Aprendizaje de representaciones latentes de timbres musicales con autocodificadores y autocodificadores variacionales
Learning latent representations of timbre using autoencoders and variational autoencoders

Trabajo de Fin de Grado

Curso 2024–2025

Autor

Pablo García López

Directores

Miguel Palomino Tarjuelo

Jaime Sánchez Hernández

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

Aprendizaje de representaciones latentes de timbres musicales con autocodificadores y autocodificadores variacionales
Learning latent representations of timbre using autoencoders and variational autoencoders

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Pablo García López

Directores
Miguel Palomino Tarjuelo
Jaime Sánchez Hernández

Convocatoria: *Junio 2025*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

26 de mayo de 2025

Resumen

Aprendizaje de representaciones latentes de timbres musicales con autocodificadores y autocodificadores variacionales

En este trabajo de fin de grado se aplica el aprendizaje profundo a la reconstrucción y generación de audio musical, considerando dos arquitecturas: autocodificadores (AEs) y autocodificadores variacionales (VAEs). A diferencia de la generación simbólica de música, que opera sobre formatos de más alto nivel como MIDI, adoptamos un enfoque musical no simbólico y trabajamos directamente sobre representaciones de más bajo nivel como son las muestras de audio y sus correspondientes representaciones frecuenciales.

Tras una revisión de la historia de la composición algorítmica y de varios métodos de síntesis de audio, se utilizan autocodificadores convolucionales para comprimir y reconstruir notas musicales, y autocodificadores variacionales para sintetizar audio mediante el muestreo dentro de representaciones ocultas aprendidas. A lo largo del proyecto, se investigan diversas configuraciones y arquitecturas de modelos con el objetivo de mejorar la calidad de reconstrucción y generación de audio.

Palabras clave

Aprendizaje profundo, Autocodificadores, Autocodificadores variacionales, Espacio latente, Espectrograma, Red Convolucional.

Abstract

Learning latent representations of timbre using autoencoders and variational autoencoders

In this Bacherlor's Thesis, deep learning is applied to musical audio reconstruction and generation, considering two architectures: autoencoders (AEs) and variational autoencoders (VAEs). Unlike symbolic music generation, which operates on higher-level formats such as MIDI, we adopt a non-symbolic approach and work directly with lower-level data such as raw audio samples and their corresponding frequency-domain transformations.

Following an overview of the history of algorithmic composition and several methods of audio synthesis, convolutional autoencoders are used to compress and reconstruct musical notes, and variational autoencoders to synthesize audio by sampling within learned hidden representations. Throughout the project, various model settings and architectures are investigated in an attempt of enhancing reconstruction quality and generation performance.

Keywords

Deep Learning, Autoencoders, Variational Autoencoders, Latent space, Spectrogram, Convolutional network.

Contents

List of figures

List of tables

Chapter 1

Introduction

1.1. Preliminaries and objectives

In recent years, deep learning has revolutionized generative tasks in fields like image synthesis, natural language processing, and audio production. Within music, research has generally split into *symbolic* approaches (focusing on note events, pitches, and durations in formats like MIDI) and *non-symbolic* approaches (focusing on raw audio waveforms or spectrograms).

Commercial digital audio workstations (DAWs) and synthesizers already allow users to generate audio with great precision. However, these are often not driven by deep learning based methods. Moreover, there is compelling interest in exploring new audio possibilities achieved by learned latent representations, e.g., timbres that might not exist in standard synthesizer libraries.

This bachelor's thesis focuses on the implementation of two different, though related, deep learning architectures applied to music: autoencoders (AEs) and variational autoencoders (VAEs). AEs will allow us to see how well musical fragments can be mimicked, while VAEs enable music generation through latent space sampling. While the results may not surpass already existing commercial synthesizers, such models can reveal new pathways for interactive sound design and serve as a starting point for other research projects.

Regardless, we would like this work to serve as an introduction and guide for students and anyone interested in the use of deep learning in music. While we do not assume extensive knowledge from the reader, we will also not go into excessively detailed explanations to keep the text accessible.

1.2. Work Plan

This project will proceed through an iterative, experimentation-driven process. We begin by diving into the fundamentals of audio representation (sampling, STFT, and spectrograms), to ensure a solid foundation for the following chapters. The next phase is the implementation of a convolutional autoencoder: selecting its architecture, tuning parameters such as channel widths and latent dimensionality, and training under varied STFT window and hop settings to balance fidelity against computational cost. Once a satisfactory autoencoder is established, we will transition to the variational autoencoder and deal with the main problems that arise with these models. Throughout, our idea is to record each experiment's configuration, training and validation curves archived, and representative audio samples. Our goal is to have, by the end of this workflow, a clear narrative of model development, performance trade-offs and audio outputs.

Chapter 2

State of the art

In this chapter, we aim to provide a brief overview of the evolution of algorithmic composition and, secondly, explore non-symbolic music generation more in depth.

2.1. Brief history of algorithmic composition

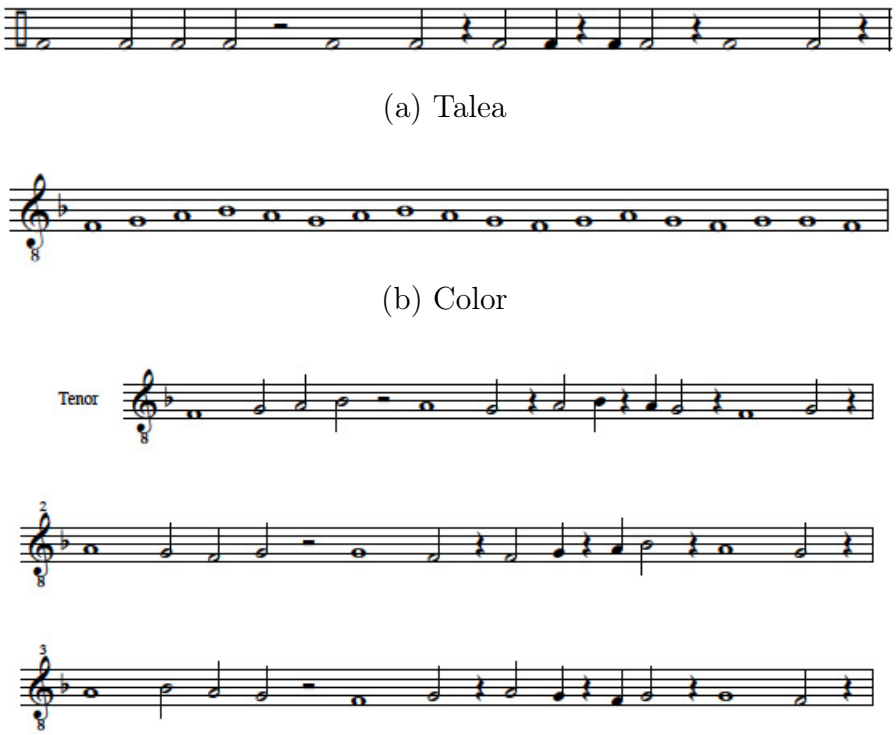
Algorithmic composition is the process of using some formal process to make music with minimal human intervention (Alpern, 1995) and can be divided into two main categories: non-computer-aided and computer-aided methods. The reader should note the following sections are nothing but a succinct run-through of algorithmic composition and will necessarily be incomplete in terms of their content.

2.1.1. Non-computer-aided methods

Algorithmic composition dates back thousands of years. In Ancient Greece, philosophers such as Pythagoras (500 B.C.) viewed music as fundamentally linked to mathematics, believing that musical harmony reflected universal order (Simoni, 2003). These ancient Greek "formalisms", however, are rooted mostly in theory, and their strict application to musical performance itself is probably questionable (Grout et al., 2010). Therefore, it can't really be said that Ancient Greek music composition was purely algorithmic in the sense we have defined it, but it undoubtably set the path towards important formal extra-human processes.

Ars Nova marked a pivotal shift in musical thought, where composers such as Philippe de Vitry and Guillaume de Machaut began to disentangle rhythm from pitch and text. By systematically applying rhythmic patterns, known as the *talea*, to fixed melodic cells called the *chroma*, they developed a method of composition that can be seen as an early form of algorithmic music-making (Simoni, 2003). This approach can be better understood by looking at Figure 2.1, which represents the talea, chroma and the mapping between them of *De bon espoir-Puisque la douce-*

Speravi by Guillaume de Machaut.



(a) Talea

(b) Color

(c) Mapping between them

Figure 2.1: Elements of the isorhythmic motet *De bon espoir–Puisque la douce–Speravi* by Guillaume de Machaut: (a) talea, (b) color, and (c) mapping between them. Retrieved from (Simoni, 2003).

In the Renaissance and the Baroque periods, algorithmic methods became more explicit through forms like the canon, where composers, like Johann Sebastian Bach, created strict rules dictating how single melodies are to be imitated by multiple voices at different times.

A famous Classical-era example is Mozart's *Musikalisches Würfelspiel* ("Dice Music") in which musical phrases were randomly assembled by dice rolls to allow any composer to form a waltz, explicitly employing chance-based algorithmic composition (Maurer, 1999).

The 20th century introduced more complex algorithmic techniques through serialism, where composers like Arnold Schoenberg and Alban Berg employed systematic tone-row matrices (see Figure 2.2) to structure their compositions through fixed rules. Composers such as John Cage and Karlheinz Stockhausen later incorporated chance and probabilistic methods, further extending the tradition of algorithmic music before the advent of computers (Simoni, 2003).

	I_0	I_{10}	I_3	I_4	I_2	I_1	I_{11}	I_9	I_8	I_7	I_5	I_6	
P_0	E♭	D♭	G♭	G	F	E	D	C	B	B♭	A♭	A	R_0
P_2	F	E♭	A♭	A	G	G♭	E	D	D♭	C	B♭	B	R_2
P_9	C	B♭	E♭	E	D	D♭	B	A	A♭	G	F	G♭	R_9
P_8	B	A	D	E♭	D♭	C	B♭	A♭	G	G♭	E	F	R_8
P_{10}	D♭	B	E	F	E♭	D	C	B♭	A	A♭	G♭	G	R_{10}
P_{11}	D	C	F	G♭	E	E♭	D♭	B	B♭	A	G	A♭	R_{11}
P_1	E	D	G	A♭	G♭	F	E♭	D♭	C	B	A	B♭	R_1
P_3	G♭	E	A	B♭	A♭	G	F	E♭	D	D♭	B	C	R_3
P_4	G	F	B♭	B	A	A♭	G♭	E	E♭	D	C	D♭	R_4
P_5	A♭	G♭	B	C	B♭	A	G	F	E	E♭	D♭	D	R_5
P_7	B♭	A♭	D♭	D	C	B	A	G	G♭	F	E♭	E	R_7
P_6	A	G	C	D♭	B	B♭	A♭	G♭	F	E	D	E♭	R_6
	RI_0	RI_{10}	RI_3	RI_4	RI_2	RI_1	RI_{11}	RI_9	RI_8	RI_7	RI_5	RI_6	

Figure 2.2: Serialism matrix. Retrieved from `https://www.musictheory.net`.

2.1.2. Computer-Aided Methods

The advent of computers in the mid-20th century significantly advanced algorithmic composition, introducing computational techniques that expanded creative possibilities. Early pioneers like Lejaren Hiller and Leonard Isaacson composed the *Illiac Suite* (1957), one of the first pieces generated entirely by computer algorithms (Hiller and Isaacson, 1979). They utilized a generator/modifier/selector framework, where musical materials were algorithmically created, modified, and selected based on predefined rules (Maurer, 1999).

Composer Iannis Xenakis introduced *stochastic music*, employing probabilistic methods to generate musical structures. For instance, in his work *Atrées* (1962), Xenakis used probability distributions and random number generators to determine musical elements (Xenakis, 1992).

Computer-aided algorithmic composition can be categorized into three main approaches:

1. Stochastic systems: they incorporate randomness, ranging from simple random note generation to complex applications of chaos theory and nonlinear dynamics (Nierhaus, 2009).

2. Rule-Based systems: these utilize explicitly defined compositional rules or grammars, similar to earlier non-computer methods like the Renaissance canons or serialist compositions we have talked about. Notable examples include William Schottstaedt's automatic species counterpoint program and Kemal Ebcioglu's CHORAL system, which generate music based on historical compositional rules (Cope, 1991).

3. Artificial Intelligence systems: these systems extend rule-based methods by allowing a computer to develop or evolve compositional rules autonomously. David Cope's Experiments in Musical Intelligence (EMI) exemplifies this approach, analyzing existing compositions to create new music emulating specific composers' styles (Maurer, 1999).

2.2. Non-symbolic music generation

In Section 2.1 we gave an overview of historical algorithmic composition along with its two main branches: non-computer-aided and computer-aided methods, which largely focus on *symbolic* or high-level approaches. In this section, however, we turn our attention to *non-symbolic* music generation, where the emphasis is on generating and shaping audio signals directly.

We begin with an overview of foundational digital synthesis systems, which provided the bedrock for modern audio generation. We then discuss recent AI-based approaches, including various deep-learning architectures capable of producing music at the waveform (or spectrogram) level.

2.2.1. Traditional Synthesis Systems

Before the advent of deep learning, sound synthesis relied on signal processing techniques like additive, subtractive, and FM synthesis. These methods, rooted in mathematical and physical principles, shaped much of 20th-century electronic music. This subsection outlines their core ideas and historical significance, as well as a brief mention of more recent approaches such as granular, physical modeling, and spectral synthesis.

2.2.1.1. Additive Synthesis

Additive synthesis is a sound creation method based on the Fourier Theorem, which states that any sound can be decomposed into a sum of sine waves, or partials (Fourier, 2009). By controlling the frequency, amplitude, and phase of each partial, one can construct complex timbres from these elementary components. Historically, this idea finds early expression in acoustic instruments such as the pipe organ (see Figure 2.3), where multiple pipes combine to produce rich harmonic textures, and in pioneering electronic devices like the Telharmonium—often considered one of the first additive synthesizers.

The method was further advanced in the mid-20th century through the work of Max Mathews at Bell Labs, who demonstrated the vast potential of digital additive synthesis for generating evolving and intricate soundscapes (Mathews, 1963). Although the flexibility of additive synthesis allows a precise crafting of any sound, its complexity made it less practical compared to the more cost-effective subtractive synthesis during the analog era. With the rise of digital signal processing, however, additive synthesis experienced a revival. This influenced the appearance of modern hybrid synthesizers that incorporate both additive and subtractive techniques (Roads, 1996; Tagi, 2023a).

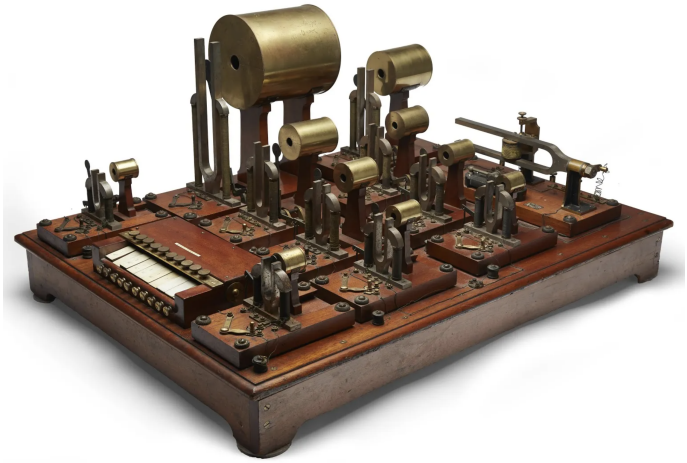


Figure 2.3: Pipe organ created by Hermann von Helmholtz around 1862. Retrieved from `https://shorturl.at/VuT2w`.

2.2.1.2. Subtractive Synthesis

Subtractive synthesis is one of the most widely used methods in sound synthesis systems. Conceptually, this approach is not harder to understand than additive synthesis: starting with a complex waveform as the raw material, we want to shape it by filtering out unwanted frequencies, much like sculpting a figure from a block of marble. What do we shape this raw signal with? Well, a subtractive synthesizer primarily uses these components:

- Oscillators: are responsible for generating the initial complex waveforms rich in harmonics.
- Filters: which remove (or subtract) selected frequency components. This can be done with filters such as the so-called low-pass or high-pass, which respectively remove high and low frequencies.
- Amplifiers and envelope generators: amplifiers control the overall level of the sound over time while an envelope generator is a tool that shapes how a sound evolves when a note is played by controlling four different dimensions (see Figure 2.4):

  1. Attack: how quickly the sound reaches its peak.
  2. Decay: how fast it drops from the peak to a steady level.
  3. Sustain: the level at which the sound holds while the note is sustained.
  4. Release: how rapidly the sound fades after the note is released.

  These stages allow to craft sounds that can be sharp and percussive or smooth and evolving (Hahn, 2022).

- LFOs (Low-Frequency Oscillators): LFOs operate at very low frequencies that are below the threshold of human hearing and can create effects like vibrato

or tremolo, therefore bringing the possibility of adding movement and life to a sound (Tagi, 2023c).

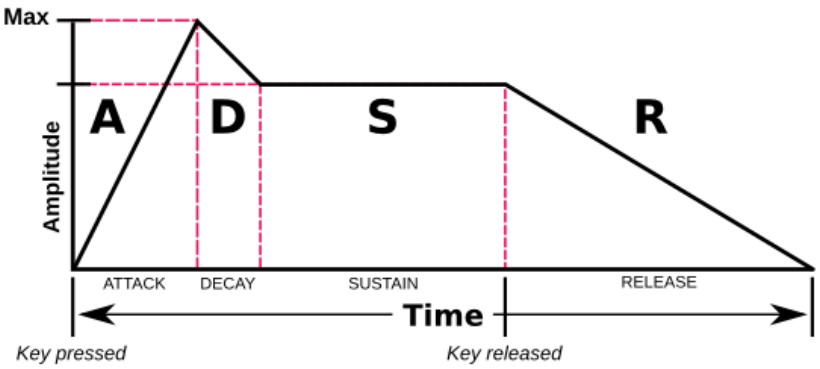


Figure 2.4: Illustration of Attack (A), Decay (D), Sustain (D) and Release (R). Retrieved from `https://shorturl.at/Zj3UQ`.

Historically, subtractive synthesis dates back as far as 1930 with instruments such as the Trautonium and continued to be used throughout the 20th century, for example, by Robert Moog's Minimoog (Réveillac, 2024).

2.2.1.3. Frequency Modulation (FM) Synthesis

Frequency modulation synthesis (FM synthesis) is a method of sound design in which one oscillator, known as the *modulator*, modulates the frequency of another oscillator, called the *carrier*, which allows one to create new frequency components without filters (see Figure 2.5). In simple terms, rather than "sculpting" a sound by removing frequencies (as in subtractive synthesis), FM synthesis generates complex spectra by dynamically altering the pitch of a carrier with a modulating signal (Cymatics, 2025).

FM synthesis is the result of John Chowning's experiments in 1967 at Stanford University: by using sine waves (using one to modulate the frequency of another) Chowning discovered that a variety of new timbres could be generated (Cymatics, 2025).

FM synthesis revolves around the building block of an *operator*, that typically includes an oscillator, an amplifier, and an envelope generator (recall we have talked about these in subtractive synthesis). Operators can serve as carriers, modulators, or both, and they are arranged in various configurations or algorithms to produce different sound textures (Tagi, 2023b).

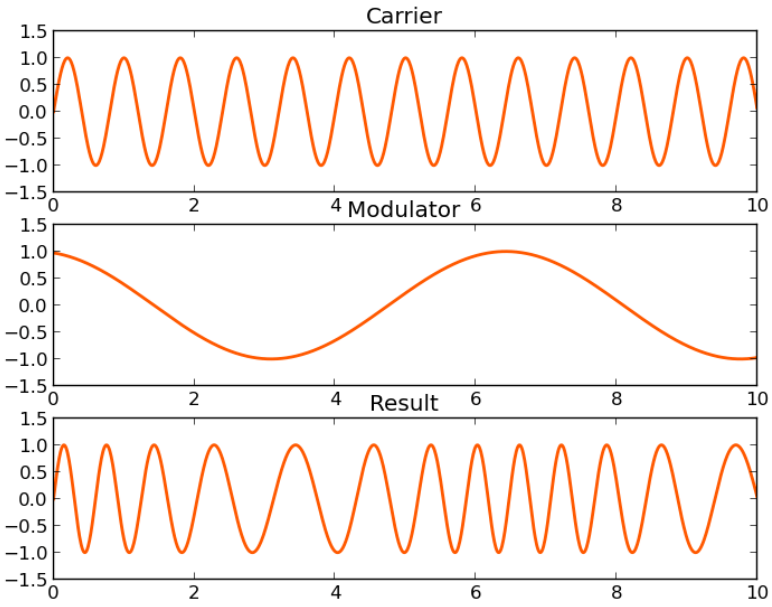


Figure 2.5: Illustration of a carrier, a modulator and the output. Retrieved from (Cymatics, 2025).

2.2.1.4. Other Approaches: Granular, Physical and Spectral Modeling

Beyond the traditional methods of additive, subtractive, and FM synthesis, there do exist other important and more modern sound generation techniques. We will very briefly talk about three of them.

Granular synthesis works by breaking a sound into tiny segments called grains. These grains can then be individually rearranged to create a rich variety of sound textures, from subtle ambiences to complex, glitch-like effects (Roads, 1996).

Physical modeling synthesis takes a different route by simulating the behavior of real-world systems, such as vibrating strings, which allows for realistic emulations of acoustic instruments. (Smith, 1996)

Spectral modeling involves analyzing a sound's frequency content (often with Fourier techniques) and then resynthesizing it by manipulating its spectral components. This way, interpolation and morphism between sounds can be achieved in a simpler way than with other traditional synthesis methods (Serra and Smith, 1990).

2.2.2. Modern AI-Driven Non-Symbolic Music Generation

Unlike the traditional systems based on handcrafted signal-processing algorithms, deep learning methods for non-symbolic music generation learn representations directly from data. They typically produce raw audio waveforms or time-frequency representations, such as spectrograms. In recent years, several influential neural

architectures have emerged, capable of generating musical audio directly at the waveform level. Our model will also follow this paradigm.

2.2.2.1. Waveform Modeling Approaches

WaveNet is a neural network initially designed for generating realistic speech audio directly from waveform samples. WaveNet operates by predicting each audio sample based on previously generated samples, using dilated causal convolutional layers. These dilations expand the receptive field, allowing the network to capture both fine-grained details and wider temporal context, which seems essential for modeling realistic audio textures. Although initially designed for text-to-speech synthesis, WaveNet was quickly adapted for music and demonstrated its effectiveness in capturing musical features at the waveform level (van den Oord et al., 2016).

Another significant development was the introduction of SampleRNN (Mehri et al., 2017), a hierarchical recurrent neural network (RNN) architecture specifically created to handle the complexity of raw audio generation. SampleRNN models waveforms at multiple temporal scales by stacking RNN layers hierarchically, allowing each layer to focus on different aspects of musical structure. Higher layers manage broader temporal dependencies, capturing long-term patterns, while lower layers handle local audio details (Maurer, 1999).

One more significant breakthrough in non-symbolic music generation was achieved with Generative Adversarial Networks (GANs). An important example is *GANSynth*, developed by *Google Magenta*, which synthesizes audio notes using generative adversarial networks operating in the frequency domain (Engel et al., 2019). Unlike WaveNet and SampleRNN, which sequentially generate each sample, GANSynth produces entire audio clips simultaneously by generating spectrograms and instantaneous frequency components. This approach results in more realistic and coherent musical timbres. Additionally, GANSynth allows for audio synthesis control, which enables independent manipulation of pitch and timbre.

Chapter 3

Audio representation basics

In this chapter, we explain the fundamentals of audio and their most frequent representations. We find this necessary in order for the reader to possess a foundational and intuitive understanding of the deep learning model we have built and of which we will talk about in Chapter 5.

First, we will give a brief introduction of what audio is and its basic components. Next, both time and frequency domain representations will be explained, along with subtopics related to each of them.

3.1. Introduction to audio data

According to Oxford's dictionary, sound is the collection of vibrations that travel through the air or another medium and can be heard when they reach a person's or animal's ear. This is probably the definition any regular could have come up with, but in order to deeply understand sound, a closer look at its physical meaning is needed.

A common approach is to model sound as a wave that propagates through some medium. Like any other wave, it is constituted by (see Figure 3.1):

- Amplitude: or simply the distance (measured in meters) of the wave from the resting position at a given point in time. Humans perceive amplitude as loudness. The bigger the amplitude, the louder the wave will sound.
- Frequency, period and wavelength: these three properties are closely related to the speed of the wave. Frequency is the number of oscillations of the wave during some period of time; the period and the wavelength are respectively the time and distance it takes for the wave to start repeating itself. Mathematically, if f, T, λ and v, respectively denote the frequency, period, wavelength and speed of the wave, we have: $\lambda = v \cdot T = v/f$. Frequency is measured in Hertz (Hz), the period in seconds, and the wavelength in meters.

Since frequency, period and wavelength are all directly or inversely related, explaining how humans perceive one of them allows us to understand the rest. In particular, we perceive the frequency of a sound as its pitch, which tells us how high or low the sound is. The higher the sound, the higher its frequency will be, and therefore the more oscillations per second the wave will go through.

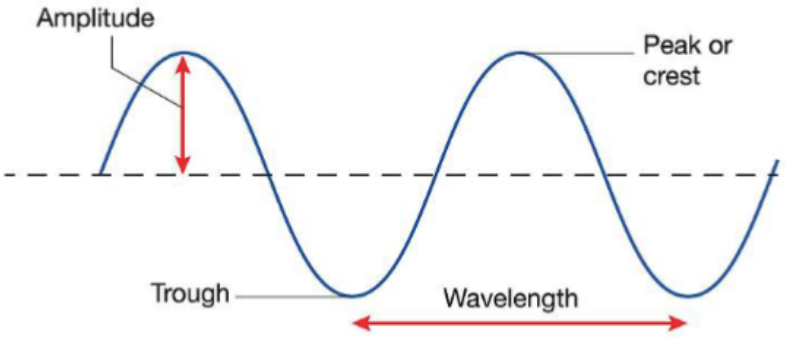


Figure 3.1: A wave can be characterized by its amplitude, frequency and wavelength. Retrieved from `https://shorturl.at/KtTHs`.

Now that we know what sound is, we can turn our attention to audio data, where audio refers to any recorded, transmitted or reproduced sound. Naturally, a sound wave is a continuous curve containing an infinite number of values over time. If we want to digitally represent this audio wave, it is clear we cannot digitally store an infinite amount of information about it. Instead, the sound wave is converted into a collection of discrete values, also known as a digital representation (HuggingFace, 2023). In fact, the different types of audio files formats, such as .mp3, .wav, etc., correspond to the way the digital representation of an audio wave is encoded.

In the following sections we will discuss two of the most important digital representations of audio that are used today: time and frequency domain representations.

3.2. Time-Domain representation

A time-domain representation refers to a way of representing signals as they change over time. The usual way of representing sound in the time domain is the waveform, where the amplitude of the signal is plotted against time. In digital systems, this waveform is commonly stored using pulse code modulation encoding (PCM), which captures the amplitude values at regular intervals.

3.2.1. Sampling and sampling rate

In order to capture the wave's information through time, we apply sampling – the process of measuring the value of a continous signal at a fixed and finite amount of time steps (see Figure 3.2).

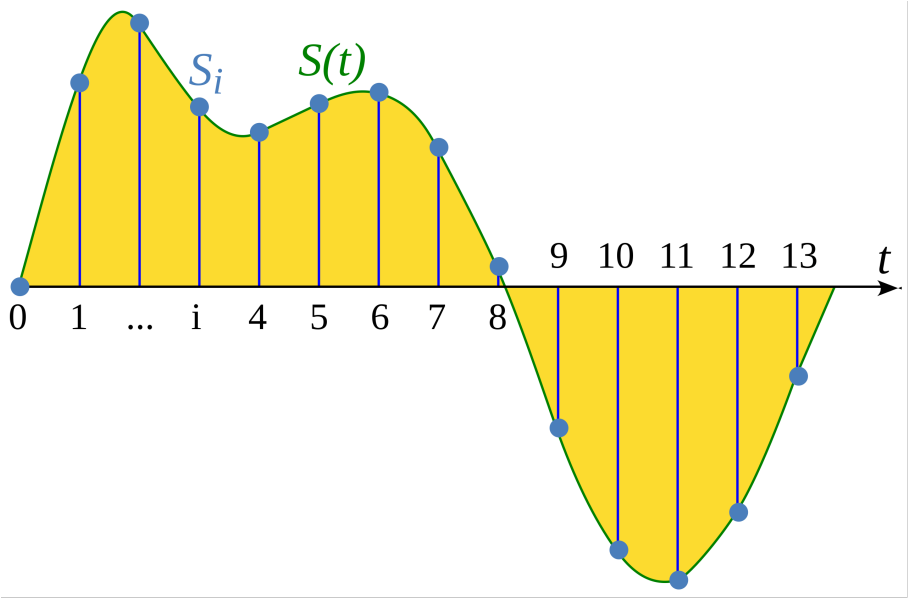


Figure 3.2: Example of sampling of an audio wave through time. Retrieved from (HuggingFace, 2023).

The sampling rate[1] is the number of samples taken in one second and is measured in Hz. For example, songs uploaded to Spotify are uploaded with a sample rate of 44.1 kHz, so each second of audio is represented by 44100 samples. For comparison, high-resolution audio has a sampling rate of 192 kHz.

The choice of the sampling rate determines the Nyquist frequency, which is the highest frequency that can be captured by the system and always equals half the sampling rate. For instance, if the sampling rate is 16kHz, then the highest frequency the audio representation will be able to model is 8 kHz.

3.2.2. Amplitude and Bit Depth

The sampling rate tells us how often samples are taken but, what exactly are we sampling? Each sampled value corresponds to the *amplitude* of the sound wave at a given instant in time. For sound waves traveling through air, the amplitude represents the deviation of air pressure from its ambient (resting) level — not the physical displacement we had previously introduced. This deviation is a physical quantity, usually measured in pascals (Pa).

Besides deciding *how often* we sample, there is another important aspect to take into account when converting a continous signal into a digital form: bit depth. This metric determines the resolution of the amplitude values: the number of distinct levels into which the continuous signal is quantized. Common audio formats use 16-bit or 24-bit depth, corresponding to 65,536 and 16,777,216 possible levels, respectively.

[1] A great resource for visualizing the sampling rate is `https://jvbalen.github.io/notes/waveform.html`.

Quantization introduces rounding error, which manifests as quantization noise. The higher the bit depth, the finer the resolution and the lower the resulting noise. In practice, 16-bit audio already provides noise levels below the threshold of human hearing, making it sufficient for most use cases (HuggingFace, 2023).

To better align with how we perceive sound intensity, amplitude values are often expressed on a logarithmic scale — in *decibels* (dB). In acoustics, this is referred to as the sound pressure level (SPL), defined as:

$$\text{SPL (dB)} = 20 \cdot \log_{10}\left(\frac{p}{p_{\text{ref}}}\right),$$

where:

- p is the measured sound pressure, in pascals (Pa),
- p_{ref} is the reference pressure, typically $20\,\mu$Pa.

It is worth noticing that although pressure is measured in pascals, SPL in decibels is a *dimensionless quantity*, since it represents a ratio of pressures on a logarithmic scale.

This logarithmic representation mirrors human perception: we are more sensitive to small changes at low intensities than at high ones. For example, an increase of about 6 dB is perceived as roughly twice as loud. In real-world audio, 0 dB SPL corresponds to the quietest sound the average human ear can hear, and louder sounds have positive dB values.

In digital audio systems, amplitude is typically expressed in decibels relative to full scale (dBFS). Here, 0 dBFS represents the amplitude of a full-scale digital signal, that is, the reference level against which all other amplitudes are measured. All other levels are negative, indicating lower amplitudes. As a rule of thumb, every decrease of 6 dB approximately halves the amplitude, and signals below -60 dBFS are generally considered inaudible.

3.2.3. Waveform

We are now in position to talk about waveforms, which are nothing but a plot of the sampled values of a sound over time illustrating the changes in its amplitude. This visualization comes in handy for identifying specific features of audios such as its overall loudness or individual sound events (see Figure 3.3[2]).

It is worth noting that the waveform from Figure 3.3 is not periodic (unlike the one in Figure 3.1), because the sound it represents is not a pure sinusoid. Instead, it may be a sum of multiple non-periodic sinusoids with different frequencies, or even a random signal like white noise.

[2]For consistency, the same 4 second clip from ABBA's *Lay all your love on me* will be used throughout the chapter.

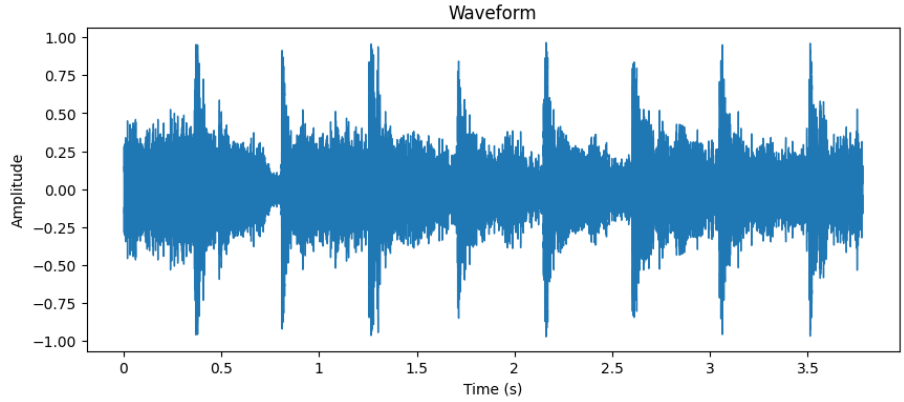


Figure 3.3: A waveform plot of a 4 second clip of ABBA's *Lay all your love on me*.

3.3. Frequency-domain representation

The frequency-domain representation provides a different way of representing sounds. While time-domain representation focuses on how a signal changes over time, frequency-domain analysis emphasizes the individual frequencies that make up the signal. This has a significant number of applications. For example, in speech separation, which aims to identify the different speakers in a conversation, different sound sources are separated based on their frequencies.

In this section, we will explore the key concepts in frequency-domain representation, starting with the frequency spectrum, then moving on to the Discrete Fourier Transform (DFT) and its applications, followed by an examination of spectrograms.

3.3.1. Fourier transform and the frequency spectrum

As we have already stated, the frequency-domain representation of an audio signal is a way to decompose it into a sum of pure periodic components, each corresponding to a specific frequency.

In order to compute a signal's individual frequencies, the Fourier transform is used. The foundation is Fourier's Theorem, which states that any periodic function can be expressed as a sum of sine and cosine functions (or equivalently, complex exponentials[3]) with specific amplitudes and phases. Here, the phase of each component describes its position within its cycle at a given time — that is, how much the wave is shifted horizontally relative to a reference. This mathematical transformation breaks down signals into simpler sine and cosine waves, each corresponding to a specific frequency (see Figure 3.4).

The Fourier transform of a continuous signal $x(t)$ is given by:

[3]By Euler's Identity, we have $e^{i\theta} = \cos\theta + i\sin\theta$.

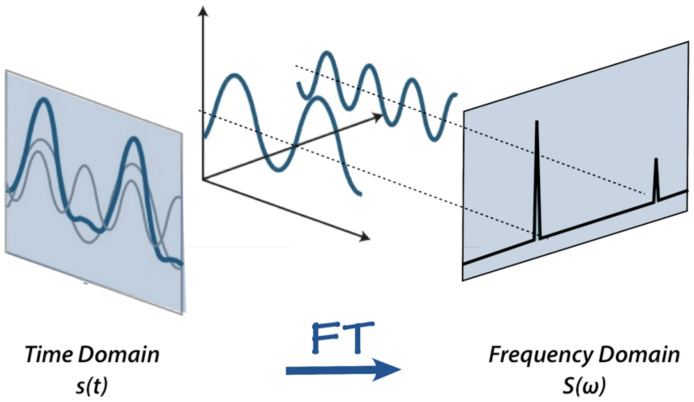


Figure 3.4: Effect of applying the Fourier transform to a signal. Retrieved from `https://shorturl.at/8D73u`.

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi ft}\, dt$$

where:

- $X(f)$ is the Fourier transform of the signal $x(t)$ that that tells us how much of each frequency f is present in the original time-domain signal.
- $x(t)$ is the original signal in the time domain.
- $e^{-i2\pi ft}$ represents oscillations at a frequency f. This expression combines sine and cosine components, and the negative sign indicates the oscillation direction.

The DFT is a version of the Fourier Transform specifically designed for discrete signals, which are composed of a finite number of samples. By applying the DFT, we can approximate the frequency content of a non-periodic, sampled signal. If we have a sequence of N samples $\{x_k\}_{k=0}^{N-1}$, the DFT transforms them into a series of complex numbers:

$$X_k = \sum_{n=0}^{N-1} x_k \cdot e^{-i2\pi \frac{k}{N} n} \tag{3.1}$$

Essentially, it takes the signal's discrete samples and computes a frequency representation over a finite duration. In fact, taking the modulus of the output of the DFT gives us the amplitude information at a given moment, while the angle between the real and imaginary components of the output provides the so-called phase spectrum.

But what if we want to see how the frequencies of an audio change over time? The solution to this question is to compute the audio's spectrogram, a very informative audio representation that allows us to jointly visualize time, frequency and amplitude, all in the same graph.

In order to compute a spectrogram, the Short Time Fourier Transform (STFT) is used. This algorithm (see Figure 3.5) first divides the signal into possibly overlapping and brief segments or windows (usually lasting a few milliseconds). Secondly, it applies the DFT to each of them in an efficient way with the Fast Fourier Transform (FFT) algorithm. Finally, all collected spectra are stacked through the time axis, forming the spectrogram. Thus, when looking at the resulting spectrogram (see Figure 3.7), each vertical stripe represents the frequency spectrum at a specific point in time, seen from the top.

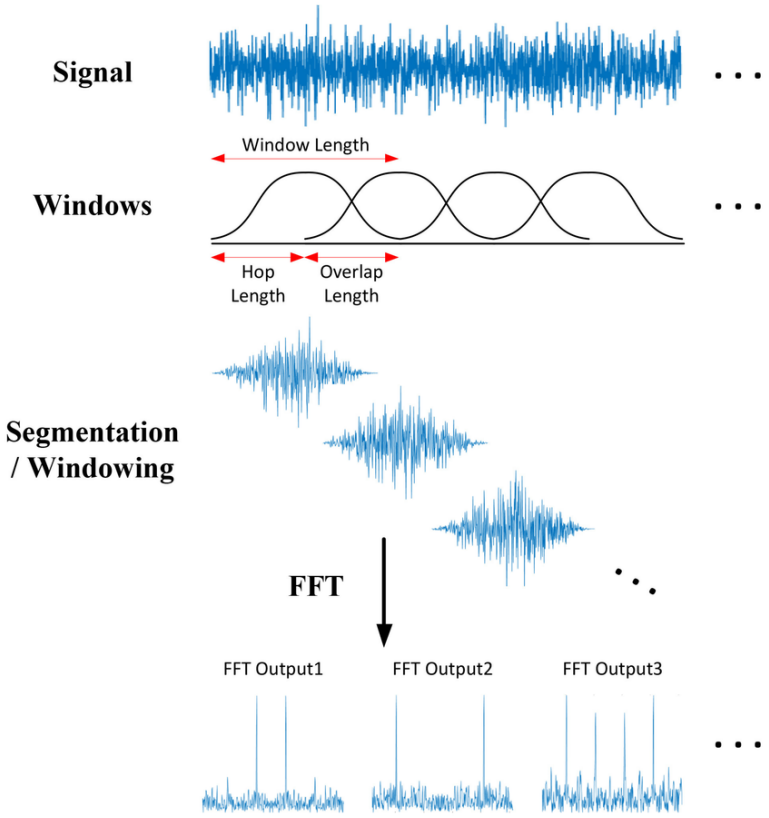


Figure 3.5: Short Time Fourier Transform diagram. Retrieved from `https://shorturl.at/JGODG`.

Note that overlapping windows are crucial in the STFT because they improve the chances of capturing short-lived events in the signal. Without this overlap, transient features can fall between two consecutive windows and be either poorly represented or entirely missed in the time-frequency analysis. We can think of it like scanning a text with a narrow spotlight: if we move the light in non-overlapping steps, some words might fall in the dark gaps and go unread.

An example of a spectrogram can be seen in Figure 3.6. In it, the reader might have the impression that the spectrogram is either incorrect or lacks meaningful

information. However, the observable "black void" does contain relevant data — it's just not visually apparent due to the use of a linear amplitude scale. By re-plotting the same spectrogram with a logarithmic (dB) scale, the previously hidden details in the darker areas become clearly visible in Figure 3.7.

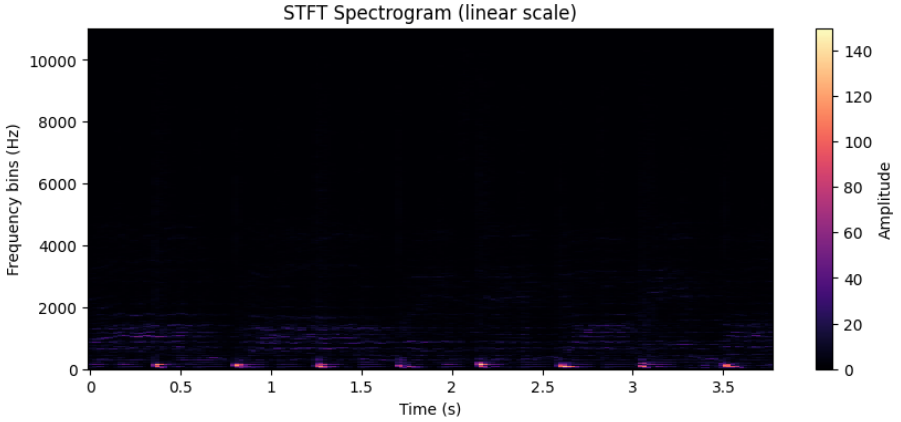


Figure 3.6: Resulting linear spectrogram after applying the STFT to a 4 second clip from ABBA's *Lay all your love on me*.

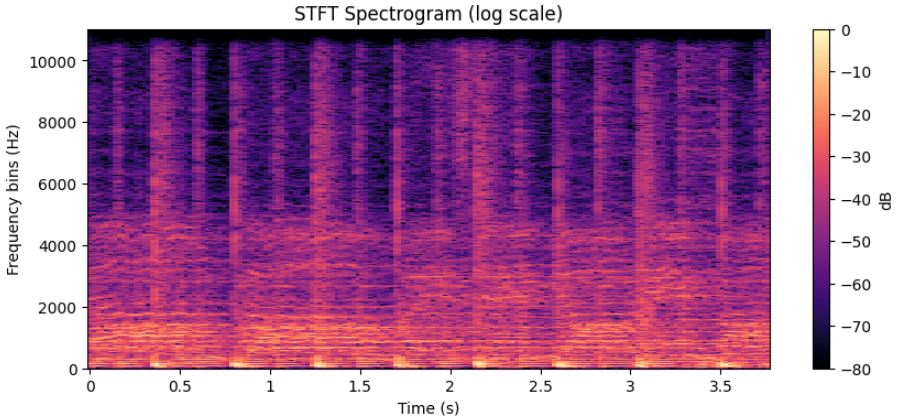


Figure 3.7: Resulting log spectrogram after applying the STFT to a 4 second clip from ABBA's *Lay all your love on me*.

An important observation about the STFT is that it is an invertible function, so it's possible to turn the spectrogram back into the original waveform. This means the spectrogram and the waveform are really just different views of the same data.

The discrete case inverse STFT can be computed as:

$$x_k = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{-i2\pi \frac{k}{N} n} \tag{3.2}$$

While the continuous STFT is theoretically invertible, in practice we work with sampled signals and apply a discrete STFT. This discrete version is also invertible,

but it can only reconstruct the discrete signal it was computed from — not the original continuous-time signal. Since sampling inherently introduces some information loss, the inverse STFT only recovers an approximation of the original waveform.

3.3.2. Mel spectrograms

A mel spectrogram (mel is short for "melody") is a kind of spectrogram characterized by changing the measurement of the frequency axis. In particular, while in a standard spectrogram the frequency axis is linear and is measured in Hz, a mel spectrogram applies a set of filters, also known as mel filterbank, to each spectrum, which transforms the frequencies to a logarithmic scale, commonly known as mel scale. The mel scale is non-linear and compresses higher frequencies while expanding lower ones.

But why might mel spectrograms be useful? Humans don't perceive changes in sound in a linear manner, but a logarithmic manner instead. This can be observed empirically[4] when listening at two pairs of sounds that differ in 50Hz but one pair has a much higher frequency value than the other: the change from 300 to 350 Hz is much more evident than the change from 8000 to 8050 Hz, which wouldn't happen if we were able to perceive these changes linearly. Thus the mel scale, and mel spectrograms, allow to model frequency-domain representations with a higher fidelity to how humans perceive sound, a fact that should probably be taken into consideration when training a neural network on an audio task.

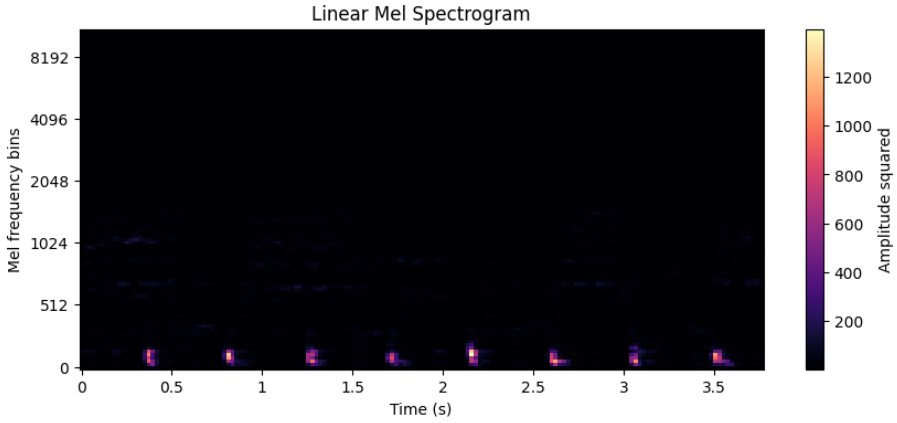


Figure 3.8: Linear mel spectrogram of a 4 second clip of ABBA's *Lay all your love on me*.

As with Figures 3.6 and 3.7, the dark region in Figure 3.8 should not be interpreted as a loss of information. Rather, it reflects how information is visually obscured when using a linear amplitude scale. Once again, converting the amplitude to decibels reveals these hidden details, making the structure of the spectrogram more perceptible.

[4]Tests can be done at `https://onlinetonegenerator.com/`

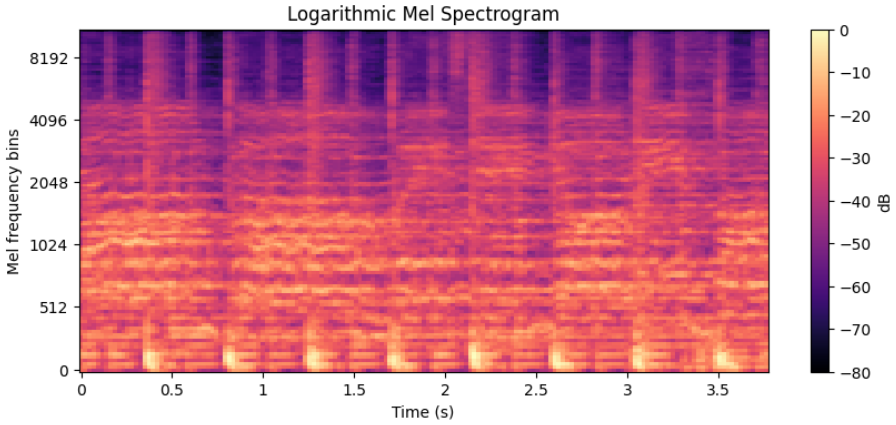


Figure 3.9: Logarithmic mel spectrogram of a 4 second clip of ABBA's *Lay all your love on me*.

3.4. Practical Considerations in Audio Representation

In this section, we briefly discuss the trade-offs involved in selecting Short-Time Fourier Transform (STFT) parameters for converting audio signals into spectrograms. An overview of the specific parameter values used by our model is provided in Chapter 5.

3.4.1. STFT parameters and signal reconstruction

The size and quality of the input spectrograms we feed to our model depends on the parameters with which we obtain said spectrograms (Manilow et al., 2020). It is therefore important that we understand the role each of these parameters play:

- Window type: determines the shape of the short-time window applied to each audio segment. Different window shapes control how the signal is weighted, especially at the edges, to minimize spectral leakage (the spreading of frequencies into neighboring ones). Windows with smoother edges, like the Hann window, reduce leakage but come with a trade-off between time resolution and frequency resolution. A sharper window (e.g., rectangular) has better time resolution but more leakage. In our models, we use the Hann window, as it strikes a good balance between leakage reduction and frequency clarity, making it a common choice for deep learning audio tasks.
- Window length: specifies the number of samples that each short-time window contains. The window length significantly influences the frequency resolution of the spectrogram: longer windows provide higher frequency resolution, while shorter windows provide better time resolution. The trade-off between time and frequency resolution is visible in Figure 3.11.

- Hop length: designates how many samples are skipped between two consecutive short-time windows. The shorter the hop length is, the more detailed the time axis will be (see Figure 3.12) and the larger the computational load will be. On the other hand, a longer hop length can result in a more compressed time axis, potentially causing a loss of temporal information.

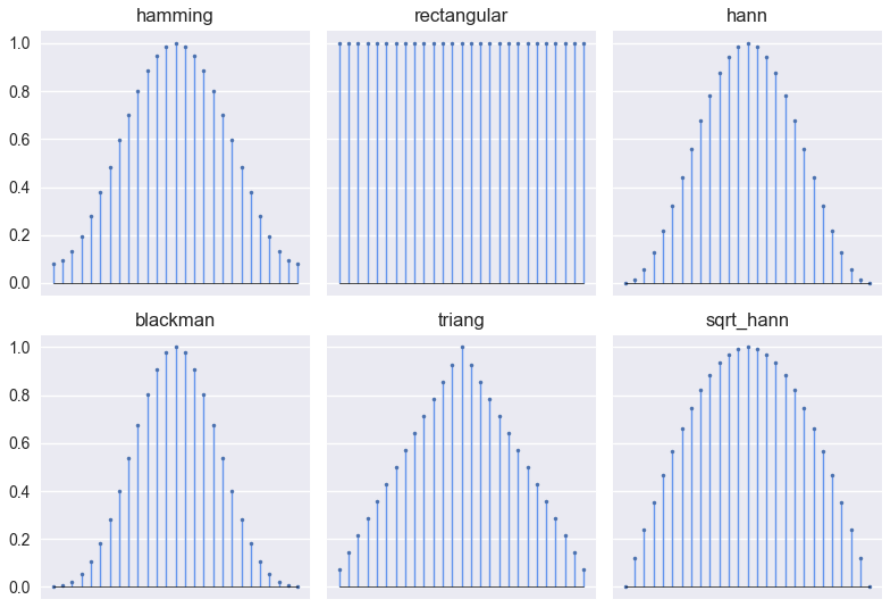


Figure 3.10: The choice of window function can significantly influence the spectral resolution and leakage effects in the spectrogram. Retrieved from `https://shorturl.at/UsEsq`.

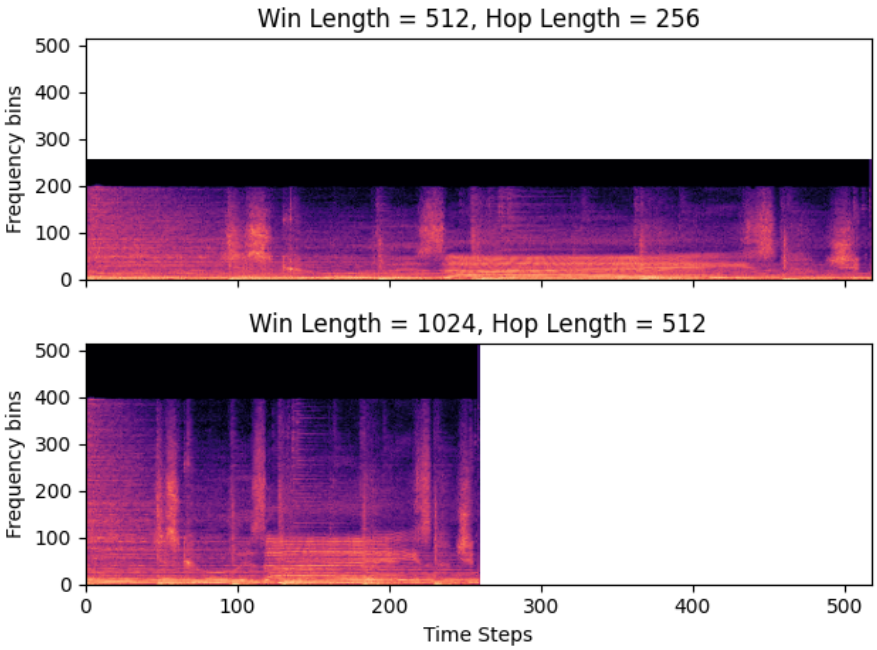


Figure 3.11: A shorter window results in finer time details but poorer frequency resolution, while a longer window captures more frequency details but at the cost of time resolution. Retrieved from `https://shorturl.at/UsEsq`.

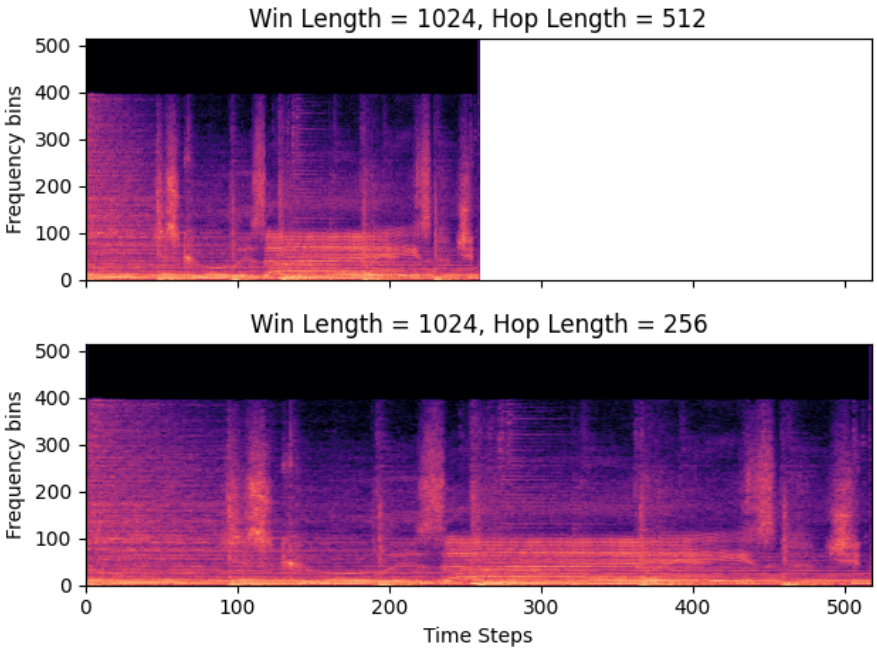


Figure 3.12: The smaller the hop length the more times a particular segment of the audio signal is represented in the STFT. Retrieved from `https://shorturl.at/UsEsq`.

Chapter 4

Introduction to deep learning and autoencoder-based models

This chapter will introduce the reader to a basic understanding of what deep learning is and its main components. The aim is not to go into detail but rather gain the necessary intuition to be able to grasp an end-to-end deep learning architecture. This will be useful for those who are introducing themselves in the topic to better comprehend our project and decisions within it.

Once we have done this, we will go a little further by explaining from both a broad and a detailed perspective the deep learning models and architectures we have used in this project: convolutional neural networks, autoencoders and variational autoencoders. For the sake of completeness, we will also briefly introduce conditional variational autoencoders.

4.1. Deep learning

Deep learning is a subset of machine learning that uses large neural networks to model complex patterns in data. A neural network consists of units called neurons, organized in layers: an input layer, one or more hidden layers, and an output layer. Each neuron applies a weighted sum of its inputs, adds a bias, and passes the result through a non-linear activation function.

A typical feed-forward pass through a single neuron can be expressed as:

$$z_j = \sum_i w_{ij}\, x_i + b_j, \quad a_j = \sigma(z_j), \tag{4.1}$$

where x_i denotes the inputs, w_{ij} are the weights connecting input i to neuron j, b_j is the bias term, z_j is the neuron's pre-activation, $\sigma(\cdot)$ is a non-linear activation function (such as ReLU, sigmoid, or tanh), and a_j is the neuron's output. Stacking many such neurons into multiple layers allows deep networks to learn hierarchical representations of data (Goodfellow et al., 2016).

If we want a model to learn about some data, we need to train it. Training a model involves finding the best weights and biases to minimize a loss function. This function measures how far off the model's predictions are from the actual targets. A common loss for regression problems is the Mean Squared Error (MSE):

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{4.2}$$

The optimization is usually done using gradient descent with the gradients of the weights in the network computed, using the backpropagation technique. During backpropagation, the chain rule is applied to propagate the error signal from the output layer back through the hidden layers, and by that means adjusting each weight to reduce the overall error in a direction guided by the negative gradient of the loss. Thanks to this, weights are updated iteratively using an optimizer like stochastic gradient descent (SGD) or Adam.

In order to train and evaluate a deep learning architecture we need a dataset from which to gather data. This data is usually divided into three parts: training, validation, and test sets. The training set is used to learn the model parameters, the validation set is used to tune hyperparameters and avoid overfitting (a situation where the model "memorizes" data instead of learning it), and the test set evaluates the model's generalization ability. There exist several ways to try to avoid overfitting, such as regularization or early stopping (Goodfellow et al., 2016).

4.2. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep learning models particularly well-suited for data with a grid-like topology, such as images (2D grids of pixels) or audio spectrograms (2D time-frequency grids). A CNN introduces two key concepts: local receptive fields and weight sharing. Rather than connecting every input unit to every neuron in the next layer, as in a fully-connected network, a convolutional layer uses a small filter, also known as *kernel*, that slides across the input to produce feature maps. This filter is a learnable matrix of weights applied to local regions of the input, detecting specific local patterns (e.g., edges, textures) wherever they might appear. The same set of filter weights is reused for every location in the input (convolution and weight sharing), which greatly reduces the number of parameters and makes the model more efficient (LeCun et al., 1998; Krizhevsky et al., 2012).

A typical CNN architecture consists of an input layer, followed by repeated stacks of convolutional layers, activation functions (like ReLU), and pooling layers. Pooling layers aggregate information in local neighborhoods (for example taking the maximum or average of that region), reducing the spatial dimensions of the feature map and attempting to capture the most important characteristics of the data. Repeated convolution added to pooling operations allow the network to extract increasingly

abstract features at deeper layers, while gradually reducing dimensionality. Ultimately, one or more fully connected layers consolidate the extracted features for the final prediction (LeCun et al., 1998; Krizhevsky et al., 2012). A nice visual of a typical CNN architecture can be seen in Figure 4.1.

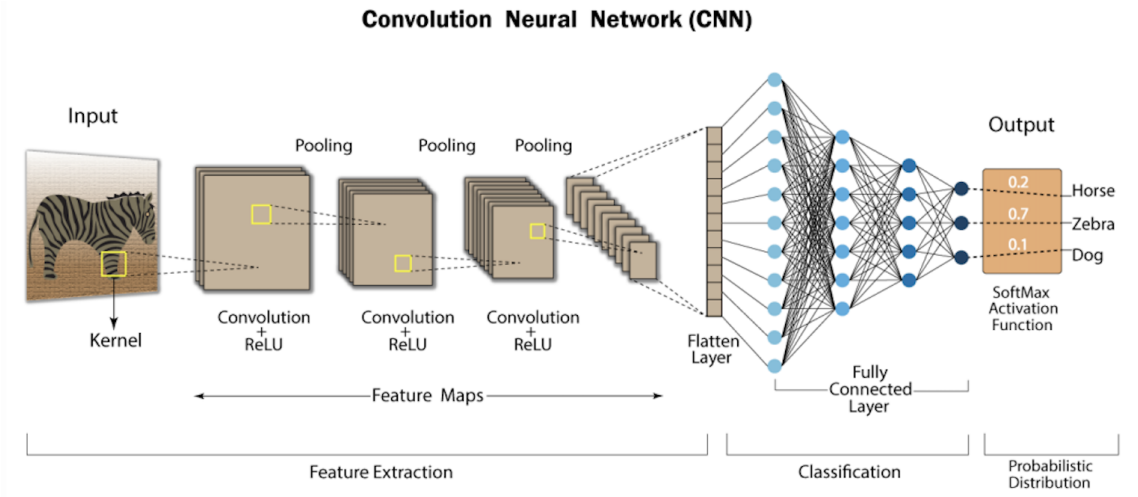


Figure 4.1: Typical CNN architecture. Retrieved from `https://shorturl.at/1uqcP`

CNNs have been extremely successful in computer vision tasks. Classic examples include LeCun's LeNet-5 for handwritten digit recognition (LeCun et al., 1998) and the AlexNet network that won the 2012 ImageNet competition (Krizhevsky et al., 2012). Both architectures demonstrated the power of deep CNNs on large-scale image data.

In our project, we used CNN layers as the main components of the autoencoder, a type of network we will talk about in a short time. In our case, CNNs were used to process spectrograms derived from the NSynth dataset (Engel et al., 2017). Spectrograms can be viewed as 2D representations of audio signals (time vs. frequency), and are therefore fit to convolutional operations.

Additionally, recent work has explored CNNs for interactive and explanatory purposes in various domains, including audio generation. For example, CNN Explainer (Wang et al., 2020)[1] demonstrates how convolutional kernels learn from image data, and similar principles could extend to audio, where convolutional layers automatically discover patterns corresponding to timbral or temporal events.

4.3. Autoencoders

An autoencoder is a type of neural network made up of two main components: an encoder and a decoder. The encoder compresses the input x into a typically lower-dimensional latent representation h, and the decoder reconstructs an output

[1]This is a great resource to closely understand how CNNs work.

$\hat{x}$ from this representation so that $\hat{x}$ closely matches the original input x (Michelucci, 2022; Bank et al., 2021). By minimizing a reconstruction loss between x and $\hat{x}$, the autoencoder is forced to learn the most salient features of the input. For example, a simple mean squared error (MSE) reconstruction loss is:

$$\mathcal{L}_{\text{AE}} = \|x - \hat{x}\|^2, \tag{4.3}$$

where $\|\cdot\|^2$ indicates element-wise squared difference.

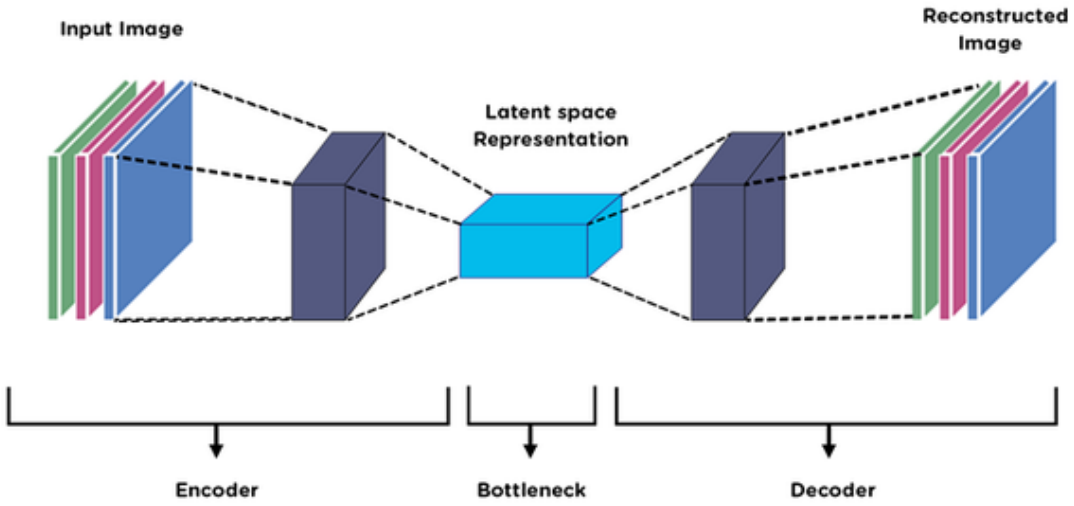


Figure 4.2: General structure of an autoencoder. The network consists of an encoder that compresses the input into a latent representation, and a decoder that reconstructs an output from it. Retrieved from `https://shorturl.at/esPtm`.

In the simplest form of autoencoder, both encoder and decoder are neural networks (often mirrored architectures) and h is a fixed-size vector (or tensor) of lower dimension than x. The hope is that h is an *informative* representation, meaning it captures the essential factors of variation in the data while discarding noise or irrelevant details. This learned latent space can then be useful for tasks like dimensionality reduction, visualization or anomaly detection (where high reconstruction error highlights anomalies).

There exists different types of autoencoders, and each of them serves a different functionality:

- **Denoising autoencoders** add noise to the input and train the model to reconstruct the original clean input, which encourages the network to learn robust features rather than simply memorizing the data (Michelucci, 2022).
- **Sparse autoencoders** impose a sparsity penalty on the latent representation, encouraging the network to use only a small number of active neurons for any given input. This often leads to the discovery of meaningful, disentangled features.

- **Convolutional autoencoders** apply convolutional layers in the encoder and decoder, which are especially effective for spatial or temporal data like images or spectrograms, since they preserve local structure. In our project, we use a convolutional autoencoder to learn compact representations of musical sounds through spectrograms.

In this work, we have implemented a convolutional autoencoder, meaning both encoder and decoder are made up of convolutional layers. To be more precise, the encoder is formed by convolutional layers which downsample data and bring it to the latent space, while the decoder is formed by so called transposed convolutional layers which upsample data from the latent space to its original dimensions. We would like to note that an autoencoder's output might not always naturally match the size of the autoencoder's input. Some sort of padding might be necessary for the dimensions to align up at the end. Therefore, it might be helpful for the reader to test with some examples how the output dimensions of convolutional and transposed convolutional layers are computed[2] and observe this phenomenon.

Ultimately, an autoencoder can learn an informative and compressed representation of data in an unsupervised manner. However, this deep learning architecture is not enough for the purposes of our thesis, since our aim is to be able to generate data from the learned distribution of samples. For this reason, we now introduce variational autoencoders.

4.4. Variational Autoencoders (VAEs)

Latent variable models are a common tool in probabilistic modeling, particularly when dealing with data that is assumed to be generated by hidden or unobservable factors. These latent variables z are not directly accessible, but they influence the observed data x. For example, in medical domains, health is often treated as a latent quantity inferred from observable tests, such as blood tests or pressure; in audio signals, latent variables could correspond to features such as pitch, rhythm, or timbre. We assume a generative process where z is sampled from a prior distribution $p(z)$, and then x is sampled from a likelihood $p_\theta(x|z)$. This leads to a joint distribution:

$$p_\theta(x, z) = p_\theta(x|z)p(z)$$

The marginal likelihood of the observed variable is then given by:

$$p_\theta(x) = \int p_\theta(x|z)p(z)dz$$

[2]PyTorch provides the respective output size formulas for convolutional and transposed convolutional layers in `https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html` and `https://docs.pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html` respectively.

In most realistic models, the integral in the expression above is not tractable. As a result, computing the exact posterior $p_\theta(z|x)$ is infeasible:

$$p_\theta(z|x) = \frac{p_\theta(x,z)}{p_\theta(x)} = \frac{p_\theta(x|z)p(z)}{\int p_\theta(x|z)p(z)dz}$$

To overcome this issue, variational inference introduces an approximate posterior $q_\phi(z|x)$ drawn from a simpler, tractable family of distributions (typically, a diagonal Gaussian). The problem then is to find a q_ϕ that is as close as possible to the true posterior. To measure this closeness, we use the Kullback-Leibler (KL) divergence, which quantifies how one probability distribution differs from another. More formally, the KL divergence between two distributions $q(z)$ and $p(z)$[3] is defined as:

$$\mathrm{KL}(q(z) \parallel p(z)) = \int q(z) \log \frac{q(z)}{p(z)}\, dz,$$

a quantity that is always non-negative and equals zero only when $q(z) = p(z)$ almost everywhere.

In the VAE's case, KL divergence expresses how much information is lost when using the approximate posterior $q_\phi(z|x)$ instead of the true posterior $p_\theta(z|x)$. The optimal approximate posterior is therefore obtained by minimizing:

$$q^*_\phi(z|x) = \arg\min_{q_\phi} \mathrm{KL}(q_\phi(z|x) \parallel p_\theta(z|x))$$

To analyze this objective more concretely, we fix a datapoint x and derive a decomposition of the KL divergence:

$$\begin{aligned}
\mathrm{KL}(q_\phi(z|x) \parallel p_\theta(z|x)) &= \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)}\, dz \\
&= \int q_\phi(z|x) \left(\log q_\phi(z|x) - \log p_\theta(z|x)\right) dz \\
&= \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] - \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(z|x)] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] - \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p(x,z)}{p(x)}\right] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] - \mathbb{E}_{q_\phi(z|x)}[\log p(x,z)] + \log p(x)
\end{aligned}$$

This gives us the identity:

$$\mathrm{KL}(q_\phi(z|x) \parallel p_\theta(z|x)) = \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x) - \log p(x,z)] + \log p(x)$$

Rearranging terms we obtain:

$$\mathbb{E}_{q_\phi(z|x)}[\log p(x,z) - \log q_\phi(z|x)] = \log p(x) - \mathrm{KL}(q_\phi(z|x) \parallel p_\theta(z|x))$$

[3]Note that $\mathrm{KL}(q(z) \parallel p(z)) \neq \mathrm{KL}(p(z) \parallel q(z))$.

Since the KL divergence is non-negative, the left-hand side forms a lower bound on the marginal log-likelihood. This quantity is known as the Evidence Lower Bound (ELBO) (Patacchiola, 2021):

$$\text{ELBO} = \mathbb{E}_{q_\phi(z|x)}[\log p(x,z) - \log q_\phi(z|x)] = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p(x,z)}{q_\phi(z|x)}\right]$$

We now unpack this expression further to obtain a more interpretable form:

$$\begin{aligned}
\text{ELBO} &= \mathbb{E}_{q_\phi(z|x)}[\log p(x,z) - \log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z) + \log p(z)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] + \mathbb{E}_{q_\phi(z|x)}[\log p(z)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \text{KL}(q_\phi(z|x) \parallel p(z))
\end{aligned}$$

This final form of the ELBO highlights two competing forces during training: the first term pushes the decoder to reconstruct the input well given the latent variable, while the second term encourages the approximate posterior to stay close to the prior.

We are now in position to introduce variational autoencoders.

A variational autoencoder (VAE) is a generative model that implements variational inference using neural networks. The encoder network outputs parameters of a diagonal Gaussian $q_\phi(z|x)$—that is, a mean $\mu(x)$ and diagonal covariance $\sigma(x)$—representing the distribution over the latent space. This is in contrast to a standard autoencoder, which instead outputs a reconstruction attempt of the input (Kingma and Welling, 2014).

In theory[4], when computing the two terms in the ELBO expression, we deal with two one-dimensional vectors of length D, i.e., $x, \hat{x} \in \mathbb{R}^D$. Let's derive a more specific version of the ELBO for our case. Recall our goal is to maximize ELBO:

$$\text{ELBO} = \mathbb{E}_{q_\phi}[\log p_\theta(x|z)] - \text{KL}(q_\phi(z|x) \| p(z)),$$

In practice, we maximize ELBO by minimizing $-$ELBO:

$$-\text{ELBO} = -\mathbb{E}_{q_\phi}[\log p_\theta(x|z)] + \text{KL}(q_\phi(z|x) \parallel p(z)).$$

Assuming a Gaussian decoder

$$p_\theta(x|z) = \mathcal{N}\big(x;\, \hat{x}(z),\, \sigma^2 I\big),$$

[4]In our PyTorch code, each example is a 4D tensor rather than a flat D-vector, but we collapse the last three dimensions into a single dimension when computing per-sample losses. This derivation therefore assumes 1D vectors for clarity, although it is algebraically equivalent to the actual multi-dimensional implementation.

where $\hat{x} := \hat{x}(z) \in \mathbb{R}^D$ denotes the decoder's output (i.e., its reconstruction of the original input x), given latent sample z and noting

$$\log p_\theta(x|z) = -\frac{1}{2\sigma^2}\sum_{i=1}^{D}\big(x_i - \hat{x}_i\big)^2 \; - \; \frac{D}{2}\,\ln\big(2\pi\sigma^2\big),$$

the reconstruction term becomes:

$$-\,\mathbb{E}_{q_\phi}[\log p_\theta(x|z)] = \mathbb{E}_{q_\phi}\left[\sum_{i=1}^{D}\frac{(x_i-\hat{x}_i)^2}{2\sigma^2} + \frac{D}{2}\ln(2\pi\sigma^2)\right].$$

Dropping the constant $\frac{D}{2}\ln(2\pi\sigma^2)$ (its gradient in the loss function will be zero, so it doesn't affect us in terms of minimization) the per-sample loss becomes

$$-\text{ELBO} = \frac{1}{2\sigma^2}\sum_{i=1}^{D}(x_i-\hat{x}_i)^2 + \text{KL}\big(q_\phi(z|x)\,\|\,p(z)\big).$$

By realizing

$$\sum_{i=1}^{D}(x_i-\hat{x}_i)^2 = \|x-\hat{x}\|_2^2 = D\,\text{MSE}(x,\hat{x}),$$

we equivalently write

$$-\text{ELBO} = \frac{1}{2\sigma^2}\,\|x-\hat{x}\|_2^2 + \text{KL}(q_\phi(z|x)\,\|\,p(z)) = \frac{D}{2\sigma^2}\,\text{MSE}(x,\hat{x}) + \text{KL}(q_\phi(z|x)\,\|\,p(z)).$$

Let's not forget about the KL term: how can we compute it? It turns out that the KL divergence between the approximate posterior $q_\phi(z|x)$ and the prior $p(z)$, two Gaussian distributions, is given by the following expression (Bernstein, 2023):

$$\text{KL} := KL\big(q_\phi(z|x)\,\|\,p(z)\big) = -\frac{1}{2}\sum_{j=1}^{L}\big(1+\log\sigma_j^2 - \mu_j^2 - \sigma_j^2\big),$$

where $\mu = (\mu_1,\ldots,\mu_L)$ and $\sigma = (\sigma_1,\ldots,\sigma_L)$ are the encoder's outputs specifying the mean and standard deviations of the approximate posterior $q_\phi(z|x) = \mathcal{N}(\mu,\,\text{diag}(\sigma^2))$, and L is the dimensionality of the latent space.

This way, we arrive at our first version of the VAE loss function:

$$\mathcal{L}(x,\hat{x}) = \frac{D}{2\sigma^2}\,\text{MSE}(x,\hat{x}) + \text{KL}$$

Although this objective works well on simple benchmarks such as MNIST, in the context of complex spectrogram data the full-strength KL term from the first epoch often drives the encoder to collapse the posterior onto the prior. This phenomenon is called "posterior collapse" and causes a degradation of generative diversity. To

prevent this, we introduce a time-dependent weight β on the KL term, annealing from 0 to 1 over a warm-up period. The resulting practical loss is

$$\mathcal{L}_\beta(x, \hat{x}) = \frac{D}{2\sigma^2} \mathrm{MSE}(x, \hat{x}) + \beta \, \mathrm{KL}$$

To make backpropagation through the stochastic sampling operation possible, VAEs employ the reparameterization trick (Kingma and Welling, 2014). Instead of sampling z directly, we sample $\epsilon \sim \mathcal{N}(0, I)$ and compute:

$$z = \mu(x) + \sigma(x) \odot \epsilon$$

where $\odot$ denotes element-wise multiplication. This formulation allows gradients to flow through μ and σ during optimization.

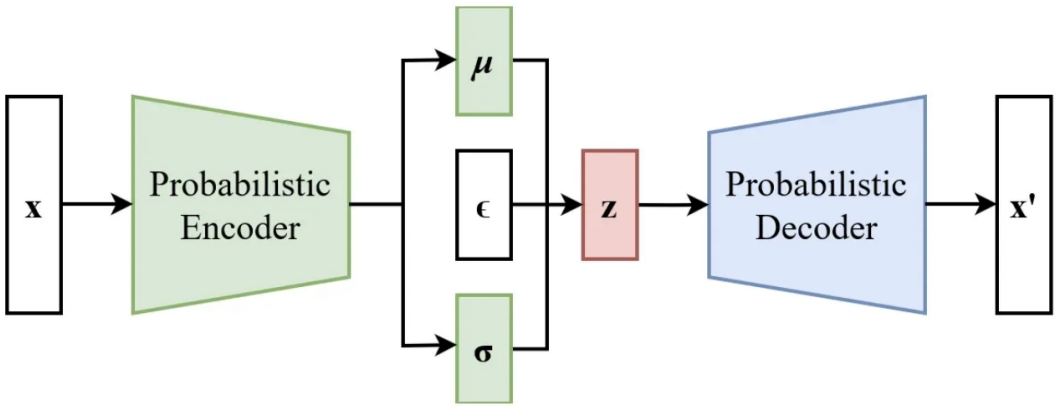


Figure 4.3: Variational autoencoder architecture (with reparameterization trick). The encoder (green) maps an input x to parameters $\mu(x)$ and $\sigma(x)$ of a Gaussian distribution $q_\phi(z|x)$ over the latent variable z. A latent sample z is drawn (by combining μ, σ with a random noise ϵ) and passed through the decoder (blue) to produce a reconstruction x'. Retrieved from `https://shorturl.at/uBd3M`.

Once trained, the decoder can generate new data by sampling from $z \sim p(z)$ (in our case from a $\mathcal{N}(0, I)$) and computing x' by passing the sampled value through the decoder. The KL regularization ensures that the latent space is well-behaved, so samples drawn from the prior typically result in realistic outputs. Moreover, interpolation between latent codes leads to smooth transitions in the data space. These properties make VAEs powerful tools for generative modeling in domains such as images, text, and audio (Michelucci, 2022).

4.5. Conditional Variational Autoencoders (CVAEs)

This section extends the discussion on variational autoencoders by briefly introducing conditional variational autoencoders (CVAEs), which, although not imple-

mented or evaluated in this work, provide a natural conceptual extension relevant to the task at hand.

A conditional variational autoencoder (CVAE) is an extension of the VAE that allows us to introduce conditioning information into the encoding and decoding process (Sohn et al., 2015). While the standard VAE is useful for modeling the underlying distribution of data, the CVAE provides additional flexibility by incorporating external information, allowing the generation process to be controlled or guided in a structured way. For example, in our case, we might be interested in generating musical notes of a specific instrument or pitch. The CVAE framework enables this by introducing a condition variable c, which is supplied to both the encoder and the decoder networks.

More concretely, the encoder models the conditional distribution $q_\phi(z|x,c)$, while the decoder models $p_\theta(x|z,c)$. The condition c can be any auxiliary information relevant to the data, such as a class label or one-hot vector. Providing c to the encoder allows the encoding of x to depend explicitly on the context, while passing c to the decoder allows the model to generate outputs that are consistent with that condition.

The training objective for a CVAE mirrors that of a VAE but includes conditioning on c throughout:

$$\mathcal{L}(x,c;\theta,\phi) = \mathbb{E}_{q_\phi(z|x,c)}\left[\log p_\theta(x|z,c)\right] - \mathrm{KL}(q_\phi(z|x,c) \parallel p(z|c))$$

In practice, the prior $p(z|c)$ is often chosen to be the standard normal distribution $\mathcal{N}(0,I)$ for all c, simplifying optimization while still enabling effective conditioning.

One of the main advantages of applying a CVAE in our use case would be that it uses label information to structure the latent space more efficiently. In a standard VAE, the model might dedicate part of the latent representation to implicitly encode categorical distinctions; for instance, separating pianos from violins if those instrument types lead to major differences in the input. The CVAE, however, is explicitly told what the instrument is via the condition c, which frees up the latent dimensions to capture other relevant variations such as articulation, dynamics, or timbre. This often leads to better use of the latent space and more targeted generation quality for each category (Sohn et al., 2015).

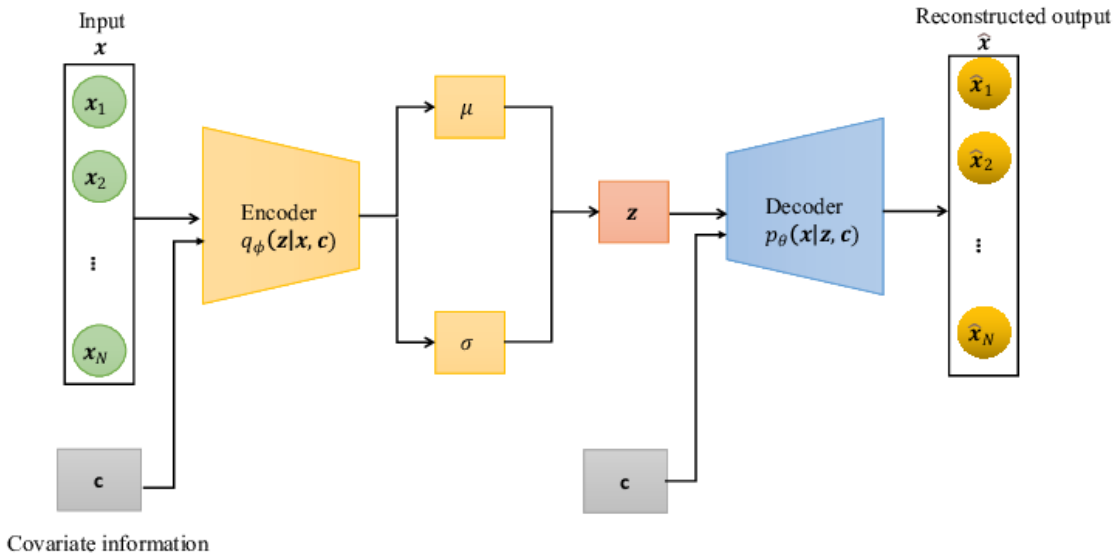


Figure 4.4: Conditional variational autoencoder architecture. The encoder (yellow) maps an input x and condition c to a latent distribution $q(z|x, c)$, and the decoder (blue) defines a conditional distribution $p(x|z, c)$ from which a sample x' can be drawn. Retrieved from `https://arxiv.org/abs/2211.02847`.

Chapter 5

Experiments and results

In this chapter we present the findings obtained from our experiments with the two deep learning architectures employed in this work: autoencoders and variational autoencoders.

Although our primary objective was to generate music notes using the latter, we consider it essential to also analyze and discuss the results achieved with the former. For each of the models, we dive into its design choices, training parameters and quantitative and/or qualitative results.

Before proceeding with the chapter, we would like to note that while some architectural decisions were informed by experimental results, others were more exploratory in nature, mainly drawing inspiration from existing architectures without extensive empirical justification.

5.1. Autoencoder

This section details the architecture, training setup, and results of the convolutional autoencoder used in our experiments. We begin by describing the model design and the reasoning behind key architectural choices. Next, we outline the training configuration and dataset preparation. Finally, we present both quantitative and qualitative evaluations of the model's performance.

5.1.1. Model design and training setup

The autoencoder model employed in this work is a convolutional autoencoder consisting of three hidden layers as well as flattening/unflattening and linear layers in both encoder and decoder. The choice of three layers was guided by a trade-off between empirical performance and computational efficiency. In preliminary experiments, deeper models offered only marginal improvements at significantly higher

computational cost.

Internally, each convolutional layer in the encoder consists (in this order) of the convolutional functional block and ReLU activation and BatchNorm layers. These last ones weren't originally considered but helped the model stabilize training and generalization. The number of channels in convolutional block was varied across different configurations; in our final setup, we used 16, 32 and 64 channels respectively in each of the encoder's convolutional layers and the same values, but mirrored, for the decoder's transposed convolutional layers. We did try to increment these values but found that by doing so the computational cost rose too much, making training not feasible in terms of execution time. Similarly, we experimented with different latent space dimensionalities and found that values in the range of 128 to 256 offered a good balance between compression, reconstruction fidelity and computation time. In the end, we fixed this parameter at a value of 200.

To connect the convolutional encoder to the latent vector representation, we introduced a flattening layer followed by a fully connected (linear) layer which allowed us to transform the final convolutional feature maps into a latent vector of dimension one, containing as many components as the latent space dimension we had trained the model with. Symmetrically, in the decoder, we used a linear layer followed by an unflattening operation to reshape the latent vector back into a suitable set of feature maps that could be processed by the transposed convolutional layers.

A visual of the autoencoder's high-level architecture can be seen in Figure 5.1.

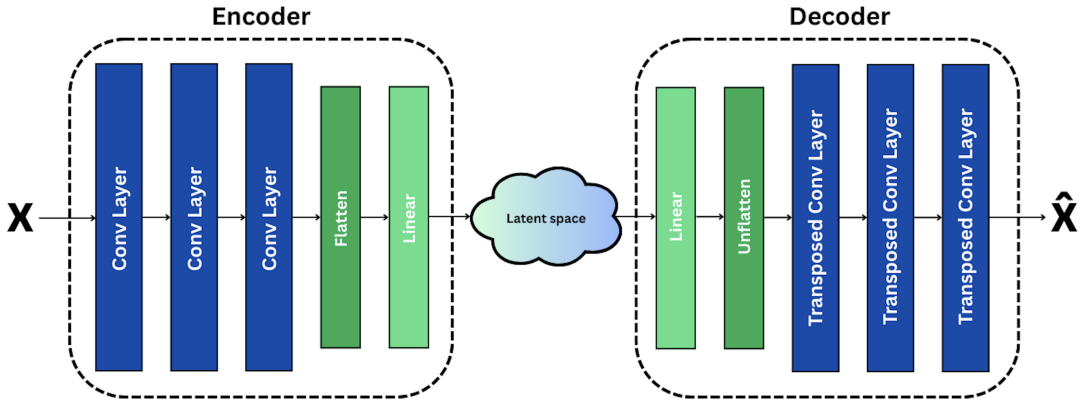


Figure 5.1: Autoencoder high-level architecture. On the left, $\mathbf{x}$ is a 4D tensor input compressed by the encoder onto the latent space and brought back by the decoder to a 4D tensor $\hat{\mathbf{x}}$ of the same shape as $\mathbf{x}$. Note the encoder's and decoder's symmetry.

The input to the model is a 4D tensor of shape `[B, C, H, W]`, where `B` is the batch size, `C` the number of channels, and `H` and `W` the height and width of the time-frequency representation of the audio. Initially, we used mel spectrograms as input — treating them as 2D grayscale images with a single channel, i.e., `C = 1`. However, this approach resulted in poor reconstructions, likely due to the loss of phase information inherent in mel spectrograms. To address this, we switched to a

richer representation by computing the STFT of the signal and explicitly separating its magnitude and phase components. These two components were then treated as separate input channels, yielding an input tensor of shape `[B, 2, H, W]`. Although reconstructions were still not perfect, this representation retained more of the original signal's structure and led to noticeably better performance compared to using mel spectrograms alone.

Having convolutional layers in our architecture, our encoder needed some form of downsampling. To this purpose, we compared the use of strided convolutions and pooling layers. We observed that using a stride greater than one in the convolutional layers yielded better audio quality in the reconstructions. Our interpretation is that this improvement stems from the learnability of the stride mechanism, in contrast to fixed pooling operations. Moreover, the additional computation introduced by strided convolutions was not prohibitively high.

In the decoder, we used transposed convolutions to upsample the latent representation. These layers employed a stride of 2 (being equal to the stride used for downsampling in the encoder) to progressively reconstruct the original input shape. To ensure the final output matched the original input dimensions[1], we applied zero-padding at the end of the decoding process. In Section 5.2, in which we dive into our observations and results from the VAE model, we explain how an unwanted generated audio characteristic might have been stemming from this type of padding and was therefore replaced by other kinds. However, in the autoencoder model, the reconstruction accuracy didn't seem to be downgraded by zero-padding enough for us to worry. We also explored using the `output_padding` parameter in the transposed convolutions to achieve precise output sizing. However, doing so imposed strict constraints on the input dimensions: only certain input sizes would result in valid outputs without violating `output_padding` restrictions. This made experimentation less flexible, as we would have had to carefully tailor the input dimensions to the architecture.

The loss function we utilized for the autoencoder is the Mean Squared Error (MSE), averaging the squared difference between the pixels over the input and output images. The Adam optimizer has been used for optimizing our autoencoder as it has become very popular for efficiency and adaptive learning rate.

We also used a `ReduceLROnPlateau`[2] learning rate scheduler that monitored the validation loss during training. If the validation loss increased after three epochs, the scheduler would reduce the learning rate by half to attempt to mitigate this issue.

Regarding overfitting, we initially had set the model to train for 50 epochs, but we found that the validation loss usually began to increase continuously around epoch 30 (see Figure 5.2). To work around this problem, we decided to limit the

[1]Recall that a decoder's output falls short, in general, with respect to the encoder's input, making size readjustment usually a necessity.

[2]Unlike other schedulers that adjust the learning rate based on a fixed schedule or after a set number of epochs, `ReduceLROnPlateau` adjusts the learning rate only when the validation loss plateaus.

training to not more than 30 epochs.

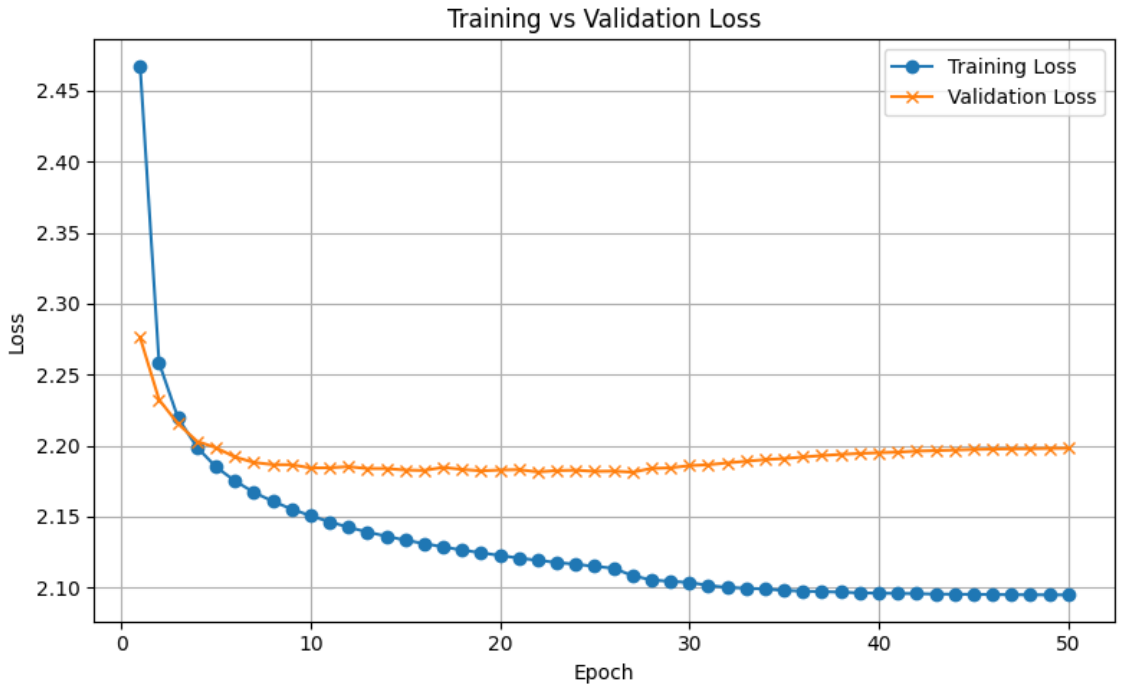


Figure 5.2: After epoch 30, the validation loss kept increasing, likely indicating model overfitting.

We also experimented with data normalization at two different stages: directly on the raw waveform and on the STFT spectrogram. In both cases, we observed a drop in performance. Normalizing the waveform aimed to standardize the input amplitudes early on, while normalization at the spectrogram level aimed to make use of the structured frequency-time representation to try to achieve a more consistent scaling. However, neither approach led to improved results; in fact, models trained on normalized data exhibited significantly worse accuracy, with some test performances dropping as low as 1.42% (the metric used is explained in Subsection 5.1.2).

The training was performed on CUDA-enabled GPUs, which significantly accelerated the computations and enabled us to handle large batches and complex calculations more efficiently. Specifically, the machine in which we trained the model had the following characteristics: a 12^{th} Gen Intel Core i7-12700K processor (3.60 GHz), 64.0 GB of RAM, and an NVIDIA RTX 3090 GPU (CUDA-enabled).

The training parameters that yielded the best testing accuracy can be found in Table 5.1 from Section 5.1.2

5.1.2. Quantitative evaluation

In this subsection, we present the evaluation of the autoencoder's performance using a specific metric applied to the test set, which we will first describe. Follow-

ing that, we will include a table that displays the testing accuracies for different configurations of model parameters.

The following per-sample accuracy formula was applied to the test set:

$$\text{accuracy}_i = \max\left(0, 1 - \frac{\|x_i - \hat{x}_i\|_2^2}{\|x_i\|_2^2 + \varepsilon}\right) \times 100\% \tag{5.1}$$

where:

- x_i is the input for sample i,
- $\hat{x}_i$ is the reconstructed output from the model,
- $\|\cdot\|_2^2$ denotes the squared L2 norm,
- ε is a small constant added to prevent division by zero.

The denominator $\|x_i\|_2^2 + \varepsilon$ plays a crucial role in normalizing the reconstruction error relative to the energy of the original input. This normalization makes the metric scale-invariant, meaning that inputs with different magnitudes are evaluated fairly.

Observe that when $x_i = \hat{x}_i$, the reconstruction is perfect, and the accuracy for that sample reaches 100%. On the other hand, when the input and output are significantly different — that is, when the reconstruction error $\|x_i - \hat{x}_i\|_2^2$ is larger than the normalization term $\|x_i\|_2^2$ — the fraction in the formula can exceed 1. In such cases, the resulting accuracy becomes negative. To prevent this and keep the interpretation consistent (as a percentage between 0% and 100%), we clip negative accuracies and define the minimum accuracy as 0.

Once the per-sample accuracy formula is defined, we can simply define the total accuracy as the mean over all accuracies:

$$\text{accuracy} = \frac{1}{N}\sum_{i=1}^{N} \text{accuracy}_i \tag{5.2}$$

where N is the total number of samples evaluated.

A comparison of testing accuracies based on different model configurations (all of them with a learning rate of 0.0001) can be observed in Table 5.1, in which we can observe that the model performed better when increasing the input size.

Two noteworthy implementation details emerged during experimentation. The first relates to PyTorch's `DataLoader` and its `num_workers` argument. When set to 8, training times increased drastically: up to 40 minutes per epoch. Upon investigation, this slowdown appeared to come from the overhead of worker processes creation. Setting `num_workers` to 0 resolved the issue and restored reasonable training speeds.

Row	T loss	V loss	Epc	T	FFT	Hop	Ht	Wd	Lat.	Acc.
1	2.2486	2.5183	30	14.20	800	100	800	633	128	47.99%
2	2.3486	2.6483	20	11.02	1000	100	1000	631	128	52.56%
3	1.8873	2.4540	20	10.56	1500	200	1500	313	128	62.79%
4	2.4951	3.2069	20	10.09	2048	350	2048	178	128	64.50%
5	2.3375	3.0431	20	10.44	2048	350	2048	178	200	65.57%
6	2.4041	3.5994	20	11.91	3000	300	3000	204	200	69.07%
7	2.3468	4.0879	20	13.27	4000	300	4000	201	200	71.18%

Table 5.1: Relevant training and transformation parameters and corresponding testing accuracy. The columns represent the following: T loss: Training loss, V loss: Validation loss, Epc: Number of epochs, T: Average time per epoch in hours, lr: Learning rate, FFT: FFT size, Hop: Hop length, Ht: Input height, Wd: Input width, Lat.: Latent dimension, Acc.: Testing accuracy.

The second observation concerns the use of batch normalization. Prior to introducing BatchNorm layers, the autoencoder achieved a test accuracy of approximately 51%. After incorporating BatchNorm and training on larger input sizes, performance improved significantly, with testing accuracy reaching up to roughly 71%.

5.1.3. Qualitative evaluation

Listening to the reconstructed audio samples revealed one consistent artifact worth noting: many reconstructions, particularly those corresponding to low frequency inputs, tend to contain noticeable background noise, while higher-pitched sounds tended to be reconstructed with greater clarity and less audible noise. A possible explanation for this behavior is that these higher frequency sounds are distributed across a broader set of STFT frequency bins and can therefore be detected better by the model.

Perhaps a more insightful evaluation of the model's performance can be gained by examining the actual audio reconstructions. Unfortunately, we can not directly do so on this document. However, we have provided several examples, which can be found in the project's Github repository[3]. Each folder number corresponds to the row number in Table 5.1 and contains:

1. A subdirectory named `examples` that includes pairs of original and reconstructed audio files. Each `Xr.wav` file represents the reconstruction of the original file `X.wav`.
2. A plot file `losses.png` showing the evolution of training and validation losses over epochs.
3. A `configs.txt` file containing the model's configuration parameters in JSON format.

[3] `https://github.com/pabgarcialopez/CS-thesis/tree/main/examples/AE`

Unfortunately, Github does not allow to upload files as large as the `.pth` files representing the trained models. However, the reader can replicate any corresponding model by using the configuration parameters present in each `configs.txt`.

5.2. Variational Autoencoders

In this section we focus on the specific adaptations and empirical findings for our Variational Autoencoder (VAE) on NSynth spectrograms. Architectural details shared with the convolutional autoencoder are omitted; instead we highlight only the key modifications, training strategies, and qualitative analyses unique to the VAE.

5.2.1. Architectural modifications

The VAE inherits the three-block convolutional encoder and decoder from Section 5.1, but with three essential changes. First, the encoder's final layers now output both a mean vector $\mu(x) \in \mathbb{R}^L$ and a log-variance vector $\log \sigma^2(x) \in \mathbb{R}^L$, rather than a single deterministic embedding. Second, in the decoder we remove the final `ReLU` activation so that its output can take negative values, which is crucial for faithfully reconstructing the phase channel, since it naturally ranges over $[-\pi, \pi]$. Third, the input sizes of the spectrograms were reduced with respect to those from the autoencoder model. This was necessary due to the increased computation time we found in the VAE model. Particularly, the input size we mostly used for VAEs was 1500 pixels high and 251 pixels wide. All other model parameters, such as layer widths, strides, or padding strategies remain identical to the autoencoder design.

5.2.2. KL–Annealing strategy

We first tried a naïve weighted sum of reconstruction and regularization given by

$$\mathcal{L} = \alpha \,\mathrm{MSE} + (1 - \alpha)\,\mathrm{KL},$$

Nonetheless, this approach lacked a variational interpretation and led to poor generative samples. We therefore abandoned this approach and shifted towards the losses expressions derived in Section 4.4.

Training under the unmodified ELBO caused the KL term to drop to zero almost immediately (Figure 5.3), a classic sign of posterior collapse that effectively reduces the VAE to a deterministic autoencoder.

As explained in Section 4.4, the standard way to prevent this posterior collapse is to apply a time-dependent weight β to the KL term. Early in training we have $\beta \approx 0$ so that the model may learn accurate reconstructions; over a warm-up of 8 epochs we slowly ramp β towards 1.

To quantify how different warmup schedules impact both reconstruction and sample quality, we trained eight VAE variants with warmup lengths of eight or ten epochs, baselines of 0 or 0.1, and cosine, linear, logistic, or cyclical shapes. Across these runs we found that purely cosine ramps often produced audible background hiss, while linear ramps, though mostly free of clicks, sometimes failed to build sufficient KL pressure, leading to overly deterministic outputs. Logistic schedules with a small baseline overcame these issues: the variant with an eight-epoch logistic ramp and a 0.1 baseline yielded the cleanest reconstructions, avoided posterior collapse, and hardly ever was clicking observed. A final cyclical experiment confirmed that restarting the ramp can reintroduce mild clicks, so we settled on the single-pass logistic warmup for our main results.

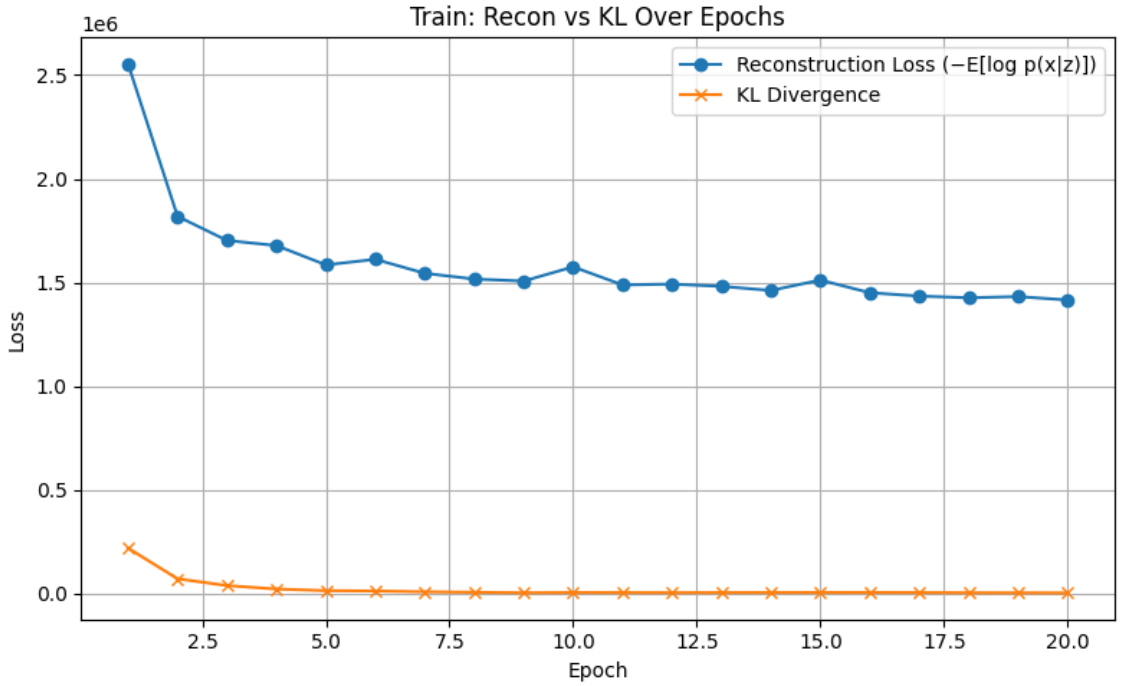


Figure 5.3: Evolution of the KL divergence term during VAE training with the standard ELBO: the KL rapidly collapses to zero, indicating posterior collapse and effectively turning the model into a deterministic autoencoder.

5.2.3. Padding and click artifacts

When generating audio samples from the latent space, we observed they exhibited a persistent clicking sound that appeared at seemingly regular intervals towards the end of the signal, and probably all through it (see Figure 5.4). Upon closer inspection, these clicks seemed to be spaced exactly by the hop length used in our STFT-based synthesis pipeline. Let's remember that if the hop length is H samples (i.e. a frame advance of $\Delta t = H/f_s$ seconds at sampling rate f_s), then each analysis window overlaps with its neighbor by $N - H$ samples, where N is the transform size. Any slight mismatch or residual at the frame boundaries during the overlap process

will probably be translated as some impulse every H samples. This would explain why the clicking noise is strictly equidistant, rather than random or confined to the end of the audio.

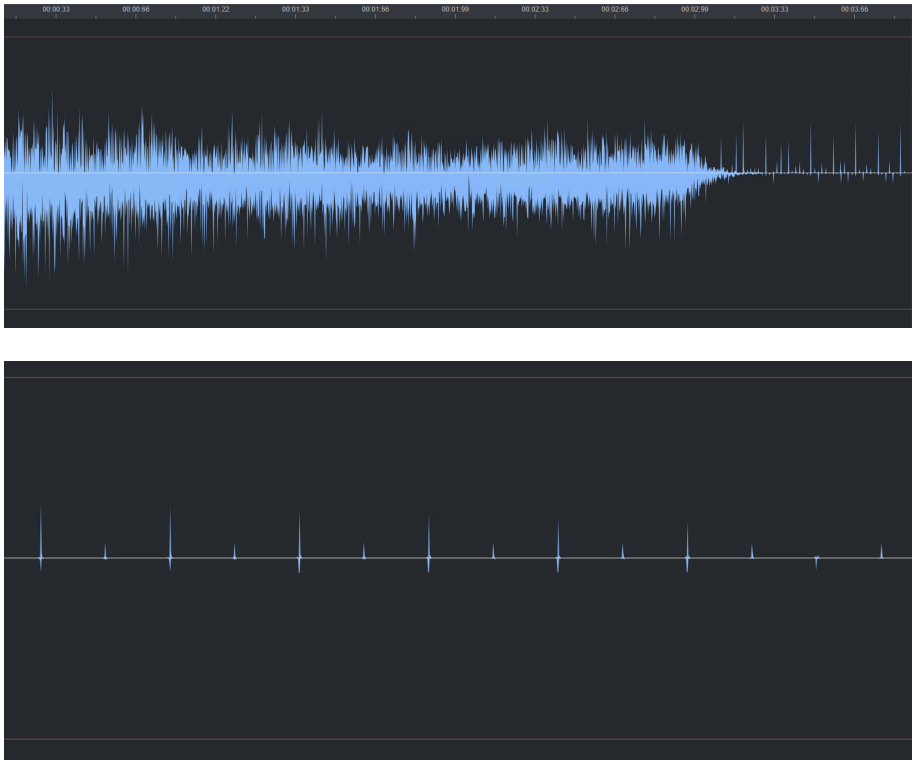


Figure 5.4: Top: full-length generated waveform exhibiting occasional impulses throughout the signal. Bottom: zoomed-in view showing the regular click train spaced by the STFT hop length.

Originally, we suspected that zero padding of the decoder's spectrogram output was responsible. In our first implementation, immediately after the VAE produced its reconstructed magnitude frames, but before applying the inverse STFT, we padded the spectrogram with zeros in time to match the model's input size.

Intuitively, inserting zero-valued columns at the end of the spectrogram introduces a hard discontinuity that can manifest as a click when mapped back to the time domain via ISTFT. To mitigate this, we experimented with two alternative padding schemes applied immediately after the VAE's decoder, but once again before the inverse STFT. In "reflexive" padding, if T frames were missing, we appended the mirror image of the last T columns of the decoder output, which using our intuition, should preserve smoothness. In "hold" padding, we instead replicated the final column T times, thereby avoiding any abrupt jump to zero. Despite these changes, the clicking artifact remained both in timing and amplitude identical to the zero-padded case. Furthermore, because padding affects only the final few frames, it cannot explain an impulse at every hop interval throughout the entire signal. Indeed, true zero padding in the spectrogram would produce at most a single click near the end of the waveform, not a regular click train spaced by the hop length.

Therefore, it seems plausible that the regular clicks arise more from imperfect overlap reconstruction under our chosen hop length and windowing than from any padding scheme. Further testing on window–hop pairs is needed to come closer to the true cause of this issue.

5.2.4. Generative sampling on Fashion-MNIST

As a sanity check, we trained the same[4] VAE architecture on Fashion-MNIST (Xiao et al., 2017), which contains 70000 grayscale 28x28 images of ten garment categories: T-shirts, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags and ankle boots.

Results varied depending on whether KL annealing was used: training with no annealing yielded the best latent space sampling, while using the linear, cosine and logistic annealing shapes translated to a poorer latent space. This can be observed respectively in Figures 5.5 and Figure 5.6. The reason why KL annealing wasn't beneficial here probably has to do, as commented in Section 4.4, with the simplicity of the dataset at hand.

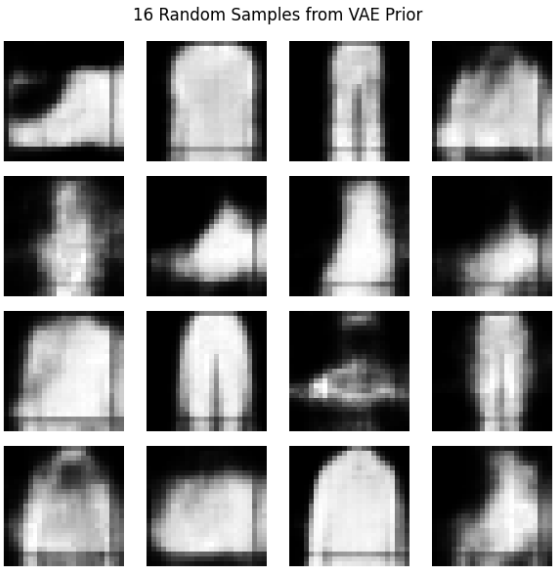


Figure 5.5: Randomly sampled reconstructions from our VAE trained on the Fashion-MNIST clothing dataset. Each image was obtained by drawing $z \sim \mathcal{N}(0, I)$ and passing it through the decoder. No KL annealing was perfomed in this case.

[4]Just changing the number of channels from 2 to 1 and adding a final `Sigmoid` layer in the decoder to make the output lie in the interval $[0, 1]$, since in the case of the Fashion-MNIST dataset, we deal with greyscale pixels with values in this range.

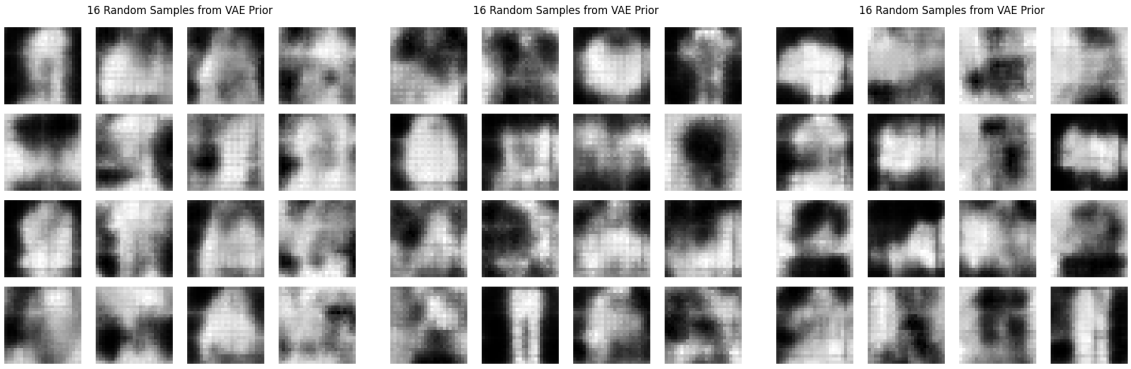


Figure 5.6: Fashion-MNIST VAE samples under three KL-annealing schedules. From left to right: linear, cosine and logistic annealing.

5.2.5. Latent-Space PCA analysis

To probe how well the VAE organized NSynth data, we encoded 1000 test spectrograms, extracted their latent means $\mu \in \mathbb{R}^{200}$, and applied PCA to reduce them to two dimensions. PCA is a dimensionality reduction technique that projects high-dimensional data into a lower-dimensional subspace, preserving as much variance as possible in the process.

In an ideally disentangled model, one would observe distinct clusters per instrument class. In our 2-D PCA projection (Figure 5.7), however, the encoded means mostly collapse into a dense cloud with only very slight instrument-specific separation. That said, reducing a 200-dimensional latent space down to two principal components can certainly mask subtler structures that might only be visible when using three, four or higher dimensions. Still, the fact that our randomly sampled outputs exhibit low diversity and tend to sound like minor variations of the same timbre supports the view that the VAE has not yet learned a richly organized latent space.

In conclusion, while our VAE did learn to some extent to generate a meaningful latent space, there is still a wide margin of improvement here. Some ideas for the model's refinement are proposed in Chapter 6, where some future work is outlined.

5.2.6. Experiment artifacts

All files produced by each VAE run are available in the project's GitHub repository[5]. Each experiment folder contains:

- `configs.txt`: the JSON configuration for that run.
- `examples/`: five audio clips sampled from the learned latent space.

[5] `https://github.com/pabgarcialopez/CS-thesis/tree/main/examples/VAE`

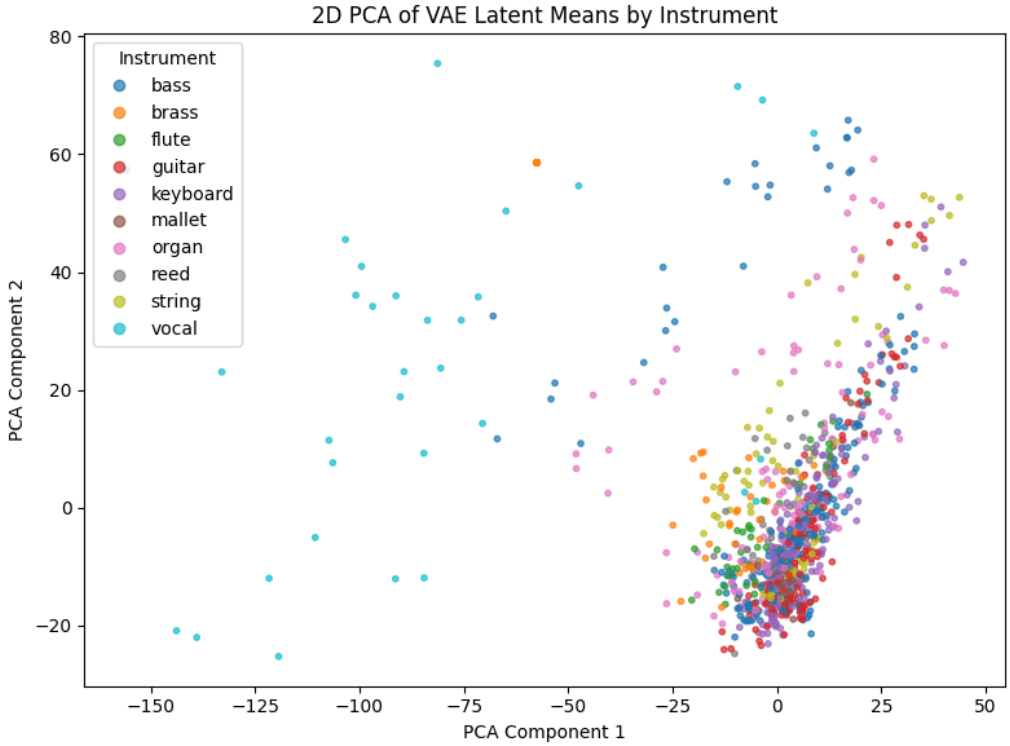


Figure 5.7: Two-dimensional PCA projection of 1000 latent means μ extracted from test samples. Colors represent different instrument classes. While some instrument-specific structure is visible, the overlap between clusters suggests limited disentanglement.

- `losses.png`: training vs. validation loss curves.
- `PCA.png`: two-dimensional PCA projection of latent means.
- `term_losses.png`: reconstruction (MSE) vs. KL divergence over epochs.
- `beta_vs_kl.png` (when present): the annealed β schedule plotted against the KL term.

As it happened in the case of AEs, the `.pth` files corresponding to the trained models are too big for Github's file system, so we see ourselves forced to leave them out. However, with the configurations present in each example, the reader can easily recreate the training conditions necessary for each model.

Chapter 6

Conclusions and future work

6.1. Conclusions

In this thesis, we have explored musical timbre reconstruction using convolutional autoencoders and the generative synthesis of novel timbres via variational autoencoders, both operating on STFT-based spectrogram representations.

The convolutional autoencoder demonstrated that a three layer encoder–decoder, using strided convolutions, batch normalization, and STFT magnitude plus phase as a two-channel input, can compress 4000-frame spectrograms into a 200-dimensional latent space and reconstruct them with up to 71.18% per-sample accuracy. Limiting training to 30 epochs, applying a ReduceLROnPlateau scheduler, and zero-padding only at the final layer prevented overfitting while preserving audio fidelity. We believe these design choices represent a good balance of reconstruction quality and computational cost.

Building on this backbone, our variational autoencoder introduced three key modifications: an encoder that outputs both mean and log-variance vectors, removal of the final ReLU to allow negative phase values and slightly reduced input dimensions to fit GPU memory. Without β-annealing, the KL term collapsed to zero, reverting the model to a deterministic autoencoder. Of the annealing strategies tested, an eight-epoch logistic ramp (with a 0.1 baseline) offered the best trade-off: it prevented posterior collapse, produced clean and click-free audio samples, and still encouraged latent variability. Simpler cosine or linear ramps either introduced noise or weakened the KL penalty, while a cyclical restart re-invoked clicking artifacts. That said, we can't forget that reducing a 200-dimensional latent space to two components may obscure a richer structure.

Moreover, generated sounds exhibited a regular clicking artifact aligned with the STFT hop length, perhaps suggesting overlap-add imperfections, though the exact cause remains undiagnosed. On the simpler Fashion-MNIST task, any form of annealing degraded sample clarity, underscoring that β-schedules must be tailored

to dataset complexity. Overall, our experiments confirm that STFT-based inputs and careful architectural tweaks enable both faithful reconstruction and generative sampling, yet significant work remains to arrive at a truly disentangled, diverse latent space and to eliminate synthesis artifacts.

6.2. Future work

A natural next step is a systematic hyperparameter and architecture sweep, exploring latent dimensionality, channel widths, network depth, learning rates, and β-warmup schedules. One could use automated search methods such as Bayesian optimization, in which a simple model of the performance metric is built and smart decisions on which hyperparameters to tweak are made.

An immediate priority is to eliminate the encoder–decoder dimension mismatch so that no post-decoder padding is required. To the best of the author's knowledge, there is no universally adopted solution to this problem in convolutional autoencoders—it remains surprisingly under-addressed in the literature. If padding cannot be avoided, it becomes essential to characterize precisely how each padding scheme (zero, reflect, hold, etc.) alters the reconstructed spectrogram's frequency content and timbral qualities. Another worth considering approach to fill the incomplete spectrogram would be performing some kind of interpolation from it.

A second avenue for future work is investigation into the source of the persistent clicking artifact. While our experiments suggest that padding discontinuities alone cannot explain the regular click train, other factors, such as imperfect overlap-add reconstruction under specific window–hop settings—may be at fault. Experiments that vary hop length, window function, and overlap type (e.g., Hann vs. Hamming) should clarify whether clicks arise from residual boundary mismatch or algorithmic artifacts in the inverse STFT. Understanding the root cause would bring both AE and VAE models to perform better.

Another path one could take from this work is, once the VAE produces cleaner and more varied audio, extending it to a conditional VAE that enables controlled synthesis. For instance, one could condition on instrument class or pitch, and even build a user interface where the specific condition for audio generation could be selected.

Beyond mean-squared metrics and PCA, introducing perceptually motivated audio-quality measures would be a very interesting and useful step as well. As of now, VAEs were only judged based on our biased perception of how good the sampled sounds are. It is our understanding that metrics such as the Fréchet Audio Distance (Kilgour et al., 2019), which compares statistical summaries of features extracted from real and generated audio, can provide audio-specific feedback on how realistic the sampled sounds are.

Finally, evaluating these architectures on alternative or narrower datasets, such as single-instrument recordings or monophonic melodies, could reveal whether a

simpler data distribution helps the model learn a more meaningful latent representation. If so, insights from these settings may inform improvements for handling the full diversity of NSynth.

Bibliography

Alpern, A. Techniques for algorithmic composition of music. 1995.

Bank, D., Koenigstein, N. and Giryes, R. Autoencoders. 2021.

Bernstein, M. N. Variational autoencoders. `https://mbernste.github.io/posts/vae/`, 2023. Accessed: 2025-05-22.

Cope, D. *Computers and musical style*. A-R Editions, Inc., USA, 1991.

Cymatics. Sound design basics: Fm synthesis. `https://cymatics.fm/blogs/production/sound-design-basics-fm-synthesis`, 2025. Accessed: 2025-03-13.

Engel, J., Agrawal, K. K., Chen, S., Gulrajani, I., Donahue, C. and Roberts, A. Gansynth: Adversarial neural audio synthesis. In *International Conference on Learning Representations (ICLR)*. 2019.

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Norouzi, M., Eck, D. and Simonyan, K. Neural audio synthesis of musical notes with wavenet autoencoders. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, 1068–1077. JMLR.org, 2017.

Fourier, J. B. J. *The Analytical Theory of Heat*. Cambridge Library Collection - Mathematics. Cambridge University Press, 2009.

Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. Book in preparation for MIT Press.

Grout, D. J., Burkholder, J. P. and Palisca, C. V. *A History of Western Music*. W. W. Norton & Company, New York, 8th edn., 2010.

Hahn, M. Subtractive synthesis: Learn synthesizer sound design. `https://blog.landr.com/subtractive-synthesis/`, 2022. Accessed: 2025-03-13.

Hiller, L. A. and Isaacson, L. M. *Experimental Music; Composition with an Electronic Computer*. Greenwood Publishing Group Inc., USA, 1979.

HUGGINGFACE. Introduction to audio data - hugging face audio course. https://huggingface.co/learn/audio-course/chapter1/audio_data, 2023. Accessed: 2024-03-02.

KILGOUR, K., ZULUAGA, M., ROBLEK, D. and SHARIFI, M. Fréchet audio distance: A reference-free metric for evaluating music enhancement algorithms. In *Interspeech 2019*, 2350–2354. 2019. ISSN 2958-1796.

KINGMA, D. P. and WELLING, M. Auto-encoding variational bayes. In *2nd International Conference on Learning Representations (ICLR)*. 2014.

KRIZHEVSKY, A., SUTSKEVER, I. and HINTON, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, Vol. 60(6), 84–90, 2012.

LECUN, Y., BOTTOU, L., BENGIO, Y. and HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, Vol. 86(11), 2278–2324, 1998.

MANILOW, E., SALAMON, J. and SEETHARAMAN, P. Representing audio. https://source-separation.github.io/tutorial/basics/representations.html, 2020. Accessed: 2024-03-01.

MATHEWS, M. V. The digital computer as a musical instrument. *Science*, Vol. 142(3592), 553–557, 1963.

MAURER, J. A. A brief history of algorithmic composition. https://ccrma.stanford.edu/~blackrse/algorithm.html, 1999. Accessed: 2024-03-01.

MEHRI, S., KUMAR, K., GULRAJANI, I., KUMAR, R., JAIN, S., SOTELO, J., COURVILLE, A. and BENGIO, Y. Samplernn: An unconditional end-to-end neural audio generation model. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. 2017.

MICHELUCCI, U. An introduction to autoencoders. 2022.

NIERHAUS, G. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer Publishing Company, Incorporated, 1st edn., 2009.

PATACCHIOLA, M. Evidence, kl-divergence, and elbo. https://mpatacchiola.github.io/blog/2021/01/25/intro-variational-inference.html, 2021. Accessed: 2025-05-10.

ROADS, C. *The Computer Music Tutorial*. MIT Press, Cambridge, MA, USA, 1996.

RÉVEILLAC, J.-M. *Synthesizers and Subtractive Synthesis 1: Theory and Overview*. Wiley-ISTE, 2024.

SERRA, X. and SMITH, J. Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal*, Vol. 14(4), 12–24, 1990.

SIMONI, M. *Algorithmic Composition: A Gentle Introduction to Music Composition Using Common LISP and Common Music*. Michigan Publishing, University of Michigan Library, Ann Arbor, MI, 2003.

SMITH, J. O. Physical modeling synthesis update. *Computer Music Journal*, Vol. 20(2), 44–56, 1996.

SOHN, K., LEE, H. and YAN, X. Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems* (edited by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama and R. Garnett), Vol. 28. Curran Associates, Inc., 2015.

TAGI, E. Synthesis methods explained: What is additive synthesis? `https://www.perfectcircuit.com/signal/what-is-additive-synthesis`, 2023a. Accessed: 2025-03-13.

TAGI, E. Synthesis methods explained: What is fm synthesis? `https://www.perfectcircuit.com/signal/what-is-fm-synthesis`, 2023b. Accessed: 2025-03-13.

TAGI, E. Synthesis methods explained: What is subtractive synthesis? `https://www.perfectcircuit.com/signal/what-is-subtractive-synthesis`, 2023c. Accessed: 2025-03-13.

VAN DEN OORD, A., DIELEMAN, S., ZEN, H., SIMONYAN, K., VINYALS, O., GRAVES, A., KALCHBRENNER, N., SENIOR, A. and KAVUKCUOGLU, K. Wavenet: A generative model for raw audio. In *9th ISCA Workshop on Speech Synthesis Workshop (SSW 9)*, 125. 2016.

WANG, Z. J., TURKO, R., SHAIKH, O., PARK, H., DAS, N., HOHMAN, F., KAHNG, M. and CHAU, D. H. P. Cnn explainer: learning convolutional neural networks with interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 27(2), 1396–1406, 2020.

XENAKIS, I. *Formalized Music: Thought and Mathematics in Composition*. Harmonologia series. Pendragon Press, 1992.

XIAO, H., RASUL, K. and VOLLGRAF, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. 2017.