Generación automática de notas musicales mediante autocodificadores variacionales condicionales
Automatic generation of music notes through conditional 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

Generación automática de notas musicales mediante autocodificadores variacionales condicionales
Automatic generation of music notes through conditional 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

Resumen

Generación automática de notas musicales mediante autocodificadores variacionales condicionales

Un resumen en castellano de media página, incluyendo el título en castellano. A continuación, se escribirá una lista de no más de 10 palabras clave.

Palabras clave

Aprendizaje profundo, Autocodificadores Variacionales Condicionales

Abstract

Automatic generation of music notes through conditional variational autoencoders

An abstract in English, half a page long, including the title in English. Below, a list with no more than 10 keywords.

Keywords

Deep Learning, Conditional Variational Autoencoders

Contents

List of figures

List of tables

Chapter 1

Introduction

"Predicting the future isn't magic, it's Artificial Intelligence"
— Dave Waters

1.1. Motivation 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 a 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 therefore focuses on the implementation of three different, through related, deep learning architectures applied to music: Autoencoders (AEs), Variational AEs (VAEs), and Conditional VAEs (CVAEs). AEs will allow us to see how well musical fragments can be mimicked, while VAEs and CVAEs make possible music generation through latent space sampling. CVAEs aditionally allow this sampling to be conditioned on some preference. While the results may not surpass the polish or versatility of commercial synthesizers, such a model can reveal new pathways for interactive sound design and serve as a starting point to other research projects.

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

1.2. Work Plan

This section describes the work plan to follow in order to achieve the objectives outlined in the previous section.

(Pongo aquí esto mejor yo creo) We will need to define a metric for the loss function, in order to quantify how good the sample generation provided by the CVAE is.

Chapter 2

State of the Art

In this chapter, we aim to first provide a brief overview of the evolution of algorithmic composition and, secondly, explore non-symbolic (i.e., low-level) 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 succint run-through of algorithmic composition and will necessarily be uncomplete (in terms of its 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 and Palisca, 1996). 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 Figures 2.1, 2.2, and 2.3, which

respectively represent the talea, chroma and the mapping between them of *De bon espoir-Puisque la douce-Speravi* by Guillaume de Machaut.

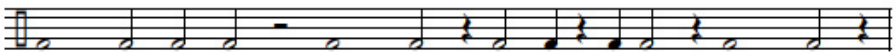

Figure 2.1: Talea of the isorhythmic motet *De bon espoir-Puisque la douce-Speravi* by Guillaume de Machaut. Retrieved from (Simoni, 2003).

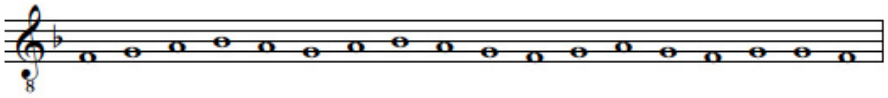

Figure 2.2: Color of the isorhythmic motet *De bon espoir-Puisque la douce-Speravi* by Guillaume de Machaut. Retrieved from (Simoni, 2003).

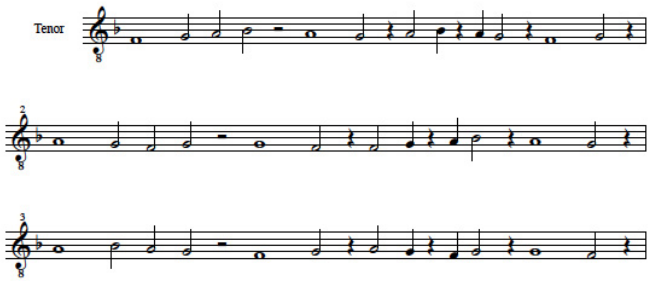

Figure 2.3: The tenor of *De bon espoir-Puisque la douce-Speravi* by Guillaume de Machaut. Retrieved from (Simoni, 2003).

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 2.4) 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).

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

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

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, 1959). 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. Although this thesis aims to ultimately employ a conditional variational autoencoder for generating musical notes, understanding the broader ecosystem of audio-focused methods places our work in context.

2.2.1. Traditional Synthesis Systems

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, 1822). 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 2.5), 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.

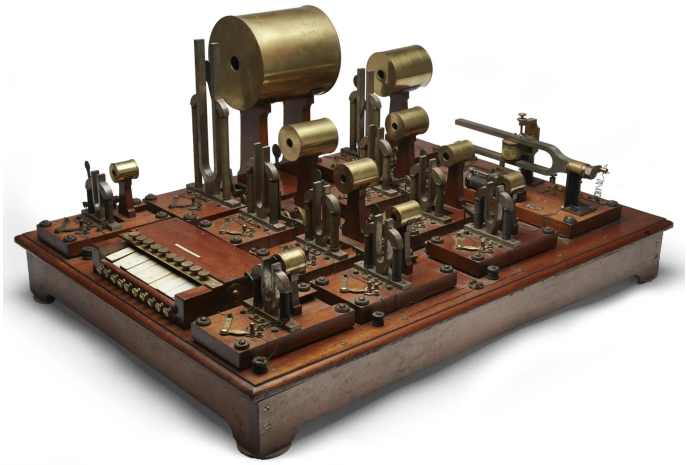

Figure 2.5: Pipe organ created by Hermann von Helmholtz around 1862. Retrieved from `https://shorturl.at/VuT2w`.

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 substractive techniques (Roads, 1996; Tagi, 2023a).

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 substractive synthesizer primarily uses these components:

- Oscillators: are the 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 2.6):

  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 simple stages allow you 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, 2023b).

Historically, subtractive synthesis dates as back 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).

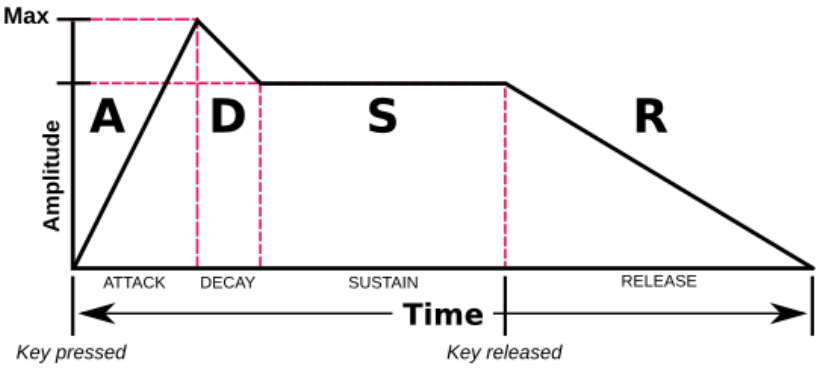

Figure 2.6: Illustration of Attack (A), Decay (D), Sustain (D) and Release (R). Retreived from `https://shorturl.at/Zj3UQ`.

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 to create new frequency components without filters (see Figure 2.7). 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).

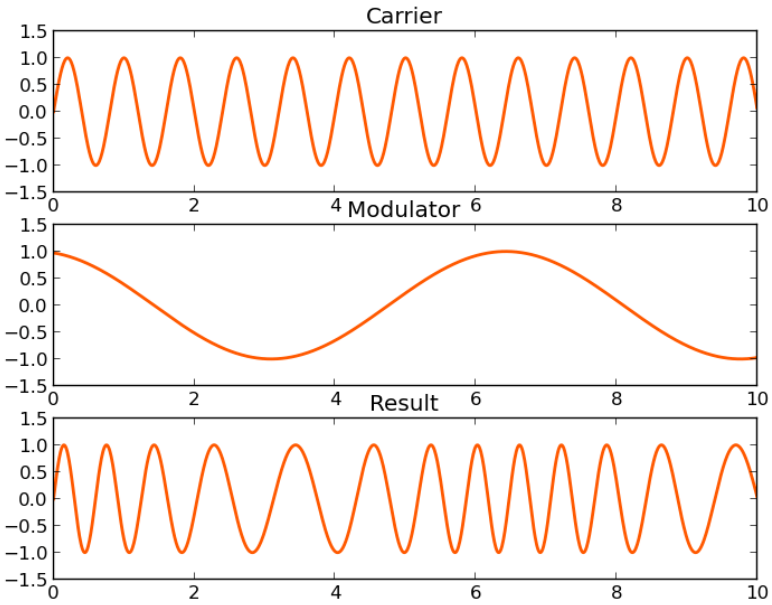

Figure 2.7: Illustration of a carrier, a modulator and the output. Retrieved from (Cymatics, 2025).

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 substractive synthesis). Operators can serve as carriers, modulators, or both, and they are arranged in various configurations or algorithms to produce different sound textures (Tagi, 2023c).

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, 1998).

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).

Another 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, our aim is to explain the fundamentals of audio and their most frequent representations. We see this necessary in order to be able to at least have a shallow 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 the 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 anyone 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: it is 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, 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.

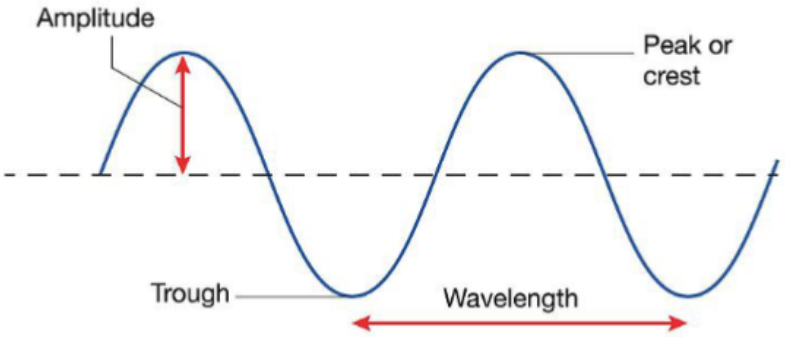

Figure 3.1: A wave can be characterized by its amplitude, frequency and wavelength. Retrieved from `https://shorturl.at/KtTHs`.

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 continouos curve containing an infinite number of values over time. If we want to digitally represent this audio wave it is clear we can cannot digitally store an infinite amoung 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).

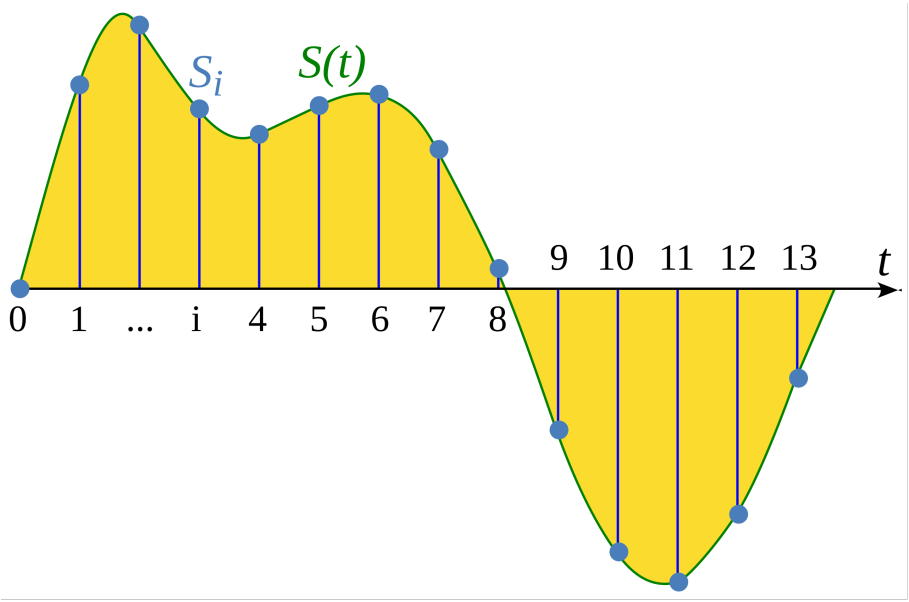

Figure 3.2: Example of sampling of an audio wave through time. Retrieved from (HuggingFace, 2023).

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, but the change in pressure caused by the wave. 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

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

or 24-bit depth, corresponding to 65,536 and 16,777,216 possible levels, respectively.

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, however, 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 which illustrates 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 this waveform 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

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

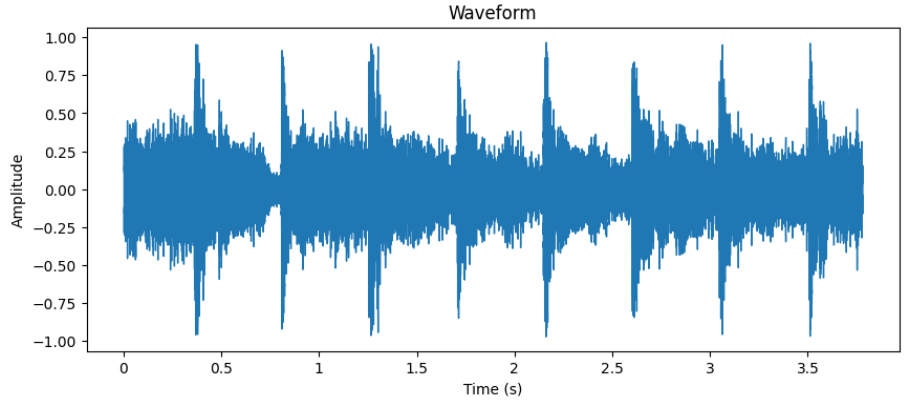

Figure 3.3: A waveform plot of a 4 second clip of ABBA's *Lay all your love on me*.

sum of multiple non-periodic sinusoids with different frequencies, or even a random signal like white noise. We will go more into depth about wave decomposition in section 3.3.

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 transforma-

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

tion breaks down signals into simpler sine and cosine waves, each corresponding to a specific frequency (see Figure 3.4).

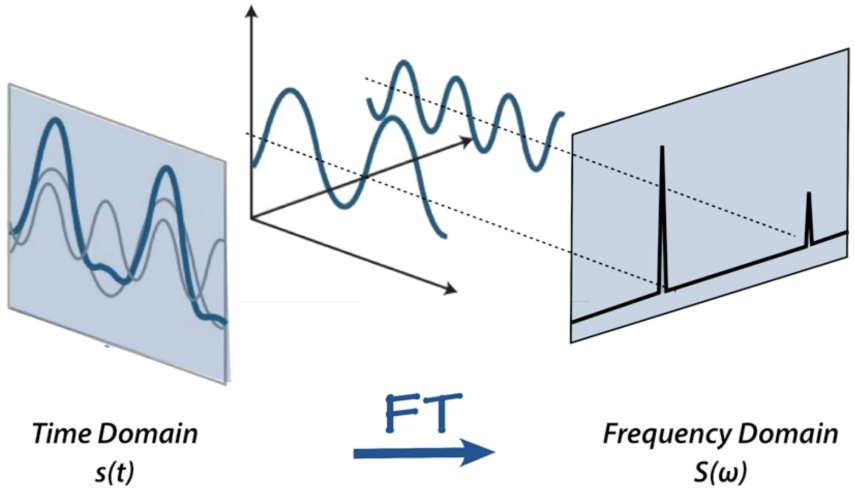

Figure 3.4: Effect of applying the Fourier transform to a signal. Retrieved from `https://shorturl.at/8D73u`.

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

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi ft}\, dt \tag{3.1}$$

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:

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

Esentially, 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 3.6), each vertical stripe represents the frequency spectrum at a specific point in time, seen from the top.

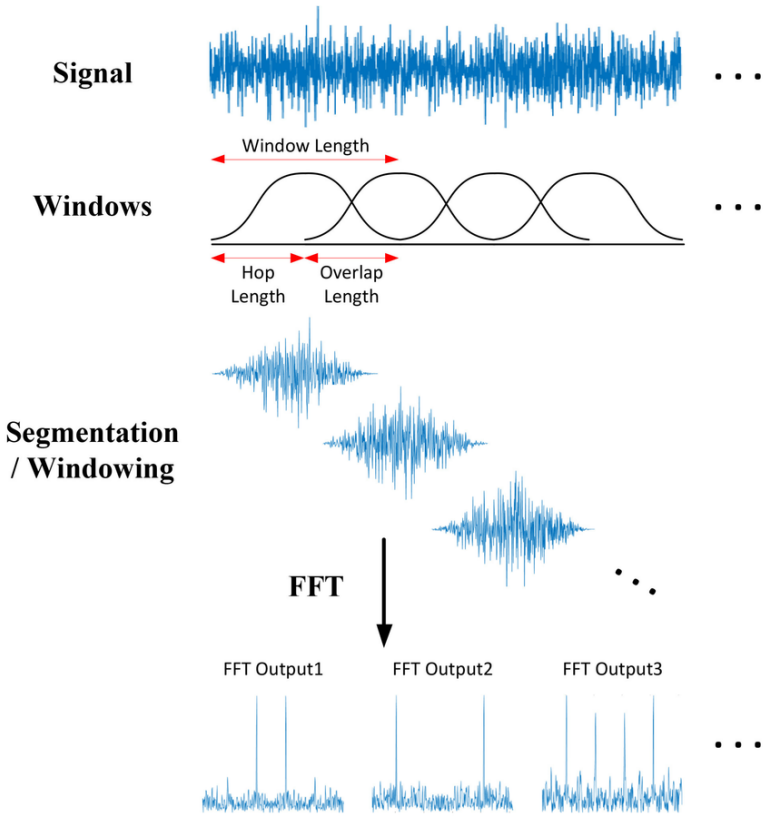

Figure 3.5: Short Time Fourier Transform diagram. Retrieved from `https://shorturl.at/JGODG`.

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 an 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, this "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.

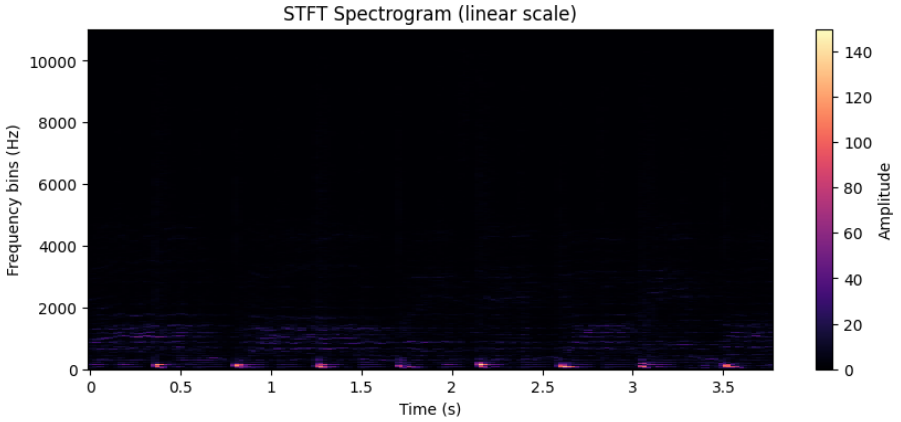

Figure 3.6: Resulting linear spectrogram after applying the STFT to a 4 second clip from ABBA's *Lay all your love on me*.

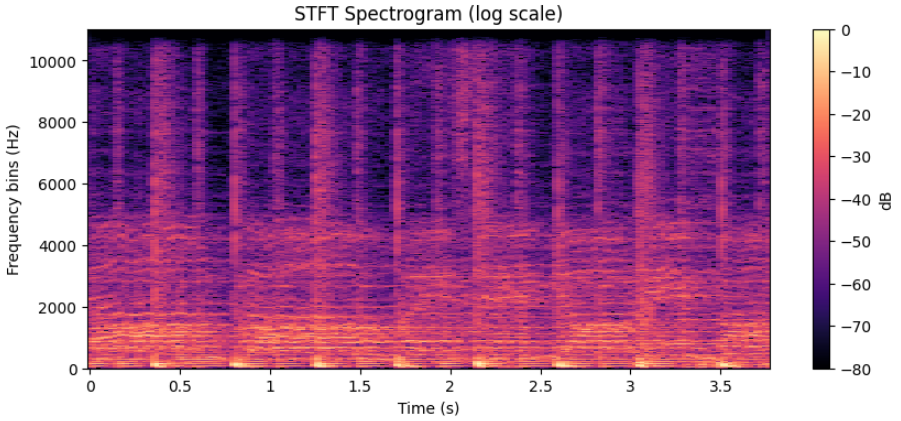

Figure 3.7: Resulting log spectrogram after applying the STFT to a 4 second clip from ABBA's *Lay all your love on me*.

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:

$$x_k = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{-i2\pi \frac{k}{N} n} \tag{3.3}$$

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.

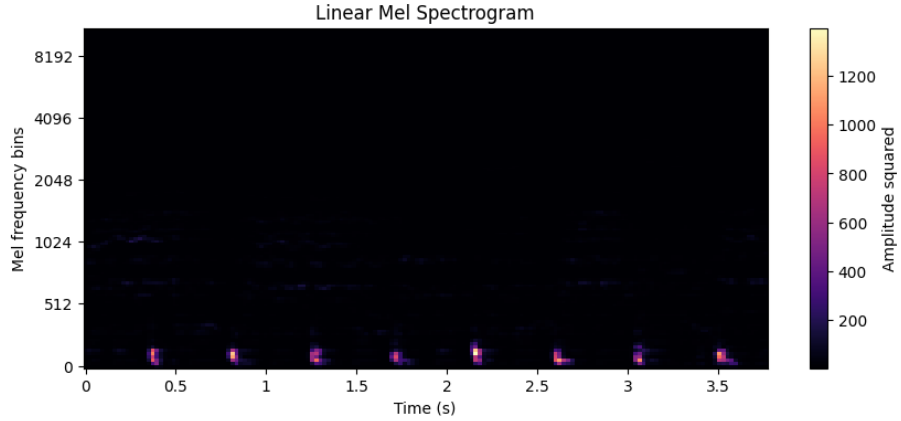

Figure 3.8: Linear mel spectrogram of a 4 second clip of ABBA's *Lay all your love on me*.

[4] `https://onlinetonegenerator.com/`

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.

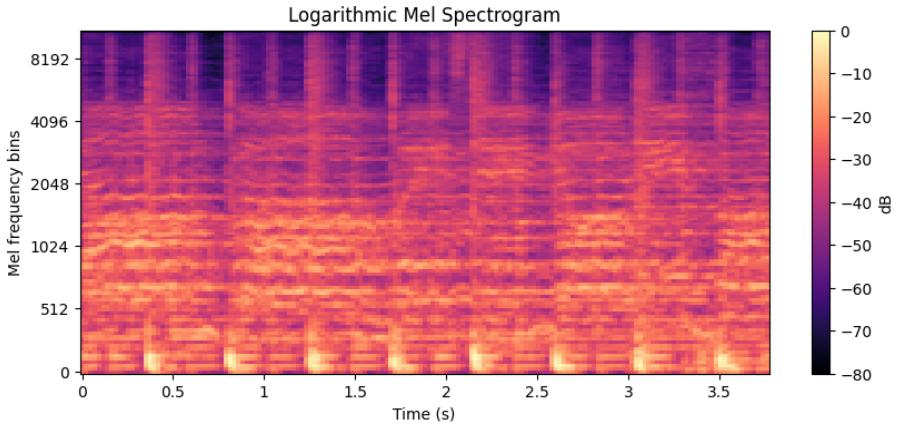

Figure 3.9: Logarithmic mel spectrogram of a 4 second clip of ABBA's *Lay all your love on me*.

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 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.

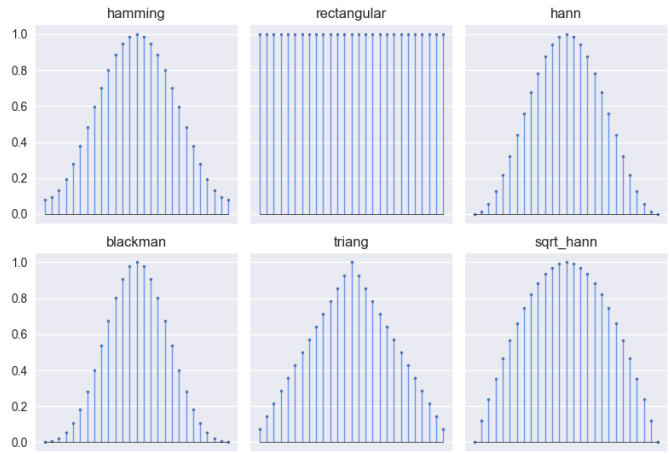

Figure 3.10: The choice of window function can significantly influence the spectral resolution and leakage effects in the spectrogram. Retrieved from `https://shorturl.at/UsEsq`.

3.5. Conclusion

In this chapter, we have explored the fundamental concepts of audio and its digital representations, which we consider essential for understanding audio tasks in deep learning. We began by discussing the basic properties of sound, such as amplitude, frequency, and wavelength, and how these relate to our perception of loudness and pitch. We then moved on to how continuous sound waves are converted into digital audio data through sampling, amplitude quantization, and bit depth, leading to time-domain representations like waveforms.

We also examined frequency-domain representations, focusing on the Fourier transform and its discrete version (DFT), which allow us to analyze an audio signal

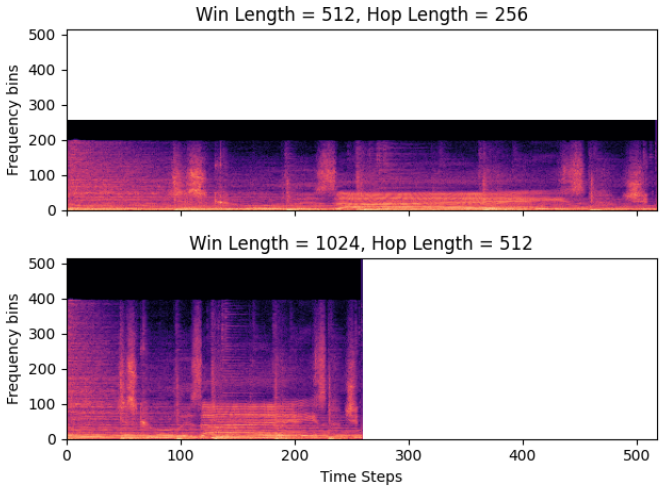

Figure 3.11: A shorter window results in finer time details but poorer frequency resolution, while a longer window captures more frequency details but at the cost of time resolution. Retrieved from `https://shorturl.at/UsEsq`.

in terms of its individual frequencies. We looked at spectrograms and mel spectrograms, which provide insights into how sound evolves over time and in the case of mel spectrograms, take into account human hearing perception. Finally, we discussed key parameters involved in creating spectrograms, such as window length and hop length.

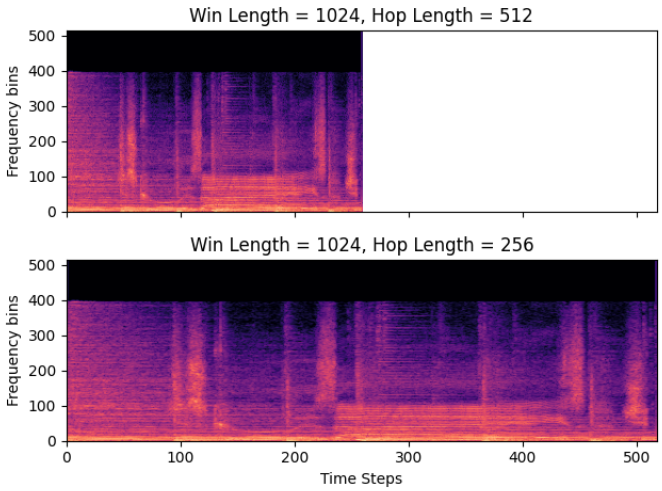

Figure 3.12: The smaller the hop length the more times a particular segment of the audio signal is represented in the STFT. Retrieved from `https://shorturl.at/UsEsq`.

Chapter 4

Introduction to Deep Learning and Conditional Variational Autoencoders

In this chapter, we will first provide the reader with 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, variational autoencoders and conditional variational autoencoders.

4.1. Deep Learning

Deep learning is a subset of machine learning that uses big 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 it, 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.

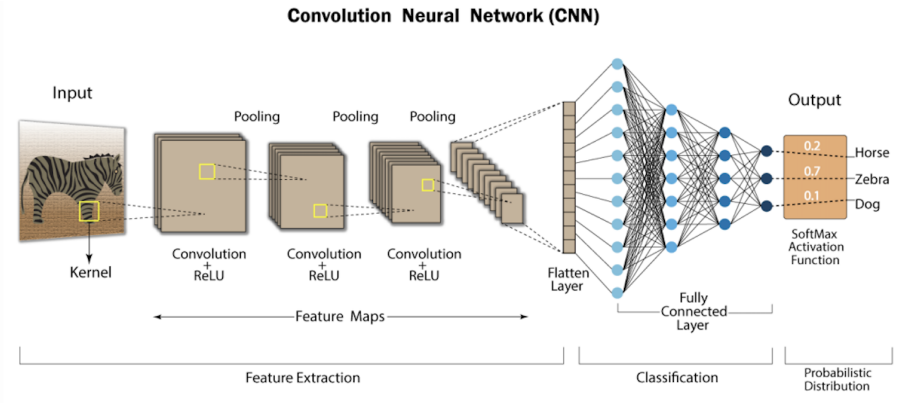

Figure 4.1: Typical CNN architecture. Retrieved from `https://shorturl.at/1uqcP`

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 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 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 $\hat{x}$ 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

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

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.

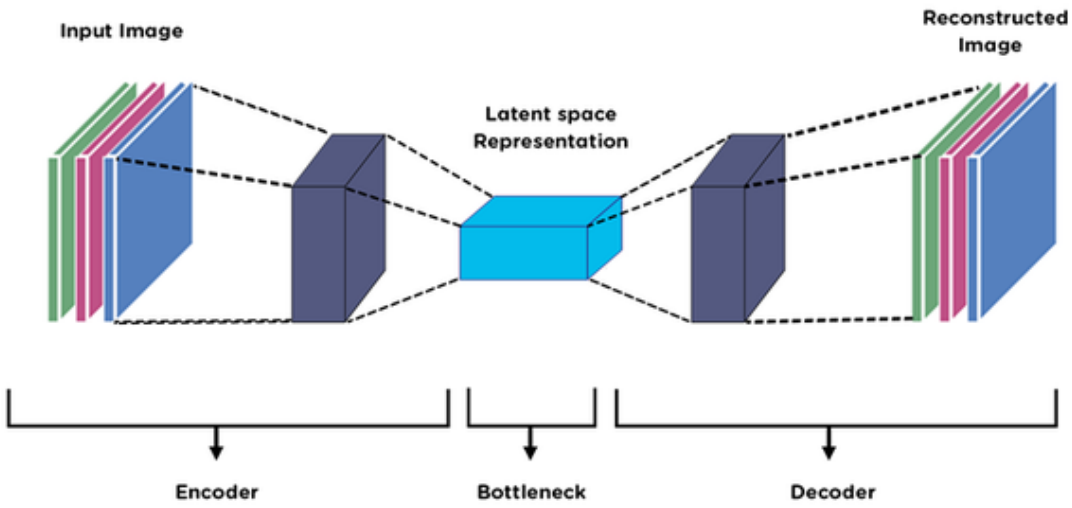

Figure 4.2: General structure of an autoencoder. The network consists of an encoder that compresses the input into a latent representation, and a decoder that reconstructs an output from it. Retrieved from `https://shorturl.at/esPtm`.

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 notes, taking advantage of local patterns in audio spectrograms.

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 we want to be able to generate data from the learned distribution of samples. For this purpose, we now introduce the variational autoencoders.

4.3.1. Variational Autoencoders (VAEs)

A Variational Autoencoder (VAE) (Kingma and Welling, 2022) is a type of generative model that builds on the autoencoder architecture but with a probabilistic twist. In a standard autoencoder, the encoder produces a deterministic code $h = f(x)$. In a VAE, the encoder instead produces a probability distribution over the latent space. Typically, given an input x, the encoder (often called the inference network in this context) outputs parameters of a distribution $q_\phi(z|x)$ —usually a Gaussian with mean $\mu(x)$ and diagonal covariance $\sigma^2(x)$— representing the probability of latent variable z given input x. We can think of this as the encoder no longer compressing x to a single point in latent space, but to a region of latent space characterized by μ and σ.

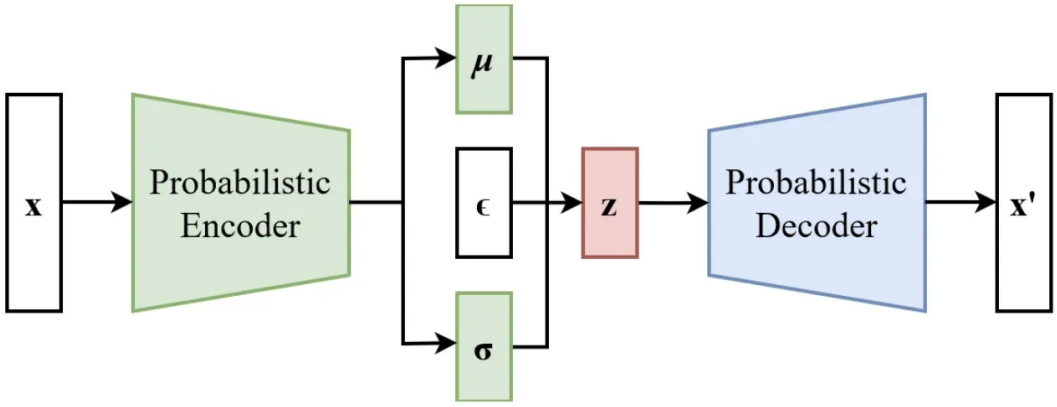

Figure 4.3: Variational autoencoder architecture (with reparameterization trick). The encoder (green) maps an input x to parameters $\mu(x)$ and $\sigma(x)$ of a Gaussian distribution $q(z|x)$ over the latent variable z. A latent sample z is drawn (by combining μ, σ with a random noise ϵ) and passed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

In the generative process of a VAE, we assume some fixed prior distribution over z, usually a simple prior $p(z) = \mathcal{N}(0, I)$ (Kingma and Welling, 2022). The decoder,

usually referred to as the generative network, defines $p_\theta(x|z)$, the probability of reconstructing x from latent z. Training a VAE involves maximizing the likelihood of the data under this generative model. However, directly maximizing the marginal likelihood $p_\theta(x) = \int p_\theta(x|z)p(z)dz$ is intractable due to the integral. VAEs address this by maximizing the evidence lower bound (ELBO), which is a substitute objective function that is easier to compute. The ELBO for a single data point x is:

$$\mathcal{L}(x;\theta,\phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \mathrm{KL}(q_\phi(z|x)\|p(z)) \tag{4.4}$$

This objective has two terms: a reconstruction term $\mathbb{E}_{q(z|x)}[\log p(x|z)]$ that encourages the decoder to reconstruct x accurately from sampled z, and a Kullback-Leibler (KL) divergence term $\mathrm{KL}(q(z|x)\|p(z))$ that regularizes the inferred latent distribution $q(z|x)$ to be close to the prior $p(z)$ (Kingma and Welling, 2022). Said in an informal manner, the Kullback-Leibler divergence term measures how much two probability distributions differ from one another.

One of the key innovations that makes VAEs trainable is the reparameterization trick (Kingma and Welling, 2022). Since sampling $z \sim q_\phi(z|x)$ is stochastic, we cannot directly backpropagate through a random sampling operation. The reparameterization trick circumvents this by expressing the sample z as a deterministic function of μ, σ, and a source of randomness ϵ that is independent of ϕ. For example, z can be obtained as:

$$z = \mu(x) + \sigma(x) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{4.5}$$

Here $\odot$ is element-wise multiplication. In this way, the randomness ϵ is pulled out of the network, and the remaining operations are deterministic and differentiable with respect to ϕ (the encoder parameters that produce μ and σ).

By optimizing the ELBO, the VAE jointly learns the encoder parameters ϕ and decoder parameters θ. At convergence, the encoder $q_\phi(z|x)$ learns to approximate the posterior distribution of latent variables given data, and the decoder $p_\theta(x|z)$ learns to generate realistic data samples from latent codes. To generate new data, one can sample z from the prior $p(z)$ (e.g. draw a random vector from $\mathcal{N}(0, I)$) and then feed it into the decoder to obtain a sample x'. Because the latent space was regularized by the KL term, samples from the prior tend to produce valid outputs, and interpolation in the latent space yields smooth interpolations in the data space (Kingma and Welling, 2022). The result is a generative model capable of not only compressing data like a standard autoencoder, but also synthesizing new plausible data. VAEs have been used in image generation, text generation, and audio synthesis, among other domains, due to these generative capabilities (Michelucci, 2022).

4.3.2. Conditional Variational Autoencoders (CVAEs)

A Conditional Variational Autoencoder (CVAE) is an extension of the VAE that allows us to introduce conditioning information into the encoding/decoding process

(Sohn et al., 2015). In many applications, we want to guide the generation process with some additional input or context. For example, in our case, we might want to generate musical notes of a certain instrument or pitch. In a CVAE, we supply an extra variable c (the condition) to both the encoder and the decoder networks in order to modulate the latent space according to that context.

Concretely, the encoder in a CVAE models $q_\phi(z \mid x, c)$ and the decoder models $p_\theta(x \mid z, c)$. The condition c could be a class label, one-hot vector, or any auxiliary information relevant to the data. By providing c to the encoder, we allow the encoding of x to depend on the context c. By providing c to the decoder, we inform the generative process about what we want to generate. The training objective for a CVAE is similar to a standard VAE, except conditioned on c:

$$\mathcal{L}(x, c; \theta, \phi) = \mathbb{E}_{q_\phi(z|x,c)}[\log p_\theta(x|z, c)] - \mathrm{KL}(q_\phi(z|x, c)\|p(z|c)) \tag{4.6}$$

In practice, $p(z|c)$ is often taken as $\mathcal{N}(0, I)$ for all c.

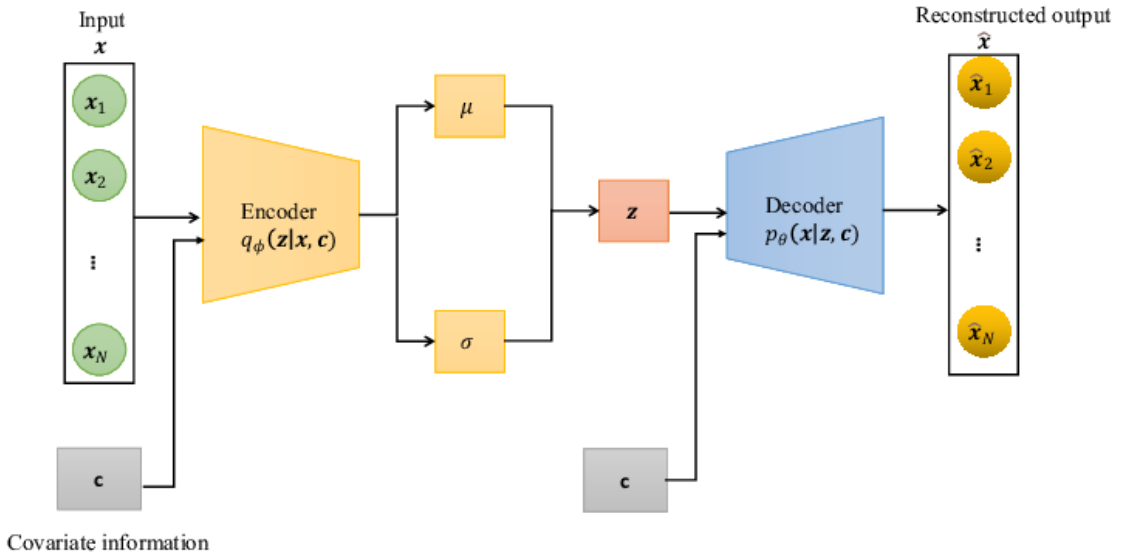

Figure 4.4: Conditional variational autoencoder architecture. The encoder (green) maps an input x and condition c to a latent distribution $q(z|x, c)$, and the decoder (blue) reconstructs the output x' conditioned on both z and c. Retrieved from `https://arxiv.org/abs/2211.02847`.

A significant benefit of the CVAE in our context is that it leverages label information to structure the latent space. In a vanilla VAE, the model might use part of the latent dimensions to encode information like "this is a piano" vs "this is a violin" if those instrument differences cause large variations in the input. The CVAE, given the instrument as input, can focus its latent dimensions on other characteristics (like timbral nuances or note dynamics), since it doesn't need to reinvent a code for the instrument identity — that's provided as c. This often leads to better utilization of the latent space and can improve generation quality for each class, because the model effectively trains separate (but related) generative pathways for each condition category (Sohn et al., 2015).

In our project, the use of a CVAE seems to be perfect for our purposes. We aim to generate musical notes from the NSynth dataset (Engel et al., 2017), and we want to be able to control certain attributes of the generated audio. The NSynth dataset consists of around 305,979 musical notes, each annotated with attributes like pitch (note), instrument type, velocity, and so on. We can thus train a CVAE that learns to generate, for example, a note of a given pitch played by a specified instrument.

Chapter 5

Experiments and results

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

This chapter is organized into five sections. The first three provide a detailed account of the parameters, design choices, and results for each of the three models introduced earlier. The fourth section presents a comparative analysis of their performance, while the final section offers concluding remarks.

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

The autoencoder architecture employed in this work is a convolutional autoencoder consisting of three hidden layers in both the encoder and the 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.

At first, the model took a 2D mel spectrogram as input, treated like a one-channel image. This choice was based on common practice in many existing models found online. The input was a 4D tensor of shape `[B, 1, H, W]`, where `B` is the batch size, 1 is the number of channels, and `H` and `W` are the height and width of the mel spectrogram. However, we found that models using this input struggled to reconstruct audio accurately, especially because mel spectrograms discard most of the phase information. It also felt limiting that perfect reconstruction from the mel spectrogram wasn't possible. Because of this, we switched to a different representation: we computed the STFT of the signal and extracted both magnitude and phase, using them as two separate channels. These were combined into a single input tensor of shape `[B, 2, H, W]`. While the resulting reconstructions were still imperfect, they were noticeably better than those obtained with mel spectrogram inputs.

Internally, the number of channels in each layer of the autoencoder was varied across different configurations; in our final setup, we used `x`, `y`, and `z` channels. Note, however, that the channel configuration was reversed in the decoder to mirror the encoder's architecture. 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 and reconstruction fidelity.

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. 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.

For downsampling in the encoder, 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 to progressively reconstruct the original input shape. To ensure the final output matched the original input dimensions, we applied zero-padding at the end of the decoding process. While this introduces a small irregularity in the decoding pipeline, we noticed minimal impact on the output quality and significantly simplified the implementation. 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.

5.1.2. Training setup

The loss function we used is the Mean Squared Error (MSE) with the mean reduction option. MSE has been widely used in autoencoder architectures and it computes the average squared difference between the input and its reconstructed output. By using the mean reduction, we ensured that the loss was averaged over all data points in the batch. For the optimization of our autoencoder, we employed the Adam optimizer, which is popular for its efficiency and adaptive learning rate.

We also used a `ReduceLROnPlateau`[1] learning rate scheduler that monitored the validation loss during training. If the validation loss increased after a certain number of epochs, the scheduler would reduce the learning rate to mitigate this issue.

In order to prevent overfitting, we experimented with the early stopping technique. Initially, we set the model to train for 50 epochs, but we found that the validation loss usually began to increase continuously around epoch 30. This can be observed in Figure 5.1. To mitigate this problem, we decided to limit the training to 30 epochs.

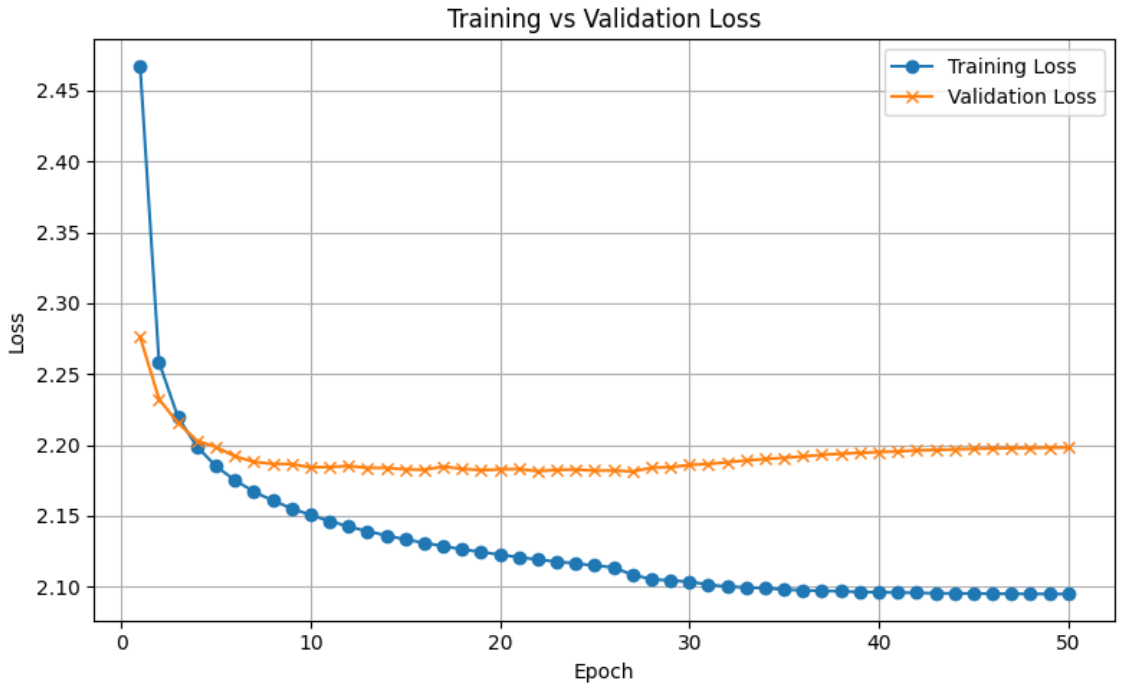

Figure 5.1: After epoch 30, the validation loss kept increasing, likely indicating model overfitting.

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

[1]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.

Parameter	Value
Train Loss	
Validation Loss	
Number of Epochs	30
Average Epoch Time (s)	
Learning Rate	
Batch Size	64
Sample Rate (Hz)	16000
FFT Size	
Hop Length	
Input Height	
Input Width	
Latent Dimension	

Table 5.1: Training Parameters

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 5.1.3).

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 final training parameters used for the final version of the autoencoder are shown in Table 5.1.

5.1.3. Quantitative Evaluation

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

T loss	V loss	Epc	T	lr	FFT	Hop	Ht	Wd	Lat.	Acc.
0.25	0.30	50	12.3	44100	1024	256	64	64	128	87.5%
0.20	0.28	100	11.5	44100	2048	512	64	64	128	89.2%
0.15	0.22	150	10.8	48000	2048	512	128	128	256	91.1%
0.30	0.35	50	15.0	44100	1024	256	64	64	128	85.3%
0.18	0.24	200	9.5	44100	2048	512	64	64	128	90.4%

Table 5.2: 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.

- $\hat{x}_i$ 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 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 acuracy 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 the model's configuration can be observed in Table 5.2.

TO BE FILLED IN CORRECTLY!!

5.1.4. Qualitative evaluation

Perhaps a more insightful evaluation of the model's performance can be taken a look at by examining the actual audio reconstructions. We have provided several examples, which can be found in the project's Github repository[2]. Each folder number corresponds to the row number in Table 5.2. Please also note that the file "`Xr.wav`" represents the reconstructed version of the audio "`X.wav`", where `X` is just a number.

TODO: input files into repo

5.2. Variational autoencoders

This section reuses many of the architecture and training details already described in Section 5.1, so they won't be repeated here. Instead, the focus is on highlighting the main differences and modifications made to the original autoencoder design. We also present the results and key findings from these changes.

5.2.1. Model Design and training setup

The VAE model introduced only a few modifications to the original autoencoder, specifically in the encoder, the forward pass, and the loss function. In the encoder, the linear layer that originally projected the input to the latent space was repurposed to output the mean, and a second linear layer was added to compute the log variance. In the forward pass, the reparameterization trick was applied using these two outputs to sample latent vectors in a differentiable way. Aside from this, the rest of the forward process remained unchanged.

The loss function, however, required a more significant adjustment. Unlike the autoencoder, which used only the MSE loss, the VAE combines the MSE term with a KL divergence term to encourage the latent distribution to match a standard normal prior (see Subsection 4.3.1 for more details). These two components were balanced using a weighting factor $\alpha \in [0, 1]$[3], and the total loss was computed as

$$\text{Loss} = \alpha \times \text{MSE} + (1 - \alpha) \times \text{KL}.$$

Aside from this change in the loss function, the rest of the training procedure—including data loading, optimizer setup, learning rate schedule, and training loop—remained essentially identical to that used for the autoencoder. The impact of different values of α on model performance is discussed in the following section.

[2] `https://github.com/pabgarcialopez/TFG-info/tree/main/examples/AE`

[3] Note that when $\alpha = 1$, the loss function reduces to that of the standard autoencoder.

T loss	V loss	Epc	T	lr	FFT	Hop	Ht	Wd	Lat.	α	Acc.
0.25	0.30	50	12.3	44100	1024	256	64	64	128	0.3	87.5%
0.20	0.28	100	11.5	44100	2048	512	64	64	128	0.3	89.2%
0.15	0.22	150	10.8	48000	2048	512	128	128	256	0.3	91.1%
0.30	0.35	50	15.0	44100	1024	256	64	64	128	0.3	85.3%
0.18	0.24	200	9.5	44100	2048	512	64	64	128	0.3	90.4%

Table 5.3: 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, α: loss function weight, Acc.: Testing accuracy.

5.2.2. Quantitative evaluaion

For audio reconstruction, we used the same metric as in the autoencoder experiments. Table 5.3 compares the reconstruction accuracy under different training configurations and values of α.

TO BE FILLED IN CORRECTLY!!

5.2.3. Qualitative evaluation

To qualitatively assess the VAE's performance, we analyzed outputs generated through both posterior and prior sampling.

When sampling from the posterior, we start with a real audio input and encode it into a distribution over latent variables — specifically, a Gaussian with a learned mean and log variance. From this distribution, we sample a latent vector and pass it through the decoder to reconstruct the input. This process is used during training and reconstruction, and the resulting audio typically resembles the original input closely. Since the latent code is directly derived from the data, the decoder operates in regions of the latent space it has seen before, leading to realistic but not necessarily diverse outputs.

Prior sampling, on the other hand, is what allows VAEs to act as generative models. Instead of using an input to guide the sampling, we draw random latent vectors from the prior distribution, usually a standard normal $\mathcal{N}(0, I)$, and feed them directly into the decoder. The idea is that the decoder has learned a meaningful structure of the data in the latent space, so random samples should still yield valid outputs. However, in our experiments, this was not entirely the case: prior samples did not produce clearly distinct sounds, but rather variations of a single base tone or structure with slight differences in texture. This suggests that the model had not fully learned to utilize the latent space in a rich or disentangled way — likely a symptom of the mismatch between the assumed prior and the true distribution of latent codes the decoder was trained on.

An interesting artifact observed in both types of generation was that many sounds had very similar pitch. This may be due to the mean layer averaging out variability in the latent representation, effectively pulling latent vectors toward a central region of the space.

Once again, we provide examples of prior and posterior sampling in the project's Github repository[4]. Each folder number corresponds to the row number in Table 5.3.

5.3. Conditional variational autoencoders

We now turn to the final model explored in this thesis: the Conditional Variational Autoencoder (CVAE). This section outlines how CVAEs extend standard VAEs, describes the modifications made to the model architecture, and presents the results obtained from our experiments.

5.3.1. Model Design and training setup

The Conditional Variational Autoencoder (CVAE) extends the standard VAE by introducing conditioning on additional information. In this setup, both the encoder and decoder receive an extra input that guides the encoding and generation process. For our experiments, we conditioned the model on the instrument type, allowing us to steer the generation toward sounds produced by specific instruments from the NSynth dataset, whose frequency can be observed in Table 5.4.

Family	**Acoustic**	**Electronic**	**Synthetic**	**Total**
Bass	200	8,387	60,368	68,955
Brass	13,760	70	0	13,830
Flute	6,572	35	2,816	9,423
Guitar	13,343	16,805	5,275	35,423
Keyboard	8,508	42,645	3,838	54,991
Mallet	27,722	5,581	1,763	35,066
Organ	176	36,401	0	36,577
Reed	14,262	76	528	14,866
String	20,510	84	0	20,594
Synth Lead	0	0	5,501	5,501
Vocal	3,925	140	6,688	10,753
Total	108,978	110,224	86,777	305,979

Table 5.4: Instrument family distribution in the NSynth dataset, categorized by source type (acoustic, electronic, synthetic).

[4] `https://github.com/pabgarcialopez/TFG-info/tree/main/examples/VAE`

5.3.2. Quantitative evaluaion

5.3.3. Qualitative evaluation

5.4. Conclusion

Conclusions and Future Work

The aim of this thesis was to explore the capabilities of three closely related deep learning models — autoencoders (AEs), variational autoencoders (VAEs), and conditional variational autoencoders (CVAEs) — in the context of musical audio data.

Conclusions

- Our autoencoder experiments achieved a testing accuracy of around 50%. While far from optimal, these results were obtained using a custom architecture built from scratch, demonstrating a solid baseline for future improvements.
- The VAE model introduced probabilistic sampling, allowing for generation of new audio samples. However, prior sampling produced limited diversity: most outputs shared similar textures and pitch, suggesting an underutilized latent space.
- The CVAE model, designed to condition generation on instrument type, remains underexplored at this stage. Results are still preliminary and require further testing and tuning to assess its ability to produce class-controlled outputs.

Future Work

- Try alternative model architectures: experiment with deeper networks, different kernel sizes, channel counts, or latent dimensions.
- Train on alternative datasets to evaluate model generalization and explore different sonic characteristics.
- Investigate the latent space in more depth: for instance why VAE-generated samples tend to have the same pitch or how interpolation behaves between

points.

- Use alternative loss functions or evaluation metrics that better reflect perceptual quality or musical relevance.

Bibliography

Alpern, A. Techniques for algorithmic compositionof musicadam. 1995.

Bank, D., Koenigstein, N. and Giryes, R. Autoencoders. 2021. Retrieved from https://arxiv.org/abs/2003.05991.

Cope, D. *Computers and musical style*. A-R Editions, Inc., USA, 1991. ISBN 0895792567.

Cymatics. Sound Design Basics: FM Synthesis. https://shorturl.at/jhdDo, 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. Accessed: March 2024.

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Eck, D., Simonyan, K. and Norouzi, M. Neural audio synthesis of musical notes with wavenet autoencoders. 2017.

Fourier, J. B. J. *The Analytical Theory of Heat*. C. G. G. and J. A. G. Balbi, 1822. Foundational work on Fourier analysis, which underpins additive synthesis.

Goodfellow, I., Bengio, Y. and Courville, A. Deep learning. MIT Press, 2016.

Grout, D. J. and Palisca, C. V. *A History of Western Music*. W. W. Norton & Company, New York, 5th edn., 1996.

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

Hiller, L. and Isaacson, L. *Experimental Music: Composition with an Electronic Computer*. McGraw-Hill, 1959.

HuggingFace. Introduction to audio data - hugging face audio course. 2023.

Kingma, D. P. and Welling, M. Auto-Encoding Variational Bayes. 2022.

KRIZHEVSKY, A., SUTSKEVER, I. and HINTON, G. E. ImageNet classification with deep convolutional neural networks. Communications of the ACM, 2012.

LECUN, Y., BOTTOU, L., BENGIO, Y. and HAFFNER, P. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998.

MANILOW, E., SALAMON, J. and SEETHARAMAN, P. Representing audio. 2020.

MATHEWS, M. Computer music. *Computer Music Journal*, Vol. 7(4), 18–37, 1963. Pioneering work in computer music demonstrating the potential of additive synthesis.

MAURER, J. A. A brief history of algorithmic composition. 1999. Retrieved from `https://ccrma.stanford.edu/~blackrse/algorithm.html`.

MEHRI, S., KUMAR, K., GULRAJANI, S., KUMAR, R., JAIN, S., SOTELO, J., COURVILLE, A. and BENGIO, Y. SampleRNN: An Unconditional End-to-End Neural Audio Generation Model. In *5th International Conference on Learning Representations (ICLR)*. 2017. Accessed: March 2024.

MICHELUCCI, U. An Introduction to Autoencoders. 2022.

NIERHAUS, G. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer, 2009.

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. 2016.

ROADS, C. *The Computer Music Tutorial*. MIT Press, 1996. A comprehensive overview of computer music techniques, including additive synthesis.

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

SERRA, X. *Spectral Modeling Synthesis: Theory and Applications*. Oxford University Press, 1998.

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. ISSN 01489267, 15315169.

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 subtractive synthesis? `https://www.perfectcircuit.com/signal/what-is-subtractive-synthesis`, 2023b. Accessed: 2025-03-13.

Tagi, E. Synthesis Methods Explained: What is FM Synthesis? `https://www.perfectcircuit.com/signal/what-is-fm-synthesis`, 2023c. Accessed: 2025-03-13.

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*. Pendragon Press, 1992.

Appendix A

Título del Apéndice A

Los apéndices son secciones al final del documento en las que se agrega texto con el objetivo de ampliar los contenidos del documento principal.

Appendix B

Título del Apéndice B

Se pueden añadir los apéndices que se consideren oportunos.