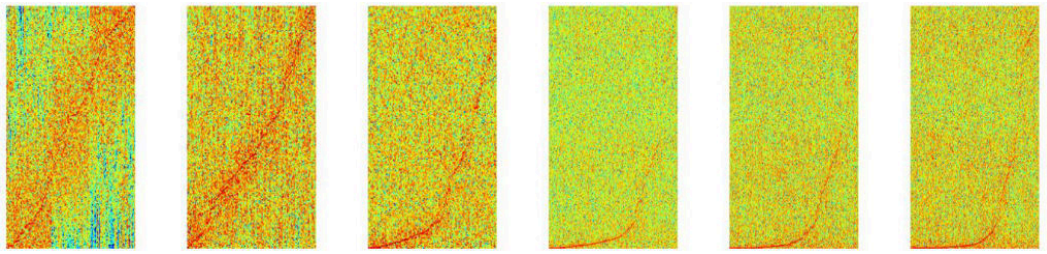
Figure 30: Deconvolution operation (Gsaxner et al. 2019, 11).

For audio applications, both 1D convolutions (also referred to as temporal convolutions) applied directly to the audio waveform and 2D convolutions applied to time-frequency transforms (most commonly the STFT) are both used (Pandey and Wang 2019; Fedorishin et al. 2021). To describe convolutional layers, Dumoulin and Visin state that "images, sound clips and many other similar kinds of data have an intrinsic structure," and share the following properties (Dumoulin and Visin 2018, 6):

1. They are stored as multi-dimensional arrays (e.g., the STFT or sliCQT)



(a) CNN architecture for audio waveforms (Abdoli, Cardinal, and Koerich 2019, 7).



(b) Audio spectral features learned by each layer (Lee et al. 2017, 4).

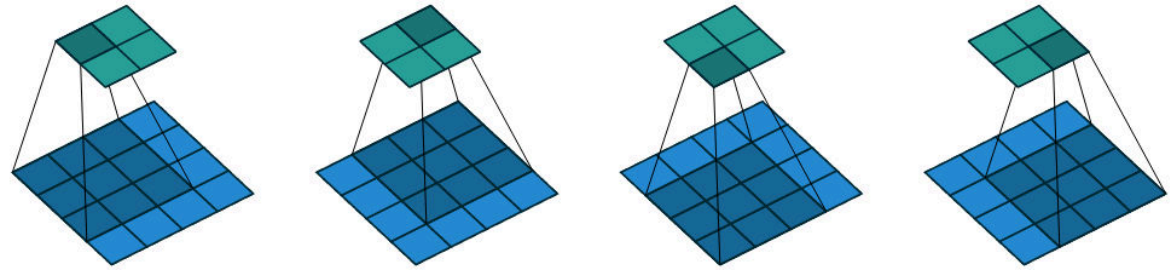
Figure 31: Example of an audio CNN architecture and spectral feature maps.

2. They feature one or more axes for which ordering matters (e.g., time and frequency for a spectrogram)
3. The channel axis is used to access different views of the data (e.g., the left and right channels of a stereo audio track)

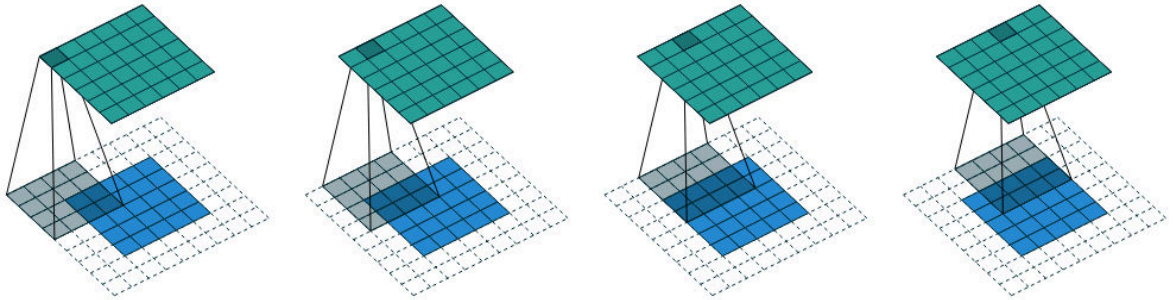
The STFT and sliCQT of stereo (2-channel) music produce a 2-channel spectrogram, such that the total dimensions of the 2D time-frequency spectrogram are time × frequency × channel. The 2D convolution kernels slide across the spectrograms to learn a feature representation, and the number of output channels determines how many feature maps are created.

The kernel parameters define the size and movement of the 2D convolution kernel in the time and frequency dimensions in each output channel. Besides the kernel size, the kernel has these additional parameters: stride, dilation, and padding (Dumoulin and Visin 2018). Figure 32 shows the behavior of these different kernel parameters. The stride is the amount by which the kernel moves; a stride larger than one implies *subsampling*, because it dictates how much of the output is retained (Dumoulin and Visin 2018). The dilation defines "holes"

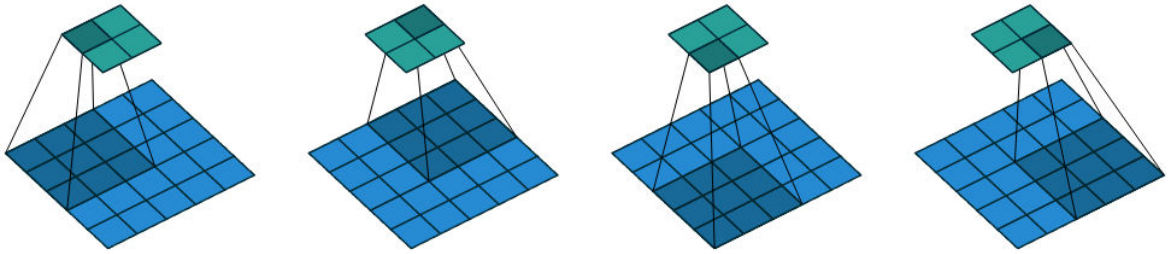
or gaps in the kernel to increase the receptive field (Dumoulin and Visin 2018), which is a computationally cheap way to increase the amount of data points considered in a feature map. The padding defines zeros concatenated to the beginning and end of an axis.



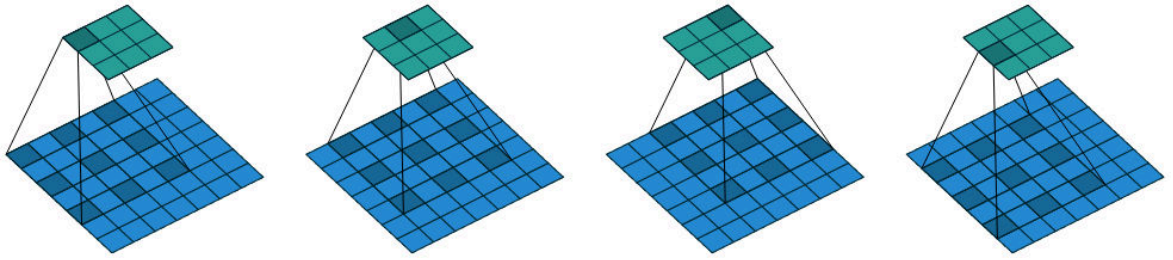
(a) 2D convolution kernel.



(b) 2D convolution kernel with padding > 0.



(c) 2D convolution kernel with stride > 1.



(d) 2D convolution kernel with dilation > 1.

Figure 32: Different behaviors of kernel parameters (Dumoulin and Visin 2018, 14, 29).

In addition to the convolution operator, another operator commonly used in CNN architectures is the pooling layer. Like the strided convolution shown in Figure 32(c), the role of

the pooling operator is to reduce the dimensionality of the input "by using some function to summarize subregions, such as taking the average or the maximum value" (Dumoulin and Visin 2018, 10). The inverse of a pooling or strided convolution layer, which reduce the dimensionality of the input, is the up-sampling layer, which "is the principal way to recover the resolution of the downsampled feature map" (Lu et al. 2020, 2). According to Lu et al. (2020), up-sampling can be implemented with a transpose convolution (or deconvolution) layer shown in Figure 30, naïve interpolation, or unpooling layers.

2.3.2 Recurrent neural networks (RNN)

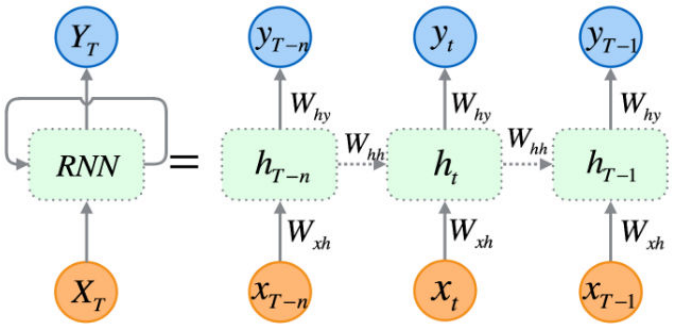
For time series data, for which the audio waveform is an exemplar given the temporal evolution of its amplitude, recurrent neural networks (RNNs) are useful since the input sequence of data is introduced back into the network with a cyclic or recurrent connection, and outputs are predicted based on past (or future, if the network is bi-directional) values of the sequence (Yu et al. 2019). RNNs are also used commonly in sequence-to-sequence, or seq2seq, models (Yousuf et al. 2020).

In the audio noise suppression model RNNoise,[20] the usefulness of the RNN and its Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997) and Gated Recurrent Unit (GRU) (Chung et al. 2014) variants for audio are described:

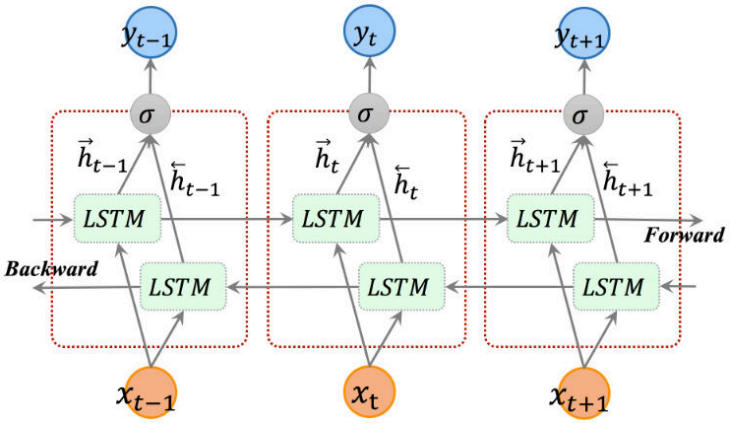
> Recurrent neural networks (RNN) are very important [...] because they make it possible to model time sequences instead of just considering input and output frames independently. [...] RNNs were heavily limited in their ability because they could not hold information for a long period of time [...] [These] problems were solved by the invention of gated units, such as the Long Short-Term Memory (LSTM), the Gated Recurrent Unit (GRU), and their many variants.

Figure 33 shows some example RNN architectures, as well as an illustration of the additional complexity in the gated LSTM and GRU units, which allow them to surpass the simple RNN.

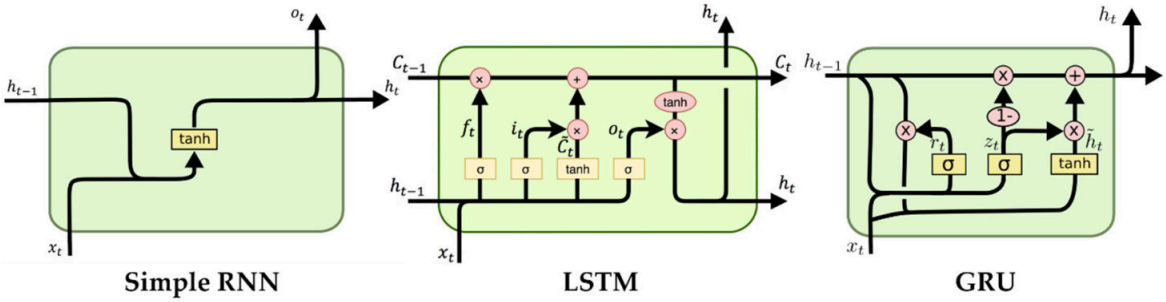
20. https://jmvalin.ca/demo/rnnoise/



(a) Regular RNN with cyclic connections to past data (Cui et al. 2019, 3).



(b) Bi-directional LSTM with cyclic connections to past and future data (Cui et al. 2019, 3).



(c) Different recurrent units (Aslam et al. 2020, 7).

Figure 33: RNN diagrams.

2.4 Software and code concepts

In this section, I will give an overview of important concepts in Python and general software and coding. The overview will cover the topics that are relevant to the methodology and experiments of this thesis in Chapter 3 and Chapter 4 respectively.

In Section 2.4.1, I will describe concepts of the Python programming language that are most relevant to this thesis, including data structures, numerical and scientific libraries, library managers for installing academic software, and random number generation.

For all of the software written for this thesis, I used Git to track changes, and made the code open-source and available on GitHub for transparency into my methodology and results. In Section 2.4.2, I will describe version control and Git, and in Section 2.4.3, I will describe open-source software and GitHub, a social website for sharing code that use Git.

2.4.1 Python programming language

Python[21] is a general-purpose programming language. It is an interpreted language,[22] which means that there is no compilation step required, and the Python code or script written by a user can be executed right away with the Python interpreter. The Python interpreter can also run statements directly for quick prototyping, without needing to write a script:

```
Python 3.9.6 (default, Jul 16 2021, 00:00:00)
[GCC 11.1.1 20210531 (Red Hat 11.1.1-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('this is python')
this is python
```

The Python interpreter is widely distributed on most modern operating systems (OS) like Linux, Windows, and OS-X. Python has support for various high-level and user-friendly data structures. Python has seen widespread adoption in academia, and especially in numerical contexts (Bayen, Kong, and Siauw 2020).

One of the core features of the Python language is the *list*.[23] The list in Python corresponds to the array data structure in computer science (Skiena 2008). It allows for the construction of an ordered list of objects. For example, a discrete-time speech signal may be described by a list of its amplitude values:

```
speech_signal = [0.15, 0.28, 0.57, 0.98, -0.59]
```

21. https://www.python.org/
22. https://www.python.org/doc/essays/blurb/
23. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Python has a rich ecosystem of academic software libraries. NumPy[24] and SciPy[25] are used for numerical computation and parallelized matrix operations that run on the CPU (central processing unit). Matplotlib[26] is a plotting library. Tensorflow[27] and PyTorch[28] are libraries for machine learning and deep learning, which also have numerical computation and parallelized matrix operations that are similar to NumPy and SciPy, except that they can run on the GPU (graphical processing unit). The use of the GPU allows for more efficient, faster, and larger parallelized matrix operations, which are essential for modern machine learning and deep learning techniques.

In NumPy, the *ndarray*[29] is the core data structure representing an n-dimensional array of objects, which can be numerical (such as int16 or float32) or general objects (such as strings). The same speech signal represented by the Python list above can be represented by a 1-dimensional ndarray of 64-bit floating-point values:

```
>>> import numpy
>>> speech_signal = numpy.asarray([0.15, 0.28, 0.57, 0.98, -0.59])
>>> print(speech_signal.dtype)
float64
>>> print(speech_signal.shape)
(5,)
```

The underlying values are stored in formats that allow for efficient numerical computations, which is why using NumPy ndarrays for numerical computation is preferred to using regular Python data structures such as lists that are not designed for speed (Walt, Colbert, and Varoquaux 2011).

In PyTorch and Tensorflow (and in general machine learning), a similar concept to the ndarray is the *tensor*. The tensor originates from the field of physics (Kolecki 2002). The tensor is also an n-dimensional numerical array, and NumPy ndarrays are interchangeable with both PyTorch and Tensorflow tensors.[30]

Tensors can be used for numerical computations using the GPU in both PyTorch and TensorFlow, much like the ndarray with NumPy and SciPy on the CPU. For example, `torch.cos()` computes the cosine of a tensor. Additionally and most importantly, the inputs and out-

24. https://numpy.org/
25. https://scipy.org/
26. https://matplotlib.org/
27. https://www.tensorflow.org/
28. https://pytorch.org/
29. https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html
30. https://www.tensorflow.org/guide/tf_numpy, https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

puts to the machine learning and deep learning constructs supported by these libraries are represented as tensors.

The `random` module of Python contains a random number generator (RNG).[31] The `seed` function initializes the random number generator,[32] which is, in fact, pseudorandom (L'Ecuyer 2010). Pseudorandom means that given the same starting seed, the exact same sequence of numbers are generated deterministically. The seed is usually a parameter to a script or Python program, such that the user can recreate the same RNG sequences for testing purposes by using the same seed.

To manage third-party libraries and packages for Python, a package manager is suggested. The built-in way of managing a sandboxed Python environment is the virtual environment.[33] This creates an isolated copy of Python where the user can install packages without risking the stability of the operating system (OS). The Python dependency manager tool, `pip`,[34] can be used to install a list of third-party packages. Packages are published to the public Python Package Index (PyPi) by the authors.[35] An example of a `pip` file, conventionally called `requirements.txt`, is shown in Code Listing 1.

```
mir_eval==0.6
tabulate==0.8.7
numpy==1.19.4
```

Code Listing 1: Example pip requirements.txt file.

There is another popular Python dependency manager called Conda.[36] Conda environments are also popular for creating reproducible Python environments for academic software. Conda supports channels for third-party package authors to publish and distribute their packages to users, similar to PyPi.[37] An example Conda environment file is shown in Code Listing 2. Conda files can also support the `pip` syntax, since `pip` is a native Python tool. The user can combine the best of both worlds by using Conda and `pip` syntax simultaneously in their Conda environment file.

Pip is installed by default with Python, but Conda needs to be installed separately. For the code in this thesis, both `pip` and Conda files are used and provided to help the readers replicate the necessary Python environments.

31. https://docs.python.org/3/library/random.html
32. https://docs.python.org/3/library/random.html#bookkeeping-functions
33. https://docs.python.org/3/tutorial/venv.html
34. https://pip.pypa.io/en/stable/cli/pip_install/#requirements-file-format
35. https://pypi.org/
36. https://docs.conda.io/en/latest/
37. https://conda.io/projects/conda/en/latest/user-guide/concepts/channels.html

```
name: test-conda-env

channels:
  - default

dependencies:
  - python=3.9
  - cudatoolkit=11
  - pip
  - pip:
    - norbert>=0.2.0
```

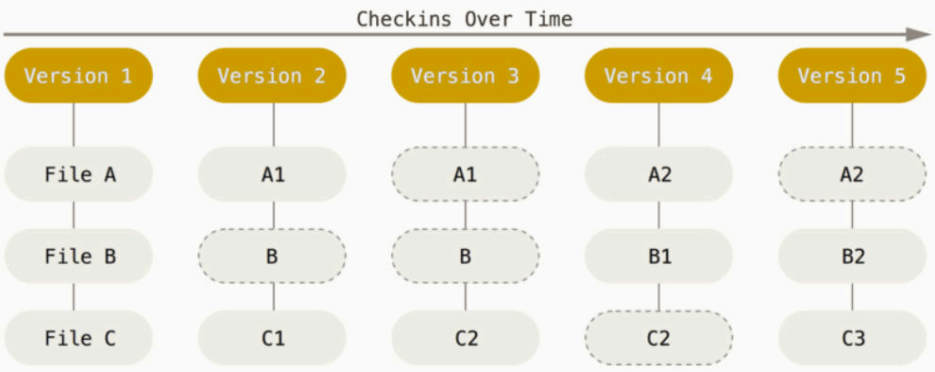
Code Listing 2: Example Conda environment.yml file.

2.4.2 Version control systems and git

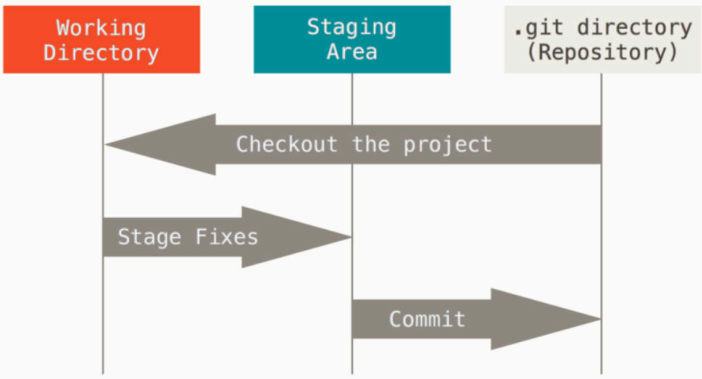
Version control systems (VCS) are systems used for tracking and managing changes to code files, and storing the history of changes in a database (Chacon and Straub 2014). Chacon and Straub state that the benefits of using a VCS is that it "allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover" (Chacon and Straub 2014, 1).

According to Chacon and Straub (2014), version control systems are either centralized or distributed. In a distributed version control systems (DVCS), each person's copy of the code contains the full history of changes. This means that people can work on their own private copy of the code, and then sync changes with a central server, rather than depending on the central server for viewing the history like in a centralized version control system (CVCS). This allows people to work independently of the central server, until they are ready to sync changes back with the other collaborators.

Git is a popular tool for DVCS written by Linus Torvalds, who is also known for having created Linux. How Git works is that it "thinks of its data more like a series of snapshots of a miniature filesystem... every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot" (Chacon and Straub 2014, 6). To work on a code project that uses Git, first the project, or *repository*, must be cloned to your local system. You make local changes and then commit them. Finally, you can push your commits back to the central server to share with the other collaborators, often with a descriptive message describing what you changed. Figure 34 shows how Git tracks a file and how a Git workflow should look.



(a) How git stores snapshots of files.



(b) Recommended git workflow.

Figure 34: How git works (Chacon and Straub 2014, 6, 8).

All of the code for this thesis were stored in Git repositories to keep track of historical changes. It conveniently allows the author to reference the Git history to describe the evolution of the project and the incremental progress.

2.4.3 Open-source software and GitHub

Fortunato and Galassi state that "free and open source software (FOSS) is any computer program released under a licence that grants users rights to run the program for any purpose, to study it, to modify it, and to redistribute it in original or modified form" (Fortunato and Galassi 2021, 1). Free and open-source software is a merging of two distinct ideas: free software[38] and open-source software.[39] While these have nuanced differences when it comes to commercial licensing, code visibility, and availability, in simplistic terms FOSS implies code that is free and open for users to read and modify. Releasing code as open-source is becoming more popular in academia, with initiatives like Papers With Code[40] and the Journal of Open

38. https://www.fsf.org/
39. https://opensource.org/
40. https://paperswithcode.com/

Source Software.[41] According to Fortunato and Galassi (2021), FOSS software is a natural complement to academic publications, given concerns with reproducibility of results.

GitHub[42] is a social website, designed for sharing and browsing software projects that use Git for version control, intended for users to share and collaborate on open-source code. Chacon and Straub describe GitHub[43] as "the single largest host for Git repositories, and [... a] central point of collaboration for millions of developers and projects" (Chacon and Straub 2014, 131). GitHub contains tools for working with Git repositories, including social elements for personal profiles or organizations. Open-source projects can be made public on GitHub such that anybody can read and download the source code. For a project hosted on GitHub, there is a file browser built into the website, which allows one to view the code and change history of a project. In this thesis, the GitHub code browser will be linked wherever appropriate. An example of the GitHub code browser showing a file from the PyTorch project's Git repository[44] is shown in Figure 35.

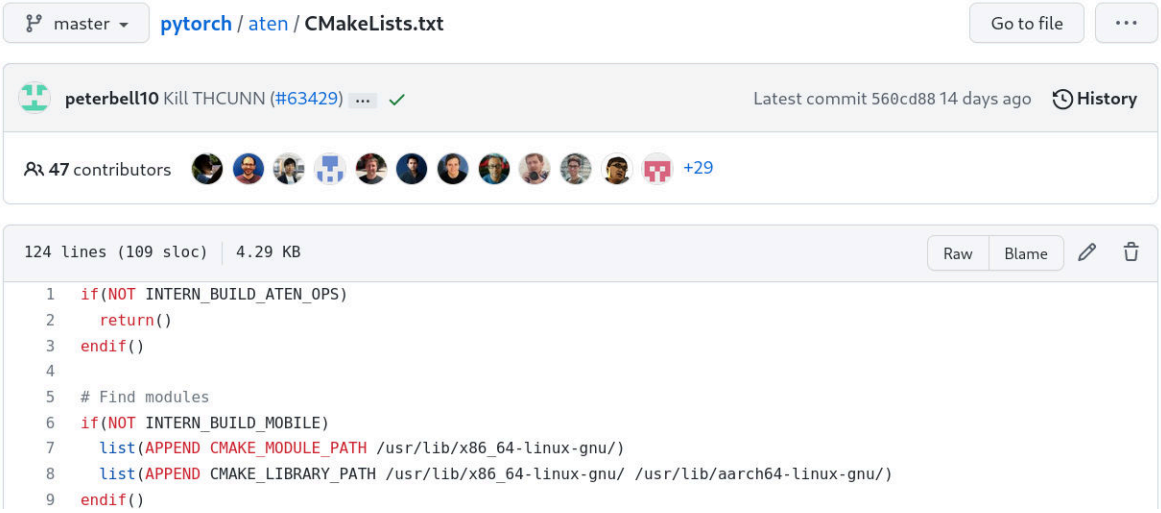


Figure 35: GitHub's code browser showing a file from the PyTorch source code.

41. https://joss.theoj.org/
42. https://github.com
43. https://github.com/
44. https://github.com/pytorch/pytorch

2.5 Music source separation and demixing

In this section, I will provide an overview of music source separation and music demixing. I will start by providing definitions, motivations, and uses of the tasks in Section 2.5.1. In Section 2.5.2, I will give an overview of historical computational approaches to general audio source separation, leading to current trends in music source separation and demixing. In Section 2.5.3, I will describe the MUSDB18 and MUSDB18-HQ datasets used for music source separation research, and in Section 2.5.4, I will describe the BSS evaluation metrics used for evaluating music source separation methods.

In Section 2.5.5, I will describe the technique of time-frequency masking, which is a popular strategy used for spectrogram-based music source separation, and oracle estimators, which are methods for estimating the upper limit of quality achievable by time-frequency masking. In Section 2.5.6, I will describe the noisy phase strategy, which is a strategy used by spectrogram-based music source separation models, where the model only estimates the magnitude STFT of the target and uses the phase STFT of the original mixed audio, or the "noisy phase." In Section 2.5.7, I will describe Harmonic/Percussive Source Separation (HPSS), which is a simple algorithm for music source separation based on the STFT spectrogram that exemplifies the time-frequency tradeoff of the STFT.

A new music source separation model, which is the result of this thesis, will be presented in Chapter 3. In Section 2.3, I described that the model will have two variants, one based on a convolutional neural network (CNN) architecture, and one based on a recurrent neural network (RNN) architecture. The RNN model architecture is based on Open-Unmix and CrossNet-Open-Unmix, two related models for music source separation, which will be respectively shown in Section 2.5.8 and Section 2.5.9. The CNN model architecture is based on the Convolutional Denoising Autoencoder (CDAE), which will be shown in Section 2.5.10.

2.5.1 Task definition and motivations

Typical music recordings are mono or stereo mixtures, with multiple sound objects (drums, vocals, etc.) sharing the same track (Rafii et al. 2018). To manipulate the individual sound objects, the stereo audio mixture needs to be separated into a track for each different sound source, in a process called audio source separation.

According to Mitsufuji et al., audio source separation and music source separation have important uses in the world today:

> Audio source separation has been studied extensively for decades as it brings benefits in our daily life, driven by many practical applications, e.g., hearing aids,

> speech diarization, etc. In particular, music source separation (MSS) attracts professional creators because it allows the remixing or reviving of songs to a level never achieved with conventional approaches such as equalizers. Suppressing vocals in songs can also improve the experience of a karaoke application, where people can enjoy singing together on top of the original song (where the vocals were suppressed), instead of relying on content developed specifically for karaoke applications (Mitsufuji et al. 2021, 1).

A common idea in survey papers is that the domains of speech enhancement and music source separation are both important subproblems of audio source separation (Rafii et al. 2018; Liu and Li 2009).

I provided definitions for music source separation and music demixing in Chapter 1, to distinguish them from the more general problem of audio source separation:

> Music source separation is the task of extracting an estimate of one or more isolated sources or instruments (for example, drums or vocals) from musical audio. The task of music demixing or unmixing considers the case where the musical audio is separated into an estimate of all of its constituent sources that can be summed back to the original mixture.

Music demixing can be considered as the reverse of the mixing process of *stems* in a recording studio, shown in Figure 36. First, I will provide a definition of what a stem is from online materials published by music production companies: a stem is a grouping of individually recorded instrument tracks that have been combined together in a common category.[45] For example, a drum stem could include all of the tracks of a drum kit (e.g., snare, tom, hihat), and a vocal stem could include all of the vocal tracks from the different singers in the song.

Music demixing can also be viewed as a combination of multiple music source separation subproblems for all of the desired target stems. To illustrate the example from Figure 36: extracting *one* of the guitar, drum, keyboard, or vocal sources (or stems) from the mixed song is music source separation, but extracting *all four* from the mixed audio at the same time is music demixing.

There are many motivations for performing this separation. Cano et al. describe that "we can remix the balance within the music [...] to make the vocals louder or to suppress an unwanted sound, or we might want to upmix a 2-channel stereo recording to a 5.1-channel

45. https://www.izotope.com/en/learn/stems-and-multitracks-whats-the-difference.html, https://blog.landr.com/stems-in-music/

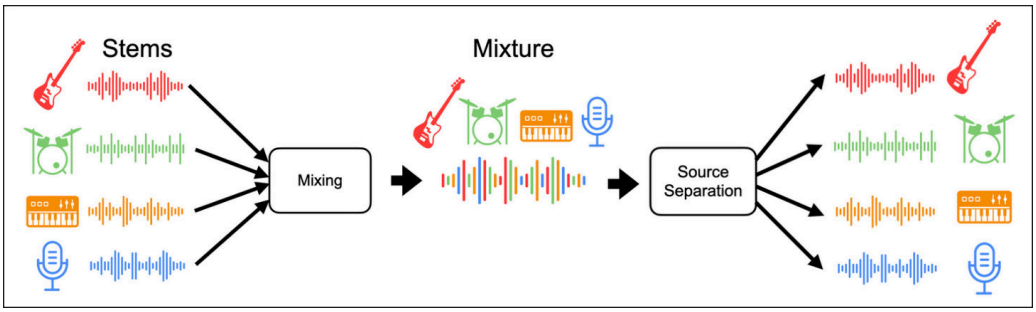


Figure 36: Music mixing and demixing block diagrams.*

*. https://www.aicrowd.com/challenges/music-demixing-challenge-ismir-2021

surround sound system... We might also want to change the spatial location of a musical instrument within the mix" (Cano et al. 2018, 31).

2.5.2 Computational approaches

Computational source separation has a history of at least 50 years (Liu and Li 2009; Rafii et al. 2018), originating from the tasks of computational auditory scene analysis (CASA) and blind source separation (BSS). In CASA, the goal is to "computationally extract individual streams from one or two recordings of an acoustic scene" (Wang and Brown 2006, 12), based on the definition of ASA (auditory scene analysis) given by Bregman (1994). BSS, first defined by Jutten and Hérault (1991), solves a subproblem of CASA which aims to recover the sources of a "mixture of multiple, statistically independent sources that are received with separate sensors" (Wang and Brown 2006, 190). The term "blind" refers to there being no prior knowledge of what the sources are, and how they were mixed together. In CASA and BSS, therefore, the mixed audio contains unknown sources combined in unknown ways that must be separated.

By contrast, in music source separation and music demixing, the sources are typically known, or have known characteristics. That is to say, in music source separation, the task is not to separate all of the distinct sources in the mixture, but to extract a predefined set of sources. Some forms of music source separation try to extract harmonic and percussive sources (Fitzgerald 2010; Fitzgerald and Gainza 2010; Driedger, Müller, and Disch 2014), which will be described in greater detail in Section 2.5.7. Another common set of separation sources are the four sources defined by the MUSDB18 (Rafii et al. 2017) dataset: vocals, drums, bass, and other. The MUSDB18 dataset will be described in more detail in Section 2.5.3. Additionally, in music source separation, the mixing process is generally assumed to

be a simple linear mixture (Cano et al. 2018), shown in equation (20):

$$x_{\text{mix}} = x_{\text{source1}} + x_{\text{source2}} + ... \tag{20}$$

A popular algorithm in BSS is Independent Component Analysis (ICA) (Liu and Li 2009; Cano et al. 2018; Rafii et al. 2018), which exploits spatial information of the sources, and assumes the sources to be independent. This technique can be used when there are as many channels in the mixture (corresponding to differently placed microphones) as the number of sources. Hyvärinen (1999) and Hyvärinen and Oja (2000) describe ICA algorithms, and Comon and Jutten (2010) and Naik and Wenwu (2014) provide an in-depth review on the history of BSS. Figure 37 shows an example of the positional considerations in a typical ICA system.

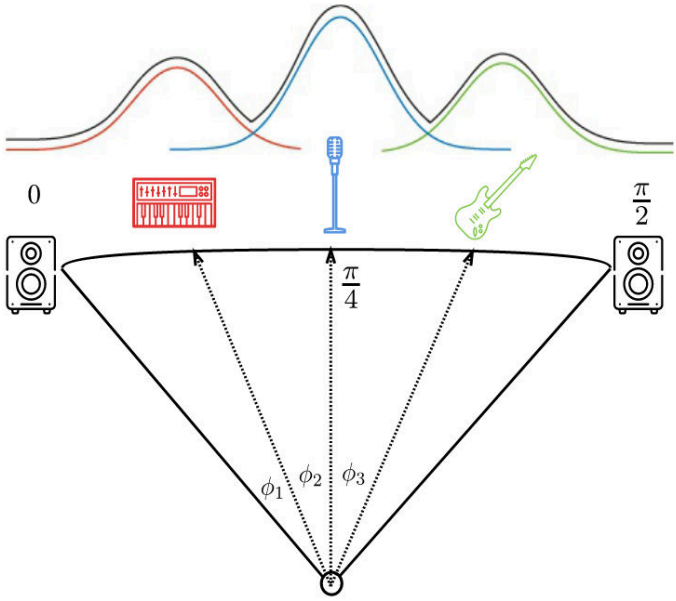


Figure 37: Position-based source separation (Cano et al. 2018, 35).

According to Rafii et al., ICA techniques arose more typically for speech denoising (Loizou 2017), and they make assumptions that cannot be generalized easily to music:

> [systems which aim] to recover clean speech from noisy recordings [...] can be seen as a particular instance of source separation. [...] many algorithms assume the audio background can be modeled as stationary. However, the musical sources are characterized by a very rich, nonstationary spectrotemporal structure. This prohibits the use of such methods. Musical sounds often exhibit highly synchronous evolution over both time and frequency, making overlap in both time

> and frequency very common. Furthermore, a typical commercial music mixture violates all the classical assumptions of ICA. Instruments are correlated (e.g., a chorus of singers), there are more instruments than channels in the mixture, and there are nonlinearities in the mixing process (e.g., dynamic range compression) (Rafii et al. 2018, 1).

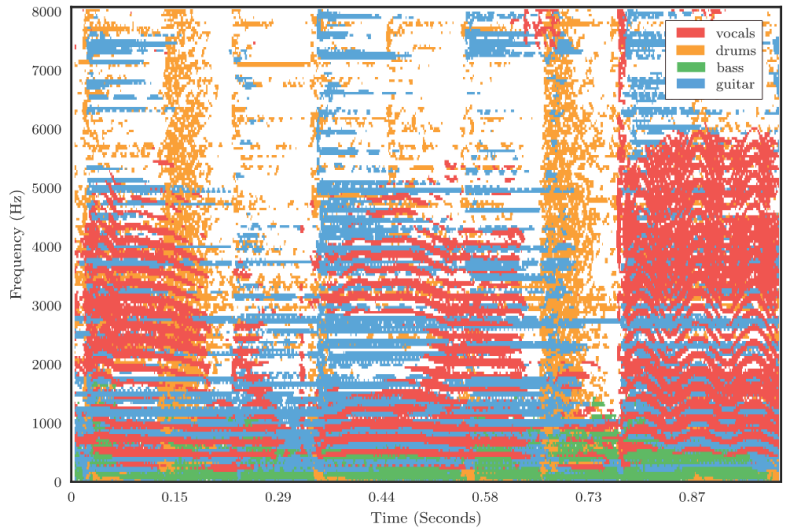
Techniques more specific to music were developed as a necessity to deal with these differences (Liutkus et al. 2013; Vincent et al. 2014). For cases where ICA cannot be applied, musical source models are more popular, which are "model-based approaches that attempt to capture the spectral characteristics of the target source can be used" (Cano et al. 2018, 36). Sources are assumed to be sufficiently different from each other, or sparse in their spectral representation (Cano et al. 2018), such that they can be extracted with time-frequency masks applied in the spectral domain. Figure 38 shows how different sources have unique spectral patterns.

Kernel Additive Modeling (KAM) is the simplest form of music source modeling (Cano et al. 2018). To estimate a music source at a given time-frequency point, KAM "select[s] a set of time-frequency bins, which, given the nature of the target source (e.g., percussive, harmonic, or vocals) are likely to be similar in value. This set of time-frequency bins is termed a proximity kernel" (Cano et al. 2018, 36). A well-known example of a KAM-based music separation algorithm is the median-filtering Harmonic/Percussive Source Separation (HPSS) algorithm (Fitzgerald 2010).

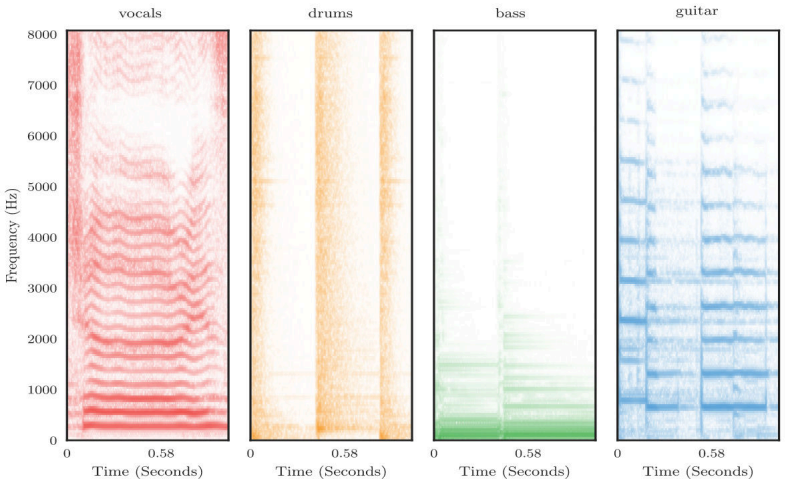
Spectrogram factorization models are more sophisticated than KAM, and the most popular spectrogram factorization model is Nonnegative Matrix Factorization (NMF) (Liu and Li 2009; Cano et al. 2018). According to Cano et al., NMF "attempts to factorize a given nonnegative matrix into two nonnegative matrices," and it can be applied to the magnitude spectrogram of the mix, M, to separate it into frequency weight matrices W and time activation matrices H (Cano et al. 2018, 37). A survey on NMF techniques by Lee and Seung (2001) covers the algorithm in more detail. Most recently, data-driven approaches based on machine learning and deep learning have significantly surpassed past approaches (Cano et al. 2018; Stöter, Liutkus, and Ito 2018). Figure 39 shows different techniques for spectral demixing.

2.5.3 Public datasets

The most popular music stem dataset used by the Signal Separation Evaluation Campaign (SiSEC) and SigSep is the MUSDB18 dataset (Rafii et al. 2017), and more recently the HQ



(a) Mixed spectrogram.



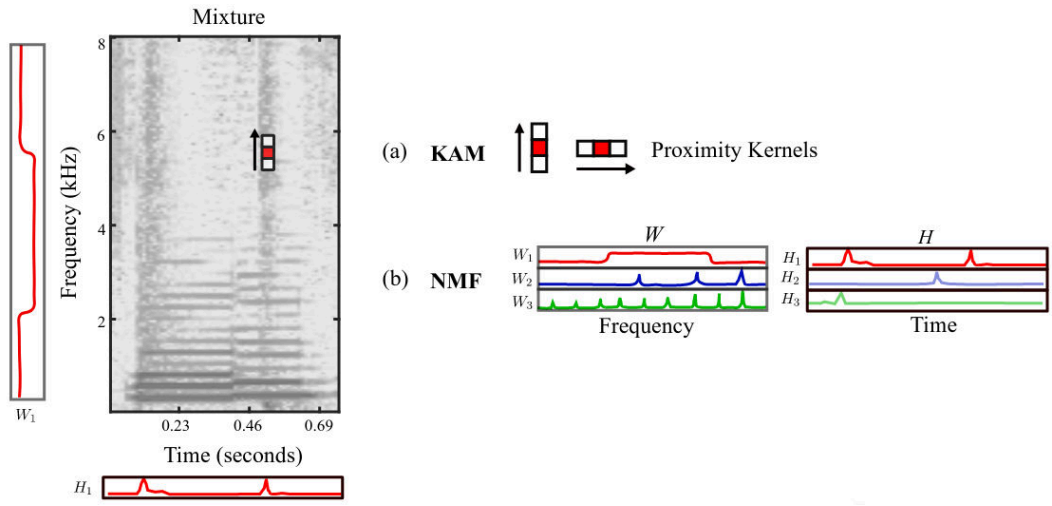
(b) Source spectrograms.

Figure 38: Sparsity of music sources in the spectral domain (Cano et al. 2018, 32).

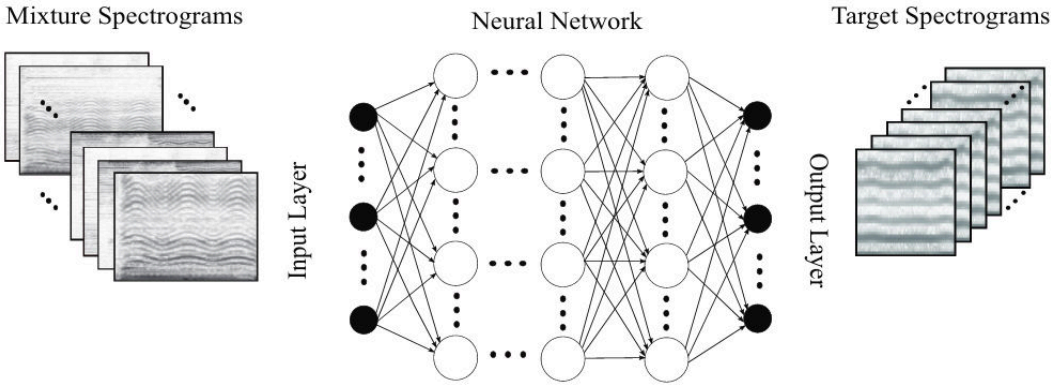
(high-quality) version (Rafii et al. 2019). MUSDB18-HQ contains stereo wav files sampled at 44,100 Hz representing stems (drum, vocal, bass, and other) from a collection of permissively licensed music, specifically intended for recording, mastering, mixing (and in this case, "de-mixing", or source separation) research. It combines earlier mixing/demixing datasets (Liutkus et al. 2017; Bittner et al. 2014).

The MUSDB18-HQ dataset has fixed train, validation, and test subsets of data. The dataset is organized into `train` and `test` folders, and a subset from the training folder is defined to be the validation set by the Python MUSDB18-HQ loader library.[46]

46. https://github.com/sigsep/sigsep-mus-db



(a) KAM vs. NMF for spectral music demixing.



(b) Deep neural networks for spectral music demixing.

Figure 39: Techniques for spectral music demixing (Cano et al. 2018, 36, 38).

2.5.4 Evaluation measures

To fairly rank and evaluate different systems for music source separation, an objective measurement of the source separation quality is necessary. Vincent, Gribonval, and Févotte (2006) and Vincent et al. (2007) introduced the BSS (Blind Source Separation) Eval metrics, a "new numerical performance criteria that can help evaluate and compare algorithms when applied on [blind audio source separation] problems" (Vincent, Gribonval, and Févotte 2006, 1463). BSS metrics have been widely used in papers introducing new music source separation models (Stöter et al. 2019; Sawata et al. 2021; Yu and Cheuk 2021; Kim et al. 2021; Kong et al. 2021; Défossez et al. 2019) and in large-scale evaluation campaigns of diverse music source separation systems (Liutkus et al. 2017; Stöter, Liutkus, and Ito 2018; Mitsufuji et al. 2021). Due to their prominence in source separation literature, I decided to use the BSS metrics in this thesis to measure the performance of my proposed model.

There are four distinct metrics that comprise BSS:

- **SDR: S**ignal to **D**istortion **R**atio
- **SIR: S**ignal to **I**nterference **R**atio
- **SAR: S**ignal to **A**rtifacts **R**atio
- **ISR:** Source **I**mage to **S**patial distortion **R**atio

Out of these four scores, SDR is the single global score that is commonly used to summarize the overall performance of a music demixing system (Nakajima et al. 2018). The SDR can be computed from equation (18):

$$SDR_{\text{instr}} = \\ 10 \log_{10} \frac{\sum_n \left(s_{\text{instr, left}}(n)\right)^2 + \sum_n \left(s_{\text{instr, right}}(n)\right)^2}{\sum_n \left(s_{\text{instr, left}}(n) - \hat{s}_{\text{instr, left}}(n)\right)^2 + \sum_n \left(s_{\text{instr, right}}(n) - \hat{s}_{\text{instr, right}}(n)\right)^2} \tag{18}$$

The SDR score is a relative scale given in decibels or dB, where $s_{\text{instr}}(n)$ denotes the ground truth waveform of the instrument, $\hat{s}_{\text{instr}}(n)$ is the estimate, and left and right refer to the two channels in the stereo dataset of MUSDB18-HQ. Given the four stems (vocals, drums, bass, other) of MUSDB18-HQ, the SDR is computed for each stem from equation (18) above. Then, the four SDR scores are combined for a total song score in equation (19):

$$SDR_{\text{song}} = \frac{1}{4}(SDR_{\text{bass}} + SDR_{\text{drums}} + SDR_{\text{vocals}} + SDR_{\text{other}}) \tag{19}$$

In the 2018 Signal Source Separation Evaluation Compaign (SiSEC), Stöter, Liutkus, and Ito (2018) introduced an evolution of BSS metrics, called BSS v4, which reduced the computational cost over the original BSS metrics used in SiSec 2016 (Liutkus et al. 2017). The new BSS v4 metrics were also made available in the Python libraries museval[47] and bsseval,[48] which were used in this thesis.

2.5.5 Time-frequency masking and oracle estimators

Diverse music source separation algorithms based on spectrograms use the technique of time-frequency masking (Cano et al. 2018; Liu and Li 2009). In this section, I will describe the concept of time-frequency masking, and oracle estimators, which can be used to compute the maximum possible quality of a given time-frequency mask from ground truth signals. Two different music source separation systems based on time-frequency masking will be described

47. https://github.com/sigsep/sigsep-mus-eval
48. https://github.com/sigsep/bsseval

in upcoming sections: Harmonic/Percussive Source Separation (Fitzgerald 2010), a KAM (kernel additive modeling) algorithm for music source separation that will be presented in Section 2.5.7, and Open-Unmix (Stöter et al. 2019), a DNN (deep neural network) for music source separation that will be presented in Section 2.5.8.

Gerkmann and Vincent (2018) describe different time-frequency masking strategies in audio source separation. A time-frequency mask (or spectral mask, or masking filter) is a matrix of the same size as the complex STFT, or its real-valued magnitude, by which the STFT is multiplied to mask, filter, or suppress specific time-frequency bins. A soft mask, or ratio mask, has real values $\in [0.0, 1.0]$, and a binary mask, or hard mask, has logical values (i.e., only zero and one). To compute a binary mask, there must be an additional real-valued parameter, θ, which is the separation factor; values below θ are set to 0, and values above θ are set to 1. According to Gerkmann and Vincent (2018), soft masks generally produce a higher quality of sound. An illustration of spectral masking is shown in Figure 40, and an example of soft and hard masking is shown in Figure 41.

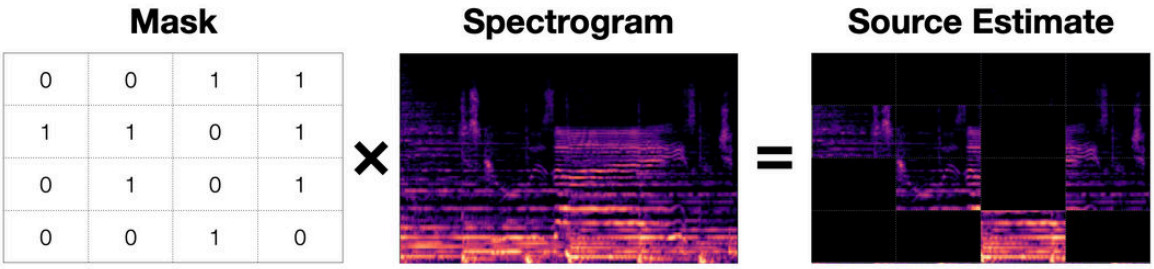


Figure 40: Simple example of applying a time-frequency mask to a spectrogram.*

*. https://source-separation.github.io/tutorial/basics/tf_and_masking.html

An example of a soft or ratio mask, used by Fitzgerald (2010) and Fitzgerald and Gainza (2010), is given by equation (21):

$$M_{\text{target}} = \frac{|\hat{S}_{\text{target}}|^p}{|\hat{S}_{\text{interference}}|^p + |\hat{S}_{\text{target}}|^p} \tag{21}$$

where $\hat{S}$ represents the complex-valued spectrogram and p represents the raised power ($p = 1$ is the magnitude spectrogram, and $p = 2$ is the power spectrogram). The binary or hard mask used by Driedger, Müller, and Disch (2014) is given by equation (22):

$$M_{\text{target}} = \frac{|\hat{S}_{\text{target}}|}{|\hat{S}_{\text{interference}}| + \epsilon} \leq \beta \tag{22}$$

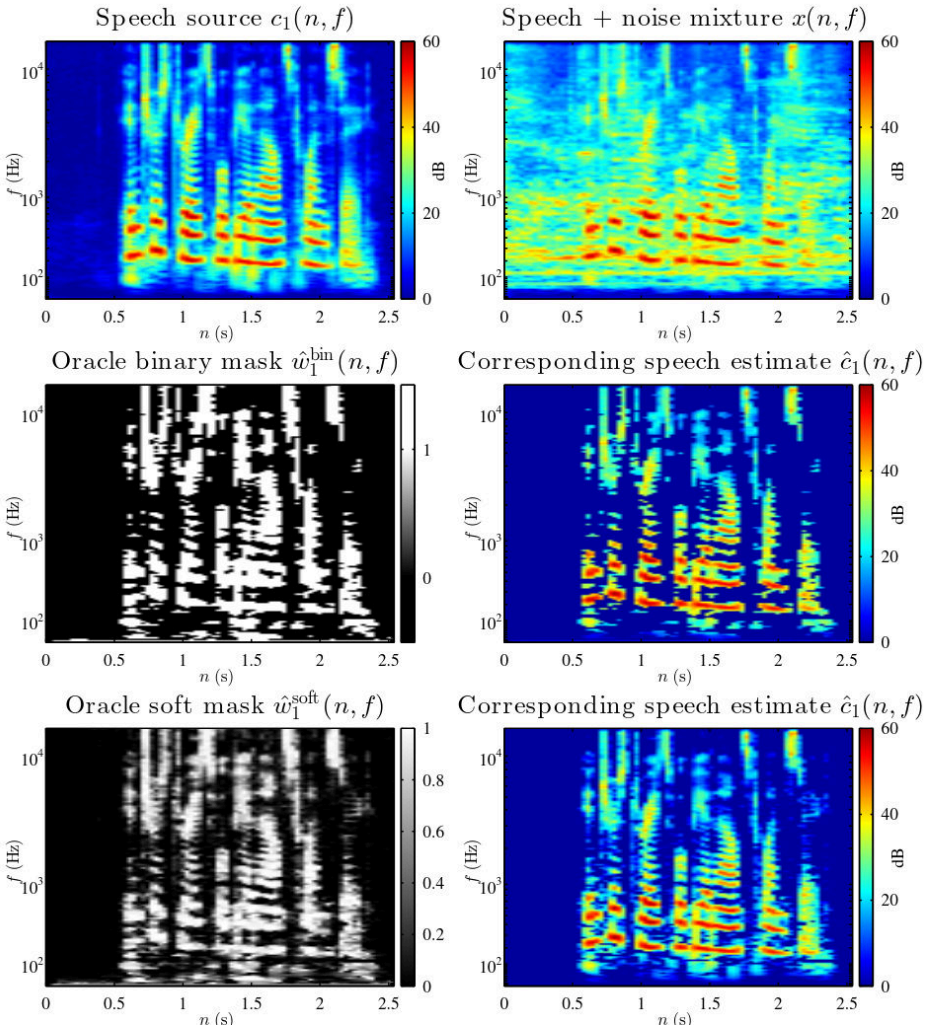


Figure 41: Results of a soft and binary oracle mask applied for speech denoising (Gerkmann and Vincent 2018, 71). When the binary mask is applied, there is a more dramatic separation, and there is starker contrast between the non-zero and zero parts of the resulting spectrogram. When the soft mask is applied, the separation is more gentle, and a range of zero to non-zero values can be seen in the resulting spectrogram.

where β is the separation factor. Note the inclusion of machine epsilon in the denominator, to avoid division by zero. One advantage of the hard mask's separation factor is that a third residual component can be extracted. This is shown for two arbitrary sources a and b in equations (23):

$$M_{\text{a}} = \frac{|\hat{S}_{\text{a}}|}{|\hat{S}_{\text{b}}| + \epsilon} > \beta, M_{\text{b}} = \frac{|\hat{S}_{\text{b}}|}{|\hat{S}_{\text{a}}| + \epsilon} \geq \beta \tag{23}$$
$$M_{\text{residual}} = 1 - (M_{\text{a}} + M_{\text{b}})$$

The oracle mask, or oracle estimator, is a perfect time-frequency mask which is computed from ground-truth data. The use of the oracle estimator is to give an idea of the upper limit of audio quality from an algorithm or machine learning model for source separation or demixing. Typically, the mask is computed from and applied to the magnitude spectrograms (Fitzgerald 2010; Fitzgerald and Gainza 2010; Driedger, Müller, and Disch 2014; Stöter et al. 2019; Grais and Plumbley 2017; Grais, Zhao, and Plumbley 2021). Discarding the phase of the complex STFT is a choice made for simplicity, although the phase may be important for source separation applications (Parry and Essa 2007).

To illustrate the calculation of the oracle mask, I will describe a simple case of a mixed song consisting of a vocal and drum track. To compute oracles, note that it is necessary to have access to the ground truth isolated source recordings of vocals and drums, in addition to the mix.

The waveforms $x_v[n]$ and $x_d[n]$ denote the isolated vocal and drum tracks respectively. The mixed song is defined by the waveform $x_m[n] = x_v[n] + x_d[n]$. To apply the music demixing task to this waveform, I want my algorithm or model to take the mixed waveform $x_m[n]$ as an input, and estimate the two source waveforms of vocals $\hat{x}_v[n]$ and drums $\hat{x}_d[n]$.

Several interesting oracle masks can be calculated from the STFTs of the mix and ground truths of two sources shown in equations (24):

$$\begin{aligned} X_m &= STFT(x_m) \\ X_v &= STFT(x_v) \\ X_d &= STFT(x_d) \end{aligned} \tag{24}$$

Stöter, Liutkus, and Ito (2018) report the performance of several oracles called IRM1, IRM2, IBM1, and IBM2. These acronyms can be understood as follows; the "I" stands for Ideal, "R|B" denotes a Ratio (soft) vs. Binary (hard) mask, "M" stands for Mask, and the trailing number is the pth power which the magnitude spectrogram is raised to. For example, the IRM1 is a soft mask between the magnitude spectrograms (since the magnitude spectrogram raised to the power of one is simply itself), and IBM2 is a hard mask between the power spectrograms (i.e., the magnitude spectrogram raised to the power of two).

The oracles for the vocal source (and the same equations apply to the drum track) are

computed from equations (25), noting that a typical default value for θ is 0.5:

$$IRM1_v = \frac{|X_v|^1}{|X_m|^1}, \qquad IRM2_v = \frac{|X_v|^2}{|X_m|^2} \tag{25}$$
$$IBM1_v = \begin{cases} 0 \text{ where } \frac{|X_v|^1}{|X_m|^1} < \theta \\ 1 \text{ where } \frac{|X_v|^1}{|X_m|^1} \geq \theta \end{cases}, IBM2_v = \begin{cases} 0 \text{ where } \frac{|X_v|^2}{|X_m|^2} < \theta \\ 1 \text{ where } \frac{|X_v|^2}{|X_m|^2} \geq \theta \end{cases}$$

To estimate the time-domain waveform from an oracle mask, the complex STFT of the mixed waveform, X_m, is multiplied by the mask, and the resultant complex STFT is inverted back to the time-domain waveform using equation (26):

$$\hat{X}_{v,\,\text{IRM1}} = X_m \cdot \text{IRM1}_v, \hat{x}_{v,\,\text{IRM1}} = iSTFT(\hat{X}_{v,\,\text{IRM1}}) \tag{26}$$

In this section, I mentioned that discarding the phase of the complex STFT and applying masks only on the magnitude spectrogram is a common choice in music source separation (Fitzgerald 2010; Fitzgerald and Gainza 2010; Driedger, Müller, and Disch 2014; Stöter et al. 2019; Grais and Plumbley 2017; Grais, Zhao, and Plumbley 2021). This is done to avoid working with the phase of the STFT (Parry and Essa 2007). Figure 42 shows how the phase spectrograms of an audio signal and random noise look similar to one another.

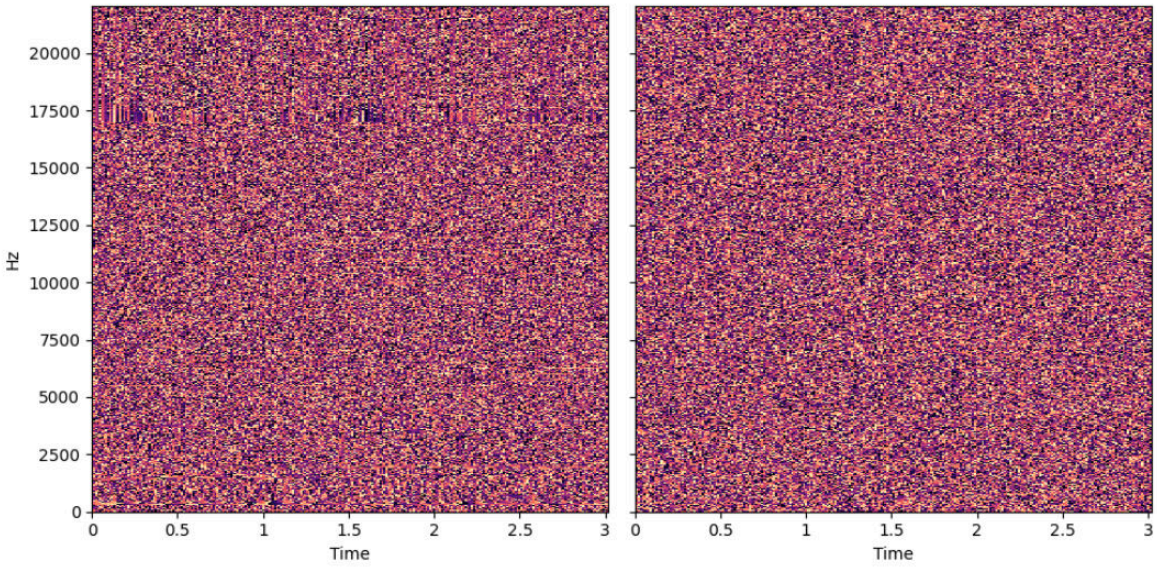


Figure 42: Phase spectrograms of an audio signal on the left and random noise on the right, showing that phase is difficult to model.*

*. https://source-separation.github.io/tutorial/basics/phase.html

Systems that only estimate the magnitude STFT of a source need a phase STFT to create a complex STFT, which can be inverted back to a time-domain waveform. In demixing and source separation literature, there is a strategy called "noisy phase" (Wichern et al. 2019; Mayer et al. 2017), where the estimated magnitude STFT of the source is combined with the phase STFT of the original mixed (or noisy) audio. The "noisy phase" strategy will be described in further detail next.

2.5.6 Noisy phase or mix-phase inversion (MPI)

In this section, I will describe the "noisy phase" strategy in music source separation, which is the strategy of estimating only the magnitude STFT of the source, and using the phase of the original mixed (or noisy) audio. I will also show the design of an oracle estimator for the "noisy phase" strategy to estimate its upper limit of quality. This is important for understanding the upper limit of performance of Open-Unmix (Stöter et al. 2019), a model for music source separation that will be described in Section 2.5.8, and that forms the base of the new model proposed by this thesis in Chapter 3.

The name "noisy phase" originates from speech separation, where a common task is to separate speech from noise. In music demixing, referring to interfering musical instruments as "noise" is inappropriate, and throughout this section and the rest of this thesis, I will use the term "mix-phase inversion" (MPI) to refer to the same idea.

Equations (27) show how the phase of the mixture and magnitude of the estimated source are used to create a waveform by some arbitrary music demixing system:

$$
\begin{aligned}
X_{\text{mix}} &= STFT(x_{\text{mix}}) \\
|X_{\text{source}}|_{\text{est}} &= MusicDemixingSystem(x_{\text{mix}}) \\
X_{\text{source, est}} &= |X_{\text{source}}|_{\text{est}} \cdot \angle X_{\text{mix}} \\
\hat{x}_{\text{source, est}} &= iSTFT(\hat{X}_{\text{source, est}})
\end{aligned}
\tag{27}
$$

In other words, the magnitude of the isolated source is combined with the phase, or the angle, of the mixed waveform, to produce a complex time-frequency transform, which is then inverted with the backward transform to obtain the estimated isolated source waveform.

The equations for the MPI oracle can be derived from equations (27). The MPI oracle can

be computed from ground-truth data, using equations (28):

$$
\begin{aligned}
X_{\text{mix}} &= STFT(x_{\text{mix}}) \\
X_{\text{source}} &= STFT(x_{\text{source}}) \\
\hat{X}_{\text{source, MPI}} &= |X_{\text{source}}| \cdot \angle X_{\text{mix}} \\
\hat{x}_{\text{source, MPI}} &= iSTFT(\hat{X}_{\text{source, MPI}})
\end{aligned}
\tag{28}
$$

The MPI oracle waveform should provide an idea of the upper limit of performance of a music source separation system that uses the MPI strategy.

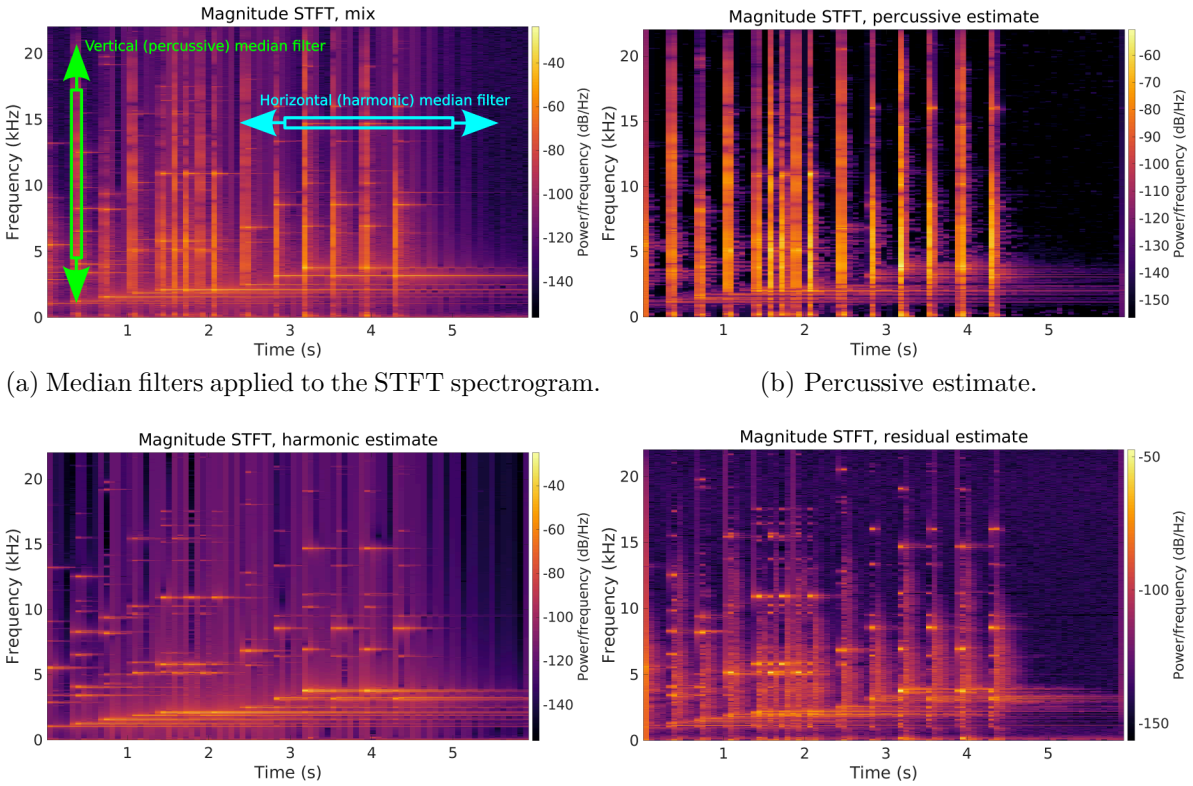
2.5.7 Harmonic/Percussive Source Separation

A simple case of music source separation is Harmonic/Percussive Source Separation (HPSS) (Cano et al. 2018). Harmonic (or steady-state, or tonal) sounds are narrowband and steady in time, while percussive (or transient) sounds are broadband and have a fast decay. Fitzgerald (2010) noted that they appear as horizontal and vertical lines, respectively, in the STFT, and applied a median filter in the vertical and horizontal directions to estimate the harmonic and percussive components. HPSS is in the category of KAM (kernel additive modeling) (Cano et al. 2018).

Median filtering is a technique for image processing where a pixel is replaced by the median value of its neighbors in a window, or filter, which slides across all pixels. Applying a median filter shaped like a horizontal rectangle (i.e., stretching in time) to the STFT causes vertical (or percussive) features to be diminished since their neighboring pixels are empty, which preserves horizontal (or harmonic) features. Similarly, applying a median filter shaped like a vertical rectangle (i.e., stretching in frequency) causes horizontal (or harmonic) features to be diminished, which preserves vertical (or percussive) features. From these estimates, soft masks are computed, which are applied to the original STFT and inverted to create the estimated harmonic and percussive signals. Driedger, Müller, and Disch (2014) replaced the soft mask with a binary/hard mask, which allows for estimation of a third component, the residual, which is neither harmonic nor percussive, and is described in further detail in Section 2.5.5.

The entire process of median-filtering HPSS with binary masks applied to the spectrogram of the glockenspiel signal is shown in Figure 43.

Driedger, Müller, and Disch (2014) also introduced a two-pass variant. The first pass separates the harmonic component using an STFT with a large window size for high frequency



(a) Median filters applied to the STFT spectrogram.

(b) Percussive estimate.

(c) Harmonic estimate.

(d) Residual estimate.

Figure 43: Outputs median filtering HPSS with binary masking applied to the glockenspiel signal.

resolution, followed by the second pass, which separates the percussive component using an STFT with a small window size for high time resolution. Fitzgerald and Gainza (2010) created a similar iterative variant using soft masks, a larger window size for the harmonic separation, and a smaller window size for the percussive separation. Additionally, they noted that when they replaced the small-window STFT with the CQT in the second pass, they obtained a separation of the human singing voice. These modifications of HPSS related to the time-frequency tradeoff of the STFT and CQT directly inspired the hypothesis of this thesis, which is that using the sliCQT instead of the STFT in a neural network for music source separation might improve the results due to the better time-frequency resolution characteristics of the sliCQT.

2.5.8 Open-Unmix (UMX)

In this thesis, my goal is to address the fixed time-frequency resolution limitation of the STFT in music source separation applications. Open-Unmix (UMX) was created by Stöter et al. (2019) as an open-source, reference implementation of a deep neural network for music

source separation based on the STFT. Due to the code being open-source and freely available, I chose to begin with UMX as the initial STFT-based model in this thesis. My goal is to replace the STFT with the sliCQT shown in Section 2.2.5.

UMX is based on time-frequency masking of the STFT, which was described previously in Section 2.5.5. UMX also only estimates the spectrogram of the target, and uses the phase of the mixed audio, which is a common strategy for spectrogram-based music demixing described previously in Section 2.5.6.

UMX uses the openly available MUSDB18 and MUSB18-HQ datasets for its training data (Rafii et al. 2017, 2019) to encourage reproducible research (Stöter et al. 2019). The Python source code of the reference implementation of UMX is openly available,[49] which uses PyTorch (Paszke et al. 2019) for GPU-accelerated numerical computation and deep learning.

In UMX, a deep neural network (DNN) is used to estimate the magnitude spectrograms of the sources from an input mixed song. The architecture of the DNN is a variant of a sequence2sequence model. UMX uses a bidirectional LSTM or Bi-LSTM architecture, which is based on two predecessor networks by Uhlich et al. (2017) and Graves and Schmidhuber (2005). Figure 44 shows the architecture of UMX, and Figure 45 shows one of its early predecessors.

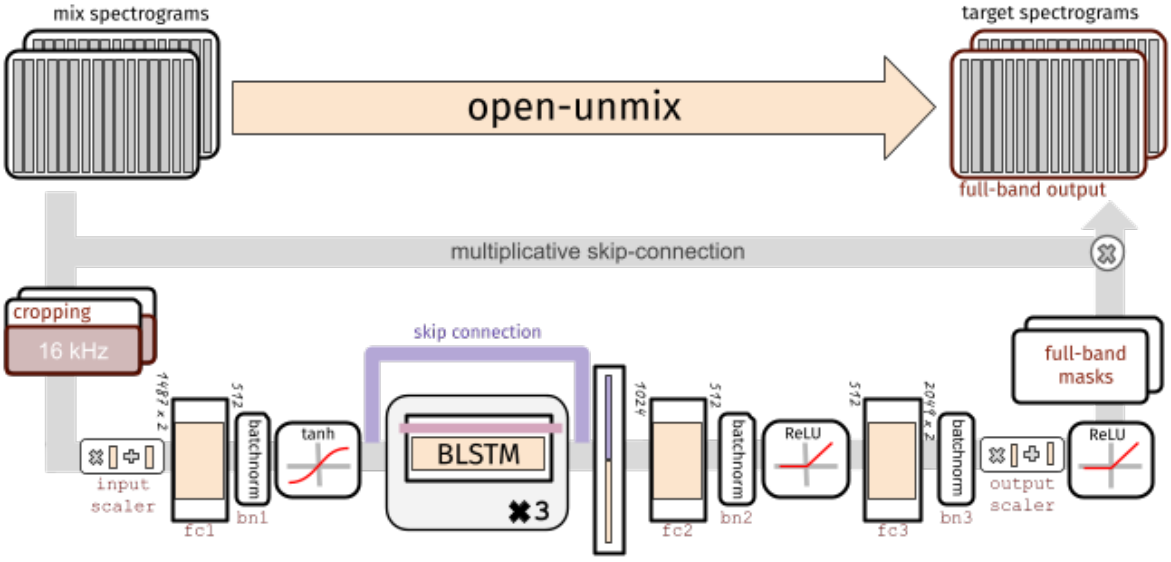


Figure 44: UMX Bi-LSTM architecture.*

*. https://github.com/sigsep/open-unmix-pytorch#-the-model-for-one-source

49. https://github.com/sigsep/open-unmix-pytorch

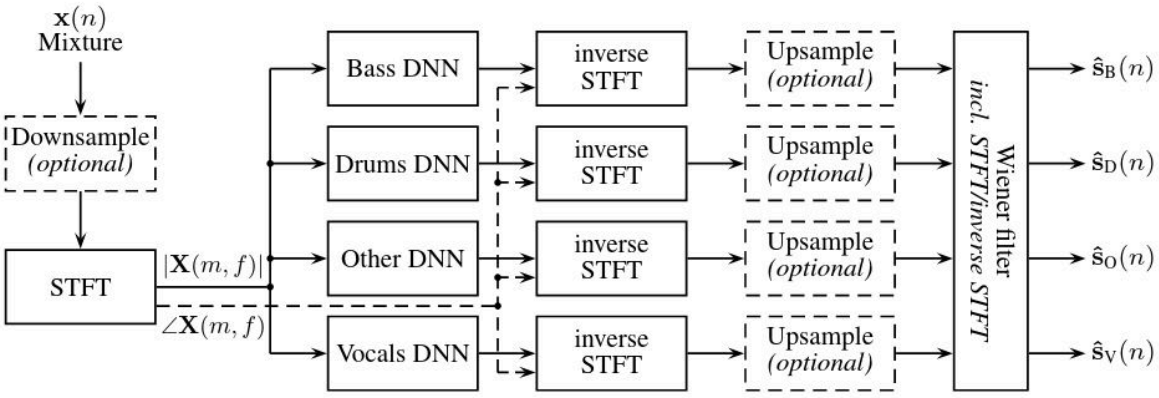


Figure 45: Simple DNN architecture for music source separation (Uhlich et al. 2017, 262).

The MUSDB18-HQ data loader is the starting point of the neural network, and it returns a single target at a time, or an (x, y) pair where x is the mixed waveform and y is the desired target. The final output of UMX is $\hat{y}$, the estimated target waveform. In between the input and output time-domain waveforms, the magnitude STFT domain is used by UMX. The important training and inference steps are as follows.

Data whitening refers to a process in machine learning where the mean μ and standard deviation σ of a training dataset X is used to normalize X, or to transform X into a normal Gaussian distribution X' with a mean of zero and a standard deviation of one (Kessy, Lewin, and Strimmer 2018). Data whitening is important because it "greatly simplifies multivariate data analysis both from a computational and a statistical standpoint," and "whitening is a critically important tool, most often employed in preprocessing" (Kessy, Lewin, and Strimmer 2018, 309).

In the training loop, the (x, y) pairs of mixed audio and the ground-truth of the target source waveform are converted to the magnitude STFT domain $(|X|, |Y|)$. To apply data whitening, the mean and standard deviation of $|X|$ are computed, or $\mu_{|X|} = mean(\{|X|\})$ and $\sigma_{|X|} = std(\{|X|\})$. The training input to the Bi-LSTM neural network is $(|X| + \mu_{|X|}) \times \sigma_{|X|}$. The Bi-LSTM neural network outputs $|\hat{Y}|$, the estimated magnitude STFT of the target sources.

The loss function is the mean-squared error (MSE) function. In the loss function, the estimated magnitude STFT of the target source is compared to the ground-truth, shown in equation (29).

$$MSE(Y, \hat{Y}) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{29}$$

where n is the number of elements in the magnitude STFT matrix.

There are four independent Bi-LSTM neural networks, shown in Figure 46, for each of the four sources: vocals, bass, drums, and other. The four Bi-LSTM models are trained until the loss stops improving. At this point, UMX is ready to perform inference.

First, the four independently-trained Bi-LSTM models are used to obtain the estimated magnitude STFTs of the four target sources. The mix phase inversion is used to create a first estimate of the waveforms for the four targets. Equations (30) show how the estimated magnitude STFT of UMX Bi-LSTM is converted to the time-domain waveform estimate:

$$
\begin{aligned}
X_{\text{mix}} &= STFT(x_{\text{mix}}) \\
|X_{\text{source}}|_{\text{est}} &= UMX(x_{\text{mix}}) \\
X_{\text{source, est}} &= |X_{\text{source}}|_{\text{est}} \cdot \angle X_{\text{mix}} \\
\hat{x}_{\text{source, est}} &= iSTFT(\hat{X}_{\text{source, est}})
\end{aligned}
\tag{30}
$$

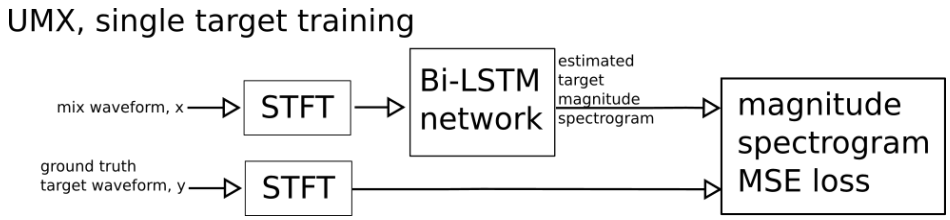
Similar equations were previously described for the mix-phase inversion (MPI) oracle in Section 2.5.6.

Finally, the four estimated magnitude STFTs are considered together, along with the phase of the STFT of the original mixed audio, to perform a post-processing step called iterative Wiener filtering and expectation-maximization (EM), jointly referred to as "Wiener-EM" (Uhlich et al. 2017; Duong, Vincent, and Gribonval 2010; Nugraha, Liutkus, and Vincent 2016a, 2016b). The output of the Wiener-EM step is a refined estimate of the waveforms of the four target sources, which is the final output of UMX.

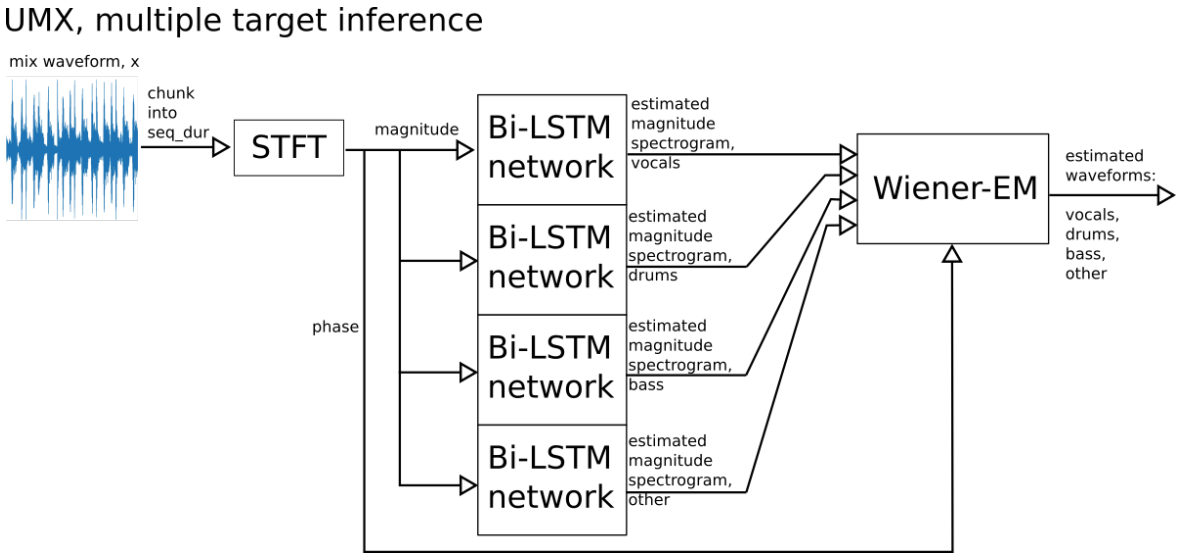
Figure 46(a) shows the training of a single UMX Bi-LSTM network, and Figure 46(b) shows the four independent networks performing inference for the four targets (vocals, drums, bass, other) of MUSDB18-HQ with the Wiener-EM post-processing step.

2.5.9 CrossNet-Open-Unmix (X-UMX)

I will now introduce a closely-related variant of Open-Unmix (UMX), called CrossNet-Open-Unmix (X-UMX) (Sawata et al. 2021). X-UMX improves on the music demixing performance of UMX without introducing any additional parameters in the neural network. Since X-UMX is mostly identical to UMX with better performance, I will be using it as the starting point for the new model that will be proposed by this thesis in Chapter 3.



(a) UMX, single target training.



(b) UMX, four targets inference.

Figure 46: UMX training for a single target and inference for four targets.

Sawata et al. (2021) combined the four separate target networks of UMX into a single model to apply the loss functions and optimizations jointly across all four targets, rather than optimizing each one separately, shown in Figure 47(b). X-UMX also includes loss computed from the time-domain waveforms, in addition to the loss measured from the magnitude spectral coefficients from the original UMX.

The loss function of X-UMX is first modified to include a multi-domain loss (MDL), which includes the existing frequency-domain loss (spectrogram MSE) in addition to a new time-domain loss. The time-domain loss is the same SDR score from the BSS metrics which was shown in Section 2.5.4.

I showed in equation (30) how to convert from the estimated magnitude spectrogram of the neural network to the waveform estimate. In X-UMX, these equations are used to compute the time-domain waveforms and apply the SDR loss function.

The next addition to the loss function of X-UMX is the combination loss (CL), which considers 14 possible combinations of the four target sources (vocals, drums, bass, other) in both

the spectral MSE loss and time-domain SDR loss. The 14 combinations are shown in Table 2.

Table 2: 14 combinations of the four targets in X-UMX.

Combination size	Combination number	Targets
1	1	bass
1	2	vocals
1	3	other
1	4	drums
2	5	bass + vocals
2	6	bass + other
2	7	bass + drums
2	8	vocals + other
2	9	vocals + drums
2	10	other + drums
3	11	bass + vocals + other
3	12	bass + vocals + drums
3	13	bass + other + drums
3	14	vocals + other + drums

Equations (31) show the computation of the total frequency-domain loss from the 14 combinations, where $|Y|$ represents the magnitude spectrogram of the ground-truth target waveform, and $|\hat{Y}|$ represents the estimated magnitude spectrogram of the neural network. The subscript of $|Y|$ and $|\hat{Y}|$ represents the target (1–4 for vocals, drums, bass, and other):

$$
\begin{aligned}
&\text{mse\_loss}_1 = \mathit{MSE}(|\hat{Y}|_1, |Y|_1) \\
&\text{(... repeat for size-1 combinations)} \\
&\text{mse\_loss}_5 = \mathit{MSE}(|\hat{Y}|_1 + |\hat{Y}|_2, |Y|_1 + |Y|_2) \\
&\text{(... repeat for size-2 combinations)} \\
&\text{mse\_loss}_{11} = \mathit{MSE}(|\hat{Y}|_1 + |\hat{Y}|_2 + |\hat{Y}|_3, |Y|_1 + |Y|_2 + |Y|_3) \\
&\text{(... repeat for size-3 combinations)} \\
&\text{mse\_loss}_{\text{total}} = \frac{1}{14} \cdot \sum_{n=1}^{14} \text{mse\_loss}_n
\end{aligned}
\tag{31}
$$

For the time-domain loss, the same 14 combinations are considered in the time domain, using the ground-truth time-domain waveforms y and the estimated time-domain waveform

$\hat{y}$ from the mix-phase inversion, shown in the equations (32):

$$\begin{aligned}
&\text{sdr_loss}_1 = SDR(\hat{y}_1, y_1) \\
&\text{(... repeat for size-1 combinations)} \\
&\text{sdr_loss}_5 = SDR(\hat{y}_1 + \hat{y}_2, y_1 + y_2) \\
&\text{(... repeat for size-2 combinations)} \\
&\text{sdr_loss}_{11} = SDR(\hat{y}_1 + \hat{y}_2 + \hat{y}_3, y_1 + y_2 + y_3) \\
&\text{(... repeat for size-3 combinations)} \\
&\text{sdr_loss}_{\text{total}} = \frac{1}{14} \cdot \sum_{n=1}^{14} \text{sdr_loss}_n
\end{aligned} \tag{32}$$

As a last step, the time-domain loss is multiplied by a mixing coefficient before adding it to the frequency-domain loss. The coefficient is set to 10 to "approximately equalize the ranges" of the two losses (Sawata et al. 2021, 4). The total loss function of X-UMX is calculated by equation (33):

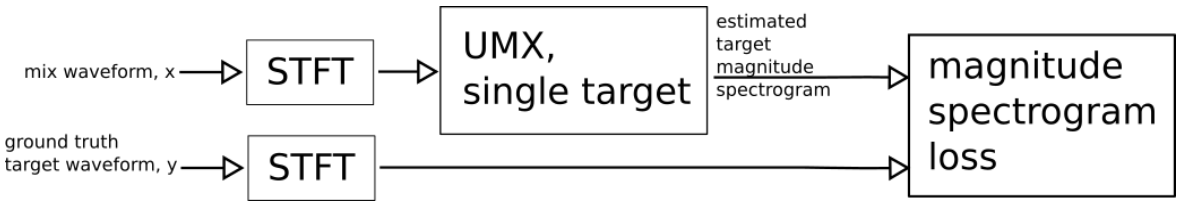
$$\text{loss}_{X-UMX} = \text{mse\_loss}_{\text{total}} + 10 \cdot \text{sdr\_loss}_{\text{total}} \tag{33}$$

Figure 47 shows how independent UMX networks for the desired four targets (vocals, drums, bass, other) are combined together in X-UMX, illustrating the key concepts of the multi-domain loss functions (MDL), which computes loss for both the magnitude spectrograms and time-domain waveforms, and the combination losses across the four targets (CL). A visualization of the multi-domain loss (MDL) and combination loss (CL) for four targets is shown in Figure 48.

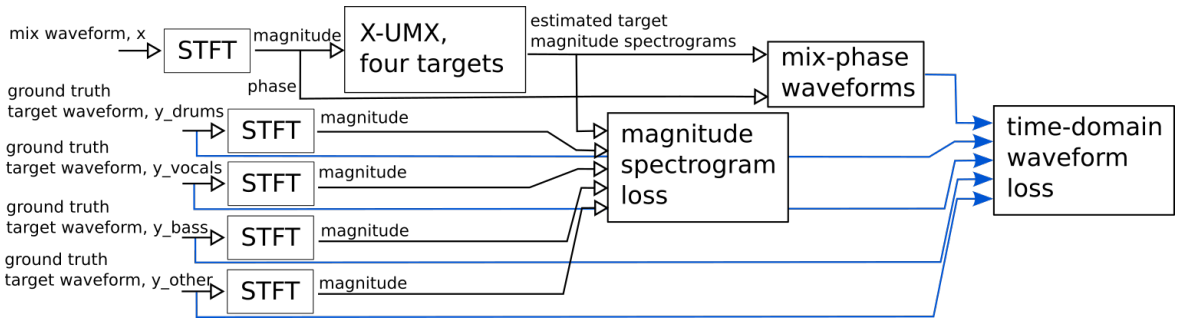
Figure 47 shows how X-UMX encompasses UMX entirely, and adds parameter-free enhancements with the new MDL and CL loss functions. For simplicity, having a single training loop and single trained model to perform the separation is easier than training four independent models. For this reason, and that the music demixing performance of X-UMX is higher than UMX, I chose to base this thesis on the X-UMX variant of UMX.

The available implementation of X-UMX[50] uses NNabla,[51] which is a different Python deep learning framework from PyTorch. The X-UMX additions are easily copied from the NNabla implementation of X-UMX into the PyTorch implementation of UMX.[52] In the upcoming sec-

50. https://github.com/sony/ai-research-code/tree/master/x-umx
51. https://nnabla.org/
52. https://github.com/sigsep/open-unmix-pytorch



(a) UMX system for a single target with loss function.



(b) X-UMX network for all four targets with MDL and CL loss functions.

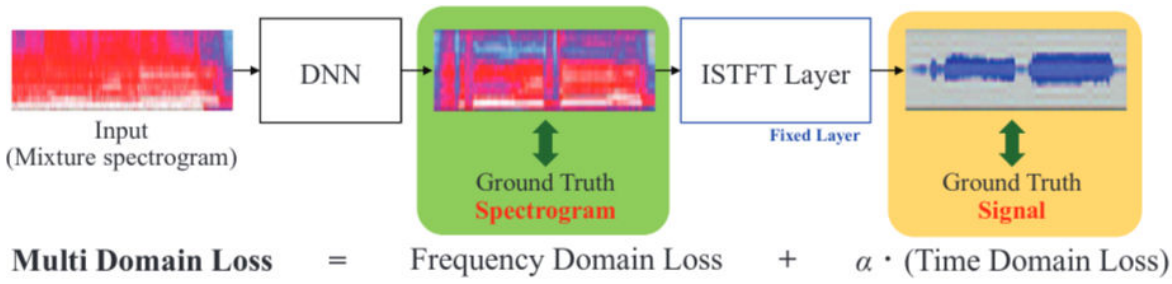
Figure 47: UMX and X-UMX loss functions compared. In UMX, a copy of the network is trained independently for each target, using single-target spectrogram loss. In X-UMX, four copies of the network are trained simultaneously for all four targets, using multi-domain and combination loss functions as shown in Figure 48.

tions, I will use X-UMX to refer to the PyTorch implementation of UMX with the changes of X-UMX included, since PyTorch is the deep learning framework chosen for this thesis.

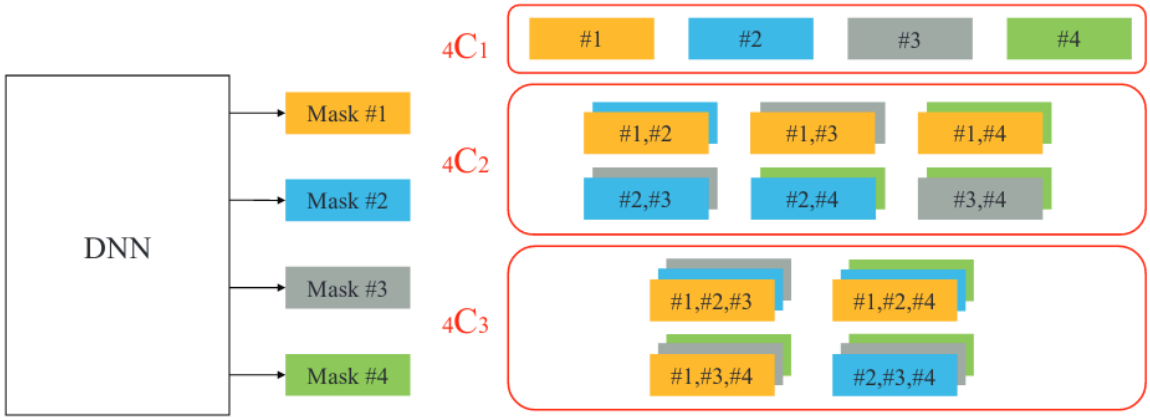
2.5.10 Convolutional denoising autoencoder (CDAE)

While Open-Unmix is based on an RNN architecture, a different type of neural network architecture based on CNN, the convolutional denoising autoencoder (CDAE), has been successfully used in music demixing (Grais and Plumbley 2017; Grais, Zhao, and Plumbley 2021). CDAEs have fewer learnable parameters than fully-connected network (FCN) or recurrent neural network (RNN) architectures with comparable music source separation performance (Grais and Plumbley 2017), resulting in a smaller model size. For this reason, I chose the CDAE as one of the models to base my work on in this thesis.

The architecture of a simple CDAE is shown in Figure 49, where 2D convolutions (in the time and frequency dimensions) are applied on the magnitude STFT. In the encoder layer, max-pooling layers are used after the convolutional layers to reduce the dimensionality of the input spectrogram and "extract robust low-dimension features from the input data" (Grais and Plumbley 2017, 1). In the decoder layer, up-sampling layers are used after the convolutional layers to increase the low-dimension encoded representation back to the



(a) Multi-domain loss (MDL), showing the frequency-domain MSE loss from the spectrograms and time-domain SDR loss from the waveforms summed with a mixing coefficient from equation (33).



(b) Combination loss (CL), showing the 14 combinations of the four targets from Table 2.

Figure 48: Diagrams of the loss functions of X-UMX (Sawata et al. 2021, 2).

original dimensions of the input spectrogram. In Section 2.3.1, I provided an overview on convolutional, deconvolutional, pooling, and up-sampling layers.

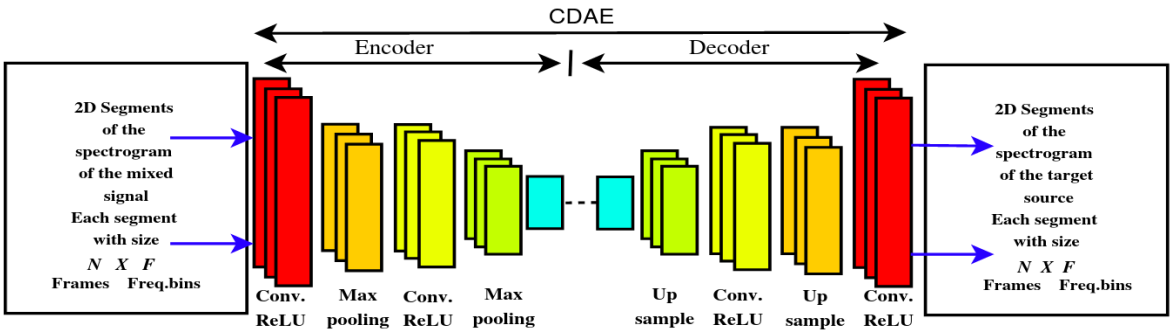


Figure 49: Diagram of a CDAE (Grais and Plumbley 2017, 2), where 2D convolutional layers are applied to the input spectrogram. Max-pooling layers are used in the encoder to reduce the dimensionality, and up-sampling layers are used in the decoder to increase the dimensionality.

3 Methodology

The proposed adaptation of CrossNet-Open-Unmix (X-UMX) to use the sliced Constant-Q Transform (sliCQ Transform or sliCQT) is named xumx-sliCQ,[53,54] and is the subject and main result of this thesis.

Decisions for the methodology were influenced by the limitations of the computer that xumx-sliCQ was developed on, e.g., the maximum GPU memory available. Refer to Appendix A for the hardware and software specifications of the computer. Appendix B contains links to all of the code referenced in this chapter.

CrossNet-Open-Unmix (X-UMX) is a near-state-of-the-art neural network for music demixing based on Short-Time Fourier Transform (STFT) masking, an overview of which was given in Section 2.5.8 and 2.5.9. The STFT has a fixed time-frequency resolution, while the sliCQT has a varying time-frequency resolution that may be more suitable for musical and auditory applications, as described in Section 2.2.4. xumx-sliCQ will be an adaptation of X-UMX which uses the sliCQT instead of the STFT, to investigate how this impacts its music demixing performance.

Figure 50 shows a block diagram for a general deep neural network (DNN) music demixing system that uses the spectrogram as its input and output representation, and uses the phase of the mixture to swap back to the time domain. This strategy was described in Section 2.5.6.

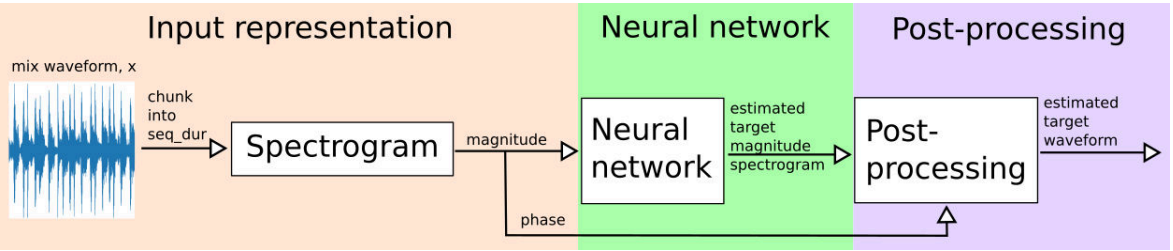
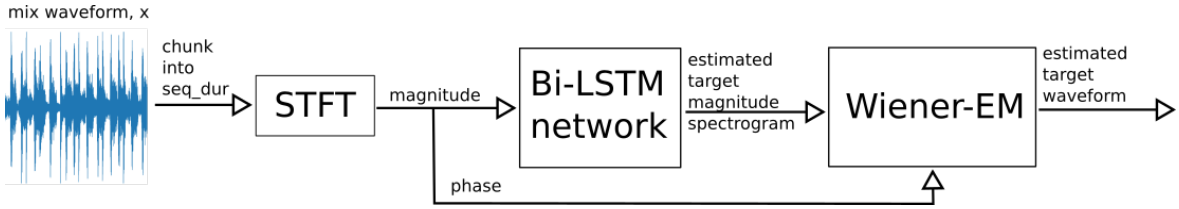


Figure 50: General DNN music demixing system with magnitude spectrograms. This diagram is shown for a single estimated waveform, but music demixing systems often estimate four waveforms for vocals, bass, drums, and other, following the example of the MUSDB18-HQ dataset.

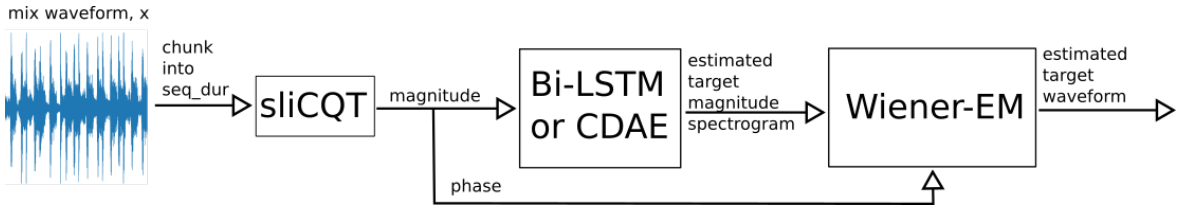
Figure 51 shows how X-UMX and xumx-sliCQ are variants of the above general system, and how they differ from each other. X-UMX uses the STFT spectrogram as the input representation of musical signals, and it uses a Bidirectional Long Short-Term Memory (Bi-LSTM)

53. https://github.com/sevagh/xumx-sliCQ/tree/main
54. xumx-sliCQ is pronounced like "X-U-M-X-slice-Q"

network architecture. xumx-sliCQ will use the sliCQT spectrogram as the input representation of music, and it will use either a Bi-LSTM or convolutional denoising autoencoder (CDAE) network architecture.



(a) X-UMX for one target.



(b) xumx-sliCQ for one target.

Figure 51: X-UMX and xumx-sliCQ compared. Note that only one target is shown for simplicity, but that both X-UMX and xumx-sliCQ estimate the four targets of vocals, drums, bass, and other from MUSDB18-HQ.

In Section 3.1, I will describe how I ported the sliCQT to use PyTorch, a Python framework to run deep neural networks (DNN) on the GPU. I will also describe new frequency scales I added to the sliCQT for music applications, and how I chose frequency scale parameters for the sliCQT that might perform well in music demixing. In Section 3.2, I will discuss the changes made to X-UMX to use the sliCQT instead of the STFT in xumx-sliCQ, the choice of Bi-LSTM and CDAE neural architectures, and the combining of the four independent target networks into a single model with the X-UMX loss functions. Finally, in Section 3.3, I will show how the post-processing iterative Wiener filtering and Expectation-Maximization (Wiener-EM) step was adapted for the sliCQT.

3.1 Input representation

First, in Section 3.1.1, 3.1.2, and 3.1.3, I will describe the differences between the output of the STFT and the sliCQT. In Section 3.1.4, I will describe how I chose an appropriate data structure to represent the sliCQT in PyTorch as a result of these differences.

Next, in Section 3.1.5, I will show the modifications made to the library to compute the transform with PyTorch on the GPU, so that it can be used in the X-UMX neural network code. I will also describe how the PyTorch sliCQT data structure differs from the

STFT.

In Section 3.1.6, I will show the addition of new frequency scales that are interesting for music applications that were described in Section 2.2.4. Recall from Section 2.2.4 that the sliCQT can be thought of as a filterbank, and the chosen frequency scale is the most important parameter that defines the spectral representation of the musical signal, which influences the results of the downstream application. In Section 3.1.7, I will explain how I added code to automatically choose the slice and transition length of the sliCQT based on the time-frequency resolution requirements of the desired frequency scale.

Finally, in Section 3.1.8, I will describe the parameter search that was run to choose frequency scales for the sliCQT with parameters that may perform best in a music demixing application. Section 3.1.9 describes modifications made to the BSS (blind source separation) metrics library to make it run faster on the GPU, with the aim to expedite the parameter search process.

3.1.1 Comparing the sliCQT to the STFT

In this section, I will compare and contrast the expected outputs of the STFT and sliCQT. This lays the groundwork for the chosen data structure to represent the sliCQT in PyTorch and its implementation, which will be shown in upcoming sections.

The STFT was described in Section 2.2.2 and its rectangular matrix output was shown in Figure 12.

The sliCQT was described in Section 2.2.5 as a realtime variant of the Nonstationary Gabor Transform (NSGT) shown in section 2.2.4. The sliCQT and NSGT both output a ragged matrix, characteristic to nonuniform frequency transforms described in 2.2.6 and shown in Figure 26. By contrast, the STFT can be thought of as a uniform frequency transform, because its frequency bins are uniformly or evenly spaced.

The sliCQT splits the input signal into fixed-size overlapping slices and computes discrete Fourier Transform (DFT) coefficients per slice, while the NSGT computes DFT coefficients for the whole input signal. To make the output of the sliCQT comparable to the NSGT, the slices must be overlap-added as described in Section 2.2.5.

From the above, there are two important and distinct characteristics that make working with the sliCQT different from the STFT:

1. The sliCQT computes the DFT coefficients for fixed-sized slices of the input signal. These slices need to be overlap-added with their adjacent slices to create the final

spectrogram. Note that the shape of the DFT coefficients of each slice is ragged, due to the nonuniform time-frequency resolution.

2. The overlap-added sliCQT spectrogram has the same ragged characteristic as the individual slices.

When designing a data structure for the sliCQT, each of these points has to be addressed separately.

3.1.2 3D shape of the sliCQT compared to the 2D STFT

The sliCQT was described in Section 2.2.5 as a realtime implementation of the NSGT, consisting of DFT coefficients computed for the input signal split into overlapping slices, shown in Figure 24. The sliCQT therefore returns the DFT coefficients of consecutive slices of length *sllen* of an input signal of length $N > sllen$. These slices need to be overlap-added with 50% of the previous and next slice to produce a spectrogram, because each slice is symmetrically zero-padded to twice its length to reduce time-aliasing, which was described in Section 2.2.5 and shown in Figure 25. This is a characteristic specific to the sliCQT. The NSGT, which is the ancestor of the sliCQT not designed for realtime processing, outputs the spectrogram directly without requiring any overlap-add.

Figure 52 shows the procedure of overlap-adding ragged adjacent slices of the sliCQT to produce the final overlap-added ragged transform.

Note that the 50% overlap is done independently for each of the time-frequency resolutions, since they have different temporal framerates and a different total number of frames per slice. For example, in the illustration above, the overlap for the lower green-colored block of frequency bins is two frames, or half of the total width of four frames, while the overlap for the upper pink-colored block of frequency bins or half the total width of six frames.

3.1.3 Raggedness of the sliCQT compared to the STFT

After overlap-adding the sliCQT, the result is a ragged matrix representing a nonuniform time-frequency transform with a different time-frequency resolution for groups of frequency bins. Figure 53 shows the output of the STFT and an arbitrary ragged nonuniform frequency transform such as the NSGT side-by-side.

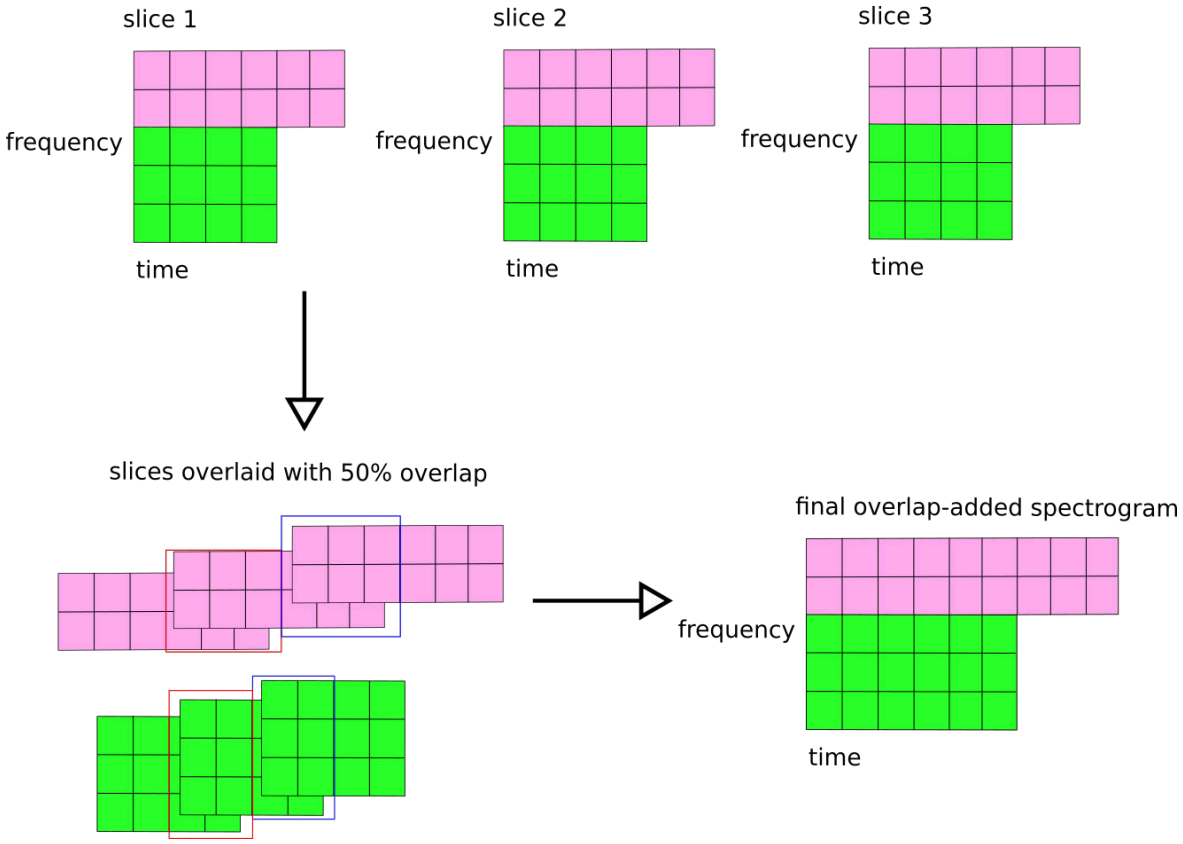


Figure 52: Overlap-adding adjacent ragged slices of the sliCQT to produce the final ragged output. The green-colored matrix represents the frequency bins analyzed with the first time-frequency resolution, and the pink-colored matrix represents the frequency bins analyzed with the second time-frequency resolution. The red rectangle shows the overlap between slice 1 and 2, and the blue rectangle shows the overlap between slice 2 and 3.

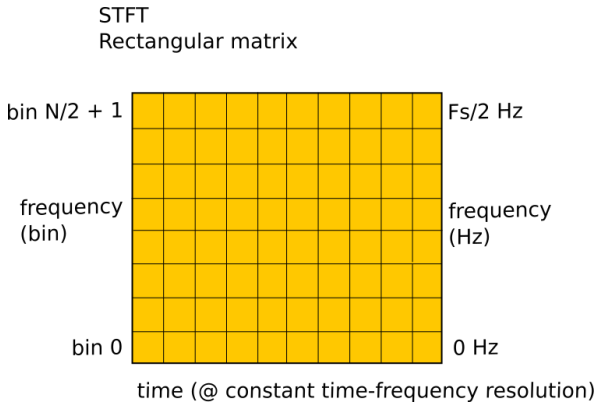
3.1.4 Choosing the data structure for the PyTorch sliCQT

When choosing a data structure to represent the sliCQT in PyTorch, the two characteristics described in the previous sections must be addressed, which are the adjacent slices that need to be overlap-added, and the raggedness of the slices and final overlap-added transform from the nonuniform time-frequency resolution.

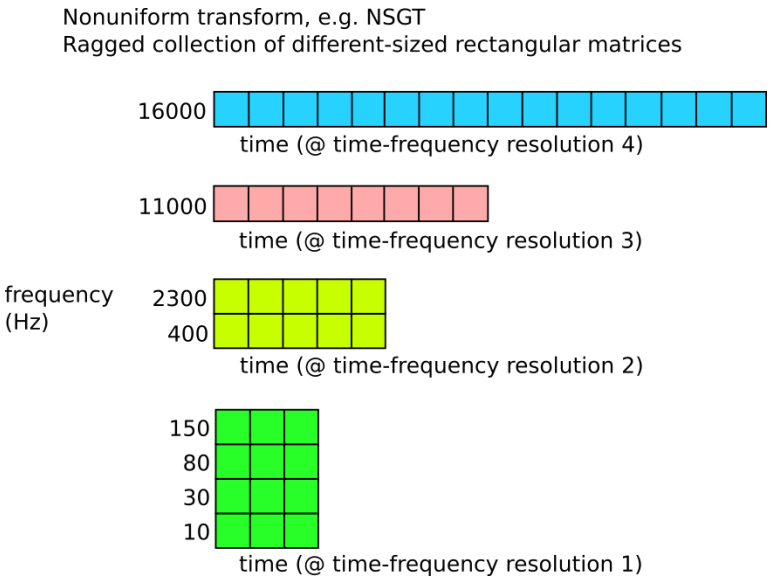
Due to limitations in the PyTorch library, jagged or ragged tensors (which are the equivalent of irregular or ragged matrices) are not yet supported.[55] That means that the ragged sliCQT cannot be represented by a single tensor, due to the irregular or ragged shape of the underlying matrix from the different time-frequency resolutions.

Therefore, to represent the raggedness of the noniform time-frequency resolution, the data structure chosen was a list of tensors. The list data structure in Python was described in

55. https://github.com/pytorch/pytorch/issues/25032



(a) STFT, a uniform time-frequency transform. The entire matrix is colored yellow, since every frequency bin is analyzed with the same (i.e., uniform) time-frequency resolution.



(b) A nonuniform time-frequency transform. The green, yellow, pink, and blue matrices represent the four different time-frequency resolutions used to analyze the different groupings of frequency bins.

Figure 53: The uniform STFT compared to a ragged nonuniform transform like the NSGT.

Section 2.4.1. Each element of the list contains a tensor for the groups of frequency bins that have the same time-frequency resolution and can therefore fit in the same tensor or matrix.

As described in the previous section, the sliCQT returns 2D time-frequency matrices per slice of the input signal, and multiple slices are needed to represent input signals longer than the slice length. Therefore, a 3D tensor with the dimensions: time, frequency, and slice are used to represent the transform of each slice. The 3D tensor, when overlap-added, is

converted to a 2D tensor with dimensions of time and frequency, which is equivalent to the 2D time-frequency dimensions of the STFT. Figure 54 shows how the list of 3D tensors is converted to a list of 2D tensors by the overlap-add procedure.

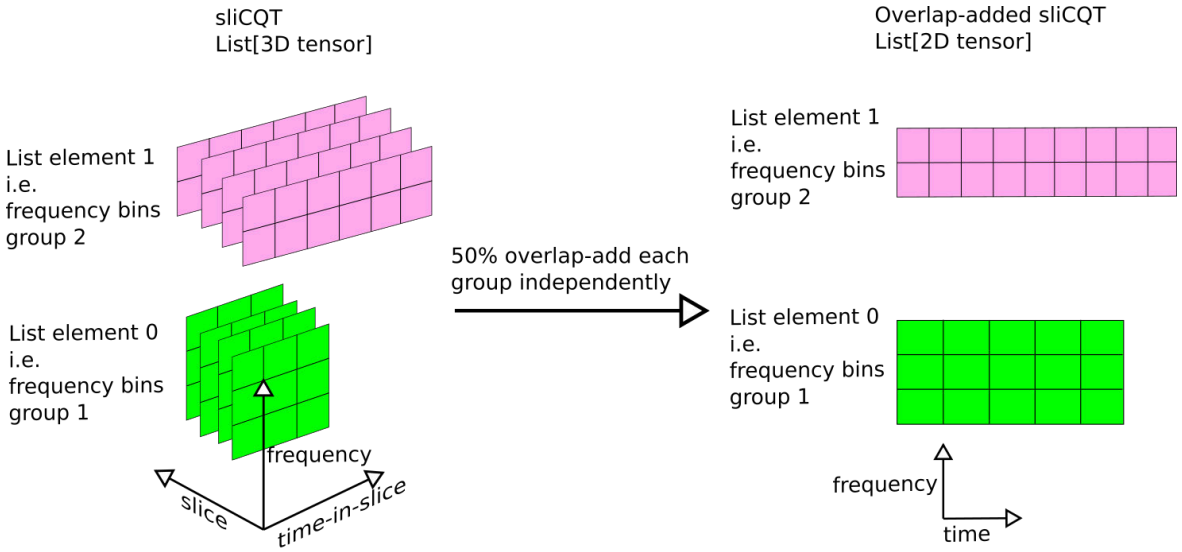


Figure 54: Overlap-adding the list of 3D tensors to create a list of 2D tensors in the sliCQT. The green-colored matrices represent the first time-frequency resolution, and the pink-colored matrices represent the second time-frequency resolution in the ragged list.

The final data structure for the sliCQT in PyTorch is a list of 3D tensors. Each 3D tensor in the list represents a grouping of frequency bins by their common time-frequency resolution. The 3D tensors of the list are each independently overlap-added to produce 2D tensors, which constitute the final time-frequency spectrogram representation of the audio signal. Figure 55 shows the data structure of the ragged overlap-added sliCQT compared to the data structure of the STFT.

3.1.5 Computing the sliCQT on the GPU with PyTorch

In this section, I will describe the procedure for porting the CPU-based code of the reference NSGT/sliCQT library to the GPU using PyTorch. This is needed to be able to use the sliCQT in the PyTorch GPU code of X-UMX. All the modifications to the library were made in my copy of the software library,[56] and the same library was embedded inside the xumx-sliCQ code.[57]

The reference NSGT/sliCQT library uses NumPy, a CPU-based Python numerical computation library. PyTorch, in addition to being a deep learning framework, is also a numerical

56. https://github.com/sevagh/nsgt
57. https://github.com/sevagh/xumx-sliCQ/tree/main

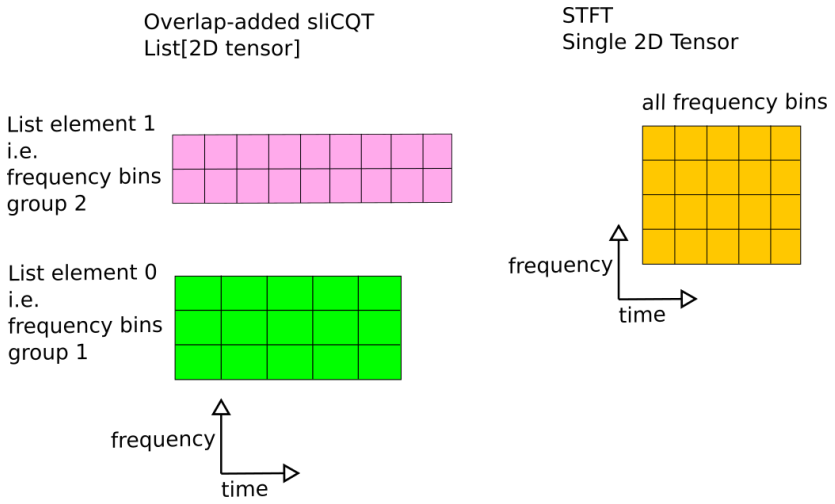


Figure 55: Ragged list of 2D tensors representing the overlap-added sliCQT spectrogram, compared to the single 2D tensor representing the STFT spectrogram. The sliCQT contains two colors, green and pink, representing two different time-frequency resolutions for different groups of frequency bins. The STFT contains a single color, yellow, representing the uniform time-frequency resolution used for all the frequency bins.

computation library that implements a large amount of NumPy's functions.

The line-profiler[58] package for Python prints a line-by-line output of the execution times of Python functions, and it was used to find performance hotspots and bottlenecks in the NSGT library. All NumPy functions and other non-GPU code was replaced with their PyTorch equivalents. To the user, the only difference was the addition of a `device` parameter to the `NSGT` and `NSGT_sliced` classes, which are the NSGT and sliCQT respectively. The choices of device are PyTorch device strings, and it defaults to the value `device="cpu"`. Using `device="cuda"` allows one to select their NVIDIA GPU. In practice, any device supported by PyTorch is also supported.

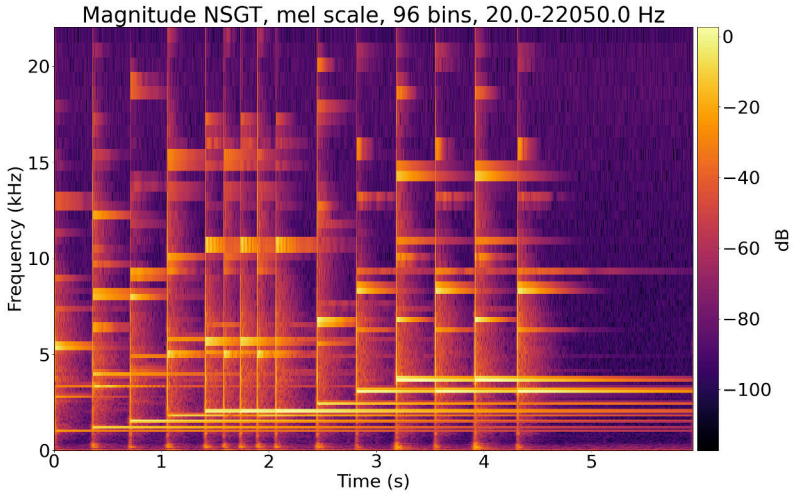
Many internal methods, functions, and utilities were modified during the port to PyTorch, and each in turn was also modified to take a `device` parameter. As these are internal to the library and not intended to be used directly by the end user, they will not be described here.

Due to requirement of the 50% overlap-add to create a spectrogram, the overlap-add utility is accompanied with a spectrogram plotting implementation. The spectrogram utility

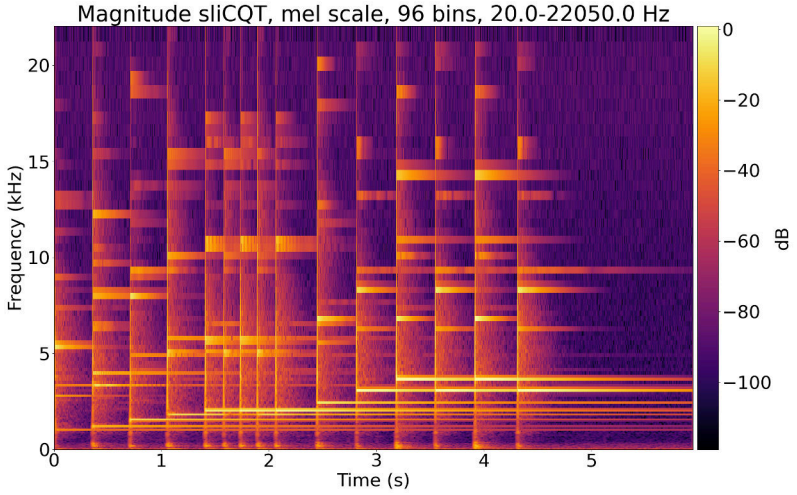
58. https://pypi.org/project/line-profiler/

function[59] uses the matplotlib[60] library functions pcolormesh[61] and colorbar[62] in its implementation.

Figure 56 compares the non-sliced NSGT spectrogram with the overlap-added sliCQT spectrogram, generated with the overlap-add and spectrogram utilities.



(a) NSGT (non-sliced), length is exact same as input signal. No overlap-add required.



(b) sliCQT with minimum automatic sllen (6,744 samples) and 50% overlap-add.

Figure 56: NSGT spectrograms for the mel scale with 96 bins in 20–22,050 Hz.

59. https://github.com/sevagh/nsgt/blob/main/nsgt/plot.py
60. https://matplotlib.org/
61. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.pcolormesh.html
62. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.colorbar.html

3.1.6 New frequency scales in the sliCQT library

In this section, I will discuss why I added new frequency scales to the reference NSGT/sliCQT library.[63]

The frequency scale is perhaps the most important parameter of the NSGT and sliCQT. It defines the nonuniform frequency bands for audio analysis desired by the user. The choice of frequency scale affects the spectral representation of the music signal. The hypothesis of this thesis is that a spectrogram generated from a musical or auditory frequency scale may improve the results of the downstream music demixing system compared to the STFT, which suffers from the time-frequency tradeoff described in Section 2.2.2.

The parameters to a frequency scale in the NSGT/sliCQT library are $f_{\min}$ and $f_{\max}$, which are the minimum and maximum frequency in Hz respectively, and the total frequency bins. The type of scale determines how the frequency bins are distributed between f_{min}–$f_{\max}$. A custom frequency scale may use additional parameters as needed.

The octave scale is the default scale used to demonstrate the NSGT and sliCQT (Balazs et al. 2011; Holighaus et al. 2013) in various open-source libraries and implementations.[64] The octave scale and log scale are almost identical, except that the bins parameter is interpreted as bins-per-octave for the octave scale, and the total number of output bins are computed using equation (10) from Section 2.2.3. In the log scale, the bins parameter is directly used as the number of output bins. In this thesis, I used the log scale to avoid the hidden bins-per-octave calculation. Appendix C contains a more in-depth comparison of the octave and log scales.

Section 2.2.4 showed examples of NSGT spectrograms with the log (Constant-Q) and mel scales, in Figures 22 and 23. These are the scales provided in the reference NSGT/sliCQT library. I also discussed that the Variable-Q scale for music and Bark psychoacoustic scale are both of interest in music and auditory applications. For this reason, I chose to implement the Variable-Q and Bark scales to supplement the Constant-Q and mel scales.

The first frequency scale I added was the Variable-Q scale,[65] which introduces an additional parameter, γ (Hz) for the offset. γ is a frequency offset added to each frequency band to reduce the Q-factor and improve the time resolution in the lower frequency bins, as described in Section 2.2.4. Figure 57 compares the Constant-Q and Variable-Q frequency scales and

63. https://github.com/grrrr/nsgt/blob/master/nsgt/fscale.py
64. https://github.com/grrrr/nsgt, https://www.mathworks.com/help/wavelet/ref/cqt.html, http://ltfat.org/doc/filterbank/cqt_code.html
65. https://github.com/sevagh/nsgt/blob/main/nsgt/fscale.py#L105

Q-factors for different values of γ. The Q-factor of Constant-Q scale was defined as the relative frequency resolution in Section 2.2.3.

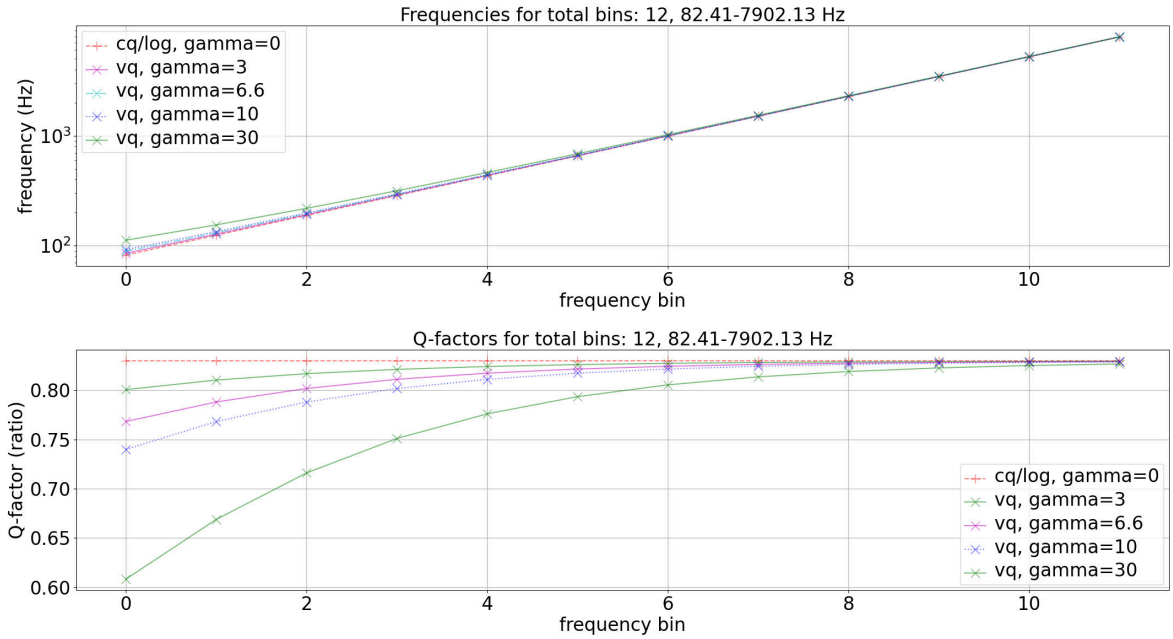


Figure 57: Frequency bins and Q-factors for the Constant-Q and Variable-Q scales.

The next frequency scale I added was the Bark psychoacoustic scale,[66] to complement the included mel scale, as discussed in Section 2.2.4. Figure 58 shows the mel and Bark frequency scales and Q-factors.

3.1.7 Automatic slice length picker

As mentioned in Section 2.2.5, the sliCQT processes the input signal in fixed-sized slices, ideal for realtime processing and for input signals with arbitrary lengths. Compared to the NSGT, the sliCQT has an additional need for slice length and transition length parameters. To analyze an input musical signal with nonuniform frequency scales, the slice and transition length of sliCQT need to be set appropriately to support the varying time-frequency resolution. I added code to the sliCQT library to automatically choose the minimum appropriate slice length and transition length for a given frequency scale.[67] This improves the user experience by only requiring the frequency scale as the parameter to the sliCQT.

66. https://github.com/sevagh/nsgt/blob/main/nsgt/fscale.py#L207
67. https://github.com/sevagh/nsgt/blob/main/nsgt/fscale.py#L36

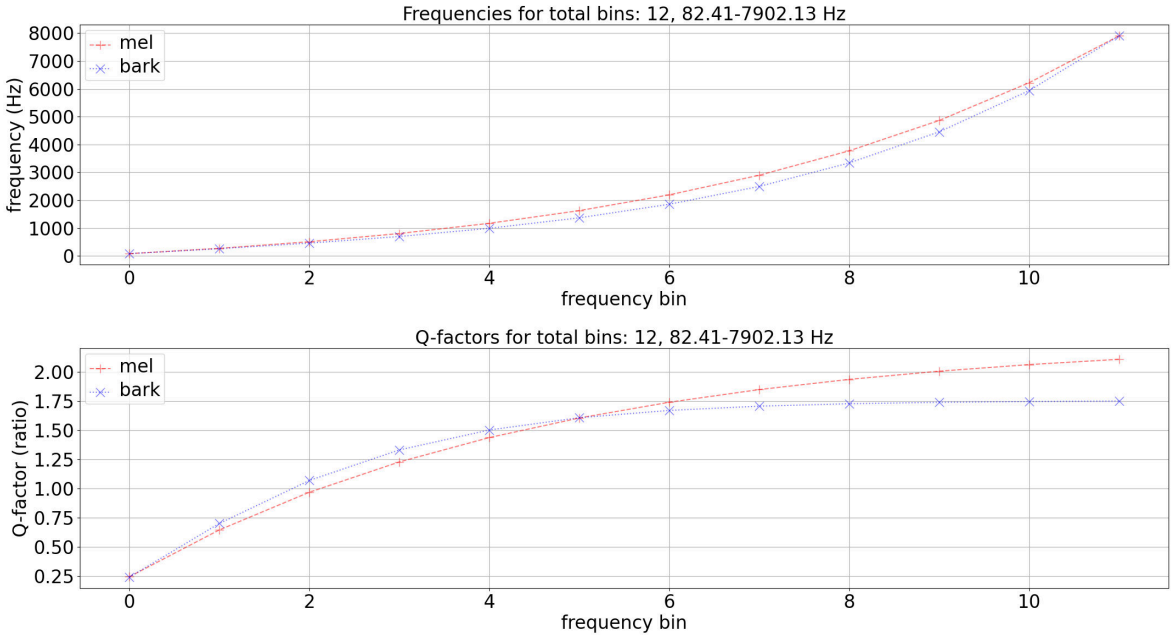


Figure 58: Frequency bins and Q-factors for the mel and Bark scales.

3.1.8 Choosing sliCQT parameters

In this section, I will design a sliCQT parameter search to discover optimal sliCQT parameters that produce the best mixed phase inversion (MPI) oracle results. MPI is the music demixing strategy where the target magnitude spectrogram is combined with the phase of the mix spectrogram to create a time-domain target waveform estimate, as was described in Section 2.5.6. The MPI oracle represents the best possible result of the MPI strategy by using the ground truth target magnitude spectrogram. Since the MPI strategy is used in X-UMX and xumx-sliCQ, my hypothesis is that maximizing the MPI oracle result would maximize the music demixing performance of the final neural network.

As discussed in Section 2.3, the validation split of a dataset is often used to tune hyperparameters, which are the parameters of a machine learning or deep learning network defined by the user. In xumx-sliCQ, the sliCQT parameters are a hyperparameter, similar to how the STFT settings of the window and hop size are hyperparameters in X-UMX.

As described in Section 3.1.6, the time and frequency resolution of the sliCQT are defined from the frequency scale, and the parameters of the frequency scale are f_{min} and f_{max}, which are the minimum and maximum frequency in Hz respectively, and the total frequency bins.

The maximum frequency f_{max} was fixed to 22,050 Hz, which is the Nyquist rate of the 44,100 Hz sample rate of MUSDB18-HQ. I made this decision after early experiments showed that

fixing the maximum frequency to 22,050 Hz in several cases led to a smaller size of the sliCQT than choosing a smaller $f_{\max}$, resulting in lower memory demands on the system. The results of these early experiments are shown in Appendix D.

The minimum frequency $f_{\min}$ was chosen from the range of 10.0–130.0 Hz, loosely based on the frequencies of the C0 (16.35 Hz) and C3 (130.81 Hz) notes of the well-tempered scale.

The total frequency bins were chosen from the range of 10–300 bins. The 300-bin maximum was determined by the maximum size of sliCQT that could fit in the 12 GB GPU memory of the NVIDIA 3080 Ti GPU. Details of the hardware limitations of the testbench computer are shown in Appendix A.

The maximum slice length was capped to 44,100 samples, which represents one second of music. This value was chosen because full-length songs are typically split into one second subsequences by several demixing papers (Grais and Plumbley 2017; Grais, Zhao, and Plumbley 2021; Défossez et al. 2019). Since the slice length and transition length are automatically picked for a given frequency scale using the new feature whose addition was described in Section 3.1.6, the parameter search discards combination of parameters whose slice length exceeds the maximum.

The random search parameters are shown in Table 3. Note that an instance of the script was run for each of the four frequency scales: Constant-Q, Variable-Q, mel, and Bark.

Table 3: Parameter ranges for the sliCQT parameter search.

Scale	Total bins	$f_{\min}$ (Hz)	Additional params
Constant-Q	10–300	10.0–130.0	n/a
Variable-Q	10–300	10.0–130.0	γ = 10.0–130.0 Hz
mel	10–300	10.0–130.0	n/a
Bark	10–300	10.0–130.0	n/a

The space of possible parameters is infinite, even with the ranges constrained as in Table 3. Therefore, an exhaustive grid search over all the possible parameters is impossible. To search for hyperparameters in a neural network, Bergstra and Bengio (2012) found that:

> [o]ften the extent of a search is determined by a computational budget, and with random search 64 trials are enough to find better models in a larger less promising space (Bergstra and Bengio 2012, 293).

I decided to use a random search with at least 64 iterations for the sliCQT parameters, since the sliCQT parameters are hyperparameters of the xumx-sliCQ neural network. I ran two

instances of the random parameter search script with 60 iterations each, resulting in 120 total iterations of the random parameter search for each of the four frequency scales. Each instance of the script used a different random seed (42 and 1337). The purpose of fixing the seed of Python's random number generator (RNG) is discussed in Section 2.4.1.

Recall from Section 2.5.4 that the Signal-to-Distortion (SDR) is one of the scores of the BSS metrics, with a unit of decibels (dB), which is commonly used as a single metric that summarizes the global performance of a music demixing system. The selected sliCQT frequency scale parameters were evaluated by computing the mix-phase waveforms, which were described in Section 2.5.6. The phase of the complex sliCQT of the mixed song and the magnitude of the complex sliCQT of the ground-truth source signal for all four sources (drums, bass, vocals, other) were combined to create the MPI (mix-phase inversion) waveforms. The median SDR was computed across the four sources and 14 validation tracks of the MUSDB18-HQ dataset, and the sliCQT parameters that resulted in the MPI waveforms with the highest SDR were selected as the final sliCQT parameters to use in the neural network.

3.1.9 Speeding up the parameter search using the GPU and CuPy

In this section, I will explore a speedup of the parameter search script using the GPU. The parameter search script described in the previous section computes the MPI oracle waveforms and measures their SDR score on the 14 validation tracks of MUSDB18-HQ. As multiple instances of the script were run for several iterations during the experimentation phase, reducing the total time of the parameter search script would allow for the script to run in a shorter time frame, leading to a faster development cycle.

As in Section 3.1.5, the Python line-profiler library was used to discover any performance bottlenecks or slow parts of the MPI oracle parameter search script. The slowest step was the calculation of the SDR score from the BSS metrics library.[68]

The bottlenecks of the BSS metrics library are caused by NumPy and SciPy operations, which are Python numerical computation libraries that use the CPU. The CuPy library[69] provides drop-in replacement functions of NumPy and SciPy, which run on NVIDIA GPUs using the CUDA compute framework.[70] This allows for acceleration of numerical operations through GPU parallelization. Note that although PyTorch was used to replace NumPy code in the earlier Section 3.1.5, CuPy is intended to be a direct replacement of NumPy and SciPy

68. https://github.com/sigsep/sigsep-mus-eval
69. https://cupy.dev/
70. https://developer.nvidia.com/cuda-toolkit

and thus represents an easier porting effort. Also, since the MPI oracle testbench is separate from the xumx-sliCQ PyTorch code, it does not need to use PyTorch.

I copied the BSS metrics library to my own GitHub user account,[71] which is called `sigsep-mus-eval` in the SigSep GitHub organization, and replaced the slowest NumPy and SciPy functions with their CuPy equivalents,[72] taking special care about GPU out-of-memory errors. The library is unchanged for end users, and the GPU is simply used whenever possible, falling back to the CPU in case of any errors, or if there is no GPU available on the user's computer.

71. https://github.com/sevagh/sigsep-mus-eval
72. https://github.com/sevagh/sigsep-mus-eval/commit/f3b7540faddc8d2b1bc91cecaf05fd20d96df7c6

3.2 Neural networks

First, in Section 3.2.1, I will look at how the X-UMX PyTorch code needs to be modified in xumx-sliCQ to replace the STFT with the ragged sliCQT described in Section 3.1.5.

Next, in Section 3.2.2, I will show two neural network architectures, Bi-LSTM and CDAE, that can be used in xumx-sliCQ. I will show how the original Bi-LSTM architecture from X-UMX and the convolutional denoising autoencoder (CDAE) architecture described in Section 2.5.10, both originally intended for the STFT, can be adapted to operate on the ragged sliCQT. In Section 3.2.3, I will describe the bandwidth parameter of X-UMX and how it was adapted for the sliCQT in xumx-sliCQ. In Section 3.2.4, I will discuss a de-overlap layer, which is necessary for any neural architecture for the sliCQT.

Finally, in Section 3.2.5, I will show how I incorporated the X-UMX idea of combining the four target models in a single model with the multi-domain loss (MDL) and combination loss (CL) functions.

3.2.1 Replacing the STFT with the sliCQT in X-UMX

In this section, I will describe the changes made to X-UMX to support the data-structure of the sliCQT shown in Section 3.1.4, which addressed the distinct traits of the sliCQT compared to the STFT: the 3D slice-wise output, and the ragged shape from the nonuniform time-frequency resolution.

Firstly, X-UMX uses a sequence duration of six seconds, meaning the training input is the STFT of six seconds of music. The dimensionality of the sliCQT for a waveform is an order of magnitude larger than the STFT for a variety of different sliCQT parameters, as was discovered during preliminary experiments shown in Appendix E. In these experiments, the sliCQT of a six second sequence used too much GPU memory during training. In xumx-sliCQ, I chose a smaller sequence duration of one second, resulting in a sliCQT with fewer coefficients that uses less GPU memory. Also, a one-second sequence duration is common in several demixing papers (Grais and Plumbley 2017; Grais, Zhao, and Plumbley 2021; Défossez et al. 2019).

In sections 2.2.6 and 3.1.4, I described the data structure of the ragged sliCQT and how it differed from the single rectangular matrix of the STFT. The ragged sliCQT outputs a list of 3D matrices, representing the sliced transform, that must be overlap-added to create a list of 2D time-frequency matrices. Recall that the list was introduced from the raggedness of the transform, because each group of time-frequency bins has a different temporal framerate, resulting in a different size tensor or matrix. This raggedness is a desired characteristic

of nonuniform time-frequency transforms like the NSGT, sliCQT, and even the original CQT.

First, the list of 3D tensors of the sliCQT must always be overlap-added to transform them to the 2D time-frequency matrix. Next, all of the operations of X-UMX that expect a single rectangular matrix of the STFT need to be modified to work on a list of matrices instead, where each separate matrix in the list consists of a group of frequency bins that share the same temporal framerate, ordered from the lowest to highest frequency bins. Lastly, the list of 2D tensors in the overlap-added sliCQT needs to be converted back to the list of 3D tensors in the original sliCQT to synthesize an audio waveform. This was achieved with a final transpose convolutional layer at the end of the neural network, whose effect is to double the time coefficients to reverse their 50% reduction from the overlap-add. Then, the 2D shape is reshaped back to the original 3D shape. The process of the de-overlap is shown in greater detail in Section 3.2.4.

3.2.2 Neural network architectures

In this section, a choice of Bi-LSTM and CDAE architectures for the ragged sliCQT in xumx-sliCQ will be presented. I copied the Bi-LSTM code from the original model,[73] making adaptations as necessary to support the sliCQT instead of the STFT. I implemented the CDAE code from the ground up, recreating the CDAE architectures described in Grais and Plumbley (2017) and Grais, Zhao, and Plumbley (2021) with the convolutional layers provided by PyTorch.

Figure 59 shows how both the Bi-LSTM and CDAE network architectures should be applied to the ragged sliCQT. Instead of having a single neural network operate on the single matrix of the STFT, which includes all of the frequency bins at the same temporal frame rate, there must be a neural network that operates on each of the sub-matrices of the ragged sliCQT.

Note that both the Bi-LSTM and CDAE architectures use the 2D overlap-added sliCQT, and need a de-overlap layer, which is discussed in Section 3.2.4.

In the original X-UMX, the neural network architecture is a dense (i.e., linear) encoder-decoder with a Bi-LSTM. The STFT in X-UMX uses a window size of 4096, which results in 2049 output frequency bins. The purpose of the dense encoder/decoder with the STFT is to reduce the 2049 frequency bins into a smaller dimension set of 512 values, which are then used as the input and output of the Bi-LSTM. In the case of the sliCQT, the output frequency bins

73. https://github.com/sigsep/open-unmix-pytorch/blob/master/openunmix/model.py#L43

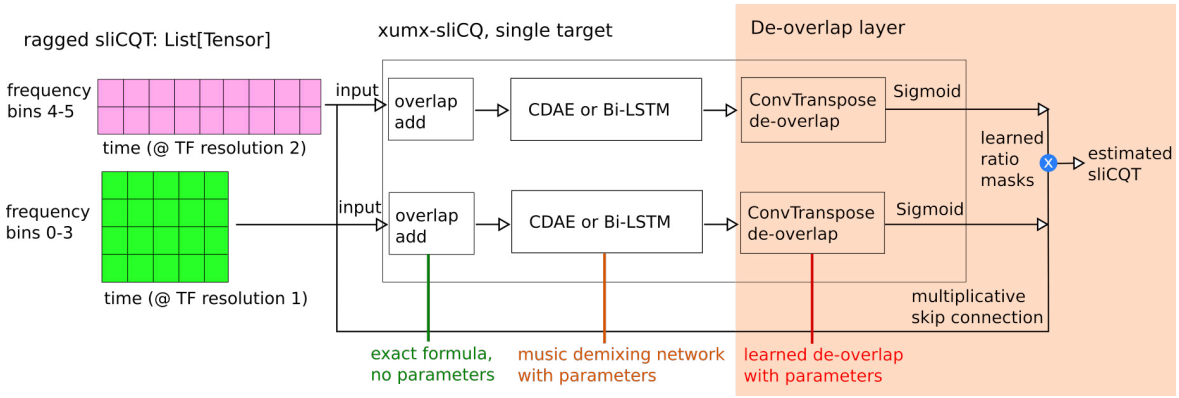
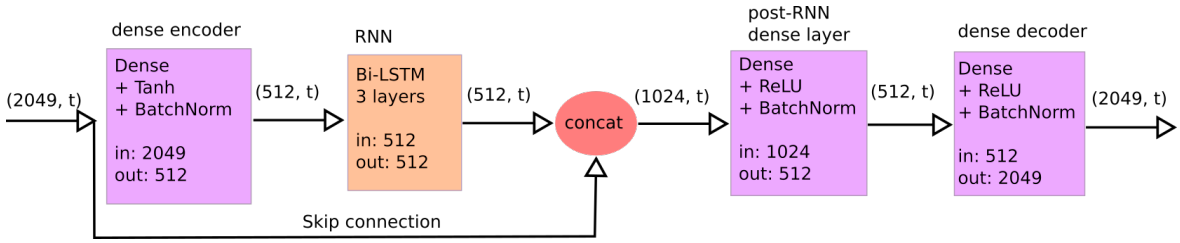


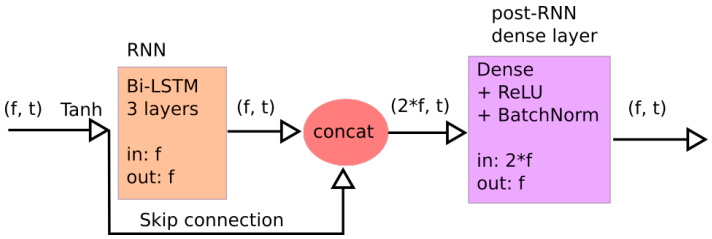
Figure 59: Block diagram of the individual CNNs of xumx-sliCQ, using a simplified ragged sliCQT for demonstration purposes. The green-colored matrix represents the first time-frequency (TF) resolution and the pink-colored matrix represents the second time-frequency resolution in the ragged transform. The de-overlap layer is described in Section 3.2.4.

are less numerous than the STFT. As described in Section 3.1.8, the maximum frequency bins of the sliCQT were capped at 300, which is a small enough value to use directly as the input to the Bi-LSTM without needing to be encoded to a lower dimension with dense layers.

Figure 60 shows the details of the Bi-LSTM for the STFT with the X-UMX default 2049 frequency bins, and how it was adapted to a sliCQT that has a smaller set of frequency bins $f \ll 2049$ (f was constrained to $[10, 300]$ in Section 3.1.8).



(a) Dense encoder/decoder + Bi-LSTM architecture for the STFT with 2049 frequency bins.



(b) Bi-LSTM architecture for the sliCQT with $f \ll 2049$ frequency bins.

Figure 60: The original X-UMX Bi-LSTM for the STFT, and the variant adapted for the sliCQT.

As discussed in sections 2.3.1 and 2.5.10, convolutional layers, and specifically convolutional denoising autoencoders (CDAE), are simple ways of defining 2D time-frequency filters that slide over a 2D time-frequency matrix of Fourier coefficients. The CDAE models of Grais and Plumbley (2017) and Grais, Zhao, and Plumbley (2021) were adapted for use in xumx-sliCQ.

Using the values from Grais, Zhao, and Plumbley (2021), the two layers of convolution in the CDAE of xumx-sliCQ use 25 and 55 channels respectively. This means that the stereo or two-channel sliCQT has 25 output feature maps (or channels) computed from the first encoder convolutional layer, and 55 output feature maps from the second encoder convolutional layer. These are reversed in the symmetric decoder layer. Figure 61 shows the CDAE architecture. The sub-matrices of the ragged sliCQT may have different dimensions for time and frequency, due to their varying frequency bins and temporal framerates, so the 2D convolution filter sizes are adapted to the input size. After each layer, a 2D batch normalization (BN) and nonlinear activation function, specifically the rectified linear unit (ReLU) activation, are applied (Grais, Zhao, and Plumbley 2021).

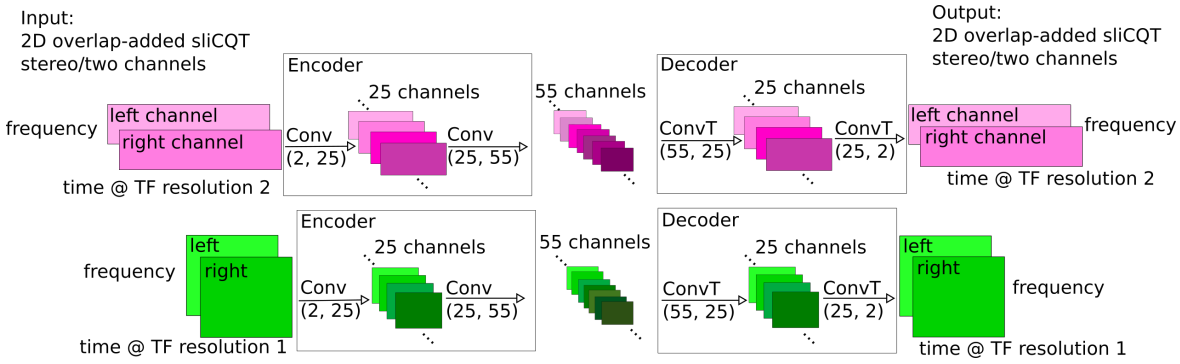


Figure 61: Symmetric encoder/decoder architecture of the xumx-sliCQ CDAE. The green-colored matrices represent the first time-frequency (TF) resolution and the pink-colored matrices represent the second time-frequency resolution of the ragged sliCQT.

The exact parameters and configuration of the convolutional kernels and layers are described in Tables 4, 5, 6, and 7. The parameters were initially chosen from the kernel values and layers in Grais and Plumbley (2017) and Grais, Zhao, and Plumbley (2021). The parameters and network architectures were repeatedly modified in an informal tuning process, and an overview of all the historical experiments of xumx-sliCQ is included in Appendix F.

3.2.3 Bandwidth parameter

X-UMX has a bandwidth parameter that controls how many of the 2049 output frequencies of the STFT with a window size of 4096 are used as an input to the neural network. The

Table 4: Kernel parameters, time dimension.

# time coefficients	CDAE layer
≤ 100	size=7 stride=1 dilation=2
> 100	size=13 stride=1 dilation=2

Table 5: Kernel parameters, frequency dimension.

# frequency bins	CDAE layer
≤ 10	size=1 stride=1 dilation=1
$\geq 10, < 20$	size=3 stride=1 dilation=1
≥ 20	size=5 stride=1 dilation=1

Table 6: CDAE encoder layers.

Layer 1	Layer 2
Conv2d$\binom{\text{in=2}}{\text{out=25}}$, BN, ReLU	Conv2d$\binom{\text{in=25}}{\text{out=55}}$, BN, ReLU

Table 7: CDAE decoder layers.

Layer 1	Layer 2
ConvTranspose2d$\binom{\text{in=55}}{\text{out=25}}$, BN, ReLU	ConvTranspose2d$\binom{\text{in=25}}{\text{out=2}}$, BN, ReLU

default value of the bandwidth parameter is 16,000 Hz, which means that the frequency bins of the STFT $> 16,000$ Hz are cropped before the STFT is input into the neural network. I implemented the same parameter in both Bi-LSTM and CDAE architectures of xumx-sliCQ, by not using frequency bins of the sliCQT $> 16,000$ Hz as inputs to the neural network.[74]

74. Note that since the sliCQT and STFT have different frequency bins and spectral bandwidths at each bin, cropping frequency bins above 16,000 Hz may result in a different effective spectral bandwidth in each transform respectively.

3.2.4 De-overlap layer

In this section, I will discuss the "de-overlap layer" that I created, which is necessary for both the Bi-LSTM and CDAE architectures of xumx-sliCQ. I described in Section 2.2.5 and 3.1.2 that the 3D sliCQT needs its adjacent slices to be overlap-added by 50% to create a spectrogram, due to the symmetric zero-padding of each slice introduced by Holighaus et al. (2013) in the sliCQT to reduce time-domain aliasing. This 50% overlap-add procedure does not have an exact inverse operation.

In the neural network of xumx-sliCQ, before applying either the Bi-LSTM or CDAE architectures, the first step is to overlap-add the 3D sliCQT to create a 2D time-frequency sliCQT. This means that in the final step of the neural network, the 2D overlap-added sliCQT needs to be transformed back to its original, non-overlap-added 3D shape. For this, I needed to create a custom layer in the neural network that could learn how to do the inverse overlap-add.

In Figure 30, I showed how the transpose convolution or deconvolution operation, which can grow its input to a larger size. For the de-overlap layer of xumx-sliCQ, I chose to use a transpose convolution layer with a kernel size of $(1, 3)$ and a stride of $(1, 2)$ where the first dimension is frequency and the second is time. This means that for each frequency bin, the kernel slides over three time coefficients to produce six time coefficients. The effect is to double the number of time coefficients, which is a reversal of the 50% reduction of time coefficients from the overlap-add procedure. Finally, the 2D sliCQT is reshaped to the original 3D shape to separate it back out into adjacent slices. The final activation after the transpose convolution layer is a Sigmoid function, which is real-valued $\in [0.0, 1.0]$. This makes the network estimate a soft or ratio mask that can then be multiplied with the input spectrogram.

3.2.5 Modifying the mixing coefficient for the MDL loss

Recall from Section 2.5.9 that X-UMX includes the multi-domain loss (MDL), with a mixing coefficient to sum the magnitude spectrogram loss and the time-domain SDR loss. The mixing coefficient in X-UMX is 10.0. In xumx-sliCQ, it is set to 0.1 to reflect the observed order of magnitude difference in the coefficients of the STFT and the sliCQT.

To implement the time-domain SDR loss, I use the auraloss Python package.[75] I used the SI-SDR (scale-invariant SDR) (Le Roux et al. 2018) variant of SDR as the loss function. SI-SDR is a simpler and more robust variant of SDR which improves on some failure cases

75. https://github.com/csteinmetz1/auraloss

of SDR (Le Roux et al. 2018).

3.3 Post-processing

In this section, I will discuss how the final post-processing step of X-UMX was modified for in xumx-sliCQ. In X-UMX, after the neural network outputs the estimates of the magnitude STFT for all four targets, the post-processing step of iterative Wiener expectation-maximization (EM) is applied to potentially improve the estimates (see Section 2.5.8).

Both the sliCQT and STFT output complex Fourier coefficients, which means that the Wiener-EM step can be run on both the sliCQT and the STFT representations of the estimated targets. Therefore, in xumx-sliCQ, I chose to implement two strategies of Wiener-EM: one that operates on the sliCQT outputs of xumx-sliCQ directly, and the other that swaps the sliCQT outputs into the STFT domain before applying Wiener-EM.

The music demixing quality and running time of both strategies will be shown in Section 4.4.3.

4 Experiment and discussion

This chapter contains the results of the experiments described in Chapter 3, and a discussion of the results. Refer to Appendix A for the hardware and software specifications of the computer which produced the results in this chapter. Appendix B contains links to all of the code for the experiments and results shown in this chapter.

The goal of this thesis is to use the sliced Constant-Q Transform (sliCQT) in CrossNet-Open-Unmix (X-UMX), a neural network for music demixing. X-UMX represents musical signals with the Short-Time Fourier Transform (STFT), which is limited by a fixed time-frequency resolution. In contrast, the sliCQT has a varying time-frequency resolution, which may be more suitable for musical applications such as music demixing.

To use the sliCQT in a neural network, I first ported the sliCQT library to use PyTorch, a Python framework for GPU-based deep neural networks. The porting effort was described in Section 3.1.5. In Section 4.1, I will show benchmark results of the port compared to the original library.

In Section 3.1.8, I described a random parameter search for the sliCQT that used the mix-phase inversion (MPI) oracle to pick sliCQT parameters. The MPI strategy, used by X-UMX and also by my proposed model xumx-sliCQ, combines the magnitude spectrogram of the estimate with the phase of the mix. Choosing sliCQT parameters that give the best MPI oracle score might result in a higher overall music demixing performance from the neural network, since the MPI oracle represents the upper limit of performance for any model that uses the MPI strategy. In Section 4.2, I will show the sliCQT parameters that were chosen by the parameter search script. In Section 3.1.9, I used the GPU-accelerated CuPy library in the BSS (Blind Source Separation) metrics library to speed up the parameter search, and I will show the benchmark results of the CuPy BSS metrics in Section 4.3.

After porting the sliCQT to PyTorch and choosing the sliCQT parameters, the next step is to train and evaluate the xumx-sliCQ neural network. First, Section 4.4.1 will list all of the training hyperparameters. Next, Section 4.4.2 will describe the training times and losses of the network architectures described in Section 3.2.2. Section 4.4.3 will show the music demixing results of the fully trained xumx-sliCQ model, using both post-processing configurations described in Section 3.3. Finally, Section 4.4.4 will describe the inference performance and model size on disk.

4.1 PyTorch port of the sliCQT

The first step is to port the sliCQT to PyTorch, a GPU-based neural network framework for Python, so that it could be used in a neural network. In Section 3.1.5, I described how I ported the original CPU-based NumPy implementation of the reference sliCQT Python library to PyTorch. In this section, I will show benchmark results of the PyTorch port.

The benchmark script for the PyTorch implementation of the sliCQT performs the forward and backward sliCQT using the Bark scale with 50 bins between 20–22,050 Hz. The sliCQT is taken on a single 3:54 minute song from the MUSDB18-HQ dataset, "Zeno - Signs." The sliCQT parameters were chosen so that the transform of the song occupied a maximum of 7.2 GB of memory and could fit on the device with the least memory, the NVIDIA 2070 Super GPU. The computation time was repeated for 100 iterations and averaged. The cost of the memory transfer of the song to the GPU is not included in the measurement. Table 8 shows the benchmark results.

Table 8: Execution times for the forward + backward ragged sliCQT.

Library	Device	Time (s)
Original	CPU	8.72
PyTorch	GPU (NVIDIA 2070 Super)	2.52
PyTorch	GPU (NVIDIA 3080 Ti)	2.38
PyTorch	CPU	1.85

These results show that the execution time of the transform improved with PyTorch in every case. The sliCQT computed with PyTorch was faster than the original library by ~4.7x, ~3.7x, and ~3.5x using the CPU, NVIDIA 2070 Super GPU, and NVIDIA 3080 Ti GPU, respectively.

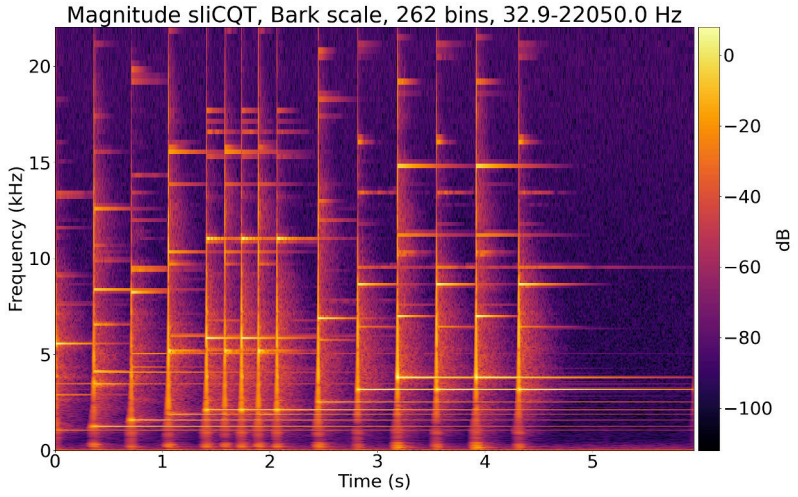
Even though the sliCQT computed with PyTorch on the GPU was not faster than the CPU for the tested parameters, having the transform on the GPU allows the use of the sliCQT inside any PyTorch-based neural network, which was the original goal of the port.

4.2 Best sliCQT parameters with MPI oracle

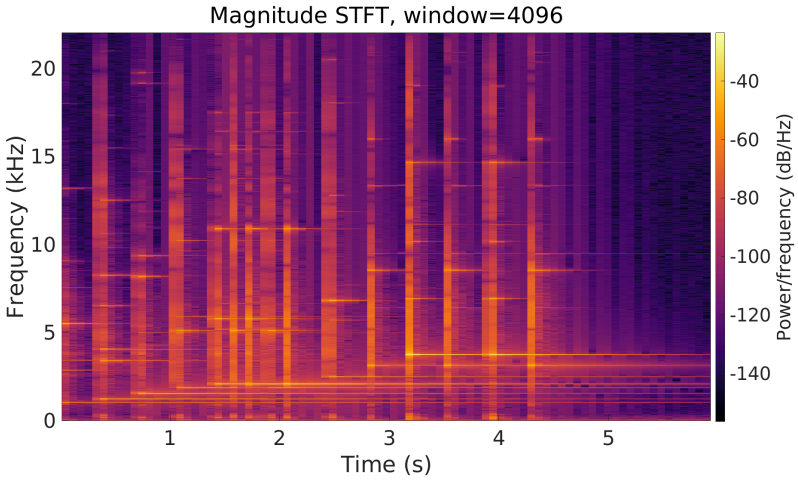
In Section 3.1.8, I described the random search to choose the parameters of the sliCQT to use in xumx-sliCQ. Since the goal of xumx-sliCQ is to try to surpass X-UMX, which uses the STFT, the first step is to try the sliCQT that has the most potential to surpass the STFT in music demixing. This was done by choosing the best Signal-to-Distortion (SDR) score on the mixed-phase inversion (MPI) oracle waveforms of validation tracks of MUSDB18-HQ. The

MPI strategy is when the estimate of the magnitude spectrogram of the target is combined with the phase of the original mixed audio, described in Section 2.5.6. The MPI strategy is used in both X-UMX and xumx-sliCQ.

The parameter search script ran for 60 iterations using random combinations of sliCQT parameters chosen from the ranges in Table 3 in Section 3.1.8. The best-performing sliCQT parameters chosen by the script used the following parameters: Bark scale, 262 bins, 32.9–22,050 Hz. These parameters were used in the final neural network, xumx-sliCQ. The slice length and transition length were chosen to be 18,060 and 4,514 samples respectively, using the automatic slice length picker described in Section 3.1.6. Figure 62 shows a visual comparison of the magnitude spectrograms of the best-performing sliCQT parameters alongside the STFT with the X-UMX default window size of 4,096 samples.



(a) Best sliCQT from MPI oracle parameter search.



(b) STFT, X-UMX default.

Figure 62: Magnitude spectrogram comparison; sliCQT vs. STFT.

The BSS metrics of the MPI oracle evaluation of these sliCQT parameters is shown in the boxplot in Figure 63, compared to the results from the STFT with different window sizes. The hypothesis of this thesis is that the varying time-frequency resolution of the sliCQT may provide an advantage in a musical application over the fixed time-frequency resolution of the STFT, and that the window size of the STFT must be changed to trade time and frequency resolution. In some music source separation papers, a short window STFT is used for a high time resolution, and a long window STFT is used for a high frequency resolution (Fitzgerald and Gainza 2010; Driedger, Müller, and Disch 2014). In particular, Driedger, Müller, and Disch (2014) use a window size of 256 for the short window STFT and 4,096 for the long window STFT, and Fitzgerald and Gainza (2010) use a window of 1,024 for the short window STFT and 16,384 for the long window STFT. Therefore, to demonstrate the effect of time-frequency resolution on music demixing, I compared the best sliCQT chosen from the parameter search with different STFT window sizes, selecting powers of two between 256 and 16,384 in the boxplot. Table 9 shows the evaluated transforms and their parameters. Note that the range of window sizes between 256–16,384 contain 4,096, which is the STFT window size of X-UMX.

Table 9: Time-frequency transforms compared in the MPI oracle boxplot.

Transform name	Transform	Parameters
slicqt-bark-262-32	sliCQT	Bark scale, 262 bins, 32.9–22,050 Hz
stft-256	STFT	256 sample window size
stft-512	STFT	512 sample window size
stft-1024	STFT	1,024 sample window size
stft-2048	STFT	2,048 sample window size
stft-4096	STFT	4,096 sample window size
stft-8192	STFT	8,192 sample window size
stft-16384	STFT	16,384 sample window size

The median SDR achieved by the chosen sliCQT parameters in the MPI oracle was 7.42 dB, surpassing the 6.23 dB achieved by the STFT with the X-UMX default window and overlap of 4,096 and 1,024 samples, respectively.

To discuss these results, the MPI oracle result indicates that a music demixing system that uses the sliCQT with 262 frequency bins between 32.9–22,050 Hz on the Bark scale has the potential to surpass a system that uses the STFT with any window size between 256–16,384 samples. To translate this potential to a real advantage in the final model, it is still a challenge to find a neural network architecture that can create a good estimate of the magnitude sliCQT spectrograms of the targets.

I stated as a hypothesis in this thesis that the additional musical information in the sliCQT might give the neural network an advantage when learning how to perform the task of music demixing, compared to the STFT. Note the sharper clarity of musical events in both time and frequency in the sliCQT spectrogram in Figure 62(a) compared to the STFT spectrogram in Figure 62(b).

4.3 CuPy acceleration of BSS metrics

In Section 3.1.9, the slowest functions of the BSS metrics library that used CPU-based NumPy and SciPy operations were swapped with their equivalent functions in CuPy, taking advantage of GPU acceleration. This was done to speed up the development cycle and execution time of the parameter search script. As a bonus side effect, running the full BSS evaluations in Section 4.4.3 were also sped up as a result of the CuPy-accelerated BSS metrics.

The benchmark script for the CuPy BSS metrics library creates a random estimated waveform for the four targets of the 14 validation tracks of MUSDB18-HQ, and computes the BSS metrics. The computation time was repeated for 10 iterations and averaged. Three copies of the benchmark script were run; one with the argument `--disable-cupy` to use the CPU code with SciPy and NumPy functions on the CPU, one with the argument `--cuda-device=0` to use the NVIDIA 3080 Ti GPU, and one with the argument `--cuda-device=1` to use the NVIDIA RTX 2070 Super GPU. The results are shown in Table 10.

Table 10: Execution times for the BSS metrics evaluation.

BSS library	Device	Time (s)
Original	CPU	1,485.77
CuPy	GPU (NVIDIA 2070 Super)	738.39
CuPy	GPU (NVIDIA 3080 Ti)	585.33

To discuss the results, there is a ~2-2.5x speedup when calculating BSS metrics with the GPU compared to the CPU, which represent a significant reduction in the time taken to evaluate music demixing systems. This speedup, aside from speeding up the parameter search and music demixing evaluation in this thesis, can be useful for larger-scale evaluation campaigns like SiSEC (Stöter, Liutkus, and Ito 2018) or MDX (Mitsufuji et al. 2021)

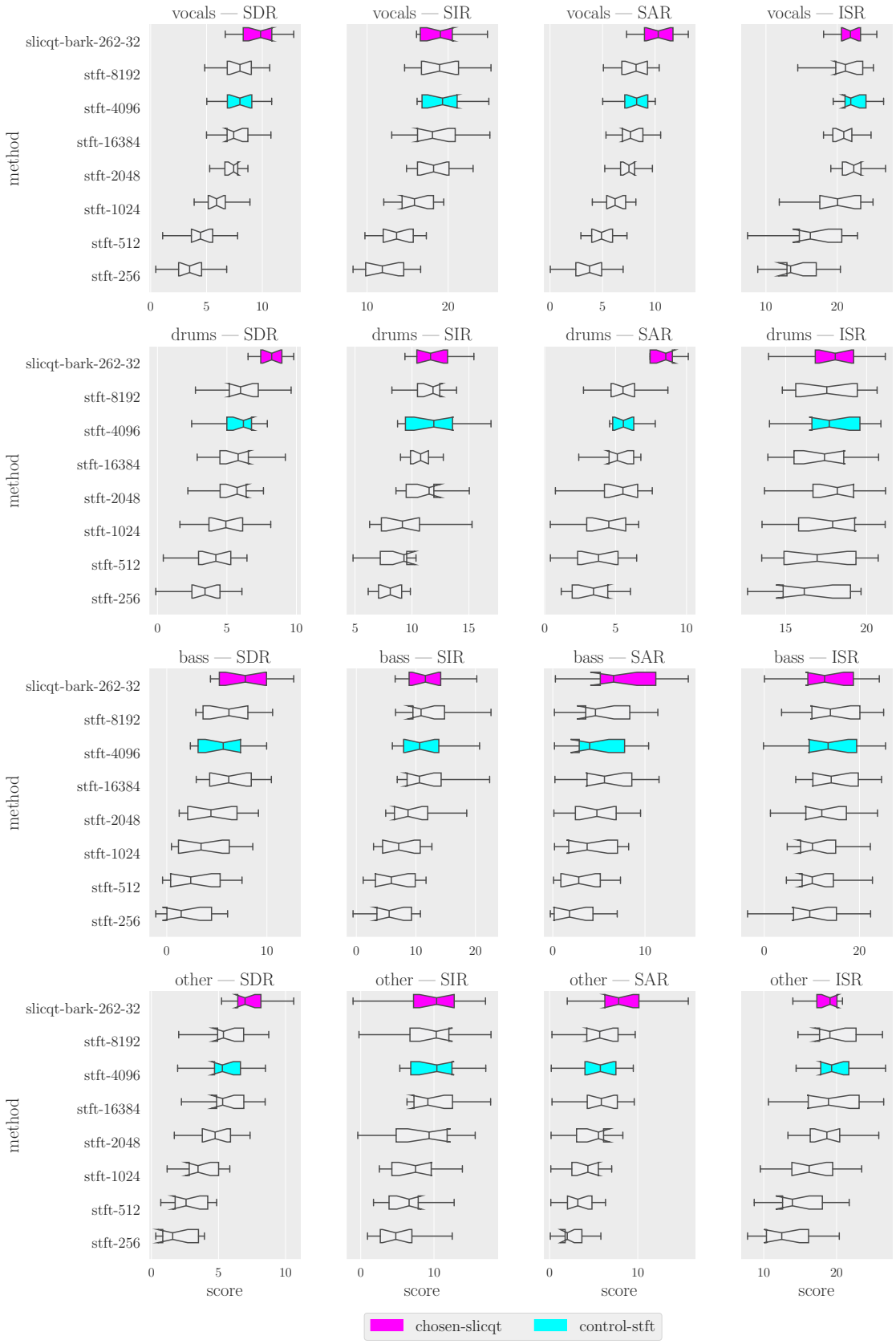


Figure 63: Boxplot for MPI oracle mask evaluations.

4.4 xumx-sliCQ neural network

In this section, I will show the hyperparameters, training procedure, and final trained performance of the xumx-sliCQ neural network, whose design was described in Section 3.2.

4.4.1 Hyperparameters in the training script

This section will cover the details of the hyperparameters of the xumx-sliCQ training script. Table 11 contains a full list of all the hyperparameters, their default values, and the justification for the choice of default value if it was not copied from X-UMX. Parameters that were introduced or modified as necessary to support the use of the sliCQT in xumx-sliCQ will point to the section of the thesis that explains the parameter and its chosen default value. Note that the choice of the Bidirectional Long Short-Term Memory (Bi-LSTM) and Convolutional Denoising Autoencoder (CDAE) network architectures of xumx-sliCQ, shown in Section 3.2, has been made a parameter in the training script.

Table 11: Hyperparameters in the xumx-sliCQ training script.

Parameter	Default	Origin
Dataset	MUSDB18-HQ	Copied from X-UMX
Learning rate	1e-3	Copied from X-UMX
Learning rate decay patience	80	Copied from X-UMX
Learning rate decay gamma	0.3	Copied from X-UMX
Weight decay	1e-5	Copied from X-UMX
Seed	42	Copied from X-UMX
Bandwidth	16,000 Hz	Copied from X-UMX
Number of channels	2	Copied from X-UMX
Number of workers for dataloader	4	Copied from X-UMX
Epochs	1,000	Copied from X-UMX
Patience	1,000	Copied from X-UMX
Frequency scale	Bark	Sections 3.1.8, 4.2
Frequency bins	262	Sections 3.1.8, 4.2
Minimum frequency	32.9 Hz	Sections 3.1.8, 4.2
Sequence duration	1 s	Section 3.2.1
Use Bi-LSTM instead of CDAE	False	Section 3.2.2
Mixing coefficient	0.1	Section 3.2.5

The optimizer used is the Adam optimizer with an adaptive learning rate scheduler, copied directly and unchanged from X-UMX.

4.4.2 Network architecture and training results

In this section, I will describe the neural network architectures and training results for both the Bi-LSTM and CDAE architectures of xumx-sliCQ. I will use the torchinfo[76] package to get a count of the total trainable parameters of the neural network, and show the loss curves of the training process with Tensorboard,[77] a neural network training visualization tool.

For the Bi-LSTM configuration of xumx-sliCQ, there are 1,889,512 trainable parameters in total. The training time per epoch within the first 10 epochs was 53 minutes, which would result in a hypothetical total of 36 days to train the full 1,000 epochs. I did not fully train this configuration, since a training time of 36 days is not feasible for this thesis.

For the CDAE configuration of xumx-sliCQ, there are 6,669,912 trainable parameters in total. The training time per epoch took 350 seconds, or almost 6 minutes, representing a total of 100 hours or 4 days of training. The total training curves for the full 1,000 epochs can be seen in Figure 64. The lowest validation loss achieved was -0.449 at epoch 583.

The music demixing performance of the trained model of xumx-sliCQ with the CDAE architecture will be shown in Section 4.4.3, and the execution time of inference and the size of the model on disk will be shown in Section 4.4.4.

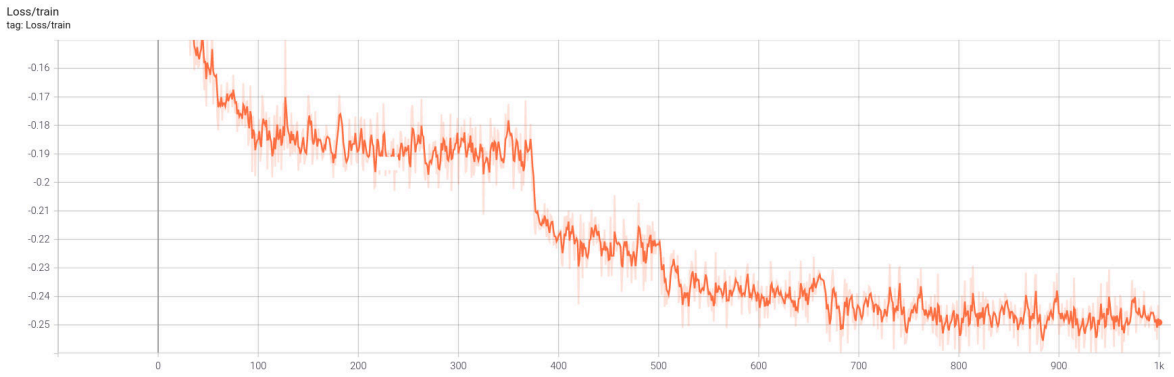
4.4.3 Music demixing results

In this section, I will compare the music demixing results of the trained models of UMX, X-UMX, and xumx-sliCQ.

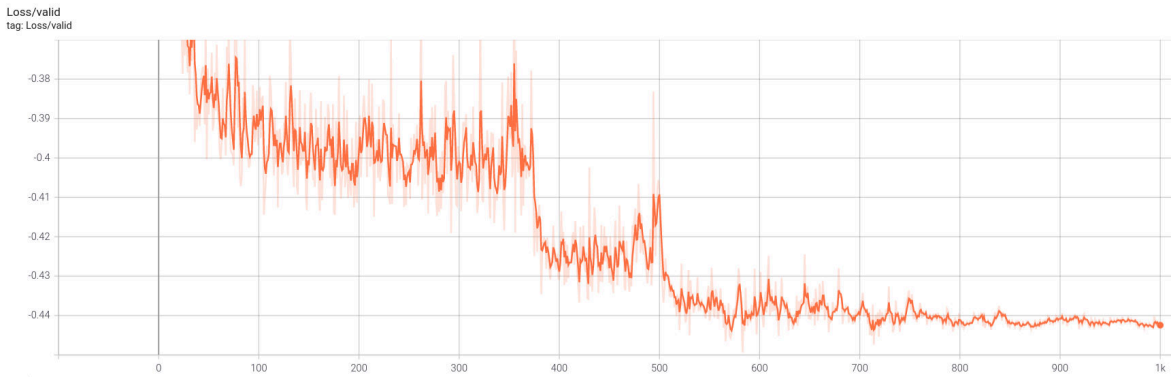
In Section 2.5.9, I described how X-UMX (CrossNet-Open-Unmix) is an evolution of UMX (Open-Unmix), and that for convenience throughout Chapter 3 and Chapter 4, I will use X-UMX to refer to the concepts of both UMX and X-UMX. In the results shown in this and the next section, I will treat UMX and X-UMX separately. They have a different music demixing performance and inference performance, due to the differences in their network architectures, shown in Figure 47, and the different deep learning frameworks used in their reference implementations.

The evaluated trained model of xumx-sliCQ uses the CDAE architecture. The Bi-LSTM architecture of xumx-sliCQ was not evaluated, due to requiring 36 days to train the full 1,000 epochs. The CDAE architecture was trained fully in four days, which is an acceptable

76. https://github.com/TylerYep/torchinfo
77. https://www.tensorflow.org/tensorboard/



(a) Train loss.



(b) Validation loss.

Figure 64: Tensorboard loss curves for xumx-sliCQ using the CDAE architecture.

training time for this thesis.

The BSS metrics for the demixing results of UMX, X-UMX, and xumx-sliCQ were computed on the 50-song test set of the MUSDB18-HQ dataset, and the boxplot is shown in Figure 65. Table 12 contains the details of the evaluated models, including the median SDR across all 50 tracks of the MUSDB18-HQ test set and all four targets. The results of xumx-sliCQ are computed for both of the Wiener Expectation Maximization (Wiener-EM) post-processing strategies, using the sliCQT or STFT. Wiener-EM will be abbreviated to WEM in the tables that follow. To maximize result reproducibility, all pretrained models and code were downloaded from their public hosted locations shown in Table 12 and stored in a separate repository to generate the results.[78]

To summarize and discuss these results, I will first note that in my evaluation, UMX and X-UMX scored a median SDR of 5.41 dB and 5.91 dB respectively. These results are in line with the published SDR scores for UMX and X-UMX, which are given as 5.41 dB and 5.79 dB respectively (Sawata et al. 2021, 4).

78. https://github.com/sevagh/xumx_slicq_extra/tree/main/mss_evaluation

Table 12: Evaluated pretrained models in the BSS boxplot.

Project	Legend	SDR	Code repository	Pretrained model
CrossNet-Open-Unmix	xumx	5.91	https://github.com/sony/ai-research-code/tree/master/x-umx	https://nnabla.org/pretrained-models/ai-research-code/x-umx/x-umx.h5
Open-Unmix	umx	5.41	https://github.com/sigsep/open-unmix-pytorch/tree/v1.0.0	https://zenodo.org/record/3370489
xumx-sliCQ sliCQT-WEM	slicq-wslicq	3.70	https://github.com/sevagh/xumx-sliCQ	https://github.com/sevagh/xumx-sliCQ/tree/main/pretrained-model
xumx-sliCQ STFT-WEM	slicq-wstft	3.67	https://github.com/sevagh/xumx-sliCQ	https://github.com/sevagh/xumx-sliCQ/tree/main/pretrained-model

xumx-sliCQ scored a median SDR of 3.67 dB with the STFT Wiener-EM step, and 3.70 dB with the sliCQT Wiener-EM step. xumx-sliCQ failed in its objective to improve the music demixing performance of UMX and X-UMX with both of its post-processing strategies.

One potential source of poor performance in xumx-sliCQ is the design choice of using an independent neural network for each of the sub-matrices of the ragged sliCQT, proposed in Section 3.2.2. Each sub-matrix of the ragged sliCQT contains frequency bins that were analyzed with the same temporal frame rate (see Section 2.2.6). A target (or source) may contain distinct frequencies that are not necessarily in the same sub-matrix of the sliCQT. Consequently, the independent neural networks might only have access to a limited subset of the time-frequency information of a target. By contrast, UMX and X-UMX use a single neural network that learns from all of the frequencies of all of the targets simultaneously in a single STFT.

Another potential source of errors in xumx-sliCQ is the de-overlap layer, shown in Figure 59 and described in Section 3.2.4. In Section 2.2.5, I showed that the sliCQT has a symmetric zero-padding step to reduce time-domain aliasing. The sliCQT must be overlap-added to create a spectrogram, a procedure shown in Section 3.1.2 that has no inverse. In xumx-sliCQ,

I implemented a learned inverse overlap by adding a transpose convolution layer added after the music demixing layers. By contrast, UMX and X-UMX do not need any extra layers, since the STFT can be used as a spectrogram directly.

Finally, the CDAE hyperparameters in xumx-sliCQ, shown in Tables 4–7 in Section 3.2.2, were perhaps kept too similar to the original STFT-based values (Grais and Plumbley 2017, 3) (Grais, Zhao, and Plumbley 2021, 3). The STFT and sliCQT, as has been shown throughout this thesis, are significantly different time-frequency transforms, and neural network architectures or hyperparameters for the STFT and sliCQT are not necessarily interchangeable. A more sophisticated approach should take into account the frequency bins of the nonlinear frequency scale and nonuniform time-frequency resolution of the sliCQT to optimize each CDAE independently.

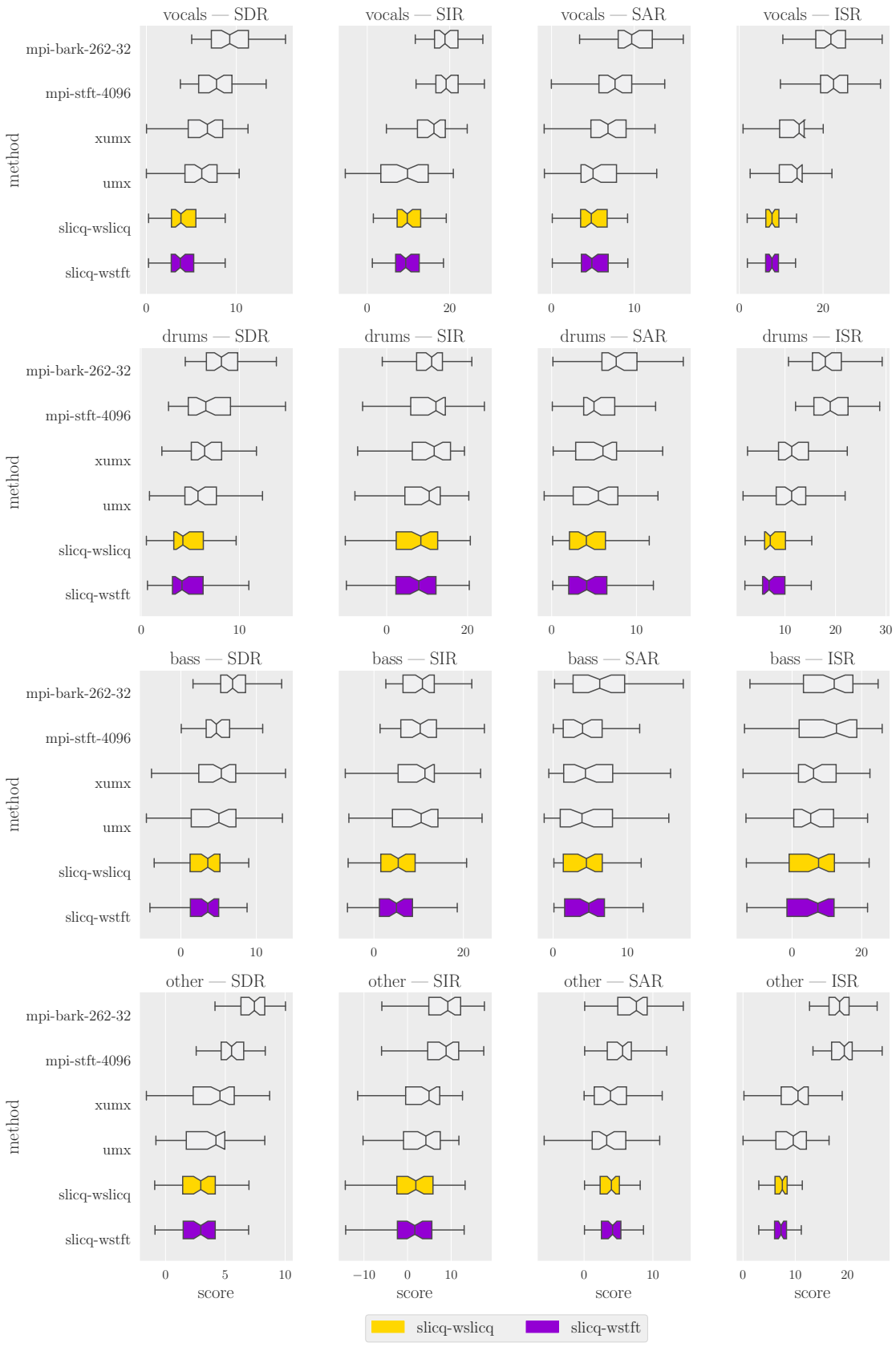


Figure 65: Boxplot of UMX, X-UMX, and xumx-sliCQ alongside the oracles.

4.4.4 Model size and inference performance comparison

In this section, I will show measurements of inference time and the size on disk of the compared models.

The pretrained UMX-HQ model[79] for the PyTorch deep learning framework has a size on disk of 137 MB for all four targets combined. The pretrained X-UMX model[80] for the NNabla deep learning framework has a size on disk of 136 MB for the four combined targets, which is almost the same size as the UMX PyTorch weights. The pretrained xumx-sliCQ model[81] for PyTorch has a size on disk of 28 MB, making it the smallest network in the comparison.

To report the inference performance, only the CPU was used. The CPU is a more universal device for performing inference, as almost every computing device (desktop, laptop, server, smartphone, etc.) has a CPU while not all have deep-learning capable GPUs (Mittal, Rajput, and Subramoney 2021; Wu et al. 2019). The time to perform the separation of a mixed song into four stems (including the Wiener-EM step) was averaged across the first 10 songs of the 50-song test set of MUSDB18-HQ. Table 13 shows the measured times. For xumx-sliCQ, both the STFT and sliCQT configurations of the post-processing Wiener-EM step were measured, denoted as STFT-WEM and sliCQT-WEM respectively.

Table 13: Execution times of CPU inference and model sizes.

Model	Size on disk	Time (s)
UMX	137 MB	27.26
xumx-sliCQ, STFT-WEM	28 MB	47.32
xumx-sliCQ, sliCQT-WEM	28 MB	91.14
X-UMX	136 MB	528.19

To discuss these results, I will first note that X-UMX shows an anomalously high execution time which is ~20x slower than UMX. This is because it uses a different deep learning framework, NNabla, compared to the PyTorch-based UMX and xumx-sliCQ. The STFT Wiener-EM configuration of xumx-sliCQ is ~1.7x slower than UMX, and the sliCQT Wiener-EM configuration is ~3.4x slower than UMX. Wiener-EM with the sliCQT only adds ~0.03 dB to the median SDR score compared to the Wiener-EM with the STFT, while doubling the running time. The pretrained model of xumx-sliCQ is ~4.9x smaller on disk than both of the original STFT-based models, using 28 MB instead of the 137 and 136 MB of UMX and X-UMX, respectively.

79. https://zenodo.org/record/3370489
80. https://nnabla.org/pretrained-models/ai-research-code/x-umx/x-umx.h5
81. https://github.com/sevagh/xumx-sliCQ/tree/main/pretrained-model

5 Conclusion

In the task of music demixing, a mixed song is separated into estimates of its isolated constituent sources or instruments that can be summed back to the original mixture. A common approach to music demixing is to apply a mask to the spectrogram of the mix to extract the desired source. Two models for music demixing are Open-Unmix (UMX) and its closely-related variant CrossNet-Open-Unmix (X-UMX), both of which use spectrogram masking with the Short-Time Fourier Transform (STFT). The STFT has known limitations that arise from its fixed time-frequency resolution, and different sources or instruments may have conflicting needs for the time-frequency resolution of their STFT for better separation results.

Methods of music demixing based on the STFT have addressed the time-frequency limitation by using multiple STFTs, or by using different time-frequency transforms like the Constant-Q Transform (CQT). The CQT has a varying time-frequency resolution, where low-frequency regions are analyzed with a high frequency resolution, and high-frequency regions are analyzed with a high time resolution, which is appropriate for both musical and auditory analysis.

The sliced Constant-Q Transform (sliCQT) is a realtime implementation of the Nonstationary Gabor Transform (NSGT). The NSGT and sliCQT provide a method for computing time-frequency transforms with complex-valued Fourier coefficients and a perfect inverse. They were originally designed to implement an invertible CQT, but they support arbitrary nonuniform frequency scales (e.g., psychoacoustic scales like the mel and Bark scales).

In this thesis, the sliCQT was explored as a replacement of the STFT in UMX and X-UMX, in a newly proposed model called xumx-sliCQ. The goal was to use the sliCQT with a custom frequency scale to see if it could improve on the STFT-based music demixing performance of UMX and X-UMX. Unfortunately, the proposed model, xumx-sliCQ, performed worse than UMX and X-UMX. However, the creation of xumx-sliCQ in this thesis led to several contributions to the current Python ecosystem of music demixing.

First, the PyTorch implementation of the NSGT/sliCQT allows these transforms to be used in future GPU neural networks, and it sped up the computation of the transforms compared to the original library by ~4x. Second, the GPU acceleration of the BSS (Blind Source Separation) metrics library can speed up evaluations of music demixing and source separation by ~2x. Finally, xumx-sliCQ demonstrated the first working prototype of a sliCQT-based deep neural network for music demixing.

5.1 Future work

To conclude my thesis, I will discuss ideas for xumx-sliCQ that I think are worth exploring, and that may lead to improved variants.

The NSGT and sliCQT were shown to support arbitrary nonuniform frequency scales in Section 2.2.3. In Section 3.1.8, I described how I chose a frequency scale for the sliCQT in the task of music demixing by searching for the highest SDR score of the mix-phase oracle, initially introduced in Section 2.5.6. There can be any number of approaches to choosing the frequency scale that might improve the results of xumx-sliCQ. For example, the frequency scale can be tailored to the frequency range of a specific musical instrument.

In Section 4.4.3, I provided some of my guesses as to why xumx-sliCQ failed to achieve its objective of beating UMX and X-UMX.

My first guess was that targets may have their frequencies distributed into independent, unconnected neural networks in xumx-sliCQ, due to how a different neural network is applied to each sub-matrix in the ragged sliCQT, shown in Section 3.2.2. In UMX and X-UMX, the neural network operates on all of the frequency bins at once on the single matrix of the STFT. Schörkhuber et al. (2014) proposed a single matrix form for the sliCQT, which may be of interest in a future version of xumx-sliCQ.

My next guess was that there could be an additional source of errors in xumx-sliCQ because of the necessary de-overlap layer, shown in Section 3.2.4. In Section 2.2.5, I described that the need to overlap-add the sliCQT comes from the symmetric zero-padding applied to each slice, to reduce time-domain aliasing (Holighaus et al. 2013). One idea is to remove the symmetric zero-padding from the sliCQT; this will eliminate the need for a de-overlap layer, but it will also reintroduce time-domain aliasing, which may have a detrimental effect on the music demixing quality.

My last guess was that neural network architectures and hyperparameters used in xumx-sliCQ were originally intended for the STFT, leading to subpar performance when copying them for the sliCQT in xumx-sliCQ. Better results might be obtained by performing a more in-depth search for optimal neural network architectures and hyperparameters tuned to the time-frequency characteristics of the sliCQT.

6 References

Abdoli, Sajjad, Patrick Cardinal, and Alessandro Lameiras Koerich. 2019. "End-to-end environmental sound classification using a 1D convolutional neural network." *Expert Systems with Applications* 136: 252–263. doi:10.1016/j.eswa.2019.06.040.

Allen, Jont B., and Lawrence R. Rabiner. 1977. "A unified approach to short-time Fourier analysis and synthesis." *Proceedings of the IEEE* 65 (11): 1558–1564. doi:10.1109/PROC.1977.10770.

Aslam, Muhammad, Jae-Myeong Lee, Hyung-Seung Kim, Seung-Jae Lee, and Sugwon Hong. 2020. "Deep learning models for long-term solar radiation forecasting considering microgrid installation: A comparative study." *Energies* 13 (1): 1–15. doi:10.3390/en13010147.

Balazs, Peter, Monika Doerfler, Florent Jaillet, Nicki Holighaus, and Gino Angelo Velasco. 2011. "Theory, implementation and applications of nonstationary Gabor frames." *Journal of Computational and Applied Mathematics* 236 (6): 1481–1496. doi:10.1016/j.cam.2011.09.011.

Balazs, Peter, Nicki Holighaus, Thibaud Necciari, and Diana Stoeva. 2017. "Frame theory for signal processing in psychoacoustics." In *Excursions in Harmonic Analysis, Volume 5,* edited by Radu Balan, John J. Benedetto, Wojciech Czaja, Matthew Dellatorre, and Kasso A. Okoudjou, 225–268. Springer. doi:10.1007/978-3-319-54711-4_10.

Bastanlar, Yalin, and Mustafa Ozuysal. 2014. "Introduction to machine learning." *Methods in Molecular Biology* 1107: 105–128. doi:10.1007/978-1-62703-748-8_7.

Bayen, Alexandre, Qingkai Kong, and Timmy Siauw. 2020. *Python programming and numerical methods: A guide for engineers and scientists.* 1st ed. Elsevier.

Bergstra, James, and Yoshua Bengio. 2012. "Random search for hyper-parameter optimization." *Journal of Machine Learning Research* 13 (10): 281–305. http://jmlr.org/papers/v13/bergstra12a.html.

Beysolow II, Taweh. 2017. *Introduction to deep learning in R.* Apress. doi:10.1007/978-1-4842-2734-3_1.

Bianco, Michael J., Peter Gerstoft, James Traer, Emma Ozanich, Marie A. Roch, Sharon Gannot, and Charles-Alban Deledalle. 2019. "Machine learning in acoustics: Theory and applications." *The Journal of the Acoustical Society of America* 146 (5): 3590–3628. doi:10.1121/1.5133944.

Bittner, Rachel, Justin Salamon, Mike Tierney, Matthias Mauch, Chris Cannam, and Juan Pablo Bello. 2014. "MedleyDB audio: A dataset of multitrack audio for music research." doi:10.5281/zenodo.1649325.

Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex optimization.* Cambridge University Press.

Bregman, Albert S. 1994. *Auditory scene analysis: The perceptual organization of sound.* MIT Press.

Brittain, James E. 1996. "John R. Carson and the conversation of radio spectrum [Scanning the Past]." *Proceedings of the IEEE* 84 (6): 909–910. doi:10.1109/JPROC.1996.503148.

Brown, Judith. 1991. "Calculation of a constant Q spectral transform." *Journal of the Acoustical Society of America* 89 (1): 425–434. https://www.ee.columbia.edu/~dpwe/papers/Brown91-cqt.pdf.

Brown, Judith, and Miller Puckette. 1992. "An efficient algorithm for the calculation of a constant Q transform." *Journal of the Acoustical Society of America* 92 (5): 2698. doi:10.1121/1.404385.

Burred, Juan José, and T. Sikora. 2006. "Comparison of frequency-warped representations for source separation of stereo mixtures." In *Audio Engineering Society Convention 121,* 1–8. https://www.aes.org/e-lib/browse.cfm?elib=13758.

Cano, Estefanía, Derry Fitzgerald, Antoine Liutkus, Mark Plumbley, and Fabian-Robert Stöter. 2018. "Musical source separation: An introduction." *IEEE Signal Processing Magazine* 36 (1): 31–40. doi:10.1109/MSP.2018.2874719.

Chacon, Scott, and Ben Straub. 2014. *Pro Git.* 2nd ed. Apress. https://git-scm.com/book/en/v2.

Chen, Scott Shaobing, David L. Donoho, and Michael A. Saunders. 2001. "Atomic decomposition by basis pursuit." *SIAM Journal on Scientific Computing* 20 (1): 33–61.

Christopoulos, Thomas, Odysseas Tsilipakos, Georgios Sinatkas, and Emmanouil E. Kriezis. 2019. "On the calculation of the quality factor in contemporary photonic resonant structures." *arXiv preprint arXiv:1902.09415.* https://arxiv.org/abs/1902.09415.

Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. "Empirical evaluation of gated recurrent neural networks on sequence modeling." *arXiv preprint arXiv:1412.3555.* https://arxiv.org/abs/1412.3555.

Comon, Pierre, and Christian Jutten. 2010. *Handbook of blind source separation, independent component analysis and applications.* 1st ed. Academic Press. doi:10.1016/C2009-0-19334-0.

Cooley, James W., and John W. Tukey. 1965. "An algorithm for the machine calculation of complex Fourier series." *Mathematics of Computation* 19 (90): 297–301. doi:10.1090/S0025-5718-1965-0178586-1.

Cui, Zhiyong, Ruimin Ke, Ziyuan Pu, and Yinhai Wang. 2019. "Deep bidirectional and unidirectional LSTM recurrent neural network for network-wide traffic speed prediction." *arXiv preprint arXiv:1801.02143.* https://arxiv.org/abs/1801.02143.

Défossez, Alexandre, Nicolas Usunier, Léon Bottou, and Francis Bach. 2019. "Music source separation in the waveform domain." *arXiv preprint arXiv:1911.13254.* https://arxiv.org/abs/1911.13254.

Diniz, Filipe C. C. B., Iuri Kothe, Luiz W. P. Biscainho, and Sergio L. Netto. 2006. "A bounded-Q fast filter bank for audio signal analysis." In *International Telecommunications Symposium,* 1015–1019. doi:10.1109/ITS.2006.4433420.

Domínguez, Alejandro. 2016. "Highlights in the History of the Fourier Transform [Retrospectroscope]." *IEEE Pulse* 7 (1): 53–61. doi:10.1109/MPUL.2015.2498500.

Dongarra, Jack, and Francis Sullivan. 2000. "Top ten algorithms of the century." *Computing in Science and Engineering* 2 (1): 22–23.

Dörfler, Monika. 2002. "Gabor analysis for a class of signals called music." PhD diss., Numerical Harmonic Analysis Group, University of Vienna.

Driedger, Jonathan, Meinard Müller, and Sascha Disch. 2014. "Extending harmonic-percussive separation of audio signals." In *Proceedings of the 15th International Conference on Music Information Retrieval (ISMIR),* 611–616.

Dumoulin, Vincent, and Francesco Visin. 2018. "A guide to convolution arithmetic for deep learning." *arXiv preprint arXiv:1603.07285.* https://arxiv.org/abs/1603.07285.

Duong, Ngoc Q. K., Emmanuel Vincent, and Rémi Gribonval. 2010. "Under-determined reverberant audio source separation using a full-rank spatial covariance model." *IEEE Transactions on Audio, Speech, and Language Processing* 18 (7): 1830–1840. doi:10.1109/TASL.2010.2050716. https://hal.inria.fr/inria-00435807v2/document.

Esposito, Floriana, and Donato Malerba. 2010. "Editorial: Machine learning in computer vision." *Applied Artificial Intelligence* 15 (8): 693–705. doi:10.1080/088395101317018546.

Fedorishin, Dennis, Nishant Sankaran, Deen Mohan, Justas Birgiolas, Philip Schneider, Srirangaraj Setlur, and Venu Govindaraju. 2021. *Investigating waveform and spectrogram feature fusion for acoustic scene classification.* Technical report. Detection, Classification of Acoustic Scenes, and Events 2021. http://dcase.community/documents/challenge2021/technical_reports/DCASE2021_Fedorishin_97_t1.pdf.

Fitzgerald, Derry. 2010. "Harmonic/percussive separation using median filtering." In *Proceedings of the 13th International Conference on Digital Audio Effects (DAFx-10).* http://dafx10.iem.at/papers/DerryFitzGerald_DAFx10_P15.pdf.

Fitzgerald, Derry, Matt Cranitch, and Marcin Cychowski. 2006. "Towards an inverse constant Q transform." In *Audio Engineering Society Convention 120.* http://www.aes.org/e-lib/browse.cfm?elib=13475.

Fitzgerald, Derry, and Mikel Gainza. 2010. "Single channel vocal separation using median filtering and factorisation techniques." *ISAST Transactions on Electronic and Signal Processing* 4 (1): 62–73.

Fortunato, Laura, and Mark Galassi. 2021. "The case for free and open source software in research and scholarship." *Philosophical Transactions of the Royal Society A* 379 (2197): 20200079. doi:10.1098/rsta.2020.0079.

Fourier, Jean-Baptise Joseph. 1807. "Théorie de la propagation de la chaleur dans les solides." *Institute of France.*

———. 1811. "Théorie du mouvement de la chaleur dans les corps solides." *Mémoires de l'Académie royale des sciences de l'Institute de France* 4.

———. 1822. *Théorie analytique de la chaleur.* Chez Firmin Didot, Père et Fils.

Gabor, Dennis. 1946. "Theory of communication." *Journal of Institution of Electrical Engineers* 93 (3): 429–457. http://www.granularsynthesis.com/pdf/gabor.pdf.

Gambella, Claudio, Bissan Ghaddar, and Joe Naoum-Sawaya. 2020. "Optimization problems for machine learning: A survey." *European Journal of Operational Research* 290 (3): 807–828. doi:10.1016/j.ejor.2020.08.045.

Gerkmann, Timo, and Emmanuel Vincent. 2018. "Spectral masking and filtering." Chap. 5 In *Audio Source Separation and Speech Enhancement,* 65–85. Wiley. doi:10.1002/9781119279860.ch5.

Grais, Emad M., and Mark D. Plumbley. 2017. "Single channel audio source separation using convolutional denoising autoencoders." In *IEEE Global Conference on Signal and Information Processing,* 1265–1269. doi:10.1109/GlobalSIP.2017.8309164.

Grais, Emad M., Fei Zhao, and Mark D. Plumbley. 2021. "Multi-band multi-resolution fully convolutional neural networks for singing voice separation." In *28th European Signal Processing Conference,* 261–265. doi:10.23919/Eusipco47968.2020.9287367.

Graves, Alex, and Jürgen Schmidhuber. 2005. "Framewise phoneme classification with bidirectional LSTM and other neural network architectures." *Neural Networks* 18 (5-6): 602–610.

Gray, Robert M., and Lee D. Davisson. 2004. "An introduction to statistical signal processing." Cambridge University Press.

Griffin, Daniel W., and Jae S. Lim. 1984. "Signal estimation from modified short-time Fourier transform." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32 (2): 236–243. doi:10.1109/TASSP.1984.1164317.

Gsaxner, Christina, Peter M. Roth, Jürgen Wallner, and Jan Egger. 2019. "Exploit fully automatic low-level segmented PET data for training high-level deep learning algorithms

for the corresponding CT data." *PLOS ONE* 14 (3): 1–20. doi:10.1371/journal.pone.0212550.

Hall, M. 2006. "Resolution and uncertainty in spectral decomposition." *First Break* 24 (12). doi:10.3997/1365-2397.2006027.

Heideman, Michael, Don Johnson, and Charles Burrus. 1985. "Gauss and the history of the fast Fourier transform." *Archive for History of Exact Sciences* 34 (3): 265–277. doi:10.1007/BF00348431.

Heinzel, Gerhard, Albrecht Rüdiger, and Roland Schilling. 2002. "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows." *Max-Planck-Institut für Gravitationsphysik.*

Heisenberg, Werner. 1927. "Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik." *Zeitschrift für Physik:* 879–893.

Herrera, Perfecto, Amaury Dehamel, and Fabien Gouyon. 2003. "Automatic labeling of unpitched percussion sounds." In *Proceedings of the 114th Convention of the Audio Engineering Society.* https://paginas.fe.up.pt/~ee97138/AES114-Herrera2003.pdf.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long short-term memory." *Neural Computation* 9 (8): 1735–1780. doi:10.1162/neco.1997.9.8.1735.

Holighaus, Nicki, Monika Dörfler, Gino Angelo Velasco, and Thomas Grill. 2013. "A framework for invertible, real-time constant-Q transforms." *IEEE Transactions on Audio, Speech, and Language Processing* 21 (4): 775–785. doi:10.1109/TASL.2012.2234114.

Hu, Peng, Qin Yan, Luan Dong, and Meng Liu. 2014. "An improved patchwork-based digital audio watermarking in CQT domain." In *22nd European Signal Processing Conference,* 920–923.

Huang, Dong-Yan, Minghui Dong, and Haizhou Li. 2015. "A real-time variable-Q non-stationary Gabor transform for pitch shifting." In *Proceedings of Interspeech 2015,* 2744–2748. doi:10.21437/Interspeech.2015-578.

Hyvärinen, Aapo. 1999. "Fast and robust fixed-point algorithms for independent component analysis." *IEEE Transactions on Neural Networks* 10 (3): 626–634. doi:10.1109/72.761722.

Hyvärinen, Aapo, and Erkki Oja. 2000. "Independent component analysis: Algorithms and applications." *Neural networks: the official journal of the International Neural Network Society* 13 (4-5): 411–430. doi:10.1016/S0893-6080(00)00026-5.

Iman, Mohammadreza, Hamid Arabnia, and Robert Branchinst. 2020. "Pathways to artificial general intelligence: A brief overview of developments and ethical issues via artificial

intelligence, machine learning, deep learning, and data science." In *Proceedings of the 22nd International Conference on Artificial Intelligence.*

Jacoby, Nori, Eduardo Undurraga, Malinda Mcpherson, Joaquín Valdés, Tomas Ossandon, and Josh McDermott. 2019. "Universal and non-universal features of musical pitch perception revealed by singing." *Current Biology* 29 (19): 3229–3243. doi:10.1016/j.cub.2019.08.020.

Jaillet, Florent, P. Balazs, and M. Dörfler. 2009. "Nonstationary Gabor frames." In *International Conference on SAMPling Theory and Applications.* https://github.com/ltfat/ltfat.github.io/blob/master/notes/ltfatnote010.pdf.

Jaillet, Florent, and Bruno Torrésani. 2007. "Time-Frequency jigsaw puzzle: Adaptive multiwindow and multilayered Gabor expansions." *International Journal of Wavelets, Multiresolution and Information Processing* 05 (02): 293–315. doi:10.1142/S0219691307001768.

Johri, Prashant, Sunil Kumar Khatri, Ahmad Al-Taani, Munish Sabharwal, Shakhzod Suvanov, and Avneesh Chauhan. 2021. "Natural language processing: History, evolution, application, and future work." In *Lecture Notes in Networks and Systems,* 167: 365–375. Springer. doi:10.1007/978-981-15-9712-1_31.

Jutten, Christian, and Jeanny Hérault. 1991. "Blind separation of sources, part I: An adaptive algorithm based on neuromimetic architecture." *Signal Processing* 24 (1): 1–10.

Kavalerov, Ilya, Scott Wisdom, Hakan Erdogan, Brian Patton, Kevin W. Wilson, Jonathan Le Roux, and John R. Hershey. 2019. "Universal sound separation." In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics,* 175–179. IEEE. doi:10.1109/WASPAA.2019.8937253.

Kessy, Agnan, Alex Lewin, and Korbinian Strimmer. 2018. "Optimal Whitening and Decorrelation." *The American Statistician* 72 (4): 309–314. doi:10.1080/00031305.2016.1277159.

Ketkar, Nikhil. 2017. "Stochastic gradient descent." In *Deep learning with Python: A hands-on introduction,* 111–130. Apress. doi:10.1007/978-1-4842-2766-4_8.

Khan, Asifullah, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. 2020. "A survey of the recent architectures of deep convolutional neural networks." *Artificial Intelligence Review* 53 (8): 5455–5516. doi:10.1007/s10462-020-09825-6.

Kim, Minseok, Woosung Choi, Jaehwa Chung, Daewon Lee, and Soonyoung Jung. 2021. "KUIELab-MDX-Net: A Two-Stream Neural Network for Music Demixing." *arXiv preprint arXiv:2111.12203.* https://arxiv.org/abs/2111.12203.

Kolecki, Joseph. 2002. *An introduction to tensors for students of physics and engineering.* Technical Memorandum 2002-211716. NASA. https://ntrs.nasa.gov/citations/20020083040.

Kong, Qiuqiang, Yin Cao, Haohe Liu, Keunwoo Choi, and Yuxuan Wang. 2021. "Decoupling magnitude and phase estimation with deep ResUNet for music source paration." In *Proceedings of the 22nd International Society for Music Information Retrieval Conference.*

Korpel, Adrian. 1982. "Gabor: Frequency, time, and memory." *Applied Optics* 21 (20): 3624–3632. doi:10.1364/AO.21.003624.

Kovačević, Jelena, and Amina Chebira. 2008. "An Introduction to Frames." *Foundations and Trends in Signal Processing* 2 (1): 1–94. doi:10.1561/2000000006.

Kutz, J. Nathan. 2013. *Data-driven modeling & scientific computation: Methods for complex systems.* 1st ed. USA: Oxford University Press.

L'Ecuyer, Pierre. 2010. "Pseudorandom number generators." In *Encyclopedia of Quantitative Finance,* edited by Rama Cont, 1431–1437. doi:10.1002/9780470061602.eqf13003.

Le, Quoc, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Ng. 2011. "On optimization methods for deep learning," 2011: 265–272.

Le Roux, Jonathan, Scott Wisdom, Hakan Erdogan, and John R. Hershey. 2018. "SDR: Half-baked or well done?" *arXiv preprint arXiv:1811.02508.* https://arxiv.org/abs/1811.02508.

Lee, Daniel, and H. Sebastian Seung. 2001. "Algorithms for non-negative matrix factorization." In *Advances in Neural Information Processing Systems 13 (NIPS 2000),* edited by T. Leen, T. Dietterich, and V. Tresp, vol. 13. MIT Press. https://proceedings.neurips.cc/paper/2000/file/f9d1152547c0bde01830b7e8bd60024c-Paper.pdf.

Lee, Jongpil, Taejun Kim, Jiyoung Park, and Juhan Nam. 2017. "Raw waveform-based audio classification using sample-level CNN architectures." *arXiv preprint arXiv:1712.00866.* https://arxiv.org/abs/1712.00866.

Litvin, Yevgeni, and Israel Cohen. 2011. "Single-channel source separation of audio signals using Bark scale wavelet packet decomposition." *Journal of Signal Processing Systems* 65: 339–350. doi:10.1007/s11265-010-0510-9.

Liu, Ruolun, and Suping Li. 2009. "A review on music source separation." In *IEEE Youth Conference on Information, Computing and Telecommunication,* 343–346. doi:10.1109/YCICT.2009.5382353.

Liuni, Marco, Axel Roebel, Ewa Matusiak, Marco Romito, and Xavier Rodet. 2013. "Automatic adaptation of the time-frequency resolution for sound analysis and re-synthesis." *IEEE Transactions on Audio Speech and Language Processing* 21 (5): 959–970. doi:10.1109/TASL.2013.2239989.

Liutkus, Antoine, Jean-Louis Durrieu, Laurent Daudet, and Gaël Richard. 2013. "An overview of informed audio source separation." In *Proceedings of the 14th International Workshop on Image Analysis for Multimedia Interactive Services,* 1–4. doi:10.1109/WIAMIS.2013.6616139.

Liutkus, Antoine, Fabian-Robert Stöter, Zafar Rafii, Daichi Kitamura, Bertrand Rivet, Nobutaka Ito, Nobutaka Ono, and Julie Fontecave. 2017. "The 2016 signal separation evaluation campaign." In *Latent Variable Analysis and Signal Separation. LVA/ICA 2017. Lecture Notes in Computer Science,* edited by Petr Tichavský, Massoud Babaie-Zadeh, Olivier J.J. Michel, and Nadège Thirion-Moreau, 10169: 323–332. Springer International Publishing. doi:10.1007/978-3-319-53547-0_31.

Loizou, Philopos C. 2017. *Speech enhancement: Theory and practice.* 2nd ed. CRC Press.

Lostanlen, Vincent, Joakim Andén, and Mathieu Lagrange. 2019. "Fourier at the heart of computer music: From harmonic sounds to texture." *Comptes Rendus Physique* 20 (5): 461–473. doi:10.1016/j.crhy.2019.07.005.

Lu, Hao, Yutong Dai, Chunhua Shen, and Songcen Xu. 2020. "Index Networks." *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

Maclennan, Bruce. 1994. *Gabor representations of spatiotemporal visual images.* Technical report CS-91-144. University of Tennessee, Knoxville, Computer Science Department.

Mallat, Stéphane G., and Zhifeng Zhang. 1993. "Matching pursuits with time-frequency dictionaries." *IEEE Transactions on Signal Processing* 41 (12): 3397–3415. doi:10.1109/78.258082.

Mattingley, Jacob, and Stephen Boyd. 2010. "Real-time convex optimization in signal processing." *IEEE Signal Processing Magazine* 27 (3): 50–61.

Mayer, Florian, Donald S. Williamson, Pejman Mowlaee, and DeLiang Wang. 2017. "Impact of phase estimation on single-channel speech separation based on time-frequency masking." *The Journal of the Acoustical Society of America* 141 (6): 4668–4679. doi:10.1121/1.4986647.

McAdams, Stephen. 2019. "The perceptual representation of timbre." Chap. 2 In *Timbre: Acoustics, Perception, and Cognition,* edited by Kai Siedenburg, Charalampos Saitis, Stephen McAdams, Arthur N. Popper, and Richard R. Fay, 69: 23–57. Springer Handbook of Auditory Research. Springer. doi:10.1007/978-3-030-14832-4_2.

McClellan, James H., Ronald W. Schafer, and Mark A. Yoder. 2003. *Signal processing first.* Pearson Prentice Hall.

Mitsufuji, Yuki, Giorgio Fabbro, Stefan Uhlich, and Fabian-Robert Stöter. 2021. "Music demixing challenge at ISMIR 2021." *arXiv preprint arXiv:2108.13559.* https://arxiv.org/abs/2108.13559.

Mittal, Sparsh, Poonam Rajput, and Sreenivas Subramoney. 2021. "A survey of deep learning on CPUs: Opportunities and co-optimizations." *IEEE Transactions on Neural Networks and Learning Systems:* 1–21. doi:10.1109/TNNLS.2021.3071762.

Moore, Brian C. J. 1973. "Frequency difference limens for short-duration tones." *The Journal of the Acoustical Society of America* 54 (3): 610–619. doi:10.1121/1.1913640.

———. 2013. *An introduction to the psychology of hearing.* 6th ed. Emerald Group Publishing Limited.

Müller, Meinard, Daniel Ellis, Anssi Klapuri, and Gaël Richard. 2011. "Signal processing for music analysis." *IEEE Journal of Selected Topics in Signal Processing* 5 (6): 1088–1110. doi:10.1109/JSTSP.2011.2112333.

Naik, Ganesh R., and Wang Wenwu. 2014. *Blind source separation: Advances in theory, algorithms, and applications.* 1st ed. Springer-Verlag.

Nakajima, Hiroaki, Yu Takahashi, Kazunobu Kondo, and Yuji Hisaminato. 2018. "Monaural source enhancement maximizing source-to-distortion ratio via automatic differentiation." *arXiv preprint arXiv:1806.05791.* http://arxiv.org/abs/1806.05791.

Necciari, Thibaud, P. Balazs, Nicki Holighaus, and Peter L. Søndergaard. 2013. "The ERBlet transform: an auditory-based time-frequency representation with perfect reconstruction." *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP):* 498–502. https://ltfat.org/notes/ltfatnote027.pdf.

Nugraha, Aditya, Antoine Liutkus, and Emmanuel Vincent. 2016a. "Multichannel audio source separation with deep neural networks." *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 24 (9): 1652–1664. doi:10.1109/TASLP.2016.2580946.

———. 2016b. "Multichannel music separation with deep neural networks," 1748–1752. doi:10.1109/EUSIPCO.2016.7760548.

Nyquist, Harry. 1928. "Certain topics in telegraph transmission theory." *Transactions of the American Institute of Electrical Engineers* 47 (2): 617–644. doi:10.1109/T-AIEE.1928.5055024.

Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. 1999. *Discrete-time signal processing.* 2nd. Pearson Prentice Hall.

Oppenheim, Jacob, and Marcelo Magnasco. 2012. "Human time-frequency acuity beats the Fourier uncertainty principle." *Physical Review Letters* 110 (4): 4–25. doi:10.1103/PhysRevLett.110.044301.

Pandey, Ashutosh, and DeLiang Wang. 2019. "TCNN: Temporal convolutional neural network for real-time speech enhancement in the time domain." In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 6875–6879. doi:10.1109/ICASSP.2019.8683634.

Parry, R. Mitchell, and Irfan Essa. 2007. "Incorporating phase information for source separation via spectrogram factorization." In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 2: II-661–II-664. doi:10.1109/ICASSP.2007.366322.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "PyTorch: an imperative style, high-performance deep learning library." In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019,* edited by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Pereyra, Marcelo, Philip Schniter, Emilie Chouzenoux, Jean-Christophe Pesquet, Jean-Yves Tourneret, Alfred O. Hero, and Steve McLaughlin. 2016. "A survey of stochastic simulation and optimization methods in signal processing." *IEEE Journal of Selected Topics in Signal Processing* 10 (2): 224–241. doi:10.1109/jstsp.2015.2496908.

Purwins, Hendrik, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-yiin Chang, and Tara Sainath. 2019. "Deep learning for audio signal processing." *IEEE Journal on Selected Topics in Signal Processing* 13 (2): 206–219. doi:10.1109/JSTSP.2019.2908700.

Rabiner, Lawrence, and Ronald Schafer. 2010. *Theory and applications of digital speech processing.* 1st. USA: Pearson Prentice Hall.

Rafii, Zafar, Antoine Liutkus, Fabian-Robert Stöter, Stylianos Ioannis Mimilakis, and Rachel Bittner. 2017. "The MUSDB18 corpus for music separation." doi:10.5281/zenodo.1117372.

———. 2019. "MUSDB18-HQ: an uncompressed version of MUSDB18." doi:10.5281/zenodo.3338373.

Rafii, Zafar, Antoine Liutkus, Fabian-Robert Stöter, Stylianos Ioannis Mimilakis, Derry FitzGerald, and Bryan Pardo. 2018. "An overview of lead and accompaniment separation in music." *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 26 (8): 1307–1335. doi:10.1109/TASLP.2018.2825440.

Rehr, Robert, and Timo Gerkmann. 2016. "A combination of pre-trained approaches and generic methods for an improved speech enhancement." In *Proceedings of the Speech Communication; 12. ITG Symposium,* 1–5.

———. 2019. "An analysis of noise-aware features in combination with the size and diversity of training data for DNN-based speech enhancement." In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 601–605. doi:10.1109/ICASSP.2019.8682991.

Rubinstein, Ron, Alfred Bruckstein, and Michael Elad. 2010. "Dictionaries for sparse representation modeling." *Proceedings of the IEEE* 98 (6): 1045–1057. doi:10.1109/JPROC.2010.2040551.

Sawata, Ryosuke, Stefan Uhlich, Shusuke Takahashi, and Yuki Mitsufuji. 2021. "All for one and one for all: improving music separation by bridging networks." *arXiv preprint arXiv:2010.04228:* 1–5. https://www.ismir2020.net/assets/img/virtual-booth-sonycsl/cUMX_paper.pdf.

Schörkhuber, Christian, and Anssi Klapuri. 2010. "Constant-Q transform toolbox for music processing." In *7th Sound and Music Computing Conference (SMC2010).* doi:10.5281/zenodo.849741.

Schörkhuber, Christian, Anssi Klapuri, Nicki Holighaus, and Monika Dörfler. 2014. "A Matlab toolbox for efficient perfect reconstruction time-frequency transforms with log-frequency resolution." In *Audio Engineering Society Conference: 53rd International Conference: Semantic Audio,* 1–8. http://www.aes.org/e-lib/browse.cfm?elib=17112.

Schörkhuber, Christian, Anssi Klapuri, and Alois Sontacchi. 2012. "Pitch shifting of audio signals using the constant-Q transform." In *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12),* 1–8.

Shannon, Claude E. 1948. "A mathematical theory of communication." *The Bell System Technical Journal* 27 (3): 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.

Shi, Ziqiang, Huibin Lin, Liu Liu, Rujie Liu, and Jiqing Han. 2019. "Is CQT more suitable for monaural speech separation than STFT? an empirical study." *arXiv preprint arXiv:1902.00631.* https://arxiv.org/abs/1902.00631.

Siedenburg, Kai. 2019. "Specifying the perceptual relevance of onset transients for musical instrument identification." *The Journal of the Acoustical Society of America* 145 (2): 1078–1087. doi:10.1121/1.5091778.

Siedenburg, Kai, and Monika Doerfler. 2011. "Structured sparsity for audio signals." In *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11).*

Simpson, Andrew. 2015. "Time-frequency trade-offs for audio source separation with binary masks." *arXiv preprint arXiv:1504.07372.* https://arxiv.org/abs/1504.07372.

Skiena, Steven S. 2008. *The algorithm design manual.* 2nd ed. Springer. doi:10.1007/978-1-84800-070-4.

Stöter, Fabian-Robert, Antoine Liutkus, and Nobutaka Ito. 2018. "The 2018 signal separation evaluation campaign." In *Latent Variable Analysis and Signal Separation. Lecture Notes in Computer Science,* edited by Yannick Deville, Sharon Gannot, Russell Mason, Mark Plumbley, and Dominic Ward, 10891: 293–305. Springer International Publishing. doi:10.1007/978-3-319-93764-9_28.

Stöter, Fabian-Robert, Stefan Uhlich, Antoine Liutkus, and Yuki Mitsufuji. 2019. "Open-Unmix: A reference implementation for music source separation." *Journal of Open Source Software* 4 (41): 1667. doi:10.21105/joss.01667.

Uhlich, Stefan, Marcello Porcu, Franck Giron, Michael Enenkl, Thomas Kemp, Naoya Takahashi, and Yuki Mitsufuji. 2017. "Improving music source separation based on deep neural networks through data augmentation and network blending." In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 261–265. doi:10.1109/ICASSP.2017.7952158.

Velasco, Gino Angelo, Nicki Holighaus, Monika Doerfler, and Thomas Grill. 2011. "Constructing an invertible constant-Q transform with nonstationary Gabor frames." In *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11).*

Vincent, Emmanuel, Nancy Bertin, Remi Gribonval, and Frederic Bimbot. 2014. "From blind to guided audio source separation: How models and side information can improve the separation of sound." *IEEE Signal Processing Magazine* 31 (3): 107–115. doi:10.1109/MSP.2013.2297440.

Vincent, Emmanuel, Rémi Gribonval, and Cédric Févotte. 2006. "Performance measurement in blind audio source separation." *IEEE Transactions on Audio, Speech, and Language Processing* 14 (4): 1462–1469. doi:10.1109/TSA.2005.858005.

Vincent, Emmanuel, Hiroshi Sawada, Pau Bofill, Shoji Makino, and Justinian Rosca. 2007. "First stereo audio source separation evaluation campaign: data, algorithms and results." In *Proceedings of the 7th International Conference on Independent Component Analysis and Blind Source Separation (ICA). Lecture Notes in Computer Science,* 4666: 552–559. Springer.

Voran, Stephen. 2008. "Estimation of speech intelligibility and quality." Chap. 28 In *Handbook of Signal Processing in Acoustics,* edited by David Havelock, Sonoko Kuwano, and Michael Vorländer, 483–520. Springer. doi:10.1007/978-0-387-30441-0_28.

Walt, Stéfan van der, S. Chris Colbert, and Gael Varoquaux. 2011. "The NumPy array: A structure for efficient numerical computation." *Computing in Science & Engineering* 13 (2): 22–30. doi:10.1109/MCSE.2011.37.

Wang, DeLiang, and Guy J. Brown. 2006. "Fundamentals of computational auditory scene analysis." In *Computational auditory scene analysis: Principles, algorithms, and applications,* edited by DeLiang Wang and Guy J. Brown. Wiley-IEEE-Press.

Wichern, Gordon, Joe Antognini, Michael Flynn, Licheng Richard Zhu, Emmett McQuinn, Dwight Crow, Ethan Manilow, and Jonathan Le Roux. 2019. "WHAM!: extending speech separation to noisy environments." *arXiv preprint arXiv:1907.01160.* https://arxiv.org/abs/1907.01160.

Wilkerson, Daniel Shawcross. 2014. "Harmony explained: progress towards a scientific theory of music." *arXiv preprint arXiv:1202.4212.* https://arxiv.org/abs/1202.4212.

Wu, Carole-Jean, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, et al. 2019. "Machine learning at Facebook: Understanding inference at the edge." In *IEEE International Symposium on High Performance Computer Architecture (HPCA),* 331–344. doi:10.1109/HPCA.2019.00048.

Xu, Yun, and Royston Goodacre. 2018. "On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning." *Journal of Analysis and Testing* 2 (3): 249–262. doi:10.1007/s41664-018-0068-2.

Yousuf, Hana, Michael Gaid, Said Salloum, and Khaled Shaalan. 2020. "A systematic review on sequence to sequence neural network and its models." *International Journal of Electrical and Computer Engineering* 11 (3): 2315–2326. doi:10.11591/ijece.v11i3.pp2315-2326.

Yu, Chin-Yun, and Kin-Wai Cheuk. 2021. "Danna-Sep: Unite to separate them all." *arXiv preprint arXiv:2112.03752.* https://arxiv.org/abs/2112.03752.

Yu, Yong, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. 2019. "A review of recurrent neural networks: LSTM cells and network architectures." *Neural Computation* 31 (7): 1235–1270. doi:10.1162/neco_a_01199.

Appendix A Testbench computer specifications

The hardware and software specifications of the computer which produced the results shown throughout this thesis are as follows:

- Motherboard: Gigabyte Aorus X570 Elite Wifi
- CPU: AMD Ryzen 5950X
- Memory: 64GB DDR4
- Storage: 1TB ADATA SX8200PNP NVMe
- GPU, primary: NVIDIA RTX 3080 Ti (12GB memory)
- GPU, secondary: NVIDIA RTX 2070 Super (8GB memory)
- OS: Fedora 34 Workstation Edition, 64-bit
- Linux kernel version: 5.133.10-200
- Python 3 version: 3.9.6 (default, Jul 16 2021, 00:00:00)
- NVIDIA driver version: 470.63.01
- NVIDIA CUDA toolkit version: 11.4

Appendix B Code availability

The code projects associated with this thesis are published as open-source software to encourage reproducibility of results.

They are split across the following projects:

- NSGT/sliCQT PyTorch copy from sections 3.1.5 and 3.1.6: https://github.com/sevagh/nsgt
- museval (BSS metrics evaluation) CuPy copy from Section 3.1.9: https://github.com/sevagh/sigsep-mus-eval
- xumx-sliCQ neural network from Section 3.2: https://github.com/sevagh/xumx-sliCQ
- LaTeX files and scripts for generating this thesis, including all plots and demixing results in Chapter 4: https://github.com/sevagh/xumx_slicq_extra
- Submissions made to the ISMIR 2021 Music Demixing Challenge (see Appendix F): https://gitlab.aicrowd.com/sevagh/music-demixing-challenge-starter-kit

All Python environments were designed to be reproducible with pip requirements.txt files or Conda environment files bundled with the source code:

- Pip file for oracles, trained model evaluations, boxplot creation, and performance benchmarks: https://github.com/sevagh/xumx_slicq_extra/blob/main/mss_evaluation/mss-oracle-experiments/requirements-cupy.txt
- Conda environment file for the NSGT/sliCQT PyTorch implementation: https://github.com/sevagh/nsgt/blob/main/conda-env.yml
- Conda environment file for the xumx-sliCQ neural network: https://github.com/sevagh/xumx-sliCQ/blob/main/scripts/environment-gpu-linux-cuda11.yml

I take code availability and reproducibility seriously. Feel free to e-mail me[82] if you encounter any errors, discrepancies, or difficulties with reproducing any of the results.

82. sevag.hanssian@mail.mcgill.ca, sevagh+thesis@pm.me

Appendix C Octave scale for the NSGT

The octave scale for the NSGT takes a bins-per-octave (bpo) argument from which the total number of frequency bins is computed. Equation (1) describes how to compute the total bins from the bins-per-octave setting of the octave scale:

$$K = [B \log_2(\xi_{\max}/\xi_{\min}) + 1] \tag{1}$$

where K is the total bins, B is the bins-per-octave, and $\xi_{\min,\max}$ are the minimum and maximum frequencies. This equation was shown previously in equation (10) in Section 2.2.3.

By contrast, the logarithmic scale takes the total number of frequency bins as a direct argument. Examples of the octave and logarithmic scales are shown in Figure 66.

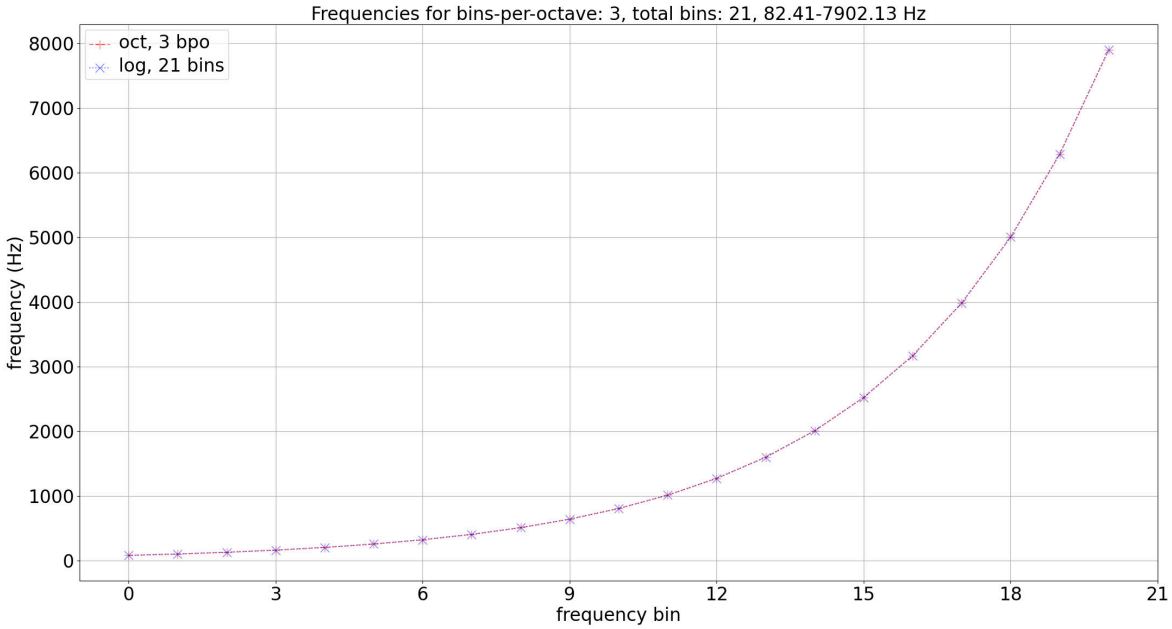


Figure 66: The octave and log scales compared. The minimum frequency for both scales is set to $\xi_{\min} = 82.41$ Hz, which is the frequency of the $E2$ musical note. The maximum frequency for both scales is set to $\xi_{\max} = 7{,}902.13$ Hz, which is the frequency of the $B8$ musical note. Note that both scales are identical. The octave scale uses $B = 3$ bpo, resulting in $K = 21$ frequency bins, from equation (1). The log scale uses 21 frequency bins.

Appendix D sliCQT dimensionality: fixing $f_{\max}$ to the Nyquist rate

In Section 3.1.8, I described that in the parameter search for the frequency scale of the sliCQT, the maximum frequency $f_{\max}$ was fixed to 22,050 Hz (the Nyquist rate of the 44,100 Hz sample rate of the music data set), because it led to a smaller sliCQT than $f_{\max} < 22,050$ Hz in several cases. Note that this is not always true, and depends on the other sliCQT parameters. Out of 809 examples evaluated for this section, 51 of them had a larger transform with the smaller value of $f_{\max}$, and 758 of them (the majority) had a smaller transform when using the smaller value of $f_{\max}$. The evaluation was done on one song from the MUSDB18-HQ dataset, "Skelpolu - Human Mistakes," which has a duration of 324.69 seconds, or 14,318,640 samples at a 44,100 Hz sample rate.

Table 14 shows some examples of sliCQT parameters for the Constant-Q scale from the more rare case where a larger frequency range with $f_{\max}$ = Nyquist rate led to a smaller transform than a smaller frequency range with $f_{\max}$ < Nyquist rate. Table 15 shows more typical counter-examples, where smaller frequency ranges result in smaller transforms.

Table 14: sliCQT examples where $f_{\max}$ = Nyquist rate resulted in a smaller transform.

Total bins	$f_{\min}$ (Hz)	$f_{\max}$ (Hz)	Number of coefficients
11	96.8	14,771.2	214,263,720
11	96.8	22,050	182,892,400 (-14%)
26	27.7	18,202.8	320,350,992
26	27.7	22,050	294,027,200 (-8%)
64	106.7	20,962.8	292,568,064
64	106.7	22,050	281,151,360 (-3.7%)
203	74.4	21,726	323,409,856
203	74.4	22,050	318,891,744 (-1.5%)

Table 15: sliCQT examples where $f_{\max}$ = Nyquist rate resulted in a larger transform.

Total bins	$f_{\min}$ (Hz)	$f_{\max}$ (Hz)	Number of coefficients
26	88.4	20,341.5	299,574,080
26	88.4	22,050	329,643,392 (+10%)
135	23.4	18,935.7	334,379,880
135	23.4	22,050	397,612,800 (+18%)
289	36.8	18,874.5	310,746,672
289	36.8	22,050	362,147,328 (+16%)

Appendix E STFT and sliCQT dimensionality compared

In Section 3.2.1, I described that X-UMX uses the STFT of six-second audio sequences as its input, and that I had to use one second sequences for the sliCQT in xumx-sliCQ so that it could fit in the GPU memory during training. This is because the sliCQT of an audio sequence is larger than the STFT. In this section, I will show several different configurations of sliCQT to support this claim.

Out of 122 examples evaluated for this section, in every single case the sliCQT was larger than the STFT. The evaluation was done on one song from the MUSDB18-HQ dataset, "Skelpolu - Human Mistakes," which has a duration of 324.69 seconds, or 14,318,640 samples at a 44,100 Hz sample rate.

Table 16 shows some examples of sliCQT parameters for the Constant-Q scale compared to the dimensionality and number of coefficients of the STFT. Alongside the STFT with a window size of 4,096 samples and overlap of 1,024 samples as used by X-UMX, I included a smaller STFT with a window size of 1,024 and overlap of 256 samples, and a larger STFT with a window size of 8,192 and an overlap of 2,048 samples.

Table 16: sliCQT examples with the Constant-Q scale, compared to the STFT.

Transform	Tensor shape	Number of coefficients
STFT, 1024	(2, 513, 18646)	19,130,796
STFT, 4096	(2, 2049, 4663)	19,108,974
STFT, 8192	(2, 4097, 2332)	19,108,408
sliCQT, 91 bins, 107.13–10078.31 Hz	(880, 2, 91, 752)	120,440,320
sliCQT, 20 bins, 39.75–22050 Hz	(2157, 2, 19, 3184)	260,979,744
sliCQT, 171 bins, 58.14–22050 Hz	(331, 2, 171, 3036)	343,681,272
sliCQT, 25 bins, 20.43–22050 Hz	(970, 2, 25, 8632)	418,652,000

Appendix F xumx-sliCQ experiments

In Section 3.2.2, I mentioned a repeated tuning process where I evaluated different neural network architectures and parameters for xumx-sliCQ. In this appendix, I will describe the nature of these experiments.

Before starting this thesis in May 2021, I worked on two related projects; a project[83] which used the NSGT (Nonstationary Gabor Transform) for music demixing using a CDAE (convolutional denoising autoencoder) neural architecture, and a project[84] which explored different time-frequency transforms and the time-frequency tradeoff in music source separation algorithms based on spectrogram masking. In the second project, I used Open-Unmix as a high-performance baseline.

In May 2021, Sony, the major Japanese technology company,[85] and Inria, a French national research institution,[86] sponsored a music demixing research challenge on the crowdsourced AI platform, AIcrowd.[87] The challenge was called the ISMIR 2021 MDX (Music Demixing) Challenge, with an associated satellite workshop planned for the ISMIR 2021 conference. The participants were encouraged to either modify the baselines (like Open-Unmix) or submit their own custom code to win the challenge. The MDX challenge ran from May to July 2021, giving participants three months to work on their neural networks. I chose to align my master's thesis project with the challenge, and made my goal the modification of Open-Unmix to use the sliCQT (sliced Constant-Q Transform).

The collaborative environment of the challenge was very fun, and I made 32 total submissions[88] to the challenge, resulting in a final place of 34 on the leaderboard. In my submission process, I tested different neural architectures and parameters with both the CDAE and Bi-LSTM variants until one of them achieved good results. My work in the challenge resulted in the creation of xumx-sliCQ,[89] the final model of this thesis. I wrote an article on xumx-sliCQ,[90] and I participated in the virtual ISMIR 2021 conference, where I showed xumx-sliCQ at the poster session[91] of the MDX 21 challenge.

83. https://github.com/sevagh/MiXiN
84. https://github.com/sevagh/Music-Separation-TF
85. https://www.sony.com/en/
86. https://www.inria.fr/en
87. https://www.aicrowd.com/challenges/music-demixing-challenge-ismir-2021
88. https://gitlab.aicrowd.com/sevagh/music-demixing-challenge-starter-kit
89. https://github.com/sevagh/xumx-sliCQ
90. https://mdx-workshop.github.io/proceedings/hanssian.pdf, https://arxiv.org/abs/2112.05509
91. https://github.com/sevagh/xumx_slicq_extra/blob/main/drafts-and-blobs/mdx21-poster.pdf