FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Synthesizing Soundscapes from Textual Input:

Development and Comparison of Generative AI Models

Márcio Duarte

Mestrado em Engenharia Informática e Computação

Supervisor: Luís Paulo Reis

Second Supervisor: Gilberto Bernardes

October 11th, 2023

Synthesizing Soundscapes from Textual Input:
Development and Comparison of Generative AI Models

Márcio Duarte

Mestrado em Engenharia Informática e Computação

Approved by . . . :

President: Henrique Lopes Cardoso

Referee: Fabien Gouyon

October 11th, 2023

Resumo

Hoje em dia, o áudio é um elemento essencial na maioria do conteúdo produzido online. Normalmente, é trabalho manual que está por detrás dos terabytes de áudio publicados diariamente. Se algum produtor desejar um som específico, tem de o pesquisar em bases de dados online, sintetizá-lo ou até gravá-lo. Essa quantidade de trabalho é uma restrição à criação de conteúdo, principalmente se esses sons forem elaborados ou muito específicos.

Os ficheiros de áudio apresentam apenas uma dimensão, *i.e.*, a amplitude da sua onda sonora recolhida num determinado intervalo de tempo. Em comparação, os ficheiros de imagem apresentam três dimensões de dados. Independentemente, as dependências temporais constituem um desafio no som, uma vez que são mais complexas e intrincadas do que as das imagens. Por exemplo, existe a expetativa de que o timbre de um determinado instrumento ou o ruído de fundo de um determinado ambiente sonoro se mantenha ao longo do tempo.

Esta dissertação apresenta uma investigação exaustiva de modelos avançados de IA generativa. Sugere alguns modelos e executa outros. Redirecciona a sua atenção para um modelo baseado em GANs que opera no espaço latente. Ao contrário de sistemas anteriores que limitam a síntese de áudio a domínios específicos, como a música ou a fala, o sistema proposto ultrapassa estas restrições, gerando áudio a partir de qualquer pedido textual. Outro foco desta dissertação é o desenvolvimento de diversas estruturas de modelos de deep learning, todas baseadas em abordagens modernas, bem como os desafios que surgem quando se trabalha com hardware limitado.

Dado que já existe trabalho significativo com modelos que criam imagens ou realizam a conversão de texto em fala, a expetativa é que um modelo generativo de áudio de ponta produza resultados satisfatórios. Os resultados deste trabalho terão impacto na velocidade do processo de produção para criadores de conteúdo, engenheiros de som e todos os interessados na criação de produtos sonoros.

Abstract

Nowadays, audio is a fundamental aspect of most online content. Generating terabytes of audio content daily relies heavily on manual labor. Content creators often find themselves tackling arduous tasks such as researching sounds from online databases, synthesizing complex audio, or even resorting to self-recording. This substantial workload poses a significant hurdle to content creation, particularly when dealing with intricate or highly specific sounds.

Unlike image files, which present three-dimensional data dimensions and are utilized by mainstream generative models, audio files are restricted to a single dimension, representing the amplitude of sound waves at specific time intervals. However, the challenge of long-term dependencies in audio is more complex than in images. Instrument timbre and persistent background noise are factors that further complicate the audio synthesis process.

This dissertation presents an extensive investigation of advanced generative AI models. It suggests certain models and carries out others. It redirects its attention to a model based on GANs that operates within the latent space. Unlike previous systems that limit audio synthesis to specific domains such as music or speech, the proposed system overcomes these constraints by generating audio from any textual prompt. Another focus of this dissertation is the development of varied deep learning model structures, all based on cutting-edge methods, as well as the challenges that arise when working with limited hardware.

The main contribution of this work focuses on investigating cutting-edge generative AI models, as well as developing and enhancing generative AI frameworks customized for audio synthesis. This involves thoroughly exploring model components, training strategies, and performance benchmarks. This research bridges a critical gap in the current AI landscape by allowing the creation of diverse and contextually rich audio content from textual cues. It not only demonstrates the capabilities of AI-powered audio synthesis, but also offers a useful resource for researchers and professionals exploring this growing field.

Acknowledgements

In embarking on this thesis development journey, I am grateful for the support and guidance provided by many individuals and institutions. Their contributions played pivotal roles in shaping the trajectory of this work.

I am grateful to my distinguished thesis supervisors, Luís Paulo Reis and Gilberto Bernardes. Their dedicated mentorship and guidance have been pivotal in guiding this research. Their expertise and encouragement have enlightened every step of this journey, and for that, I am profoundly appreciative.

The Artificial Intelligence and Computer Science Laboratory (LIACC) deserves thanks for the necessary server infrastructure for my project's development.

The academic atmosphere at the Department of Informatics and FEUP (Faculdade de Engenharia da Universidade do Porto) contributed to my intellectual curiosity and academic growth. The institution's commitment to excellence supported my development as a student.

I am grateful for the support of my family, as well as the contributions of individuals, institutions, and communities involved in this endeavor. To my mother, father, and brother, Fábio, I appreciate their support. I pay tribute to my beloved cat, Kikka, whose presence has brought joy to my life and serves as an inspiration for my work. I am grateful to my close friends for their enduring support throughout this journey.

I humbly acknowledge the larger AI community as a critical driving force behind technological advancement. The collaborative effort in this community has been the primary driving force behind the advancements that will propel us into the future.

I appreciate the music world, as it has been a crucial aspect of my identity, in addition to my pursuit of computer science. Music stimulates emotions, demonstrates complex patterns, and shapes our perception of the world, which has motivated me during my journey. Being a musician, I find parallels between harmonies in melodies and algorithms. Furthermore, I express gratitude to the institutions and communities that uphold musical expression. Their dedication to the art of sound has enriched my own creative ventures and surely influenced how I explore sound in this thesis.

This journey demonstrates the effectiveness of collaboration, shared contributions, and the synergistic relationship between disciplines. This work testifies to our joint pursuit of knowledge and innovation, and I am grateful to have contributed to this exceptional and harmonious effort.

Márcio Duarte

"To go wrong in one's own way
is better than to go right in someone else's."

Fyodor Mikhailovich Dostoyevsky

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AE autoencoder

AI artificial intelligence

API Application Programming Interface

AR autoregressive

AT Audio Transformer

BCE binary cross-entropy

BPE byte pair encoding

CLIP Contrastive Language–Image Pre-training

CNN convolutional neural network

CPU Central Processing Unit

CSS concatenative sound synthesis

DARN deep autoregressive network

DCGAN deep convolutional generative adversarial network

DFT Discrete-Fourier-Transform

DL deep learning

DNN deep neural network

dVAE discrete variational autoencoder

ELBO evidence lower bound

GAN generative adversarial network

GB gigabyte

GLIDE Guided Language to Image Diffusion for Generation and Editing

GPU graphics processing unit

GRU gated recurrent unit

Hz Hertz

KL Kullback–Leibler

LIACC Artificial Intelligence and Computer Science Laboratory

LLM Large Language Model

LSTM long short-term memory

MAD mean absolute deviation

MAE mean absolute error

ML machine learning

MOS mean opinion score

MPD multi-period discrimintor

MS-VQ-VAE multi-scale vector quantised variational autoencoder

MSD multi-scale discriminator

MSE mean squared error

NaN not a number

NLP natural language processing

RAM Random Access Memory

ReLU rectified linear unit

RNN recurrent neural network

RVQ residual vector quantizer

SGD stochastic gradient descent

STFT Short-Time Fourier Transform

tanh hyperbolic tangent

TTS text-to-speech

VAE variational autoencoder

VQ vector quantized

VQ-VAE vector quantized variational autoencoder

XOR exclusive OR

Chapter 1

Introduction

Contents

Audio is essential for shaping human experiences and interactions in the digital age. Audio content, ranging from podcasts and music to sound effects and immersive environments, is pervasive in our lives and enhances multimedia experiences. However, generating high-quality, diverse, and contextually relevant audio is still a time-consuming and labor-intensive process. As the demand for audio content continues to increase, there is a need for innovative solutions that can simplify the process and enable creators to generate various audio content with less effort.

Recent advances in artificial intelligence (AI) and deep learning (DL) are revolutionizing different domains, such as image generation and text-to-speech synthesis. These cutting-edge models exhibit exceptional abilities to generate high-quality content from basic textual inputs. This thesis is motivated by these successful models and explores the potential of AI-driven generative models to synthesize audio snippets based on textual descriptions. This thesis is not constrained to specific domains, like music or speech, but instead studies and implements systems suitable for a wide range of audio prompts.

The motivation behind this work lies in the potential benefits that an end-to-end audio generative model can offer content creators, sound engineers, and other stakeholders in the audio production ecosystem. By reducing the time and effort required to generate unique and high-quality audio samples, this research aims to contribute to the democratization of audio content creation and facilitate new avenues for creative expression.

This chapter provides an introduction to the research study and sets the stage for the exploration of the topic. It aims to supply a comprehensive overview of the present study's context, motivation, objectives, and structure.

The context information can be found in section 1.1. Its focus is to provide the necessary background for the research study and to explain its importance. Section 1.2 explores the motivation behind the study. The purpose of this section is to explain the necessity of the research and the knowledge gap it aims to address. Next, in section 1.3, the research study's objectives are described. This section aims to explain the specific objectives and outcomes the research seeks to accomplish. The structure of the dissertation is presented in Section 1.4. This section offers an organized overview of the remaining chapters.

1.1 Context

According to the University of York [126], "Computer Science is the study of computation and information". In practice, engineers and computer scientists typically create programs that machines execute. Traditionally, these programs consist of a sequence of instructions for the computer to follow.

Currently, processing and analyzing vast datasets is imperative for most endeavors. Computer science enables the possibility of these commodities, ranging from running hospitals to creating songs for streaming platforms.

Currently, most endeavors require processing and analyzing vast datasets. These commodities are only possible with computer science, from running hospitals to creating the songs one listens to on a streaming platform.

This amount of data has given rise to a new type of application. They do not need to have these instructions wired in. Instead, algorithms learn these from data. To this end, we call this machine learning (ML).

Since computers were invented, the scientific community has wondered whether they can learn [83]. ML is a field devoted to understanding and building methods that let machines "learn" – that is, methods that leverage data to improve computer performance on some set of tasks [7]. Modern AI was born in 1958 when F. Rosenblatt [108], in an attempt to understand the capability of higher organisms for perceptual recognition, generalization, recall, and eventually thinking, proposed three fundamental questions:

1. "How is information about the physical world sensed, or detected, by the biological system?"

2. "In what form is information stored, or remembered?"

3. "How does information contained in storage, or in memory, influence recognition and behavior?"

With this, Rosenblatt theorized a mathematical system called the perceptron (explained in Section 2.1.2.1 that, by following the supposed behavior of neurons in one's nervous system, was the central piece of a hypothetical system capable of answering these questions.

Then, during the 1960s, much work was put into convergence algorithms for the perceptron and models based on it. Both deterministic and stochastic methods were proposed [41]. However, in 1969, Minksy and Papert [79] published a book demonstrating the limitations of perceptrons. Namely, the authors showed that perceptrons could only represent linear functions, and simple non-linear functions such as exclusive OR (XOR) were impossible. As a result, the study of AI was mainly halted until the 1980s. This period is usually called *the first winter of AI* [41].

During the 1980s, studies of learning under multilayer neural networks went underway [41]. In 1986, Rumelhart et al. [110] described a new learning procedure for networks of neurons. The procedure adjusts the weights of the network's connections to minimize a measure of the difference between the expected and the actual output, an error function. This method is still used nowadays and is called *backpropagation*.

Fradkov et al. [41] argue that an intensive advertisement of the success of backpropagation and other computational advances produced great hope for future successes. However, real successes were not happening, and investments in ML decreased again in the early 1990s. This period is called *the second winter of AI* [77].

The turn of the millennium saw a new rise in ML technologies; this time, it was, until now, for good. According to Fradkov et al. [41], this was due to three trends that emerged:

1. The appearance of big data. Dealing with huge amounts of data is an interest not only to a small portion of scientists but to the whole market.

2. Reduced cost of parallel computing with both software (with, for instance, Google's MapReduce [25]) and hardware (with an investment in specialized hardware for ML from companies such as NVidia).

3. A newfound interest by scientists in new, more complex, ML algorithms, denominated *deep neural networks (DNNs)*.

The critical idea of this new ML is that it can infer plausible models to explain the observed data. A machine can use such models to make predictions about future data and make rational decisions based on these predictions [44]. It involves training a model on a large dataset to learn patterns and relationships in the data and then make predictions or decisions based on those patterns. For instance, a computer program can learn from medical records which treatments are most effective

against new diseases, or houses can learn from experience to optimize energy costs based on the particular usage patterns of their occupants.

In traditional ML, the features input into the models were usually hand-picked by humans, which leads to errors. New models learn intermediate representations— a vector of features — from data. The model itself usually performs this feature extraction with more layers [45]. Hence, deep learning (DL). DL algorithms consist of multiple layers of interconnected nodes and are trained to learn complex patterns and relationships in the data. DL algorithms can automatically learn and extract features from data. They are particularly well-suited for tasks such as image and audio processing.

The use of DL in everyday applications increased during the 2010s. Nowadays, DL is relied upon for various computer-made tasks including text translation, recommender systems, fake news detection, spam filtering, image captioning, and even self-driving cars [24]. Generative models are a key tool for creating new data.

Generative models, which are a type of DL model, can create synthetic data that is similar to a given training dataset. Generative models can produce new data that conforms to the distribution learned from the training data. In contrast to common learning objectives such as classification or regression tasks, which focus on labeling inputs or estimating mappings, generative models aim to replicate and capture hidden statistics in observed data [61].

The scientific community turned its heads to generative models a few years ago. These models allow a variety of new applications and products. For instance, DALL-E [101] and DALL-E 2 [100] allow the generation of images given any textual input (see Sections 2.1.5.2 and 2.1.5.5. GPT-3 is an extensive language model [15] that powers applications such as ChatGPT, capable of generating text. Modern text-to-speech (as seen in Section 2.2.4.1 applications also rely on these technologies.

Because of the relevance and effectiveness of these new generative technologies, DL has achieved mainstream status, and the general public is aware of the capabilities of these algorithms. Given the rise of AI in the 2010s with predictions and classifications, it is plausible that the 2020s will be a decade of generative applications. Automating manual work to enhance human creativity has never been more feasible. Generative applications span diverse domains such as images, text, and audio. However, even in the context of a well-defined frame of reference and optimal individual categorization, it's important to recognize that models inevitably involve reduction to averages. This raises concerns about undesirable convergence and oversimplification of media [39].

For the purpose of this thesis, audio is broadly classified into three main categories: music, speech, and soundscapes. Music consists of organized tones and rhythms created by humans or other living entities [90]. Speech encompasses all vocalizations produced by humans, which are used for communication and expression [57]. Soundscapes refer to an acoustic environment that is experienced

and understood by individuals in context, including natural and human-made sounds [63, 114]. These three categories encompass most of the sounds people encounter.

There are two types of soundscapes: those found in the physical world and capable of being recorded, and those that can be artificially created through methods like mathematical equations. When it comes to the latter, one can imagine sounds such as white or brown noise - which are simply repetitive mathematical patterns that can be generated with ease through computation. To the best of the author's knowledge, all remaining sounds were either recorded or generated through sampling or other creative techniques, such as capturing the behavior of vibrations [125].

Using neural networks, deep generative models can produce audio from parameters. These models learn hidden data patterns to create new samples that match the training data's distribution [61].

Despite significant advances in generative technologies, research on audio synthesis has fallen behind. Although image generation has achieved impressive levels of realism and text generation is capable of passing medical exams [121], differentiating when a DL process generates a specific sound is still relatively easy. Most of the sound-related research is focused on text-to-speech (TTS) applications, which still need further improvement. Recent developments suggest a changing landscape.

In 2023, companies have significantly increased their investments in advancing the audio synthesis capabilities for music, voice, and soundscapes. The renewed focus aims to close the gap between state-of-the-art image generation models and general sound synthesis tools. The objective is to develop advanced algorithms capable of creating highly realistic audio outputs in various domains.

1.2 Motivation

In recent years, there has been a significant increase in the use of ML techniques for audio processing tasks, such as sound synthesis, audio restoration, and speech recognition. The ability of ML algorithms to learn and extract complex patterns from large datasets has shown promising results in improving the quality and efficiency of audio processing tasks.

Furthermore, integrating ML techniques in sound generation technologies can revolutionize how one creates and experiences sound. It can provide new avenues for artists and musicians to explore their creativity and produce unique and innovative audio content. It can also offer new possibilities for sound design in various industries, such as film, gaming, and virtual reality.

The current need for studies in sound generation technologies highlights the need for further research and development. This dissertation endeavors to establish itself as a significant study in this field. It offers high-quality resources to researchers and developers to investigate the potential and limitations of ML techniques for sound synthesis.

In today's world, digital technologies are reshaping our relationship with music and sound by enabling innovative capabilities [122]. This study strives to investigate and broaden this impact by offering a novel tool that bolsters human potential in sound creation. Additionally, the findings can be a valuable resource for audio processing researchers, developers, and practitioners. The research findings offer guidance for future research and development endeavors concerning sound generation technologies and provide valuable insights into the most effective practices and techniques for utilizing ML in audio processing.

Overall, this study's significance and potential impact make it a worthwhile and valuable contribution to the field of audio processing and machine learning.

1.3 Objectives

To address the core motivations behind this research, this dissertation aims to undertake a comprehensive study of DL and, more specifically, generative deep learning models in the context of sound synthesis.

The goals include conducting a comprehensive survey of existing DL architectures for audio generation while analyzing their strengths and limitations. In addition, novel approaches will be proposed and developed to further advance the field.

By pursuing these revised objectives, this research aims to provide valuable insights into state-of-the-art in sound synthesis using DL methods. Furthermore, it aims to provide practical guidance for future advances in creating high-quality audio outputs based on textual inputs.

In order to accomplish this implementation, some specific goals are set:

1. Make a study of the current state-of-the-art deep learning architectures, focusing on generative ones.
2. Examine prior algorithms that can process sound for augmentation, feature extraction, or other purposes.
3. Make a study of the current state-of-the-art architectures used to develop sounds artificially.
4. Develop end-to-end systems that can synthesize sound from any given text input, while accounting for hardware constraints and ensuring reliable performance.
5. Evaluate the systems' ability to generate a sound from the given textual input accurately.

1.4 Dissertation Structure

The present dissertation commences with a comprehensive examination of the current state-of-the-art technologies pertaining to sound generation. The analysis encompasses sound generation and

delves into the realm of deep learning and generative deep learning architectures, which are not limited to sound. Subsequently, the dissertation examines additional tools, such as data augmentation for sound and sound analysis. Then, the focus shifts to a more in-depth study of generative deep learning technologies specific to sound, including vocoders, end-to-end tools, and other related terms, which are thoroughly explained.

The following section of the dissertation focuses on formulating and defining the problem at hand. Subsequently, practical applications of the developed technologies are demonstrated, and the dissertation investigates the various decisions and consequences that arise in developing such a system.

The methodology and approach adopted to fulfill the thesis's objectives are outlined in the solution section. An overview of the work carried out, its results, and the work plan is presented in detail.

Finally, the dissertation concludes by assessing the extent to which the objectives proposed in the introduction have been met and by presenting a summary of the findings. To facilitate navigation and ease of reference, each chapter of the dissertation includes a table of contents.

Chapter 2

Overview of the State-of-the-Art Techniques

Contents

This chapter provides an objective review of the pertinent literature on DL and audio processing. Its goal is to offer readers a comprehensive understanding of the current state of knowledge in the field and its evolution. Recent advancements in tools for sound comprehension and generation are enhancing existing models, enabling the modeling of new sounds more efficiently and rapidly [122]. Specifically, this chapter plays the following essential roles:

1. To establish the context for the research problem: By reviewing the existing literature, the chapter sets the stage for the research problem and provides a basis for understanding its significance and importance.
2. To identify gaps in the literature: The chapter helps to identify areas where further research is needed, as well as potential opportunities for contribution.
3. To provide a foundation for the research design: The chapter helps to inform the design of the research study by highlighting previous research and its limitations.
4. To demonstrate the originality of the research: By reviewing the existing literature, the chapter helps demonstrate the originality of the research problem and the thesis's contribution to the field.
5. To position the thesis within the larger context of the field: The chapter helps to position the thesis within the larger context of the field, demonstrating its relevance and significance.

The chapter is divided into two main sections: Background and Related Work. The Background section, present in 2.1, discusses previous work that is important for understanding the context of the research problem, even though it does not address the same problem as the thesis. On the other hand, the Related Work section, present in 2.2, focuses on issues that are similar or closely related to the research problem addressed in the thesis. This chapter serves as a foundation for the research problem and contributes to the overall contribution of the thesis.

2.1 Background

This dissertation is situated in a vast body of research on DL. This section positions the current dissertation within this context. This work is not intended to explain other technologies that solve similar problems, but rather to explain other technologies that are useful in addressing the problem at hand.

The dissertation aims at readers with basic ML knowledge, but not necessarily basics on DL. No basis on sound besides 8th-grade physics is needed, also.

The dissertation will start by explaining how sound can be processed digitally in subsection 2.1.1, keeping this objective in mind. Then, in subsection 2.1.2, it will provide a comprehensive review of general deep learning architectures and techniques, emphasizing their importance in sound synthesis. In subsection 2.1.3, the subsequent part will concentrate on crucial techniques used in developing the suggested models. Section 2.1.4 will discuss the deep learning frameworks used in this research. Subsection 2.1.5 will explore state-of-the-art models for data generation that are unrelated to audio but can serve as references, such as image generators, to provide further context.

2.1.1 Sound

A digital audio signal — often called a waveform — captures alterations in sound pressure over-time. This waveform represents how sound waves develop and spread throughout space. Distinct sounds that constitute our auditory experience can be perceived and interpreted through analyzing the changes in frequency and amplitude within the waveform. Waveforms encode the richness of sound, enabling us to capture and manipulate it for diverse purposes such as analysis, synthesis, and artistic expression.

Sound is a continuous phenomenon but can be discretized by taking samples at a specific time rate. The number of samples taken in one second is known as the "sampling rate." The standard value for this is 44,100 Hz. The sampling rate has a direct impact on the accuracy of the signal [34].

With the discrete version of time, it becomes possible to represent sound through an array digitally. Knowledge of the array and the sampling rate enables a computer to reconstruct the sound. These arrays can become quite large. For instance, stereo sound with a sampling rate of 44,100 Hertz (Hz) needs to accommodate 1,411,200 bits per second [34].

2.1.1.1 Short-Time Fourier Transform

Even though digital sound media can be encoded as one-dimensional data [87], it can be converted. By using a Discrete-Fourier-Transform (DFT), the array can be represented in the frequency domain [95]. Since the contents of a sound sample typically vary over time, DFTs can be computed over successive time frames of the signal. This operation forms the basis of the Short-Time Fourier Transform (STFT). The equation for STFT is as follows:

$$STFT\{x(n)\}(m,\omega) = \sum_{n=-\infty}^{\infty} x(n)w(n-mR)e^{-j\omega n} \tag{2.1}$$

In this equation, $STFT\{x(n)\}(m,\omega)$ represents the time-frequency representation of the input signal $x(n)$ as a function of time index m and frequency ω. The function is a complex-valued function containing both magnitude and phase information of the signal's frequency components at different time intervals.

The summation symbol $\sum_{n=-\infty}^{\infty}$ denotes that the product of the signal, window function, and complex exponential is summed over all time indices n. The discrete-time input signal is represented by $x(n)$, sampled at integer time indices n.

The window function, represented as $w(n-mR)$, is utilized to isolate a specific time interval of the input signal. The window function is centered at the time index mR, where R is the hop or step size between consecutive sample windows.

Finally, the complex exponential term $e^{-j\omega n}$ is utilized to analyze the signal's frequency content within the windowed interval. The variable j is the imaginary unit, and ω represents the angular frequency.

In essence, the STFT equation computes the Fourier Transform of the input signal $x(n)$ within a windowed time interval, providing a time-frequency representation of the signal. The window function isolates a specific time interval of the signal, and the complex exponential term analyzes its frequency content. This process is repeated for different time indices m, resulting in a time-frequency representation that allows the study of the signal's frequency components at various time intervals.

With the STFT, one can generate spectrograms by plotting the time in the x axis and the frequency in the y axis. In short, a spectrogram is a graphical representation of the frequency content of a signal over time, typically displayed as a 2D image (see Figure 2.1 for an example).

2.1.1.2 Meaning of Spectrograms for Machine Learning

Representing sound as an image opens a multitude of opportunities. However, even though spectrograms can technically be processed using convolutional neural networks (CNNs) (see section 2.1.2.1), there is a considerable difference between a spectrogram and a standard image. In a typical image, the axes represent the same concept, the spatial position. The elements of an actual image have the same meaning independent of where they are found. A sub-object of an image does not depend on the axes. At the same time, neighbor pixels are usually highly correlated.

On the other hand, the axes of the spectrograms have different meanings [95]. Moving a set of pixels horizontally and vertically means different things. Therefore, structures such as CNN are not as helpful. One can still use them but should be careful about the shape of the filters and the axis along which the convolution is performed [95].

2.1.1.3 Soundscapes

The digitalization of sound has allowed for multiple applications and use cases. Applications can generate speech, and movies and videos can embed audio, such as soundscapes. Soundscapes are the sonic environments or sound environments that surround environments. They are the complex and dynamic mix of sounds heard in everyday life, including sounds from nature, human-made, and cultural sounds [63, 114]. In other words, a soundscape encompasses the auditory milieu characterized by a collection of naturally occurring and human-generated sounds as perceived, encountered, and comprehended within a contextual framework by individuals. It is paramount in audio content creation, augmenting the user experience across media applications by infusing emotional engagement, a greater sense of immersion, and attention [16].

Nevertheless, for audio media generation with ML, one usually finds models in the literature that solve music or speech generation, not soundscapes. This is no coincidence. These sounds are more

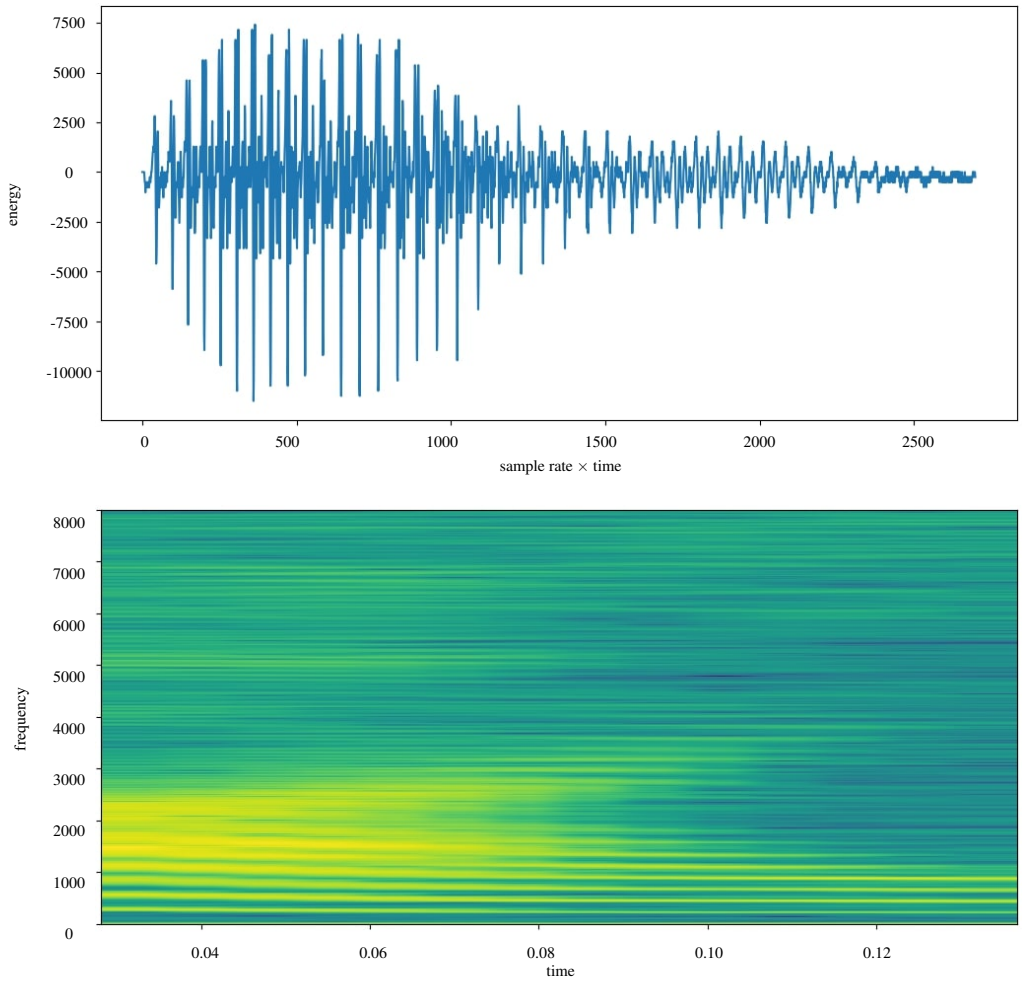


Figure 2.1: **Raw Wave vs. Spectrogram** — A comparative analysis of a sound sample from the Audio MNIST dataset (showcased in section 4.2.1.2), specifically entry number 9 of speaker Nicolas uttering the digit "five". The top plot illustrates the raw waveform with time (sample rate × time) on the X-axis and energy (amplitude) on the Y-axis, providing a temporal representation of the audio signal. The bottom plot presents a spectrogram generated using the STFT method, displaying time on the X-axis and frequency on the Y-axis, offering a time-frequency representation that reveals the spectral content and evolution of the signal over time.

straightforward and, thus, easier to generate. Speech, for instance, usually contains a single sound source (the speaker). Also, speech and music are highly structured over time and timbrically. This happens because speech is bound to grammar, and music is bound to an underlying structure. Both of them are timbrically bound to their authors. On the contrary, soundscapes have no specific structure. Hence the increased difficulty [95].

2.1.2 Deep Learning

As previously mentioned, the growth of deep learning (DL) began in the early 2000s as a response to the challenge of handling vast quantities of data. Fundamentally, DL stems from the

application of ML to process large amounts of data. To be precise, DL is a subfield of ML that employs multiple levels of information processing and abstraction to learn and represent features, as demonstrated by Deng et al. [27]. Subsequently, the extracted features can be utilized for classification, regression, and other modeling techniques. In the past, such features were manually selected by humans.

This study uses DL for sound generation because it offers several advantages over traditional sound generation techniques. By being data-driven, these models can generate new sounds based on sounds it has heard before. On traditional methods, these sounds would have to come from the inspiration of their human creator. Besides, end-to-end generation, from text to sound generation, is only possible through DL. The model has to extract features from the text, learn features from thousands or millions of sounds, and correlate both. This highly complex task can only be achieved with DL techniques.

This section presents traditional DL architectures and their evolution to generative DL architectures.

2.1.2.1 Deep Learning Architectures

Generative DL architectures establish blueprints for developing DL networks that synthesize diverse and novel data samples according to a learned distribution. These architectures entail creating latent data constructs and learning to emulate the fundamental statistical patterns found in observed data.

Generative deep neural models have been applied to tasks comprising image synthesis, text generation, and audio synthesis. Their popularity has recently surged owing to their remarkable ability to generate high-quality data and effectively model complex distributions. In the following sections, we outline the most ordinarily used generative DL architectures, presented chronologically, as summarized in Table 2.1.

This section will discuss novel architectural designs for DL. It is important to note that there is a discrepancy between the date of their conceptualization and their widespread adoption. This trend is common in machine learning because software theories have a faster rate of development compared to hardware theories.

This section does not describe all DL architectures that had some relevance. It, however, describes the ones used either by the models developed or by systems studied for the state-of-the-art.

Feedforward Neural Network (1957) To understand *feedforward neural networks* (or any neural network), it is essential to understand both Hebbian learning and the perceptron.

In 1949, inspired by the observation that neurons that fire together wire together, the psychologist Donald Hebb [52] developed a rule for adjusting the strength of connections between neurons in any neural network. The rule goes by the name of Hebbian learning and states that if two neurons

Table 2.1: Comparison of Generative Deep Learning Architectures

Model	**Year**	**Type**	**Key Characteristics**	**Inference**
DARN	2013	Autoregressive	Uses a single model to predict the probability distribution of each output token conditioned on the previous tokens	Sequential
VAE	2013	Variational Autoencoder	Learns a latent representation of the input data and generates new samples by sampling from the learned latent space	Parallel
GAN	2014	Generative Adversarial Network	Consists of a generator and a discriminator that compete in a two-player minimax game to generate realistic samples	Parallel
Normalizing Flows	2015	Flow-based models	Transforms a simple probability distribution into a complex one by applying a sequence of invertible transformations	Parallel
Diffusion	2015	Flow-based models	Uses a diffusion process to model the probability distribution of the data	Parallel
Transformers	2017	Attention-based models	Uses self-attention to capture global dependencies and generate sequences	Sequential
VQ-VAE	2018	Variational Autoencoder	Discretizes the continuous latent space by mapping each latent vector to the closest codebook vector	Parallel
MS-VQ-VAE	2019	Variational Autoencoder	Is an extension of the VQ-VAE that incorporates multiple discrete latent spaces of different scales, enabling hierarchical and diverse representations with improved abstraction levels and latent space expressiveness	Parallel

are activated simultaneously, their connection should be strengthened. The idea behind Hebbian learning is that learning occurs as a result of changes in the strengths of synaptic connections between neurons in the brain rather than solely due to changes in the activity of individual neurons.

The perceptron is a simple model first published in 1958 by F. Rosenblatt [108]. It consists of a single neuron with a binary threshold — also known as bias — and is used for binary classification tasks. A set of weights transforms the input to the perceptron. The output is determined by checking if the input is enough to trigger the bias and then applying a function to the weighted sum of inputs as shown in Figure 2.2. The output of the perceptron can be mathematically expressed as in the equation 2.2.

$$y = h(\sum_{x=1}^{N} (I_x \times W_x) - b) \tag{2.2}$$

y is the output, N is the number of inputs, I_n is the input number n, W_n is the weight related to I_n, b is the bias, and h is an activation function, usually.

The perceptron is trained using an algorithm that adjusts the weights based on the error between the actual and predicted outputs. The exciting part is that Hebbian learning can be applied to perceptrons, which was one of the bases of the future backpropagation. The perceptron can only classify linear separated sets, as shown in *Perceptrons* [79], hence the need for more complex structures.

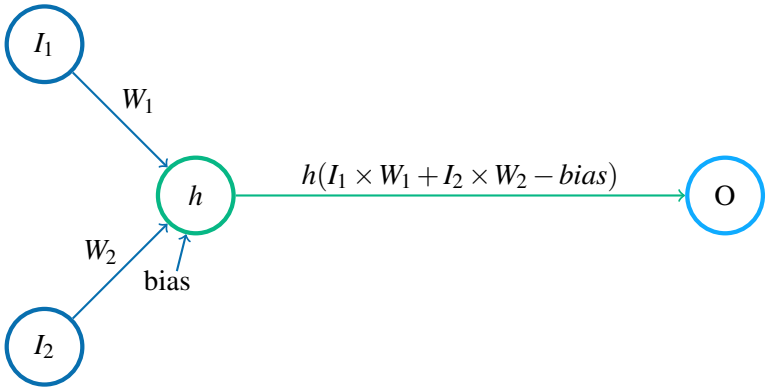


Figure 2.2: **Perceptron** — The perceptron receives multiple inputs associated with a weight. It also holds a given bias. Its output is the application of a given function (*e.g.* step function) to the weighted average of the inputs minus the bias.

A feedforward neural network is no more than a set of layers of perceptron-like neurons. In a feedforward neural network, information moves in only one direction — forward. Hebbian learning no longer applies to these networks, so more complex algorithms, such as backpropagation, are required. These networks are universal approximators [23] if they have at least one hidden layer, meaning they can approach any function given the proper configuration. They can also learn different tasks than classification, such as regression. The first functional networks, called multi-layer perceptrons, were invented in the 1980s. An example structure of a feedforward neural network can be seen in Figure 2.3.

To a certain extent, since the perceptron is the most basic feedforward neural network (consisting of only one neuron), these networks do not necessarily equate to deep learning (DL). Smaller networks have existed and been utilized for decades. Nevertheless, the foundation for most DL architectures comes from this, making it essential to recognize.

Convolutional Neural Network (CNN) (1980) During the 1950s and 1960s, Hubel and Wiesel [60], who were distinguished neurophysiologists, conducted pioneering research on the eyes of cats. Their research uncovered the presence of neurons with receptive fields that map various visual regions. Receptive fields refer to specific areas in the visual field that activate particular cells. Their studies revealed two fundamental categories of cells involved in visual processing: simple cells

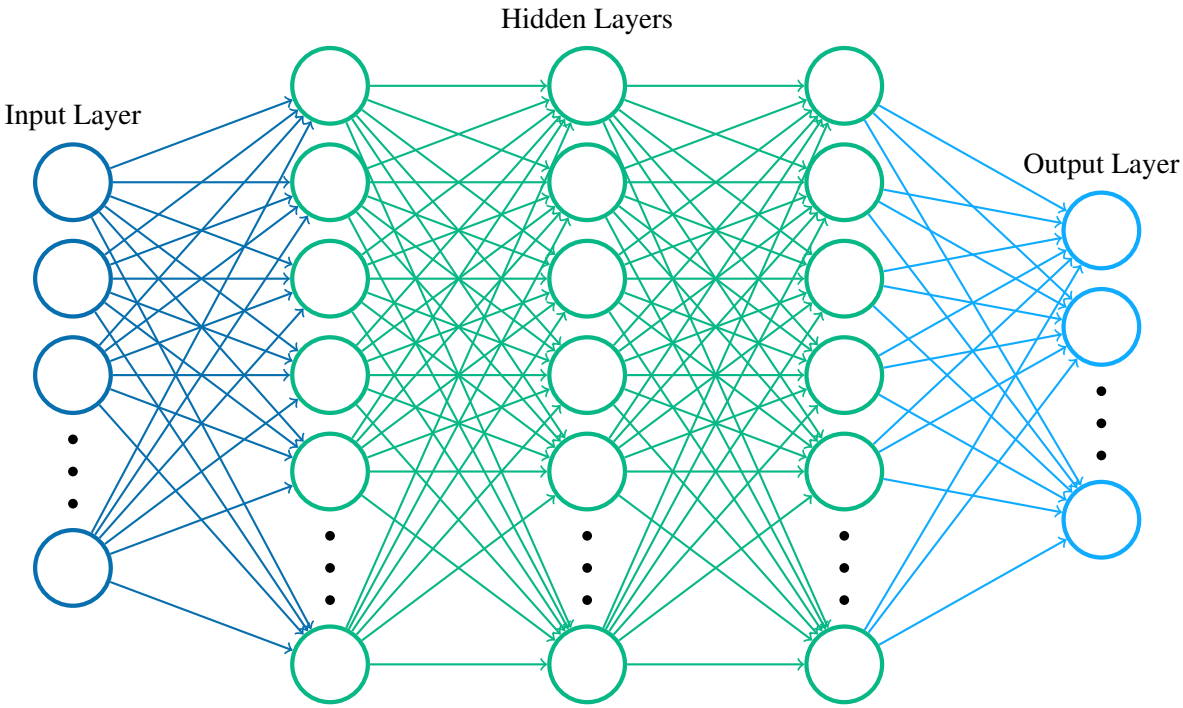


Figure 2.3: **Feedforward neural network** — The size of the input is constant. The flow of information goes through nonlinear functions on hidden layers. Both weights between nodes on different layers and the nodes' bias are trained with backpropagation.

with a strong response to edges and complex cells with larger receptive fields that prioritize the organization of edges over their precise positioning.

In 1980, Kinihiko Fukushima introduced the *neocognitron* which marked a significant milestone in the development of CNNs [42]. It was one of the first CNN architectures designed to mimic specific aspects of human visual perception. This laid the groundwork for future advances in deep learning for computer vision tasks.

CNNs are a category of DL neural networks widely used in computer vision and other sequential data types. The architecture of CNNs is specifically designed to handle inputs where data correlate with its vicinity.

The key idea behind CNNs is to learn and extract features from the input hierarchically. This is achieved through convolutional layers, where a small set of learnable filters are used to scan and transform the input image into a feature map. These networks also use pooling layers that perform down-sampling of the feature map to reduce its size while retaining important information. These operations allow the network to capture patterns and features in the input, which can then be passed to feedforward neural networks and used for classification or regression. This process is represented in Figure 2.4.

A major benefit of CNNs compared to other kinds of networks is their ability for parallelism, where a long input sequence can be handled fast [61]. This can greatly accelerate training since

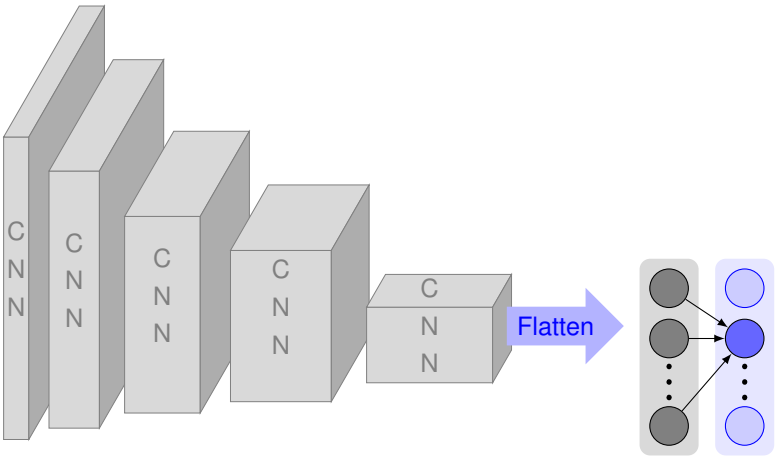


Figure 2.4: **Convolutional neural network (CNN)** — The network present in the Figure deals with 2D data, such as images. The networks' first step consists of convolutional (and pooling) layers. This first step is responsible for finding features. The deeper the convolutional layer, the bigger the receptive field; consequently, the more abstract the feature. After the features are caught, the flatten operation transforms the data into a 1D vector. And then, the network is no more than a feedforward neural network to, in this case, classify what was fed to the network.

the whole output can be processed in one forward pass. The shared weights and local connectivity among neurons in the network enable efficient computation, and using pooling layers lowers the number of parameters to be learned.

Convolutional Operations This section will discuss three common layers in CNNs: convolutional layers, pooling layers, and transposed convolutions. It also introduces the concept of masked convolutions and their applications in specific tasks.

A **convolutional layer** is a fundamental building block of CNNs (see Section 2.1.2.1).

At a high level, a convolutional layer applies a set of learnable filters to the input data, extracting local features and patterns relevant to the task. The filters are typically small and slide over the input data, computing the dot product between the filter weights and the input values at each position. This operation is called convolution and can be seen in Figure 2.5.

To be more precise, let us consider a 1D convolutional layer that takes an input tensor X of size $C_{in} \times L_{in}$, where C_{in} is the number of input channels and L_{in} is the length of the input sequence. The layer also has C_{out} filters, each of size $C_{in} \times k$, where k is the size of the filter kernel. The output of the layer is a tensor Y of size $C_{out} \times L_{out}$, where L_{out} is the length of the output sequence.

The convolution operation can be expressed as follows:

$$Y_{c,i} = \sum_{p=0}^{k-1} \sum_{k=0}^{C_{in}-1} X_{k,i+p} \cdot W_{c,k,p} + b_c \tag{2.3}$$

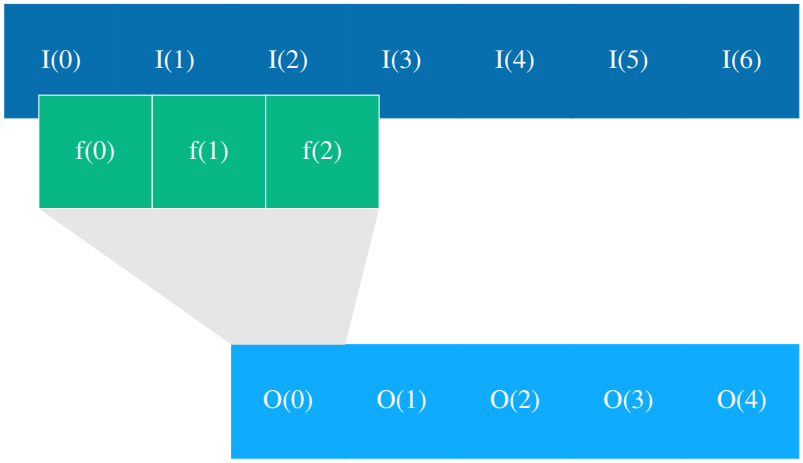


Figure 2.5: **Convolutional layer** — The presented diagram depicts a 1-dimensional convolutional layer designed to process an input signal consisting of a single channel and a length of 7. The layer employs a single filter with a size of 3, resulting in an output signal of length 5 and a single channel. Each element of the output signal is produced by convolving the corresponding filter weights with a subset of the input signal. Specifically, the first output element is generated by convolving the filter with the first three elements of the input signal. The filter is then slid along the input signal with a stride of 1, such that each subsequent output element depends on a different subset of the input signal.

where c is the index of the output channel, i is the spatial index of the output sequence, p is the spatial index of the filter kernel, k is the index of the input channel, $X_{k,i+p}$ is the input value at position $(k, i+p)$, $W_{c,k,p}$ is the weight of the filter at position (c,k,p), and b_c is the bias term for the c-th output channel.

The convolution operation is applied independently to each output channel and each spatial location of the output feature map, resulting in a set of feature maps that capture different aspects of the input data. The weights and biases of the filters are learned during training using backpropagation and gradient descent (see section 2.1.2.2), optimizing a suitable loss function for the task at hand.

Pooling is a standard operation in CNNs that reduces the spatial dimensions of the input feature maps while retaining important information. Pooling is typically applied after convolutional layers to gradually reduce the feature maps' spatial dimensions and increase the network's receptive field. There are several types of pooling, including max pooling, average pooling, L2 pooling, and more. Of these, max pooling is one of the most commonly used.

Max pooling works by partitioning the input feature map into non-overlapping rectangular regions, called pooling kernels. For each pooling kernel, the maximum value is selected and used as the output value for that region. The size of the pooling kernel and the stride determine the amount of reduction in the spatial dimensions of the feature map. The stride is the number of pixels that the pooling kernel is shifted horizontally and vertically between each pooling operation. It can be seen in the Figure 2.6

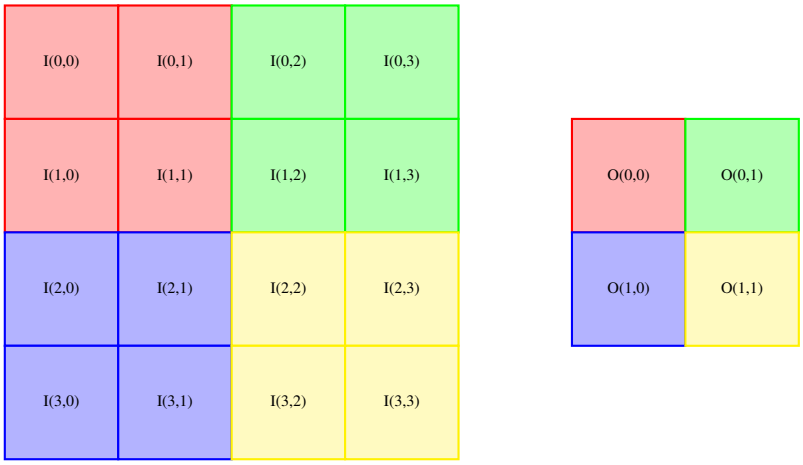


Figure 2.6: **Pooling layer** — The Figure illustrates a 4×4 input tensor with a single channel. Each colored square in the tensor denotes a single element. The pooling layer partitions the input tensor into non-overlapping 2×2 regions and applies a pooling function to each region, resulting in a 2×2 output tensor with a single channel. The number of channels remains unchanged in this scenario, implying that the pooling function is applied separately to each channel of the input tensor, and the resulting output tensor retains the same number of channels as the input tensor. The primary objective of pooling layers is to decrease the spatial dimensions of the input tensor while maintaining the essential features. In this instance, the pooling layer decreases the spatial dimensions of the input tensor by half, resulting in a smaller output tensor. The output tensor is displayed on the right-hand side of the image, where each colored square represents a single element of the output tensor. The colors of the output tensor correspond to those of the input tensor regions from which they were derived.

Max pooling has several benefits for CNNs [105]:

- It helps to reduce the model's sensitivity to minor variations in the input. This is because the maximum value within each pooling kernel is selected, which is less sensitive to slight variations than taking the average or other statistics.
- It can prevent overfitting by reducing the number of parameters in the model. This is because the pooling operation reduces the spatial dimensions of the feature map, reducing the number of parameters in the subsequent layers of the network.
- It can increase the network's receptive field by combining the information from neighboring pixels.

There are some potential drawbacks to max pooling as well. One is that it can discard some information that may be important for the task. This is because only the maximum value within

each pooling kernel is selected, and other information is discarded.

A **transposed convolution**, also known as a deconvolution, is a layer that performs the opposite operation of a convolutional layer. It takes an input tensor of size $C_{in} \times L_{in}$ and produces an output tensor of size $C_{out} \times L_{out}$, where C_{in} and C_{out} are the number of input and output channels, respectively, and L_{in} and L_{out} are the input and output lengths.

The transposed convolution applies a set of learnable filters to the input data. However, instead of sliding the filters over the input, it slides them over the output and computes the dot product between the filter weights and the output values at each position. This operation can be seen as an "unfolding" of the convolution operation. Hence the name "transposed convolution". An example can be seen in Figure 2.7.

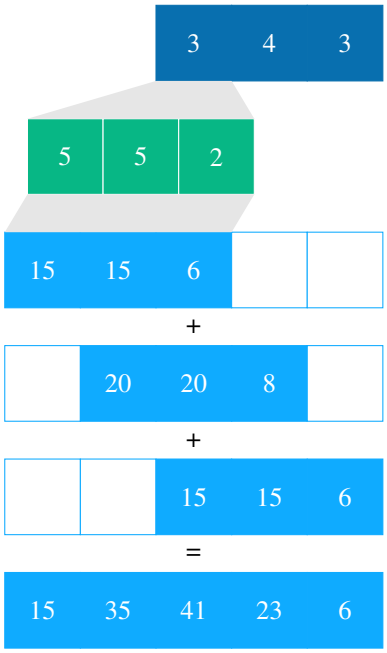


Figure 2.7: **Transposed convolution** — This illustration shows how a transposed convolution operation works. The input array has three elements $(3,4,3)$ and is shown by the dark blue rectangles at the top. The filter array also has three elements $(5,5,2)$ and is shown by the green rectangles in the middle. The output array is obtained by sliding the filter over the input array and computing the element-wise products and sums. The stride parameter controls how much the filter moves, which is 1 in this example. The light blue rectangles at the bottom show the intermediate and final output arrays. The intermediate output arrays are $(15,15,6,0,0)$, $(0,20,20,8,0)$, and $(0,0,15,15,6)$. The final output array is the sum of the intermediate output arrays $(15,35,41,23,6)$. The gray-shaded regions indicate how the filter broadcasts each element in the input array to produce an intermediate output array.

The transposed convolution can be expressed as follows:

$$X_{k,i} = \sum_{p=0}^{k-1} \sum_{c=0}^{C_{out}-1} Y_{c,i+p} \cdot W_{c,k,p} + b_k \tag{2.4}$$

where k is the index of the input channel, i is the spatial index of the input sequence, p is the spatial index of the filter kernel, c is the index of the output channel, $Y_{c,i+p}$ is the output value at position $(c, i+p)$, $W_{c,k,p}$ is the weight of the filter at position (c,k,p), and b_k is the bias term for the k-th input channel.

Like convolutional layers, the weights and biases of the filters are learned during training using backpropagation and gradient descent.

A **masked convolution** is a convolutional layer that selectively masks out specific input values based on their position in the input sequence. This masking is typically done by setting the weights of the filters to zero for certain positions in the kernel. For example, in an autoregressive (AR) language modeling task, where the goal is to predict the next word in a sentence given the previous words, a masked convolution can ensure that the model only has access to the previous, not the future words.

Another example where masked convolutions can be helpful is in audio generation tasks. In such tasks, the model takes as input a sequence of audio samples and generates a new sequence of samples that sound similar to the input. In some cases, generating audio that only depends on the previous time steps rather than the entire input sequence may be desirable. This can be achieved using a masked convolution that masks out the future time steps by setting the filter weights to zero for those positions in the kernel. By doing so, the model can only rely on the previous time steps to generate the following sample.

Masked convolutions can be implemented using standard convolutional layers with appropriate masking of the filter weights. Another approach is using a specialized masked convolutional layer that takes an additional mask tensor as input, indicating which input values should be masked.

One advantage of masked convolutions is that they can help prevent the model from overfitting to the training data by forcing it to rely on the input data available at each time step rather than the ground truth values that may not be available at test time. This can be particularly useful in tasks where the input data has a sequential or temporal structure, and the model needs to make predictions based on partial information.

Recurrent Neural Network (RNN) (1986) In 1986, Rumelhart and McClelland, psychologists, in the aftermath of the discovery of the backpropagation algorithm, wrote a book [109] where, between others, they took the declarations made in Perceptrons [79], presented in Section 1.1, and showed that, although the perceptron alone might not be enough, networks of perceptrons, with the suitable algorithms, might suffice.

One example they gave was a theoretical network that would be recursive [111]. This would be the first time the "recurrent" term would appear in the literature. They explained that a recursive network is a feedforward network where some weights are kept the same between layers. They showed that it is easy to backpropagate the error and train such a network with this simplification.

At the time, "the major problem with this procedure is the memory required". This new network was able to solve problems related to sequences. In this chapter, they showed the development of a system that would learn to complete straightforward sentences using recurrent neural networks.

Furthermore, a recurrent neural network (RNN) used nowadays is not very different from the one theorized in 1986. A RNN is a neural network derived from a feedforward neural network (see Section 2.1.2.1), where some connections between nodes can create cycles. This is because some nodes' output is the same nodes' input. This operation makes the network have memory. These networks can exhibit temporal behavior and process variable-length sequence of inputs. In 1996 it was shown that RNNs could perform any operation that a regular computer can, meaning that they are Turing complete [62].

RNN have the problem of forgetting terms seen long ago as new terms come along [55]. This may be a problem in, for instance, long texts where the model has to remember the name of a given entity that was given at the beginning of the text.

Learning happens as in feedforward neural networks with one additional step: the recurrent nodes are unrolled, as seen in Figure 2.8.

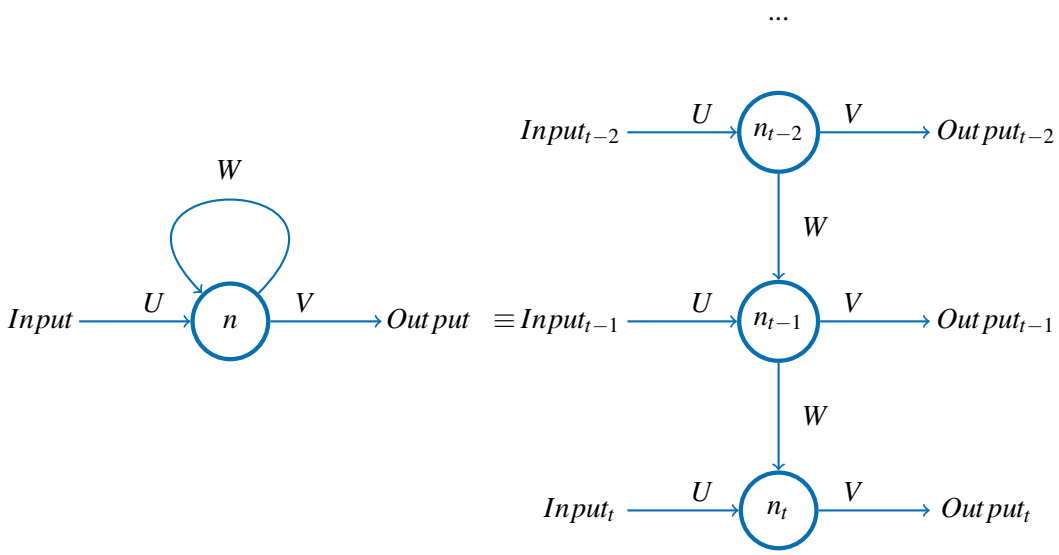


Figure 2.8: **Simple recurrent neural network** — The left side of the image displays the most straightforward representation of a RNN because it only has one recurrent neuron. Apart from the bias, the neuron has an input weight, a weight for the recursive connection, and a weight for the output. The images on the left and the right are equivalent. The image on the right represents the same simple network. However, it shows what exactly happens when it receives multiple inputs: an output is generated for each one, and a value is passed to the next iteration. One may notice that even though the inputs and outputs change, the weights on the edges do not.

New takes that solve the problems with RNNs have been developed over the years. The basis of

these models is to include learnable gates that choose what and when to remember and to forget. The most prominent examples are the Gated recurrent units (GRUs) [18], and the Long short-term memorys (LSTMs) [55].

RNN Variants RNNs are a class of neural networks designed to handle sequential data. They work by maintaining an internal state or memory, which allows them to process sequences of inputs and produce outputs that depend on previous inputs. This definition allows for other types of networks, different from the one shown in the previous section.

However, there are convincing reasons to investigate architectures beyond the basic approach. The typical RNN is not without its difficulties, particularly the development of the *vanishing gradient problem* as a major concern.

RNNs may be prone to the vanishing gradient problem due to their method of updating internal states. RNNs compute the internal state at time step t by incorporating the input of time step t and the previous state of time step $t-1$. This process generates a sequence of dependencies between the present and past states, which leads to multiple matrix multiplications during the backpropagation.

As the gradient keeps decreasing after each multiplication, it may ultimately shrink to such an extent that the neural network fails to learn long-term patterns present in the input sequence.

The prime example of an architecture that fights the vanishing gradient problem is the **long short-term memory (LSTM)** [55]. LSTMs were introduced in 1997 and have since become a popular choice for processing sequential data, especially in natural language processing (NLP).

The main idea behind LSTM is to introduce memory cells that can selectively forget or remember information from previous time steps. Each memory cell has three gates: the input gate, the forget gate, and the output gate, which control the flow of information into and out of the cell. One can see how these work in Figure 2.9.

The forget gate determines which information from the previous state should be forgotten or retained. It takes the current input, X_t, and the previous state, H_{t-1}, and computes the gate activation, f_t, a number between 0 and 1 that determines how much of the previous state should be retained or forgotten. It is calculated as

$$f_t = \sigma(W_f * [H_{t-1}, X_t] + b_f) \tag{2.5}$$

W_f is the weight matrix for the forget gate, and b_f is the bias vector.

The input gate determines which information from the current input and the previous state should be allowed into the cell. It takes the current input, X_t, and the previous state, H_{t-1}, as inputs and

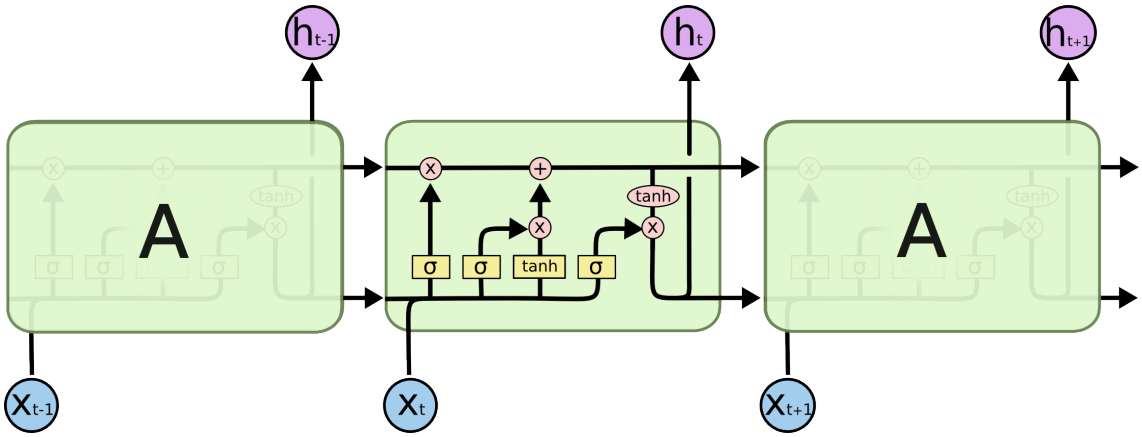


Figure 2.9: **Long short-term memory (LSTM)** — The network was taken from [21]. Each green rectangle is an LSTM cell. The blue circles represent an input at a given time, such that X_t is the input at time t. The pink circles represent the output at a given time h_t. Each LSTM cell takes three inputs: the previous cell's memory and output, and the current input, and outputs both the real output and the current memory. In the Figure, the top exiting arrow corresponds to the memory, while the bottom corresponds to the output. The yellow rectangles have learnable parameters, weights, and biases.

computes the gate activation, i_t, which is a number between 0 and 1 that determines how much of the input and previous state should be allowed into the cell. It is calculated as

$$i_t = \sigma(W_i * [H_{t-1}, X_t] + b_i) \tag{2.6}$$

where σ is the sigmoid activation function, W_i is the weight matrix for the input gate, and b_i is the bias vector.

The memory update computes the new information stored in the memory cell. It takes the current input, X_t, and the previous state, H_{t-1}, as inputs and computes the new candidate memory content, $\tilde{C}_t$. It is calculated as

$$\tilde{C}_t = \tanh(W_C \times [H_{t-1}, X_t] + b_C) \tag{2.7}$$

W_C is the weight matrix for the candidate memory update, and b_C is the bias vector.

The memory cell stores the current memory content, C_t, a combination of the previous and new candidate memory content, as determined by the input and forget gates. It is calculated as

$$C_t = f_t \times C_{t-1} + i_t * \tilde{C}_t \tag{2.8}$$

The output gate determines which information from the current memory cell should be used as output. It takes the current input, X_t, and the previous state, H_{t-1}, as inputs and computes the gate activation, o_t, a number between 0 and 1 that determines how much of the memory cell content should be outputted. It is calculated

$$o_t = \sigma(W_o \times [H_{t-1}, X_t] + b_o) \tag{2.9}$$

The hidden state, H_t, is computed by applying the output gate to the memory cell. It is calculated as

$$H_t = o_t \times \tanh(C_t) \tag{2.10}$$

By selectively forgetting or remembering information from previous time steps, LSTMs can maintain long-term dependencies in the input sequence and avoid the vanishing gradient problem that occurs in vanilla RNNs.

Other kinds of networks that handle the vanishing gradient problem have been proposed. One example is the **gated recurrent unit (GRU)** [19]. GRU is very similar to LSTM. The main difference between GRU and LSTM is in their architecture and the number of gates they use to control the flow of information. While the LSTM has the three gates mentioned above, GRU has two gates: the reset gate and the update gate. The update gate controls how much of the previous hidden state to keep, and the reset gate determines how much of the previous hidden state to forget.

Overall, GRU has a simpler architecture compared to LSTM, which makes it faster to train and requires fewer parameters. However, LSTM is generally considered more powerful and better suited for tasks that require longer-term memory, such as machine translation or speech recognition.

Another widespread helpful RNN implementation is the **bidirectional RNN** [115]. Unlike traditional RNNs, which process input sequences in only one direction, from beginning to end, bidirectional RNNs process input sequences in both directions, from beginning to end and from end to beginning, simultaneously.

The main idea behind bidirectional RNNs is to use two separate RNNs, one that reads the input sequence in the forward direction and another that reads the sequence in the backward direction. The output of the two RNNs are then combined to produce the final output.

The benefit of using a bidirectional RNN is that it allows the network to access both past and future context when making predictions about the current time step.

Autoencoder (AE) Autoencoders (AEs) are a type of neural network architecture used for unsupervised learning. Their origin is difficult to precise as the literature is vast and multiple representations of this kind of network started popping up at the end of the 1980s with different names.

The primary goal of AEs is to learn an efficient representation of the input data by encoding it into a lower dimensional space, known as the bottleneck layer, and then decoding it back to the original dimensions. AEs try to learn the identity function. The network learns to minimize the reconstruction error between the input and the reconstructed output.

The architecture of an AE typically consists of the encoder and the decoder. The encoder maps the input to the bottleneck layer, while the decoder maps the bottleneck layer back to the original dimensions. These layers can be seen in figure 2.10. The bottleneck layer acts as a bottleneck that restricts the amount of information that can be passed through, forcing the network to learn the most important features of the input data. For instance, if the size of the bottleneck layer were the same as the input and the output, the network would not learn the essential features, as a simple pass-through would suffice.

A simple use case for AEs is embeddings, for instance, word embeddings. If given multiple sentences, a model learns to transform a word in itself, passing it through a bottleneck layer; in the future, only the values in the bottleneck (the embedding) and the decoder are required to get the original word. Since, ideally, these embeddings are more feature rich than the word itself, these representations are beneficial for NLP tasks.

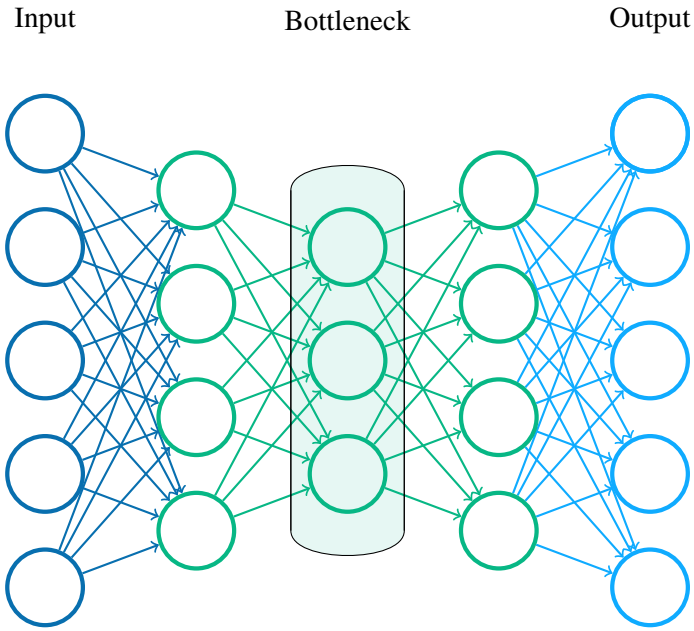


Figure 2.10: **Autoencoder (AE)** — This AE receives an input of size five and tries to learn a way to transform it into itself by passing through a bottleneck of size three. In the initial part, from the input to the bottleneck, an encoder is present, while the second part displays a decoder.

U-Net (2015) U-Net is a deep learning architecture introduced by Olaf Ronneberger et al. in 2015 for biomedical image segmentation tasks [107]. The name *U-Net* comes from the shape of

the network, which resembles the letter *U*.

The U-Net architecture consists of two main parts: an encoder path and a decoder path. The encoder path is a typical CNN (Section 2.1.2.1) that extracts features from the input image. On the other hand, the decoder path uses upsampling operations to recover the spatial resolution of the feature maps and generate a segmentation mask. Upsampling is the process of increasing the resolution of an image or signal by using, for instance, nearest neighbor interpolation, bilinear interpolation, or transposed convolution (see Section 2.1.2.1). The U-Net architecture can be seen in Figure 2.11.

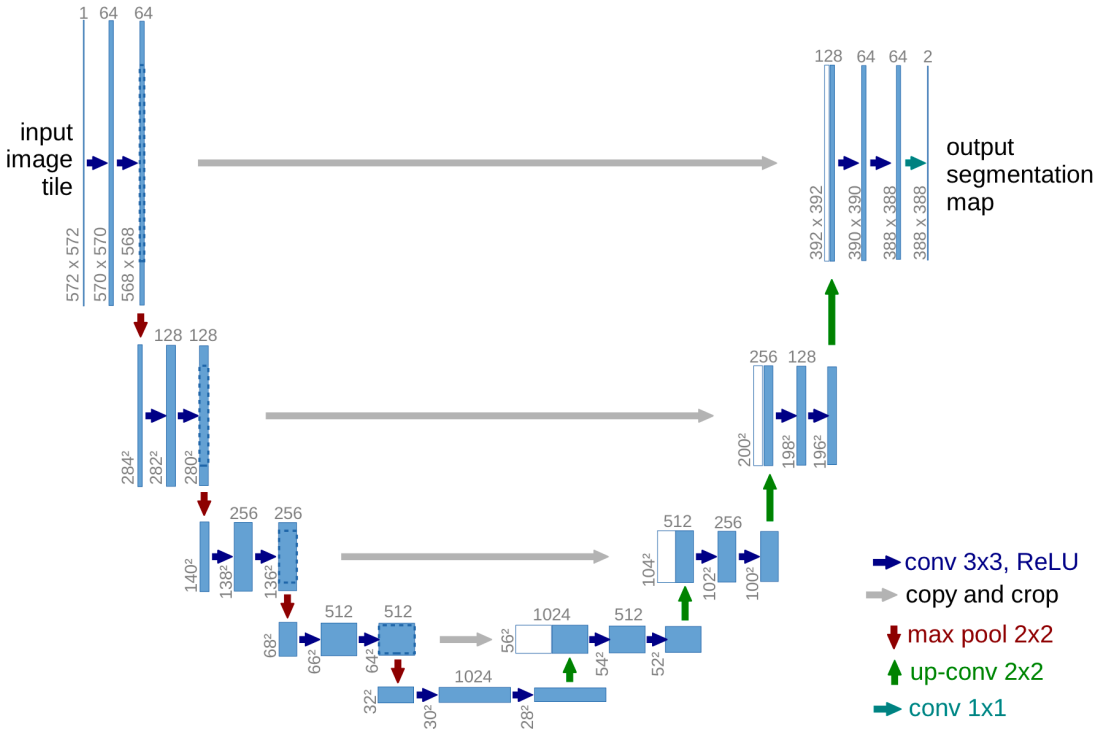


Figure 2.11: **U-Net** — This figure was taken from the original paper and follows an example for images with 572×572 pixels. Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations. One can see that at its smallest size, the feature maps were $28 \times 28 \times 1024$, and the end result provides two features maps of 388×388 pixels, meaning that this specific network could be used, for instance, for segmentation of foreground vs. background. It is also essential to notice that, to keep the original image's fidelity, there is a deconvolutional step for each convolutional one. These are concatenated (represented by the white boxes).

The encoder path extracts features from the input image. These features are then compressed into a lower-dimensional representation, which the decoder path uses to generate a segmentation mask for the input image. In this sense, the U-Net architecture can be viewed as a specialized type of AE not designed to reconstruct the input image itself.

The U-Net architecture was initially designed to classify each pixel in an image as belonging to a specific object or background: image segmentation. It combines high-level and low-level features from the input image to generate the final segmentation.

2.1.2.2 Foundations of Deep Learning

This section explores the fundamental principles of DL, with a specific focus on activation functions and backpropagation. Activation functions play a vital role in neural networks by introducing non-linearity and enabling the representation of complex relationships. This section covers the most commonly used activation functions. In addition, this section discusses the backpropagation algorithm, which plays a crucial role in training feedforward neural networks by calculating gradients with respect to network weights for iterative adjustments and optimal model performance. Furthermore, this section explores optimization techniques within deep learning.

Activation Functions Activation functions are essential to neural networks, as they introduce non-linearity into the model. Nonlinearity is important because it allows the model to learn complex relationships between inputs and outputs. This section will discuss some of the most commonly used activation functions, including sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU).

The **sigmoid** activation function is prevalent in neural networks. It is defined as:

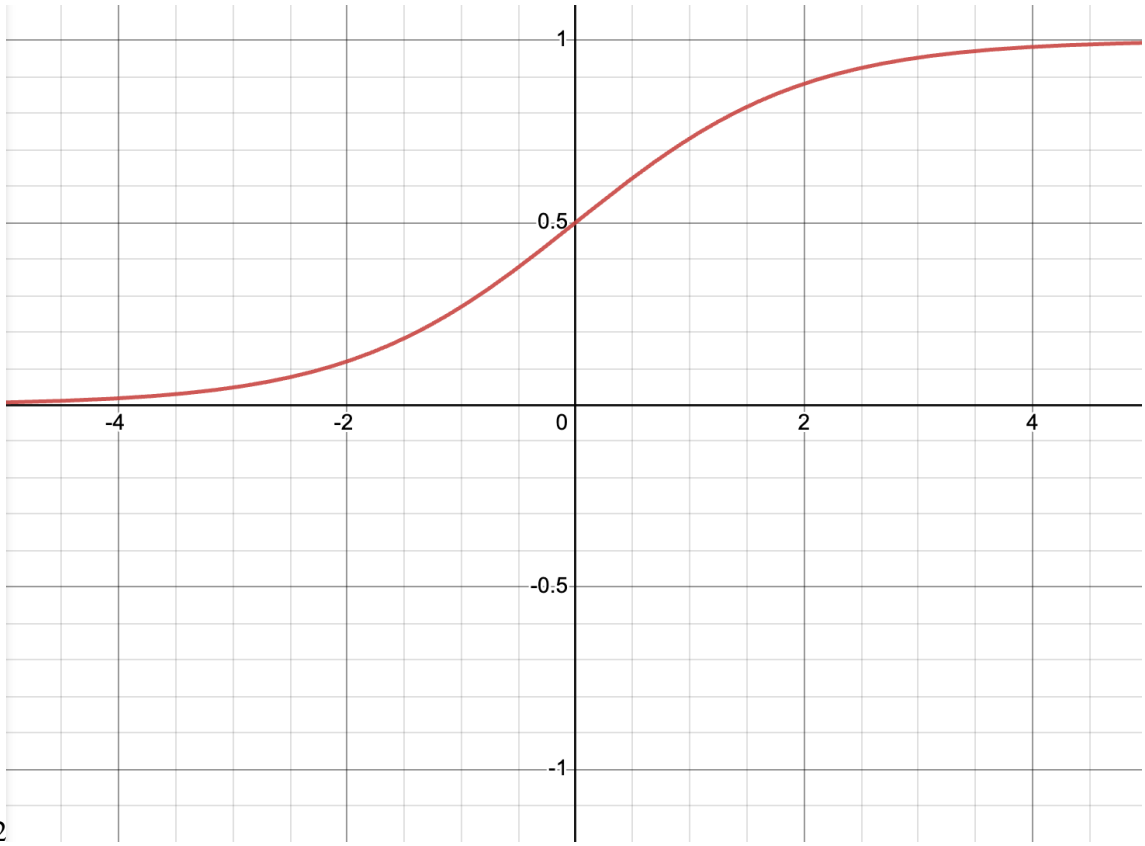
$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.11}$$

where x is the input to the function. The output of the sigmoid function is always between 0 and 1, which makes it useful for binary classification tasks. The sigmoid function is also differentiable, which is essential for backpropagation during training. Figure 2.12 visually represents this function.

However, the sigmoid function has a few drawbacks. One issue is that the function's gradient approaches zero as the input becomes very large or very small. This can cause the weights to update very slowly during training, a problem known as the *vanishing gradient* problem. Additionally, the output of the sigmoid function is not zero-centered, which can make optimization more difficult. On the other hand, if the gradients become extremely large, it can cause the weights to update too much in each iteration, leading to the *gradient explosion problem*. This can lead to the model diverging and failing to converge to a good solution.

The **hyperbolic tangent (tanh)** activation function is similar to the sigmoid function, but its output is between -1 and 1:

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.12}$$



/2/2

Figure 2.12: One can see that the values that are output are exclusively between 0 and 1 with its value being 0.5 for $x = 0$.

Tanh is differentiable and valuable for binary classification tasks like the sigmoid function. However, it has the same vanishing gradient problem as the sigmoid function. A visual representation can be seen in Figure 2.13.

The **rectified linear unit (ReLU)** activation function is a popular choice in DL. It is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2.13}$$

The ReLU function is also zero-centered, which can make optimization easier. However, this function is not differentiable at $x = 0$, which can cause problems during training. Variants of the ReLU function have been proposed to address this issue. A visual representation can be seen in Figure 2.14.

The ReLU activation has a potential problem known as the *dying ReLU* problem. When a neuron's weights become such that its input is always negative, its gradient will always be zero. In this situation, the neuron will become inactive, or *dead*, and will no longer update during training. This problem can lead to underfitting, as some neurons stop learning and contributing to the model.

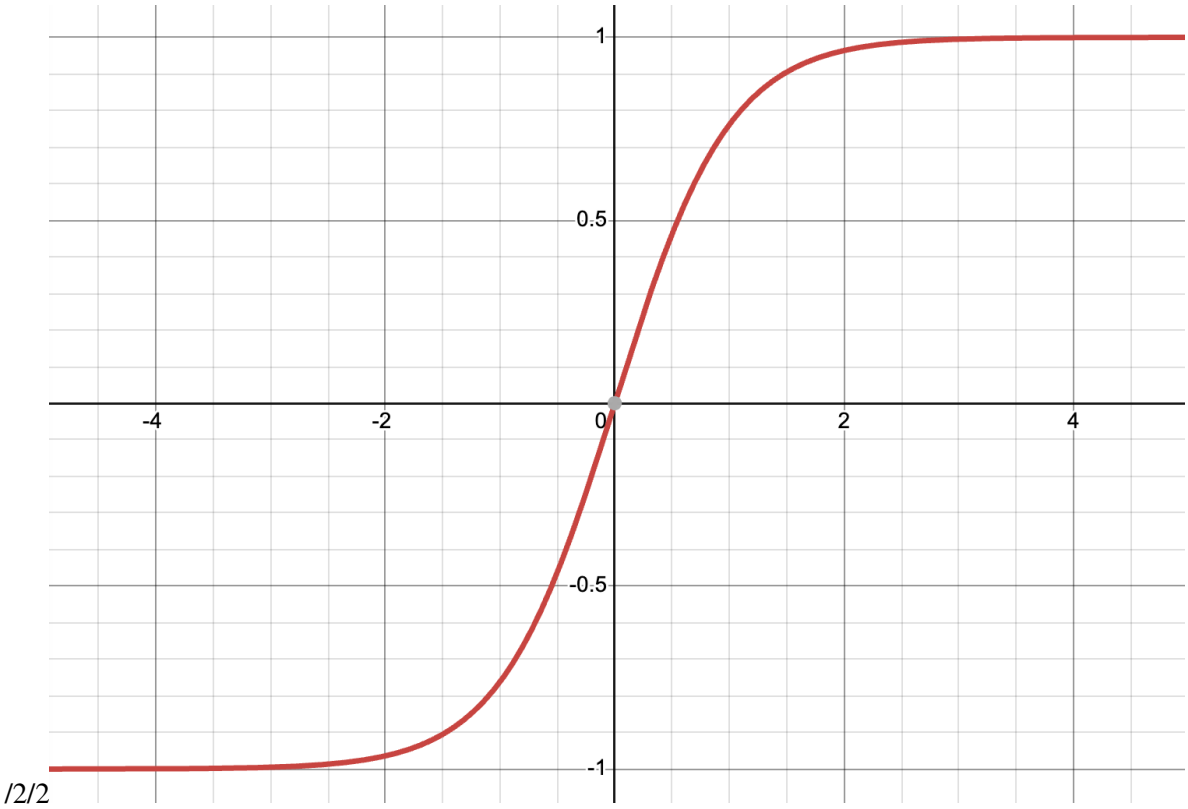


Figure 2.13: A very similar function except that its limits are present on -1 and 1

Variants of ReLU, such as **Leaky ReLU**, have been proposed to mitigate the dying ReLU problem by assigning a slight non-zero gradient for negative input values.

The Leaky ReLU function is defined as:

$$\text{Leaky ReLU}(x) = \max(ax, x) \tag{2.14}$$

where a is a small positive number such as 0.01. This non-zero gradient for negative inputs can help prevent neurons from becoming inactive during training. This function can visually be seen in Figure 2.15.

Backpropagation Algorithm for Training Neural Networks *Backpropagation* is an algorithm used to train feedforward neural networks by computing the gradient of the loss function concerning the network weights. This gradient is then used to update the weights in the opposite direction of the gradient, allowing the network to learn how to predict outputs given inputs accurately.

To understand backpropagation, it is crucial to define the loss function, which measures the network's performance on a given task. For instance, the cross-entropy loss is commonly used in

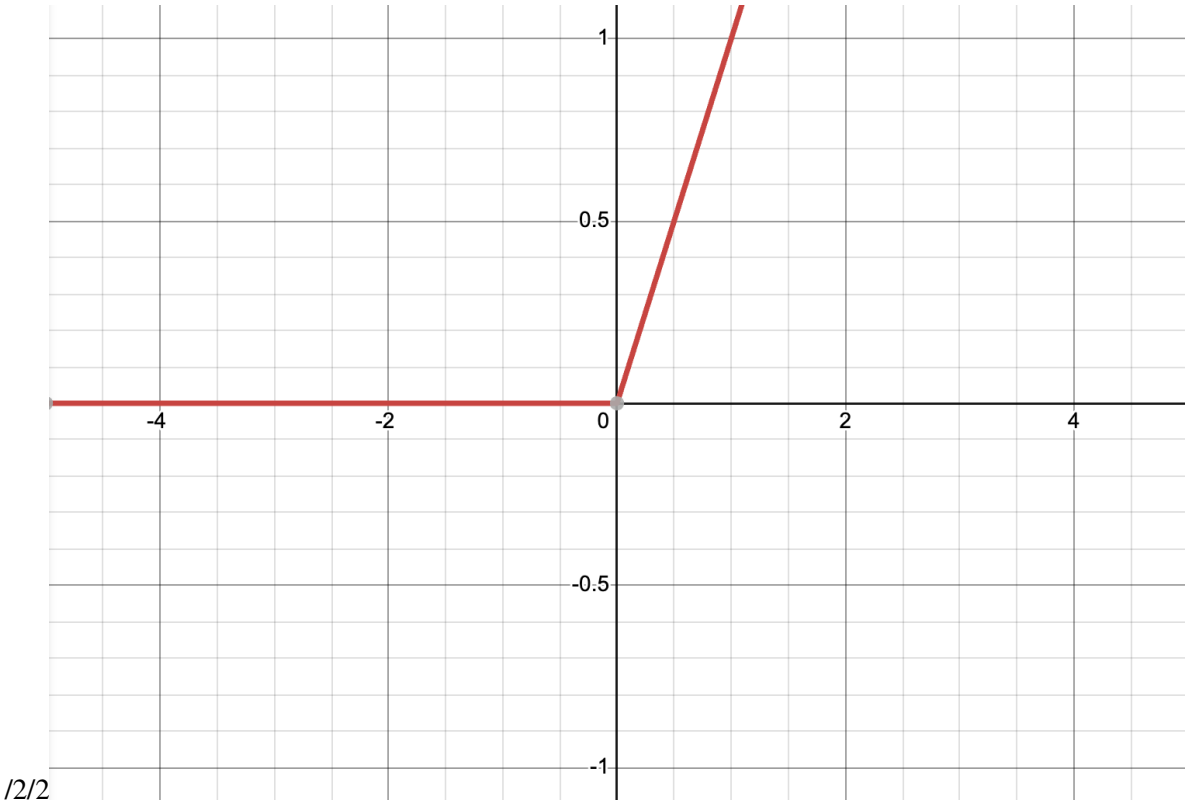


Figure 2.14: Linear for positive values, 0 for negative values.

classification tasks to quantify the difference between predicted probabilities and the correct labels. More information on loss functions can be found in Section 2.1.3.2. Backpropagation adjusts the network weights to minimize the loss function.

The backpropagation algorithm works by computing the gradient of the loss function concerning each weight in the network. This gradient tells how much the loss function would change if one were to make a small change to the weight. This gradient is then used to update the weight in the direction that reduces the loss function.

The gradient is computed using the chain rule of calculus. Considering a simple feedforward neural network with one hidden layer. The output of the network is given by:

$$y = h(\sum_{j=1}^{M} w_{2,j} h(\sum_{i=1}^{N} w_{1,i} x_i + b_1) + b_2) \tag{2.15}$$

where x_i is the i-th input, $w_{1,i}$ and $w_{2,j}$ are the weights connecting the input to the hidden layer and the hidden layer to the output, respectively, b_1 and b_2 are the biases of the hidden layer and the output, respectively, h is the activation function, such as sigmoid, (see section 2.1.2.2), and N and M are the numbers of inputs and hidden units, respectively.

The loss function is a function of the output y and the actual label t.

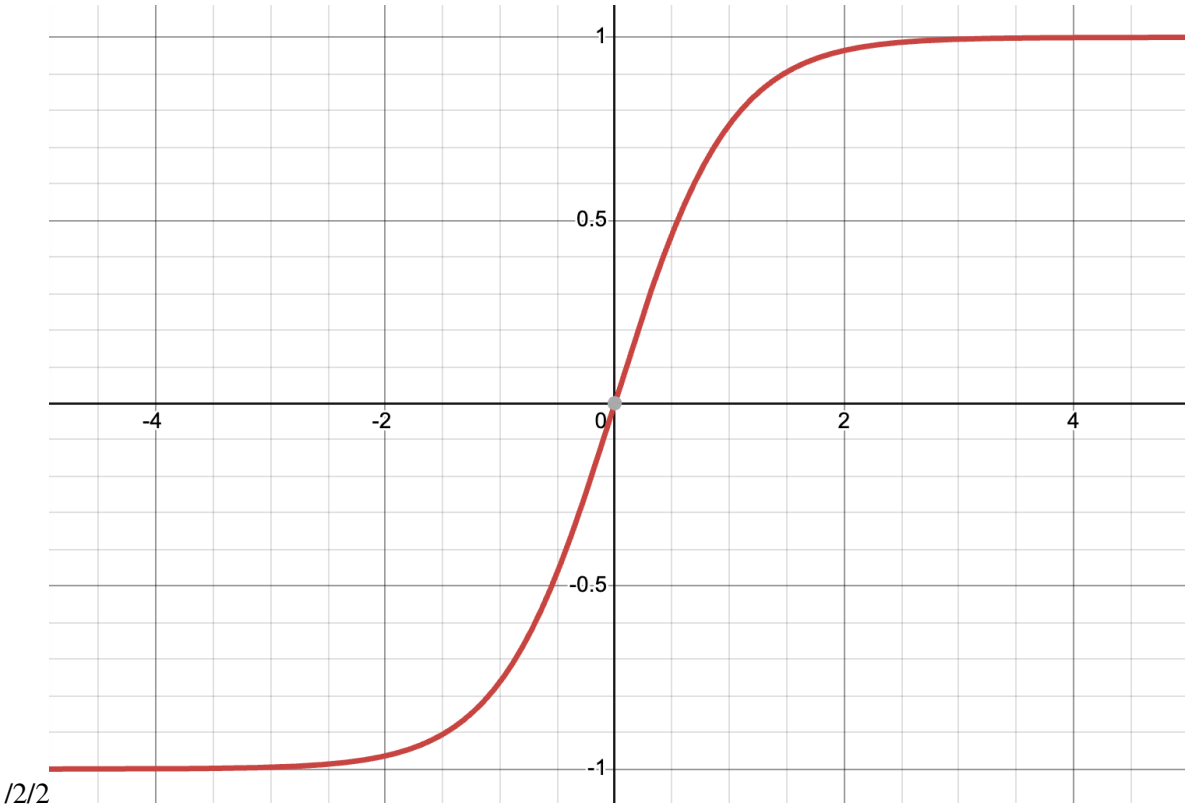


Figure 2.15: Very similar to Relu but with non-zero gradient for negative inputs.

To compute the gradient of the loss function concerning a weight $w_{i,j}$, one first needs to compute the local gradient of the output for the weight. This is given by:

$$\frac{\partial y}{\partial w_{i,j}} = h'(\sum_{j=1}^{M} w_{2,j} h(\sum_{i=1}^{N} w_{1,i} x_i + b_1) + b_2) h(\sum_{i=1}^{N} w_{1,i} x_i + b_1) w_{2,j} \tag{2.16}$$

where h' is the derivative of the activation function. One can then use the chain rule to compute the gradient of the loss function for the weight:

$$\frac{\partial E}{\partial w_{i,j}} = (y - t) \frac{\partial y}{\partial w_{i,j}} \tag{2.17}$$

Once one has computed the gradient of the loss function concerning all the weights in the network, the weights can be updated using gradient descent:

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial E}{\partial w_{i,j}} \tag{2.18}$$

where α is the learning rate, which determines the step size of the weight update.

Backpropagation can be extended to networks with multiple hidden layers using the chain rule to propagate the gradient backward through the network.

Optimization with Stochastic Gradient Descent Stochastic gradient descent (SGD) is a widely used optimization algorithm in DL. It is an iterative method that minimizes the loss function by updating the model parameters in the direction of the negative gradient of the loss function. In each iteration, SGD randomly selects a subset of the training data, called a mini-batch, and computes the gradient of the loss function concerning the parameters using the mini-batch. The parameters are then updated by subtracting the product of the gradient and a learning rate hyperparameter, which controls the step size of the update. The learning rate is typically set to a small value to ensure the stability and convergence of the algorithm.

Technically, it would be possible to use the loss of all the samples to update the model weights. However, using all the samples to update the weights, also known as batch gradient descent, can be computationally expensive and memory-intensive, especially for large datasets. In contrast, SGD updates the weights based on a randomly selected sample mini-batch, reducing the computational and memory requirements and enabling faster convergence.

Moreover, SGD introduces stochasticity in the optimization process, which can help the algorithm escape from local minima and explore different regions of the parameter space. This can improve the model's generalization performance and prevent overfitting the training data.

However, SGD can also be noisier and less stable than batch gradient descent due to the mini-batches random sampling and the gradients' fluctuation. Therefore, finding an appropriate learning rate and mini-batch size is crucial for the solutions' convergence and quality in SGD.

Formally, let θ be the vector of model parameters, $L(\theta)$ be the loss function, D be the training dataset, and B be a mini-batch sampled from D. Then, the update rule for SGD can be written as:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta_t; B) \tag{2.19}$$

where α is the learning rate, and $\nabla_\theta L(\theta_t; B)$ is the gradient of the loss function concerning the parameters evaluated on the mini-batch B at iteration t.

SGD has several advantages, such as its simplicity and low memory requirements, which make it suitable for large-scale datasets and complex models. However, it also has some limitations, such as its sensitivity to the learning rate and the mini-batch size, which can affect the solutions' convergence and quality. Therefore, several variants of SGD, such as Adagrad (that adapts the learning rate for each parameter based on its historical gradients, working well for sparse data) and Adam (explained in Section 2.1.2.2, adds a fraction of the previous update to the current update, and adapts the learning rate), have been proposed to address these issues and improve the algorithm's performance.

Optimization with the Adam Optimizer Adam is a variant of SGD (see Section 2.1.2.2) that adapts the learning rate for each parameter based on the estimates of the first and second moments of the gradients [69].

Adam addresses some of the limitations of SGD, such as the sensitivity to the learning rate and the mini-batch size, using a more sophisticated update rule incorporating information about the gradients and their history. Specifically, Adam computes a moving average of the gradients and their squares, which adjusts the updates' learning rate and momentum.

The estimates of the moments are computed as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.20}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{2.21}$$

where g_t is the gradient of the loss function with respect to the parameters at iteration t, m_{t-1} and v_{t-1} are the estimates of the moments at the previous iteration, and β_1 and β_2 are the decay rates for the moving averages of the gradients and their squares, respectively.

The update rule for Adam can then be written as follows:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t \tag{2.22}$$

where θ_t is the vector of model parameters at iteration t, α is the learning rate, $\hat{m}_t$ and $\hat{v}_t$ are the biased estimates of the first and second moments of the gradients, respectively, and ε is a small constant to avoid division by zero.

The biased estimates of the moments are computed as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.23}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.24}$$

where t is the iteration number. The bias correction is necessary to account for the fact that the estimates are initialized at zero and may be biased towards zero in the early iterations.

By using the biased estimates of the moments, Adam can handle noisy or sparse gradients and converge faster than SGD on a wide range of optimization problems. However, the choice of the

hyperparameters, such as the learning rate, the decay rates, and the epsilon value, can significantly affect the performance of Adam and should be carefully tuned for each problem.

Adam combines the benefits of both momentum and adaptive learning rates by using the moving average of the gradients to update the momentum and the moving average of the squared gradients to adapt the learning rate. Unlike SGD, which uses a fixed learning rate for all parameters, Adam adapts the learning rate individually for each parameter based on the estimate of the second moment of the gradient. This can improve the solutions' convergence and quality, especially for problems with sparse or noisy gradients. Moreover, Adam can handle non-stationary objective functions and noisy gradients, which can be challenging for other optimization algorithms.

However, Adam also has some limitations, such as its sensitivity to the choice of hyperparameters and its tendency to overshoot the optimal solution. Therefore, tuning the hyperparameters carefully and monitoring the algorithm's convergence during training is important.

2.1.2.3 Generative Deep Learning Architectures

Generative DL architectures are a subset of DL networks designed to generate new and diverse data samples from a learned distribution.

By finding latent data structures and learning to reproduce the hidden statistics behind observed data, they do so. To achieve this, the model tries to estimate an underlying probability distribution p_{data} when given a set of samples from this distribution. Thus, training a generative model involves selecting the best parameters that reduce some concept of distance/loss/error between the model and the actual distribution. As Huzaifah [61] states: "given training data points X as samples from an empirical distribution $p_{data}(X)$, we want to learn a model $p_\theta(X)$, belonging to a model family M that closely matches $p_{data}(X)$, by repeatedly changing model parameters θ". This is expressed as the problem in equation 2.25.

$$\min_{\theta \in M} d(p_{data}, p_\theta) \tag{2.25}$$

Standard functions for d are displayed in section 2.1.3.2.

These models have been applied to various tasks, such as image synthesis, text generation, and audio synthesis. Generative DL models have gained popularity recently due to their ability to produce high-quality data and model complex distributions.

This section will present the most used architectures.

Deep Autoregressive Network (DARN) First introduced in 2013, deep autoregressive network (DARN) is an architecture for generative models that are used to generate data by using an autoregressive (AR) approach [47].

AR models generate data by predicting the following sample in a sequence based on the previous samples. In the case of DARNs, the model has multiple hidden layers to do so. The idea behind it is to model the data's complex, high-dimensional probability distribution by breaking it down into a series of simple, conditionally independent distributions. This is done by learning a deep neural network that maps inputs to outputs through a series of hidden layers. This allows the network to build up a complex representation of the data distribution over time, capturing complex patterns in the data and ultimately making more precise predictions.

By applying the chain rule of probability, AR models can create a feasible density model that breaks down the probability distribution over n time steps [61], as shown in equation

$$p(X) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1}) \tag{2.26}$$

This method implies that data has a standard sequential order—the present term in the sequence (x_i) depends only on a recent window of preceding terms. Future terms are not taken into account. This is, ultimately, they assume that a data point only depends on previous ones and learn to predict the following sample is given only what has come just prior. The rationale behind this technique is similar to the one that RNNs employ. In fact, an RNN can be seen as an AR model that reduces the previous terms to a hidden state rather than giving them directly as input to a layer [61]. Besides, AR models are more straightforward and faster to train. Indeed, an AR model is a feedforward model (see Section 2.1.2.1)which predicts future values from past values. One can imagine a model similar to a feedforward neural network that is not fully connected but with only some connections regarding past inputs. This can be seen in Figure 2.16.

Variational Autoencoder (VAE) Kingma and Welling proposed the concept of variational autoencoders (VAEs) in 2013 [70]. The authors proposed a new approach to traditional AEs that utilizes a variational inference method to model complex distributions in high-dimensional spaces. Thus allowing for the generation of new, unseen data similar to the initial training.

In the realm of AEs, traditional approaches, as discussed in Section 2.1.2.1, aim to map the identity function through the utilization of encoder and decoder networks. The encoder takes the input and transforms it into a compressed vector representation, while the decoder reconstructs the input from this compressed representation. However, Variational Autoencoders (VAEs) introduce a distinct variation in their approach. Instead of modeling the input as a deterministic vector, the encoder in VAEs characterizes the input data as a probability distribution across potential representations. This distribution is typically represented by two sets of latent values: one corresponding to the mean and the other to the variance. These latent sets are often modeled using fully connected layers within the architecture.

Consequently, the VAE framework imposes a constraint on the embedded vector by confining it to a specified number of points in a hyperplane. Interestingly, this enables us to discard the

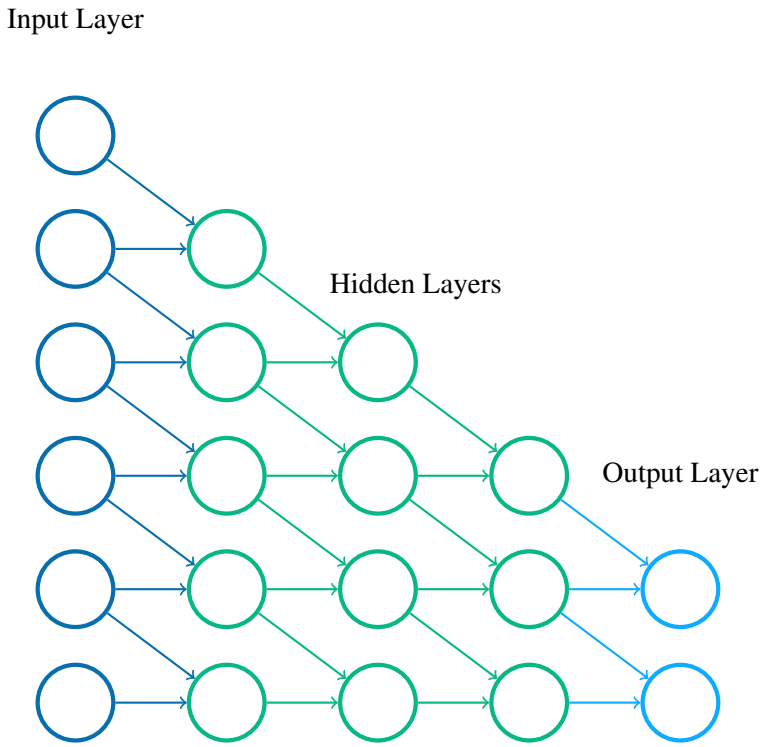


Figure 2.16: **Deep autoregressive network (DARN)** — The input of each neuron in a given layer is conditioned by the output of the 2 previous neurons in the previous output.

encoder entirely. By working with a continuous latent space, the decoder of the VAE can generate novel and diverse samples akin to the training data. Figure 2.17 provides a visual depiction of this concept, illustrating how the decoder operates based on samples from the latent distribution to generate new output.

This was encouraging, as distributions near each other would produce similar outputs. This means, it created smooth changes between data points. The explanation for this is that VAEs discover low-dimensional parameterized representations of the data [61].

For training, these networks use an objective function that aims to minimize the loss between input and output and ensure that the learned distribution is similar to a prior distribution, such as a Gaussian.

However, sometimes during training, the VAE can learn to ignore the latent variable and instead rely solely on the decoder network to generate the output. This means that the encoder network outputs the same distribution over the latent space for all input data points, resulting in a collapsed posterior distribution. In other words, the encoder fails to capture the variability in the input data, and the decoder generates outputs that are not diverse.

This phenomenon is known as *posterior collapse*, and it can occur due to various reasons, such

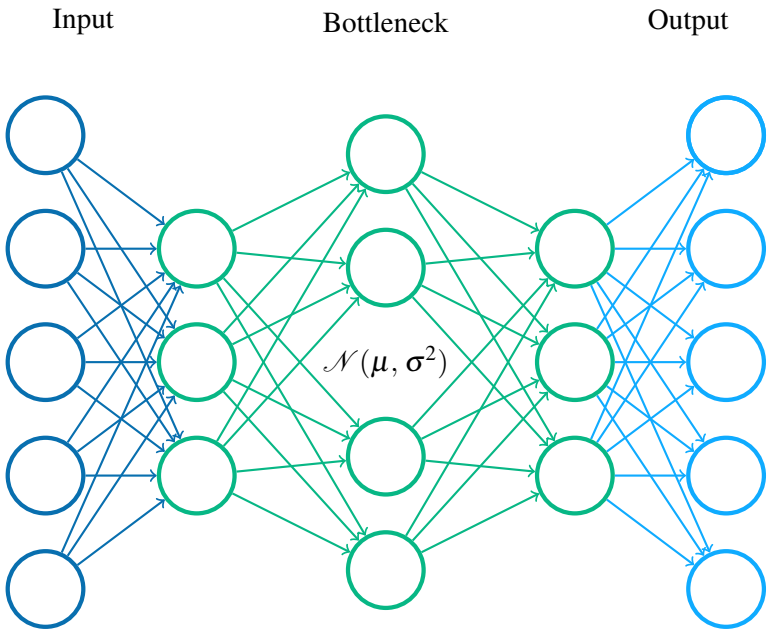


Figure 2.17: **Variational autoencoder (VAE)** — The VAE encoder operates in a manner comparable to the traditional AE, but with a notable distinction. Instead of directly mapping the input to a single latent representation, the VAE encoder translates the input into two sets of latent features: the normals and the variances. These latent features are represented in the Figure by the two sets of nodes within the bottleneck of the encoder architecture.

as a high reconstruction loss weight or a small latent space size. Posterior collapse can severely impact the performance of the VAE and result in poor-quality generated samples.

These models can be used for any generative task, such as computer vision, natural language processing, and sound generation.

Generative Adversarial Network (GAN) The paper "Generative Adversarial Networks" by Goodfellow et al. [46] introduced a novel framework for generative modeling using deep neural networks. The main idea behind generative adversarial networks (GANs) is to train two neural networks simultaneously, one generator and one discriminator.

By transforming a random noise vector z into a target distribution in some data space $\hat{X}$ (for example, spectrograms), the generator network G produces new samples. Meanwhile, the discriminator D attempts to tell apart synthetic and real data; that is, D assigns the input data, whether it is X or $\hat{X}$, a categorical label based on whether it believes the input originated from the actual data distribution $p(X)$ or the model distribution $p(z)$. Figure 2.18 shows this process.

Using a minimax optimization framework, the networks are trained in an adversarial way. The goal is that G generates a fake sample $\hat{X}$ that is given to D along with a real one X. This network then

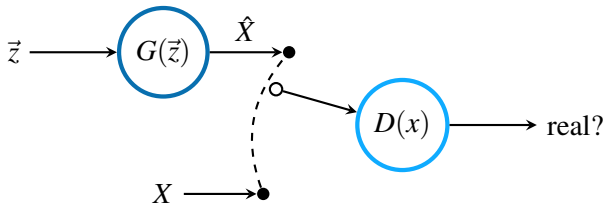


Figure 2.18: **Generative adversarial network (GAN)** — A random noise vector $\vec{z}$ is passed through the generator in $G(\vec{z})$ to create the synthetic sample $\hat{X}$. Both this and the real sample X are passed to the discriminator D that predicts which of the samples is the real one. It is important to notice that in this illustration, the circles represent entire neural networks and not simply neurons.

has to identify which is genuine and which is fabricated. D is trained to increase the probability of telling apart the real from the fake data. While G is trained at the same time for the opposite objective, that is, to deceive D by minimizing $\log(1 - D(G(z)))$ [61]. G and D are trained in turns until a Nash equilibrium is achieved.

When G creates flawless fake data that cannot be told apart from real data, a Nash equilibrium is reached. D has no clue whether the data is real or fake and just makes random guesses about the input label. In this situation, G performs at its best, and D performs at its worst. The models cannot get any better than this. This is a perfect scenario that requires effort to attain in reality. It is worth noting that the generator never sees the training samples, only the feedback given by the discriminator [61].

Once the training is done, the discriminator is thrown away, and the generator can be used to draw samples from the learned distribution of the real data. The generator has learned to associate random vectors with data samples in the target domain. These vectors usually represent some features. As a result, they cluster output data with similar features to nearby input values, offering a natural way of exploring output data with different attributes. This implies that similar input vectors will produce similar outputs [61].

Although this technique has seen great success in producing high-resolution images, it still needs to improve in the audio domain as the sections in Section 2.2 show. Besides that, even in ideal settings, it has some drawbacks. For instance, the fact that the training of the whole model implies the training of two different networks makes it unstable. It is easy to get stuck at a sub-optimal nash equilibrium. One such example is mode collapse, where the generator produces limited variations of the target distribution.

Deep autoregressive networks (DARNs) (see Section 2.1.2.3) represented the state-of-the-art in neural audio synthesis for a long time. These models are good at learning local latent structure, this is, the features of sounds over brief periods. However, they struggle with longer-term features. Besides, DARNs are very slow because they generate waveforms one sample at a time. GANs are capable of modeling global latent structure since they build the output as a whole; moreover, after training, they generate way faster [122], showing promising features for audio generation.

Normalizing Flow Models Normalizing flow models provide a flexible and robust framework for generative modeling and were first introduced in 2015 [104].

The main idea is to use the change of variables in the probability distributions technique to convert simple distributions into more complicated ones. This technique requires applying a transformation to a distribution that changes it into another, more intricate, distribution. The entire idea begins with a simple distribution (for instance, Gaussian) for a set of hidden variables z. The goal is to change this distribution into a complex one that corresponds to an output X. A single transformation is provided by a smooth and reversible function f that can relate z and X, such that $X = f(z)$ and $z = f^{-1}(X)$. Considering the complexity of X, one of these transformations might not produce a sufficiently complex distribution. Hence, multiple reversible transformations are combined sequentially, forming a "flow". Neural network layers determine each mapping function in the flow [61]. Figure 2.19 illustrates this process.

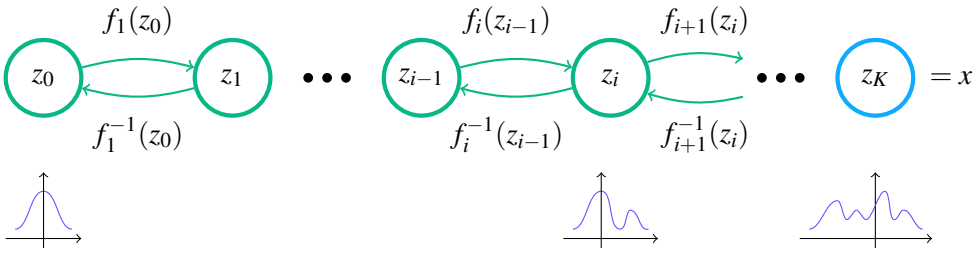


Figure 2.19: **Normalizing flows network** — This illustration was based on [133] and shows the application of multiple invertible functions f_k composed one after the other in order to build the complex output $z_K = x$ from a simple Gaussian distribution.

Accurately, let z_0 be a multivariate random variable with a distribution $p_0(z_0)$ where p_0 is, for example, a Gaussian distribution. Then, for $i = 1, ..., K$ where K is the number of flow operations, let $z_i = f_i(z_{i-1})$ be a sequence of random multivariate variables. f_i^{-1} should exist for training to occur. The final output z_K models the target distribution.

Normalizing flow models are flexible, meaning they can model various distributions by stacking multiple normalizing flows to form a deep network. This allows it to capture complex relationships between variables in the data.

These models have been proven effective for the generative modeling of high-dimensional data.

In the generative scene, these models are distinguished from the previously mentioned ones because they can speed up the generations and modeling processes [61].

Diffusion Models Until the proliferation of diffusion models, the architecture most used for data generation was the GAN (Section 2.1.2.3). The problem is that GANs are hard to train. For instance, mode collapse can happen. In mode collapse, the generator always generates the same data that fools the discriminator.

Diffusion models [117] simplify this generation process into more intuitive small steps where the work of the network is lighter and is run multiple times. This is done by taking inspiration from non-equilibrium thermodynamics. These models define a Markov chain of diffusion steps to slowly add random noise to data and then learn to reverse the diffusion process to construct desired data samples from the noise.

Practically, diffusion models use a Markov chain to gradually convert one distribution into another. This chain starts from a simple known distribution (*e.g.* a Gaussian) into a target distribution using a diffusion process. Learning in this framework involves estimating small perturbations to a diffusion process, using a network such as the U-Net (Section 2.1.2.1). Estimating small perturbations is more tractable than explicitly describing the whole distribution with a single, non-analytically-normalizable potential function. This process can be seen in Figure 2.20

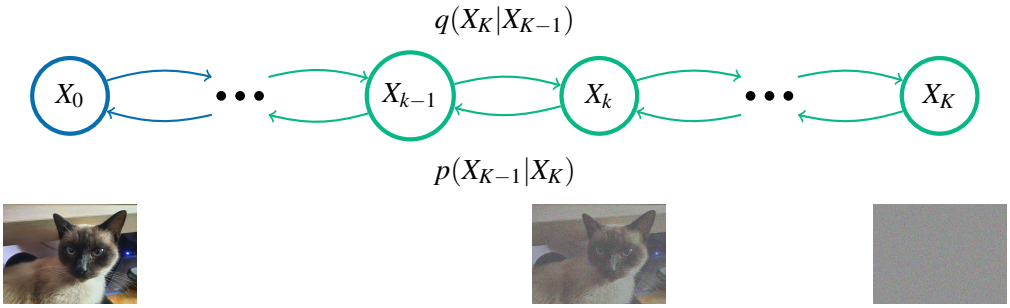


Figure 2.20: **Diffusion model** — This illustration was based on [53] and shows the process of applying Gaussian noise to an image sample through multiple steps $q(X_t|X_{t-1})$. The model will then learn the operation p that transforms X_t into X_{t-1} with $p(X_{t-1}|X_t)$ and so on until X_0. At this point, the model has generated a new data sample.

The ultimate goal is to define a forward (or inference) diffusion process which converts any complex data distribution into a simple, tractable distribution and then learn a finite-time reversal of this diffusion process which defines the generative model distribution [117].

One problem is that one needs to decide how much noise one wants to increment per iteration. For instance, if one decides to train a network that directly learns to denoise full Gaussian to a real image, then one is simply training a GAN generator. It is easier to remove a small amount of noise per iteration. The amount of noise added per iteration is a hyperparameter called a scheduler. For instance, one can add the same amount of noise per iteration, called the *linear schedule*. Multiple schedules may have different impacts.

For instance, given a linear scheduler, one can define that for $t = x$, the sample would be the original one with $k = x \times 10$ random data points with Gaussian noise. This allows data generation in different timestamps without running through all timestamps. For instance, generating a data sample with $t = 5$ would be as easy as noising $k = 50$ random data points.

To train these networks, one would give pairs of the original data sample X plus a data sample at a random timestamp X_t plus the random step t, $X_t = X + N(t)$ where N is a noising function.

The network would learn to get the noise from the data given a timestamp. This means that the network would learn to predict $N(t)$ using image segmentation. This will not always be perfect, so the network learns to predict $N\tilde{(}t)$. Then, theoretically, by applying $X_t - N\tilde{(}t)$, one gets $\tilde{X}$, which should be as close as possible to X. This process for $t = 50$ is challenging, as most of the data is Gaussian noise. However, applying the process for, for instance, $t = 1$, should be quite easy.

For inference, one gets noisy data X_t and a given timestamp t. Applying the network returns $N\tilde{(}t)$ as explained previously. By doing $\tilde{X} = X_t - N\tilde{(}t)$, one generates a bad data sample. But then, the algorithm takes $\tilde{X}$ and applies $N(t-1)$. This results in another noisy data sample with less noise. This process loops until $t = 0$. By then, a new data sample is generated.

Transformers In 2017, the introduction of the "Attention is All You Need" [128] paper marked a significant milestone in DL. Although initially introduced for NLP, the transformer architecture has proven helpful in various data generation tasks, including audio synthesis as shown in Section 2.2. This marked a paradigm shift from the conventional RNN-based models, which were earlier widely used, with some incorporating a rudimentary form of the attention mechanism.

In transformers, attention is a key component that allows the model to focus on relevant parts of the input sequence when making predictions. Mathematically, attention can be defined as a weighted sum of values based on their importance or relevance.

Let's break down the mathematical formulation of attention in transformers:

Query Q, Key K, and Value V: These are three linear transformations applied to the input sequence. The query represents the element for which we want to compute attention weights, while keys and values represent all elements in the sequence.

To calculate how much each value contributes to the output for a given query, we compute dot products between the query and all keys:

$$\text{scores} = Q \cdot K^T \tag{2.27}$$

Here, $\cdot$ represents matrix multiplication, K^T denotes transpose of matrix K.

The next step is to normalize these scores using a softmax function along dimension 1 (rows) to obtain attention weights that sum up to 1:

$$\text{weights} = \text{softmax}(\text{scores}) \tag{2.28}$$

Finally, we take a weighted sum of values using these normalized attention weights:

$$\begin{aligned} \text{attended\_values} &= V \cdot \text{weights} \\ \text{output} &= \sum(\text{attended\_values}) \end{aligned} \tag{2.29}$$

Here, softmax computes exponentiated values scaled by their row-wise sums, while sum performs summation across rows.

The resulting output represents an attended representation obtained by giving higher weightage/importance to more relevant parts of the input sequence based on similarity with respect to the query. This mathematical formulation allows transformer models to capture long-range dependencies effectively by attending to the pertinent information in the source sequence when generating predictions.

The attention mechanism allows the model to give different importance to different input parts. For instance, let us imagine a translation task English-Portuguese. If naively translated, the sentence "She is a doctor" could be translated to "Ela é um doutor". However, if, when generating the last word, the model gives some importance to the word "she", it might guess that the correct word is "doutora".

The key idea behind transformers is to completely disregard the recurrent architecture and only use attention by using self-attention. Self-attention is a mechanism that allows the calculation of the importance of each input element concerning all other elements in the input sequence. This allows the model to dynamically focus on the most relevant information at each step of the calculation instead of relying on fixed relationships between elements in the input as in traditional recurrent neural networks.

The transformer architecture, shown in Figure 2.21, serves as the basis for advanced transformers utilized in a variety of applications, such as audio synthesis. This architecture includes an encoder and decoder, both with stacked layers, which are essential to processing textual input data and producing consistent audio waveforms.

The encoder, consisting of N identical layers, aims to convert the input text into continuous representations that encapsulate crucial semantic information. Each layer features two sub-layers that enable this conversion:

1. Multi-Head Self-Attention: As explained above, this mechanism allows the encoder to model input text sequence dependencies objectively. Multiple scaled dot-product attention heads attend to various positions of the text sequence simultaneously. This process captures intricate relationships within the text in parallel, enriching the representation.

2. Position-wise Feedforward Network: Composed of two linear transformations with ReLU activation, this neural network introduces non-linear interactions within the embedded space.

These interactions bolster the model's capability to comprehend intricate semantic nuances that exist within the text.

Residual skip connections and layer normalization surround each sub-layer, enhancing training stability. By mapping input text to continuous representations, the encoder empowers subsequent stages to generate audio that aligns with the input's semantics.

The decoder, which includes M stacked layers, generates the output audio waveform while conditioned on the continuous representations from the encoder. This conditioning guarantees that the synthesized audio aligns with the intended semantics of the input text. Each decoder layer comprises three sub-layers.

1. Masked Multi-Head Self-Attention over Previous Outputs: This sublayer enables modeling of dependencies between previously generated audio samples, facilitating the creation of coherent output samples. It blocks leftward information flow, ensuring a causal relationship between created samples.

2. Multi-Head Attention over Encoder Outputs: By attending over the final encoder representations, this sublayer allows the decoder to incorporate the input text semantics into the generation process. This attention mechanism guarantees that the synthesized sample remains aligned with the intended meaning of the input.

3. The Position-wise Feedforward Network consists of two linear transformations with ReLU activation, similar to the encoder. It improves the decoder's ability to comprehend complex relationships between different samples.

Like the encoder, the decoder employs residual connections and layer normalization for stability during training. The decoder generates the output sequentially, predicting one sample at a time. The integration of multi-head attention mechanisms at both the local and global levels enables the synthesis of diverse and natural outputs. This autoregressive process, conditioned on powerful semantic representations, produces high-fidelity outputs aligned with the input.

This architectural change significantly accelerates the training and inference processes by allowing the use of larger data sets while greatly improving the generation results - thus revolutionizing the progress made in the field.

Vector Quantised Variational AutoEncoder (VQ-VAE) (2018) The vector quantized variational autoencoder (VQ-VAE), introduced in 2018, model distinguishes itself from traditional VAEs in two main aspects: the encoder network outputs discrete codes instead of continuous ones, and the prior is learned rather than static. While continuous feature learning has been the focus of many previous works, this model, introduced by [89], concentrates on discrete representations, a natural fit for complex reasoning, planning, and predictive learning.

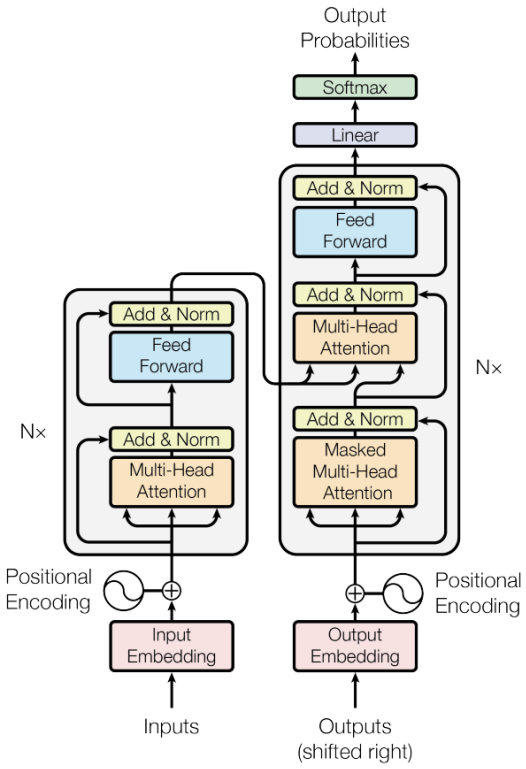


Figure 2.21: **Transformer** — This illustration was taken from [128] and shows the general architecture of the base transformer. One can see that the constituents are pretty simple. These are simple word embeddings (which are not covered in this study), self-attention, and feedforward layers. The left side of the structure is called the encoder, while the right side is called the decoder.

The VQ-VAE model combines the VAE framework with discrete latent representations through a parameterization of the posterior distribution of (discrete) latents given an observation. Based on vector quantization, this model is simple to train, does not suffer from significant variance, and avoids the "posterior collapse". As illustrated in Fig 2.22, the VQ-VAE architecture consists of an encoder, a discrete latent space, and a decoder.

The VQ-VAE defines a latent embedding space $e \in R^{N \times D}$, where N is the size of the discrete latent space (i.e., a N-way categorical), and D is the dimensionality of each latent embedding vector e_n. There are N embedding vectors $e_n \in R^D, n \in 1,2,...,N$. The model takes an input X, passed through an encoder producing output $z_e(X)$. The discrete latent variables z are then calculated by the nearest neighbor look-up using the shared embedding space e. The input to the decoder is the corresponding embedding vector e_n. This forward computation pipeline is a regular autoencoder with a non-linearity that maps the latents to 1-of-N embedding vectors.

The posterior categorical distribution $q(z|X)$ probabilities are defined as one-hot (Eq. 2.30):

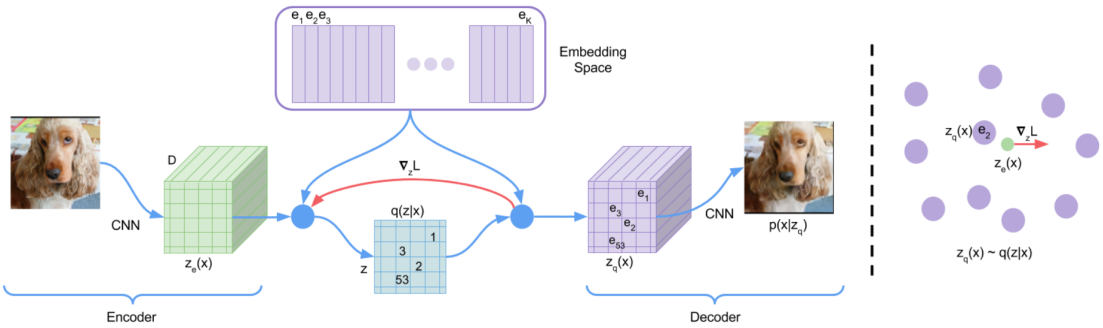


Figure 2.22: **VQ-VAE** — Taken from the original paper, this Figure presents two distinct illustrations. On the left side, a detailed diagram of the VQ-VAE architecture is provided, showcasing the flow of information through the encoder, the discrete latent space, and the decoder. On the right side, a visualization of the embedding space is displayed, where the encoder output $z(X)$ is mapped to its nearest embedding point e_2. The red arrow represents the gradient $\nabla_z L$, influencing the encoder's output adjustment. This adjustment may result in a different configuration during the subsequent forward pass, highlighting the dynamic nature of the learning process within the VQ-VAE model.

$$q(z=n|X) = \begin{cases} 1 & \text{for } n = \text{argmin}_j ||z_e(X) - e_j||_2, \\ 0 & \text{otherwise.} \end{cases} \tag{2.30}$$

where $z = e_n$ is the closest embedding vector to the encoder output $z_e(X)$. During forward computation, the nearest embedding $z_q(X)$ is passed to the decoder, and during the backward pass, the gradient $\nabla_z L$ is passed unaltered to the encoder. The overall loss function has three components to train different parts of the VQ-VAE: the reconstruction loss, the vector quantized (VQ) objective, and the commitment loss. The total training objective becomes:

$$L = \log p(X|z_q(X)) \tag{2.31}$$
$$+ ||\text{sg}[z_e(X)] - e||_2^2 \tag{2.32}$$
$$+ \beta ||z_e(X) - \text{sg}[e]||_2^2 \tag{2.33}$$

This equation combines the three following terms:

1. **Reconstruction loss** (Equation 2.31): This term represents the log probability of the input data X given the latent variable $z_q(X)$. It measures how well the model can reconstruct the input data using $z_q(X)$ as a representation. Maximizing this term would lead to a better reconstruction of the input data.

2. **VQ** (Equation 2.32): The second term measures the difference between the stop-gradient of the encoder output $z_e(X)$ and the embedding vector e. The stop-gradient operator, denoted

as sg, acts as the identity during the forward pass but has zero partial derivatives during the backward pass. This term encourages the model to use the embeddings effectively by minimizing the distance between the encoder output and the closest embedding vector.

3. **Commitment loss** (Equation 2.33): This term acts as a regularization term that measures the difference between the encoder output $z_e(X)$ and the stop-gradient of the embedding vector e. The β parameter controls the strength of this regularization. Minimizing this term would make $z_e(X)$ closer to the straight-through estimator of e.

VQ-VAE has emerged as a vital component in generative artificial intelligence, spanning domains such as image [101] and sound generation [135].

Multi-Scale Vector Quantised Variational AutoEncoder (MS-VQ-VAE) (2019) The multi-scale vector quantised variational autoencoder (MS-VQ-VAE) model is a generalization of the VQ-VAE model (see Section 2.1.2.3) that employs multiple discrete latent spaces with different scales and dimensions. Tjandra et al. proposed this model [124] to learn unsupervised hierarchical and discrete representations of complex data. The MS-VQ-VAE architecture comprises an encoder, a multiscale discrete latent space, and a decoder.

The key difference between the MS-VQ-VAE and the VQ-VAE is that the former defines a collection of latent embedding spaces $e^s \in R^{K_s \times D_s}$, where s denotes the scale index, K_s denotes the cardinality of the discrete latent space at scale s, and D_s denotes the dimensionality of each latent embedding vector e^s_i. There are K_s embedding vectors $e^s_i \in R^{D_s}, i \in 1,2,...,K_s$. The model takes an input x, encoded into outputs $z^s_e(x)$ at different scales. The discrete latent variables z^s are then obtained by the nearest neighbor look-up using the shared embedding space e^s. The decoder input is the corresponding embedding vector e^s_k. This forward computation pipeline resembles a regular AE (see section 2.1.2.1) with a non-linearity that maps the latents to 1-of-K_s embedding vectors.

The posterior categorical distribution $q(z^s|x)$ probabilities are defined as one-hot (Eq. 2.34), analogous to Eq. 2.30 in section 2.1.2.3, but with an additional scale index:

$$q(z^s = k|x) = \begin{cases} 1 & \text{for } k = \text{argmin}_j ||z^s_e(x) - e^s_j||_2, \\ 0 & \text{otherwise.} \end{cases} \tag{2.34}$$

The overall loss function consists of three components for each scale: the reconstruction loss, the VQ objective, and the commitment loss. The total training objective becomes (Eq. 2.35), analogous to Eq. 2.31 in section 2.1.2.3, but with a summation over scales:

$$L = \sum_{s=1}^{S} (\log p(x|z^s_q(x)) + ||\text{sg}[z^s_e(x)] - e^s||^2_2 + \beta ||z^s_e(x) - \text{sg}[e^s]||^2_2) \tag{2.35}$$

The benefit of using multiple codebooks and scales is that it enables the model to capture different levels of abstraction and granularity in audio signals. For instance, lower scales can encode phonetic information in speech, while higher scales can encode prosodic information. Furthermore, using multiple codebooks can enhance the diversity and expressiveness of the latent space by allowing more combinations of discrete codes.

2.1.3 Foundations for Enhancing Generative Models for Audio

To develop generative models for audio, it is necessary to address several factors that impact their performance and quality. This thesis concentrates on three main areas: data augmentation, evaluation metrics, and data embedding.

Data augmentation is the process of applying transformations to the original data to increase its size and diversity. This can help overcome the limitations of small or imbalanced datasets and improve the generalization ability of generative models. Different types of data augmentation techniques for sound generation and their effects on the model outcomes are discussed in Sextion 2.1.3.1.

Evaluation metrics are the methods used to measure the quality and diversity of the generated sounds. They provide a way to compare different generative models and assess their strengths and weaknesses. However, evaluating sound generation is not trivial, as it involves objective and subjective criteria. We review various evaluation metrics for sound generation and their advantages and disadvantages in Section 2.1.3.2

Data embedding is the technique of converting data into numerical representations that capture its essential features and characteristics. This can facilitate the learning process of generative models and enhance their expressiveness and efficiency. We explore different data embedding methods in section 2.1.3.3.

2.1.3.1 Data Augmentation

Data augmentation is crucial to ML tasks. It helps to increase the training dataset's size and improve the model's robustness. For the task at hand, there are two main types of data augmentation: acoustic and linguistic. Data augmentation can reduce overfitting and improve generalization by introducing more variations into the training set [97].

Acoustic Data Augmentation According to Abayomi-Alli et al. [1], the most commonly used data augmentation tools for audio ML tasks are the addition of noise, time shifting, pitch shifting, GAN-based methods (see Section 2.1.2.3), time stretching, and concatenation. Other techniques, such as overlapping, are also helpful. All these techniques play a critical role in increasing the size of the training dataset and providing the model with a diverse range of input data. This Section thoroughly explores these techniques and discusses their impact on the performance of sound-based ML models.

Table 2.2 provides a concise summary of the advantages and limitations of each technique.

Table 2.2: Summary of Acoustic Data Augmentation Methods

Method	**Description**	**Advantages**	**Limitations**
Addition of Noise	Adding random noise to the original audio signals.	Increases dataset diversity, helps model learn to handle noisy environments.	Noise must be chosen to align with the problem at hand.
Time Shifting	Altering the temporal structure of an audio signal by shifting it.	Helps model learn sound patterns invariant to temporal changes, improves generalization.	Trimming and shifting sections may introduce artifacts.
Pitch Shifting	Altering an audio signal's frequency.	Trains models to recognize sound patterns free from pitch variations.	May introduce artifacts or distortions.
GAN Based Methods	Utilizing GAN networks to generate synthetic audio signals.	Generates high-quality data.	Requires significant computational resources.
Time Stretching	Changing the time duration while preserving their spectral content.	Generates samples with different time durations.	Simple methods may introduce artifacts; sophisticated methods are computationally intensive.
Sound Concatenation	Joining snippets of multiple audio signals to form a new one.	Trains models that recognize specific sound sequences.	Labels associated with the original audio signals may need to be modified to represent the new signal.
Sound Overlapping	Merging two or more audio signals to form a new combined audio signal.	Helps models identify sound patterns amidst overlapping sounds.	Amplitude adjustment and normalization are crucial to prevent overpowering or clipping in the composite signal.

Addition of Noise Data augmentation for audio with the addition of noise is a common technique that involves adding random noise to the original audio signals to produce new, diverse audio samples.

The process of adding noise to audio signals involves several steps:

1. Select a noise source: This can be any type of noise, generated or real [86], such as white noise, babble noise, static noise, factory noise, jet cockpit, shouting, background noise, and others, [1]. The noises should be chosen according to the problem at hand.

2. Specify the noise level: For instance, the augmented sound might be $y' = y + 0.05 \times Wn$ where y is the initial sound, y' the augmented sound, and Wn some white noise [84].

3. Add the noise: The selected noise source is then added to the audio signal by adding the noise and audio signals element-wise.

4. Normalize the output: Finally, the resulting audio signal with added noise is normalized to prevent clipping or overloading.

Repeating these steps with different noise sources and noise levels makes it possible to generate multiple, diverse audio samples that can be used for data augmentation purposes.

Time Shifting Time shifting, also known as time warping, is a data augmentation technique that involves altering the temporal structure of an audio signal.

Time shifting involves shifting the entire audio signal by a certain amount of time, either forwards or backward. This can be achieved by adding or removing samples from the audio signal or changing the existing samples' position within the signal.

One approach to implementing time shifting involves trimming the length of the audio signal and using the trimmed sections to create new, diverse audio samples. For example, consider an audio signal of size 150. If this signal is trimmed to a length of 125, up to 25 new audio samples can be generated by shifting the trimmed sections. These new samples can be labeled the same as the original audio signal.

Time shifting by trimming and shifting sections of the audio signal can significantly impact the performance of sound-based ML models. By providing the model with diverse, time-shifted versions of the audio signal, this technique can help the model learn to identify sound patterns invariant to temporal changes, such as the presence of a particular sound event or the spoken words in an audio recording. This can lead to better generalization performance on new, unseen data and improved overall model performance.

Pitch Shifting Pitch shifting is a technique used in audio data augmentation that involves altering an audio signal's fundamental frequency.

Pitch shifting is achieved by adjusting the pitch of an audio signal positively or negatively. For example, a plus or minus two shift can be implemented [84]. This process results in audio signals that have a different pitch. This can be helpful in training models to recognize sound patterns that are free from pitch variations.

GAN Based Methods Utilizing GANs (see Section 2.1.2.3) in data augmentation for audio signals can be a powerful and effective method, albeit slower than other techniques. In this approach, a GAN network is trained on the available audio data to learn the underlying patterns and distributions present in the data. The network then generates new, synthetic audio signals similar to the input data [98].

The success of the GAN-based data augmentation method depends heavily on the quality of the GAN training, as well as the diversity of the input data. If the GAN is trained well and the input data is diverse, the generated data will be of high quality.

It is important to note that GANs require considerable computational resources and training time compared to other data augmentation techniques. However, the results obtained from GANs can be highly effective and accurate, making this approach a valuable addition to the data augmentation toolkit for audio machine learning tasks.

Time Stretching Time stretching as a data augmentation technique for audio signals involves changing the time duration of the audio signals, typically by increasing or decreasing the time axis of the audio signals. The purpose of time stretching is to generate new audio samples from the original audio signals with different time durations.

A straightforward way of implementing time stretching is to use a stretching factor. For example, if the stretching factor is 1.2, then the time axis of the audio signal is increased by 20%. To achieve this, one approach is to use a naive algorithm that duplicates some of the samples in the audio signal according to the stretching factor. However, this simple method can result in undesirable artifacts, such as pitch changes, if the stretching factor is not an integer.

More sophisticated methods, such as phase vocoder-based time stretching, can produce high-quality time stretching with minimal artifacts. These methods use time and frequency domain processing techniques to stretch the audio signal while preserving its spectral content and temporal structure. The resulting audio signal has a different time duration while preserving the original pitch [6].

Sound Concatenation Mixing up sounds, or sound concatenation, is a method for audio signals where multiple audio signals are joined to form a new and diverse audio signal. This technique can be achieved by taking snippets of multiple audio signals and concatenating them randomly or using cross-fade techniques to ensure a seamless transition between the different audio snippets.

This technique would be advantageous in a sound generation setting where one wants to make the network learn a prompt such as “dog barking and then car honking”. When applying sound concatenation, one must consider that the label will also change.

Sound Overlapping Sound overlapping, also referred to as sound mixing or audio blending, is a technical process that involves merging two or more audio signals to form a new combined audio signal. This process is currently utilized in popular data augmentation platforms [76] to aid the model in identifying sound patterns amidst overlapping sounds, which is a frequent occurrence in real-world applications.

There are several steps involved in the process of sound overlapping. The first step involves selecting multiple audio signals, which can originate from the same or different sources depending

on the desired outcome and problem at hand. The next step is to adjust the amplitude of each audio signal to achieve a balanced combination and prevent one signal from overpowering others. This step is crucial. This can be achieved by either normalizing the amplitude of each signal or scaling them based on a predetermined factor. After appropriately adjusting the amplitudes, the selected audio signals are combined by adding them element-wise. A new composite audio signal is generated using this process, which includes overlapping sounds from the original signals. It is essential to normalize the output to prevent clipping or overloading, ensuring a well-balanced and usable composite sound waveform.

Repeating these steps with different combinations of audio signals and amplitude adjustments can generate diverse composite audio samples for data augmentation purposes.

It is important to note that when using sound overlapping as a data augmentation technique, the labels associated with the original audio signals must also be considered. Sometimes, the labels may need to be combined or modified to accurately represent the new composite audio signal.

Linguistic Data Augmentation Linguistic data transfiguration involves metamorphosing textual data to augment its diversity and quantity. This can ameliorate the performance and robustness of natural language processing models that rely on text data.

For sonic milieu generation, linguistic data transfiguration provides an efficacious approach to creating more varied and verisimilitudinous soundscape descriptions from text. Nonetheless, various existing soundscape datasets discussed in Section 4.2 contain categorical labels or tags instead of descriptive annotations.

It is imperative to note that some linguistic data augmentation techniques only function when the original text is in natural language, while others can still be applied to categorical labels.

Variations in input text can facilitate the generation of soundscapes with more variability and legitimacy. The following sections cover specific linguistic data augmentation techniques and their applications for soundscape generation.

While linguistic data augmentation has several advantages, it poses some issues and challenges [116].

The main benefits of textual augmentation are as follows:

1. It can reduce the costs of collecting and annotating textual data.
2. It can improve the accuracy of models by increasing the training data size, alleviating data scarcity, mitigating overfitting, and creating variability in the data.
3. It can boost the generalizability of models by exposing them to different linguistic patterns and styles.
4. It can increase the robustness of our models by making them resilient against adversarial attacks that attempt to deceive them through expert alterations of the input sequences.

However, there are also some potential downsides:

1. It can introduce noise or errors into the data that may impact the quality and readability of the augmented text.
2. It can change or lose the original text's meaning, style, or complexity, mainly if the transformation is inappropriate or irrelevant to the task or domain.
3. It can be computationally expensive or time-consuming to generate high-quality and diverse augmented text, especially if it involves using external resources or models such as dictionaries, corpora, word embeddings, generative models, or translation models.

Various frameworks have been developed for augmenting text data linguistically. These can be broadly grouped into symbolic and neural augmentation models.

Symbolic Augmentation Models These methods employ rule-based transformations operating directly on the surface form of text via predetermined heuristics. They include:

- Rule-based augmentation replaces, inserts, or deletes tokens according to specified rules. An example is replacing named entities with alternatives [132].
- Graph-based augmentation uses graph structures to perturb the text, *e.g.* swapping adjacent adjectives and nouns [5].
- MixUp combines existing examples via interpolation to synthesize augmented instances, *e.g.* combining "The cat sat on the mat" and "The dog lay on the rug" to generate "The cat lay on the mat" [50].
- Feature-based augmentation applies transformations to word embeddings, *e.g.* adding noise to the embedding space [17].

Despite their interpretability, symbolic methods struggle with complex transformations.

Neural Augmentation Models These techniques leverage deep neural networks and large language models. They encompass:

- Back-translation, which translates text into another language and back, producing paraphrases, *e.g.* translating "The book was interesting." to French and back to English, yielding "The book was fascinating" [96].
- Generative augmentation employs generative language models to synthesize novel text, *e.g.* using an Large Language Model (LLM) such as GPT to rephrase sentences or simply fine-tune them.

While more complex, neural methods can generate diverse and realistic augmented instances.

Both symbolic and neural augmentation aim to expose models to more variability during training, helping combat overfitting and improve performance. However, symbolic methods offer interpretability, while neural methods provide more flexibility and variation.

Table 2.3: A taxonomy of text augmentation methods for transformer language models according to their algorithmic properties and underlying approaches.

	Interpretability Transparency	**Algorithmic Complexity**	**Capacity to Leverage Labels**	**Paradigmatic**
Rule-based	High	Relatively low	Incapable	Lexical substitution
Graph-based	High	Moderate	Incapable	Knowledge graph-based
Sample Combination	Medium	Moderate	Capable	Embeddings summation
Feature-based	Medium	Moderate	Incapable	Noise injection
Generative	Low	High	Capable	Text auto-generation
Back-translation	Medium	High	Capable	Inverse translation

Table 2.3 categorizes several common text augmentation strategies according to their transparency, complexity, dependence on labels, and paradigm.

Regarding interpretability, rule-based and graph-based methods exhibit high transparency since they employ explicit symbolic transformations. In contrast, the stochastic nature of generative models and back-translation compromises their interpretability.

Computational complexity also differs. Rule-based and graph-based augmentation are relatively efficient since they apply explicit symbolic transformations. In comparison, training neural networks for generative modeling and back-translation requires more computational resources.

For leveraging labels, given that some datasets mentioned in Section 4.2 contain categorical labels, and we desire descriptive annotations, it is pertinent to assess whether these techniques can transform categorical labels into text descriptions. Employing generative models or back-translation potentially enables this since the models attempt to make sense of the input and transform it into a sentence.

2.1.3.2 Evaluation Metrics

The evaluation process can provide insights into the system's performance and reveal areas that need improvement. Moreover, a proper evaluation metric helps to compare the results of different models and choose the best one. The aim of this section is to provide a comprehensive overview of the available evaluation metrics for audio generation and to provide a foundation for the evaluation of the system developed in this thesis.

In this thesis, a comprehensive evaluation framework is developed that considers two different types of evaluations. The first type of evaluation focuses on metrics that can be used for training models offline, such as loss metrics, which play a crucial role in assessing the performance of a model. The second type of evaluation focuses on evaluating the broader impacts of a model, such as its environmental impact and inference time for a generation. These evaluations are essential in ensuring that the model not only performs well on its primary task but also has minimal negative impacts on other aspects of the system.

Loss Functions Evaluating generative audio systems is challenging due to the need for a standard set of metrics to capture the quality and diversity of the generated audio samples. Different studies often use different evaluation methodologies and metrics when reporting results, making a direct comparison to other systems intricate if not impossible [130]. Furthermore, the perceptual relevance and meaning of the reported metrics, in most cases unknown, prohibit any conclusive insights concerning practical usability and audio quality.

A review and comparison of the available evaluation metrics for audio generation is essential to provide a foundation for evaluating the system developed in this thesis. This section discusses some of the commonly used metrics for evaluating generative audio systems, such as mean absolute error (MAE), mean squared error (MSE), Kullback–Leibler (KL) divergence, and evidence lower bound (ELBO). It also discusses their advantages and limitations and how they can be applied to sound generation tasks.

Mean Absolute Error The Mean absolute error (MAE) is a quantitative measure of the average magnitude of the errors between the predicted and actual values [134]. It is computed as:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.36}$$

where y_i denotes the true value, $\hat{y}_i$ denotes the predicted value, and n denotes the number of samples. The MAE is also called L1-norm loss or mean absolute deviation (MAD).

The MAE is a relevant metric for evaluating generative models as it treats all errors equally and is arguably less sensitive to outliers compared to MSE (see Section 2.1.3.2). Specifically, it indicates the average absolute difference between the predicted and actual values in the same unit as the output.

However, the MAE has some notable limitations that should be considered. For instance, it does not directly capture the perceptual quality of the generated audio samples. The perceptual quality of audio can depend on factors such as timbre, pitch, or harmony, which are not explicitly determined by the MAE. Therefore, to fully assess the quality of generative models, the MAE should be complemented with other metrics and human evaluation. Additionally, the MAE is a scale-dependent measure and cannot be used to compare predictions that use different scales.

To illustrate the difference between MAE computed on raw audio versus spectrograms, consider the following example: For a 1D raw audio sample, the MAE would measure the average absolute difference between the amplitude values of the audio signals. In contrast, if the audio data were represented as spectrograms, the MAE would measure the average absolute difference between the magnitude values of the frequency bins. In this case, the spectrograms can be treated as images with a single channel, and the MAE can be seen as a pixel-wise error metric.

It is important to consider the limitations of the MAE when evaluating generative models. For example, consider a scenario with two audio samples of a dog barking: one with a bark at one second and another with a bark at two seconds. Despite being similar sounds with different temporal positions, calculating their MAE would lead to higher-than-expected error as it does not account for temporal alignment. Therefore, it is important to use other metrics and human evaluation methods along with MAE to assess timing accuracy and perceptual quality of generated audio samples comprehensively.

Mean Squared Error Mean squared error (MSE) is a standard metric to evaluate the performance of a predictor or an estimator. It quantifies the average of the squared errors, the average squared difference between the estimated and actual values. MSE is always a non-negative value approaching zero as the error decreases. The smaller the MSE, the better the predictor or estimator [56].

MSE can be calculated as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where n is the number of data points, y_i is the true value of the variable being predicted or estimated, and $\hat{y}_i$ is the predicted or estimated value.

MSE incorporates both the variance and the bias of the predictor or estimator. The variance measures how widely spread the estimates are from one data sample to another. The bias measures the distance of the average estimated value from the true value. For an unbiased estimator, the MSE equals the variance.

MSE can compare different predictors or estimators and select the one that minimizes the MSE. For instance, in linear regression, MSE can be used to find the best-fitting line that minimizes the sum of squared errors. MSE can also evaluate the quality of a generative model that produces audio samples from textual input. In this case, MSE can measure how similar the generated audio samples are to the target audio samples regarding their amplitude values.

Unlike MAE (see Section 2.1.3.2), which assigns equal weight to all errors, MSE penalizes larger errors more than smaller ones. This means that MSE is more sensitive to outliers and may not reflect the overall discrepancy between the generated and target audio samples well. Moreover,

MSE does not account for perceptual aspects of audio quality, such as timbre, pitch, or loudness. Therefore, MSE should be used with other metrics and evaluation methods, such as KL divergence (see Section 2.1.3.2), subjective listening tests, or qualitative analysis.

MSE can be applied to sound generation tasks in different ways, depending on the representation of the audio data. Similar to MAE, it can be applied to 1D raw audio and spectrograms.

It should be noted that MSE is subject to the same temporal issue as MAE. MSE may not be effective in identifying differences in timing accuracy. Therefore, it is crucial to employ other metrics and evaluation methods that specifically focus on timing aspects and perceptual quality when assessing audio produced by text-trained models.

Cross-Entropy Cross Entropy Loss is commonly used to evaluate DL models, especially in classification or sequence generation tasks. It measures the dissimilarity between predicted and target distributions by calculating the average negative log-likelihood of predicting each class or element correctly.

Cross Entropy Loss can be applied in audio generation tasks to produce discrete elements, such as musical notes or phonemes. For instance, if one aims to create music based on textual input with particular note sequences, Cross Entropy Loss can assess the prediction accuracy of each note at every time step.

Mathematically, given data samples x_i and their respective true labels y_i, where i ranges from 1 to n, Cross Entropy Loss can be calculated in the following way:

$$CE = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C} y_{ij}\log(p_{ij}) \tag{2.37}$$

This is where: The symbol C represents the number of classes or elements. The notation y_{ij} indicates whether sample x_i belongs to class j (or has element j). Additionally, p_{ij} represents the predicted probability that sample x_i belongs to class j (or has element j).

The objective is to minimize the Cross Entropy Loss during training, so that the generative model can learn to predict precise and coherent distributions over classes or elements.

Nonetheless, it is important to consider some limitations when using Cross Entropy Loss to evaluate audio generation systems. First, the model assumes independence between individual predictions within one sample. However, this assumption may not hold for sequential audio data where the context and dependencies between elements are critical. Second, the Cross Entropy Loss does not directly capture perceptual aspects of audio quality, such as timbre or tonality. Therefore, it is advisable to combine Cross Entropy Loss with other evaluation metrics, such as MAE, MSE, or subjective listening tests to achieve a comprehensive understanding of the generative model's performance.

To summarize, the Cross Entropy Loss is commonly employed as a loss function for evaluating generative models that involve discrete element generation. Although it is applicable for audio generation tasks with categorical outputs such as music note prediction, it should be combined with other evaluation methods to gain a more comprehensive assessment of accuracy and perceptual quality.

KL Divergence Kullback–Leibler (KL) divergence, also known as relative entropy, is a non-symmetric measure of the difference between two probability distributions. It is a mathematical quantity that quantifies the distance between two probability distributions.

In simple terms, KL divergence measures the difference between the probability distribution predicted by a model and the true underlying distribution of the data. KL divergence is commonly used to evaluate generative models.

KL divergence is calculated as the expectation of the logarithmic difference between the predicted probability distribution and the actual distribution. It is a scalar value, and the smaller the KL divergence, the closer the predicted distribution is to the real distribution.

The DL system can be trained to produce sounds near the target sounds in terms of their probability distribution by using KL divergence as a loss function. The concept is that the sound does not have to be alike the input, but only its distribution. Maximizing the log-likelihood between the generated output and the given input can be seen as minimizing the KL divergence [61].

Evidence Lower Bound (ELBO) Evidence lower bound (ELBO) is a lower bound on the log-likelihood of some observed data commonly used in variational Bayesian methods [13].

The ELBO is defined as follows:

$$ELBO = E_{Z\sim q}\left[\log \frac{p(X,Z;\theta)}{q(Z)}\right] \tag{2.38}$$

where X and Z are random variables with joint distribution $p(X,Z;\theta)$, θ are the parameters of the model, and $q(Z)$ is an approximate posterior distribution for the latent variable Z. The ELBO can be seen as the difference between two terms: the expected log joint probability of the data and the latent variables under the model and the entropy of the approximate posterior distribution.

The ELBO has several desirable properties. First, it is a lower bound on the log-likelihood of the data, $\log p(X;\theta)$, also known as the evidence. Meaning that the ELBO is a quantity that is always less than or equal to the log-likelihood of the data, which is the logarithm of the probability of the data given the model parameters. The log-likelihood of the data is also called the evidence because it indicates how well the model fits the data. The higher the log-likelihood, the more evidence we have that the model is suitable for the data. However, computing the log-likelihood of the data is often intractable. Therefore, the model can be optimized more easily using ELBO.

Second, it is a tractable objective function that can be optimized concerning θ and $q(Z)$. This allows us to perform variational inference, approximating the posterior distribution $p(Z|X;\theta)$ by finding the $q(Z)$ that maximizes the ELBO. This can be done using gradient-based methods, thus being used in machine learning systems.

Third, it can be decomposed into two significant components: the reconstruction term and the regularization term. The reconstruction term is the expected log-likelihood of the data given the latent variables under the model, $E_{Z\sim q}[\log p(X|Z;\theta)]$. It measures how well the model fits the data. The regularization term is the negative KL divergence (see section 2.1.3.2) between the approximate posterior and the prior distributions, $-D_{KL}(q(Z)||p(Z))$. It measures how close the approximate posterior is to the prior. The KL divergence is always non-negative, and it is zero if and only if $q(Z) = p(Z)$. Therefore, maximizing the ELBO encourages data fidelity and posterior regularization.

The ELBO can be applied to sound generation tasks using a deep generative model such as a VAE (see section 2.1.2.3). This model can be trained by maximizing the ELBO concerning its parameters and latent variables. The ELBO can then be used to evaluate the quality and diversity of the generated sounds by comparing them to the target sounds. For instance, the ELBO for a VAE can be written as:

$$ELBO = E_{Z\sim q_\phi(Z|X)}[\log p_\theta(X|Z)] - D_{KL}(q_\phi(Z|X)||p(Z)) \tag{2.39}$$

The first term is the reconstruction term, which measures how well the decoder network reconstructs the input sound X from the latent variable Z. The second term is the regularization term, which measures the proximity of the approximate posterior distribution to a prior distribution $p(Z)$.

By maximizing the ELBO, a model learns to generate realistic sounds similar to the input sounds regarding their conditional distribution while ensuring that the latent variables have a smooth and regular structure that facilitates interpolation and manipulation.

Model Evaluation Functions The second type of evaluation in this thesis involves assessing the wider impacts and implications of the developed audio generation model. Although metrics used for training models offline provide insights into performance, it is equally important to evaluate how a model affects aspects beyond its primary task. This includes considerations such as the environmental impact and inference time during generation, among others. Understanding these broader impacts ensures that the model not only performs well in its intended purpose but also operates with minimal negative consequences or trade-offs in other areas of the system. Conducting evaluations encompassing these factors will provide a more comprehensive understanding of how well-rounded and sustainable our audio generation system is.

Evaluating Energy Expended Evaluating the amount of energy expended by a deep learning model is crucial in developing and deploying these systems. With the increasing demand for machine learning applications and the complexity of deep learning models, energy efficiency has become a critical factor in designing and deploying deep learning systems.

With the growing concern for environmental sustainability, the energy footprint of deep learning models has become an essential topic in the field. Most of the recent advances produced by deep learning approaches rely on significant increases in size and complexity [32]. Such improvements are backed by an increase in power consumption and carbon emissions. The high energy consumption of deep learning models during both the training and inference phases significantly impacts the environment, and it is imperative to address this issue.

Therefore, evaluating the amount of energy a deep learning model expends is essential in ensuring its practicality and scalability. This is a crucial step in ensuring that the deep learning models developed today are not only accurate, but also energy-efficient and sustainable for future deployment.

This evaluation can be done in two ways: physically measuring the energy expended by the machines on both learning and inference time or by approximating given average numbers per neuron, for instance.

A good model is a compromise between accuracy and complexity. If the model trains significantly longer to train or infer and does not provide way better results, in the context of this research, the model is not much better than a simpler counterpart.

2.1.3.3 Data Embedding

Data embedding is the technique of converting data into numerical representations that capture its essential features and characteristics. For sound generation, both audio and text embedding is important.

Data embedding is essential for generative models. This happens for two reasons.

First, it allows the models to work on smaller and lighter representations. This is, for instance, taking an audio sample with 5 seconds sampled at 16 kHz, representing 80 000 entries. Encoding that into meaningful features might reduce this number to a few thousand or even hundreds of entries. This allows for faster training.

Second, these representations are meaningful in ways where the raw input is not. For instance, take text embedding. The idea is that words such as “pretty” and “beautiful” have similar representations, helping the model generalize. If the model were to check the words, letter by letter, it would have a hard time realizing that some words, like these, have a relation between them.

So, text and audio embedding is essential for the task. However, audio embedding possesses several challenges, such as dealing with high-dimensional and sequential data, preserving temporal

and spectral information, and ensuring robustness and interpretability. To handle this, both feature and learning-based methods can be applied.

Feature-based embedding methods extract predefined features from the raw audio data, such as spectral, temporal, or perceptual features. These features are then input to generative models or further processed to obtain lower-dimensional embeddings. One such example would be the application of the STFT to build a spectrogram (see Section 2.1.1). Feature-based embedding methods have the benefit of being simple and interpretable, but they may also lose some information or introduce noise during the feature extraction process.

Using neural networks or other machine learning techniques, learning-based embedding methods learn embeddings directly from the raw audio data. These methods can automatically discover relevant features from the data without relying on predefined criteria. Learning-based embedding methods have the advantage of being flexible and adaptive, but they may also require more computational resources or suffer from overfitting or underfitting issues.

Text embedding has been a solved problem since the days of Word2Vec [82]. This AE (see Section 2.1.2.1) model would be trained by getting the meaning of a word taking into account the word with whom the first appeared. This model makes possible the representation of a word through a vector of latent factors.

Nevertheless, the words cannot be merely embedded for the current issue. When a user adds a text input, embedding it is imperative. This represents the entire textual input. For example, computing the average of latent factors for each input would be a naive approximation.

Nevertheless, nowadays, this is mainly solved with transformers (see Section 2.1.2.3), namely the encoder part. This part of the transformer takes a whole string and outputs a vector representation, precisely what the task needs.

After the vanilla transformer, the one that gained more prominence was BERT [28] in 2018, which introduced the conditioning of the whole input for each word inputted, not only the words that appeared before, as the vanilla transformer did. Other later encoder transformers are based on BERT or tackle a different problem.

While there are many techniques for data embedding, recent advancements in the field have led to the development of specialized models for various modalities, such as CLIP [99] for images and MuLan for audio. Instead of embedding the text and the media separately and dealing with it afterward, media and text are embedded in the same space, meaning that a textual segment and a media sample representing the same textual segment should have similar latent factors.

MuLan MuLan [59] is a state-of-the-art music audio embedding model that aims to link music audio directly to unconstrained natural language music descriptions.

MuLan employs a two-tower parallel encoder architecture, meaning two completely independent neural architectures, using a contrastive loss objective that elicits a shared embedding space between music audio and text.

Each MuLan model consists of two separate embedding networks for the audio and text input modalities. These networks share no weights, but each terminates in 2-normalized embedding spaces with the same dimensionality. The contrastive loss objective minimizes the distance between matching audio-text pairs while the distance between mismatched pairs is maximized. This approach enables MuLan to learn a joint representation of music audio and text that captures their semantic relationships.

MuLan is trained using 44 million music recordings (370K hours) and weakly-associated, free-form text annotations. The resulting audio-text representation subsumes existing ontologies while graduating to true zero-shot functionalities. MuLan demonstrates versatility in transfer learning, zero-shot music tagging, language understanding in the music domain, and cross-modal retrieval applications.

2.1.4 Deep Learning Frameworks

DL frameworks have revolutionized the field of AI, enabling researchers and practitioners to efficiently develop and deploy complex neural networks for the multiple DL tasks. These frameworks provide a wide range of tools and techniques for building, training, and evaluating deep neural networks and have significantly accelerated the pace of progress in the field. This section explores some of the most popular DL frameworks, their key features and capabilities, and how they have been used to develop state-of-the-art generative AI models for audio synthesis from textual input.

Several DL frameworks are available, and they differ in several ways, including their programming languages, ease of use, and performance. However, to the best of the author's knowledge, there is no recent study on the performance of today's DL networks. The most recent is from 2017 [93].

2.1.4.1 TensorFlow

Developed by Google, TensorFlow [78] is one of the most widely used DL. It supports both Central Processing Unit (CPU) and graphics processing unit (GPU) computations and provides a variety of Application Programming Interfaces (APIs) for building different types of neural networks. TensorFlow is written in Python, but its core functionality is implemented in C++ for optimal performance.

TensorFlow is an interface for expressing ML algorithms and an implementation for executing them. It allows computations to be executed with little or no change on various systems, from mobile devices to large-scale distributed systems. The system is flexible and can express many different algorithms, including training and inference algorithms for deep neural network models.

TensorFlow can be used with various programming languages, including Python, JavaScript, C++, and Java. Python is the recommended language for TensorFlow, but other languages' APIs may offer some performance advantages. Other languages like Julia, R, Haskell, and others have bindings.

2.1.4.2 PyTorch

PyTorch is an open-source DL framework developed by Facebook [94]. It has gained popularity due to its ease of use and dynamic computation graph, which allows for more flexible and intuitive programming. PyTorch also supports CPU and GPU computations, and although it supports Python and C++, it has a Python-first approach, making it easy to integrate with other Python libraries.

PyTorch is a popular deep-learning framework that is easy to use and learn. It has a simple and intuitive API that makes it easy to learn and use. PyTorch is also flexible and can be used for various applications.

2.1.4.3 Keras

Keras is a Python library for building and training neural network models at a high-level. It offers a user-friendly interface, and it is commonly used for DL purposes [20]. Keras is built on top of lower-level libraries such as TensorFlow (see Section 2.1.4.1). It simplifies the creation and training of neural networks by abstracting away many low-level details. Keras enables fast neural network model creation by assembling pre-built building block layers. These building blocks consist of input layers, CNN layers, RNN layers, fully connected layers, activation functions, and other components.

Keras is often regarded as more user-friendly than TensorFlow, as it offers a high-level interface that hides many low-level details.

Keras provides a simplified API for constructing and training deep learning models. Keras includes a variety of utilities for manipulating data, such as data preprocessing, data augmentation, and data visualization, facilitating data manipulation.

2.1.4.4 Conclusions on Deep Learning Frameworks

Selecting a DL framework is critical in developing a DL project. It is vital for the development of this thesis that the best framework that can offer flexibility, ease of use, and optimal performance of the DL models is selected.

PyTorch uses a dynamic computation graph, which is created for each iteration in an epoch. In each iteration, the code executes the forward pass, computes the derivatives of output *w.r.t* to the network parameters, and updates the parameters to fit the given examples. After doing the backward pass, the graph is freed to save memory. A dynamic graph can be changed on the

fly, allowing for more freedom, easier debugging, and easier experimentation with architectures and hyperparameters during the model development process. Tensorflow, on the other hand, uses a static graph. There, the library creates and connects all the variables at the beginning and initializes them into a static (unchanging) session. This session and graph persist and are reused: it is not rebuilt after each iteration of training, making it more efficient and restrictive.

Furthermore, PyTorch offers a more straightforward and intuitive API compared to TensorFlow. This feature makes it easier for developers to write and debug code. Additionally, PyTorch offers more flexibility when creating custom layers and functions, which is impossible in Keras. This flexibility enables developers to create more complex and innovative models, which can lead to better performance. Keras offers a straightforward but too simple network, while low-level Tensorflow offers a complicated and convoluted API making it challenging to focus on the real problem. PyTorch offers a good balance between simplicity and features.

One of the significant drawbacks of Keras is the need for more flexibility when customizing DL models. Keras offers a limited set of pre-defined layers, making it difficult for developers to create complex custom layers and functions. This lack of flexibility limits the ability to make changes to the architecture of a network during the development process, which can hinder the performance of the final model. When one needs more custom layers, one has to resort to the Tensorflow jungle, making the development a hassle.

Both PyTorch and TensorFlow have good hardware support. However, PyTorch has a feature that distinguishes it from TensorFlow: data parallelism. PyTorch optimizes performance by using native support for asynchronous execution from Python. In TensorFlow, one has to manually code and fine-tune every operation to be run on a specific device to allow distributed training. Running on top of TensorFlow, Keras presents the same hardware problems as the latter.

These conclusions are in table 2.4. Given all of this, and considering that ease of use, debugging, and high customization are essential for this work, the clear choice is PyTorch. One should consider its use when this work uses practical terms.

Table 2.4: Comparison of PyTorch, Raw TensorFlow, and Keras

Feature	**PyTorch**	**Raw TensorFlow**	**Keras**
Graph computation	Dynamic	Static	Static
Ease of use	Moderate	Difficult	Easy
Debugging	Good	Difficult	Moderate
Customization	High	High	Moderate
Hardware support	Good	Moderate	Moderate

2.1.5 Data Generators

Creating new data from existing data is known as generative modeling. This technique has numerous applications across various media types, including images, text, and video. However, each

media type possesses unique characteristics, so generating them requires distinct approaches and techniques. Sound, for instance, presents a different set of challenges than other media types, and hence its generation necessitates diverse techniques.

This section aims to survey some of the state-of-the-art generators for media types that are not sound-based, primarily focusing on image generators. By doing so, one hopes to comprehensively understand the different approaches and techniques employed for generating various media types.

The discussion delves into the main ideas behind these generators, their strengths and limitations, and how they can be related to or inspired by sound generation. Through this exploration, this section highlights the diversity of methods used for generative modeling and how they can be adapted to different contexts.

2.1.5.1 PixelCNN Decoders

Van den Oord et al. introduced *PixelCNNs* as discussed in [88, 127]. PixelCNNs model high-dimensional discrete data, like images.

They employ the AE architecture as described in Section 2.1.2.3, where pixels are generated sequently while conditioning on prior pixels.

The goal is to estimate a distribution over images that can be used to compute the likelihood of images and thus generate new ones. The network scans the image one row at a time and one pixel at a time within each row, using masked convolutions (see Section 2.1.2.1). For each pixel, it predicts the conditional distribution over the possible pixel values given the scanned context.

Each image x is assigned a probability $p(x)$. To do this, if one considers each pixel sequentially, row by row, such that x_k is the pixel number k, the final probability is as follows.

$$p(x) = \prod_{i=1}^{h \times w} p(x_i | x_1, \ldots, x_{i-1}) \tag{2.40}$$

Where h is the height of the image and w is the width.

For multiple colors, one can condition the colors on one another. For instance, instead of having $x_1, x_2, \ldots$, there would be $x_1^R, x_1^G, x_1^B, x_2^R$, and so forth.

2.1.5.2 DALL-E

In 2021, the initial version of OpenAI's DALL-E was launched, which introduced zero-shot text-to-image generation [101]. Zero-shot refers to the ability to generate data from textual descriptions without any specific training on those exact combinations of text and images. In other words, DALL-E's training is based on a rich variety of image-text pairs, which enables it to generate new

Table 2.5: Comparison of Data Generators

Generator Name	**Main Idea**	**Strengths**	**Limitations**
PixelCNN [88]	Uses autoregressive connections to model images pixel by pixel.	Fast and parallelizable training, high resolution and diversity, interpretable latent space.	Slow and sequential sampling, limited global coherence, difficulty in conditioning on high-level features.
DALL-E [101]	A transformer language model that creates images from text descriptions, using a dataset of text–image pairs.	Can generate novel and creative images, combine concepts in plausible ways, render text and apply transformations.	Requires large amounts of data and compute, may produce harmful or biased outputs, not publicly accessible.
Stable Diffusion [106]	A latent diffusion model that creates images from text descriptions, using a dataset of text–image pairs and a pretrained CLIP model.	Can generate realistic and detailed images, is lighter than previous state of the art models.	Requires fine-tuning for specific domains, may produce artifacts or inconsistencies, sensitive to text prompts.
GLIDE [85]	Guided diffusion-based approach for text-conditioned image generation.	High-fidelity image synthesis, classifier-free guidance preferred for photorealism and caption similarity, text-driven image editing capabilities.	Difficulty generating images for complex or unusual text prompts, slow generation speed.
DALL-E 2 [100]	An improved version of DALL-E that generates more realistic and accurate images using a diffusion model.	Has better results than DALL-E and is faster.	Still requires large amounts of data and compute, may still produce harmful or biased outputs, still not publicly accessible.

images from textual inputs, relying solely on this training. This issue closely relates to the one addressed by the present research.

The overall structure is easy to comprehend. It primarily comprises of two phases. In the initial phase, image representations or features are learned using a discrete variational autoencoder (dVAE). In this context, a dVAE is an enhanced version of the classic VAE (see Section 2.1.2.3) that employs discrete latent variables in lieu of continuous ones. Discretizing the latent space allows for explicit control and manipulation of generated images in the synthesis process using dVAEs.

On the other hand, the second stage produces image representations from textual descriptions using a transformer model, as detailed in Section 2.1.2.3.

Figure 2.23 illustrates this macro-architecture.

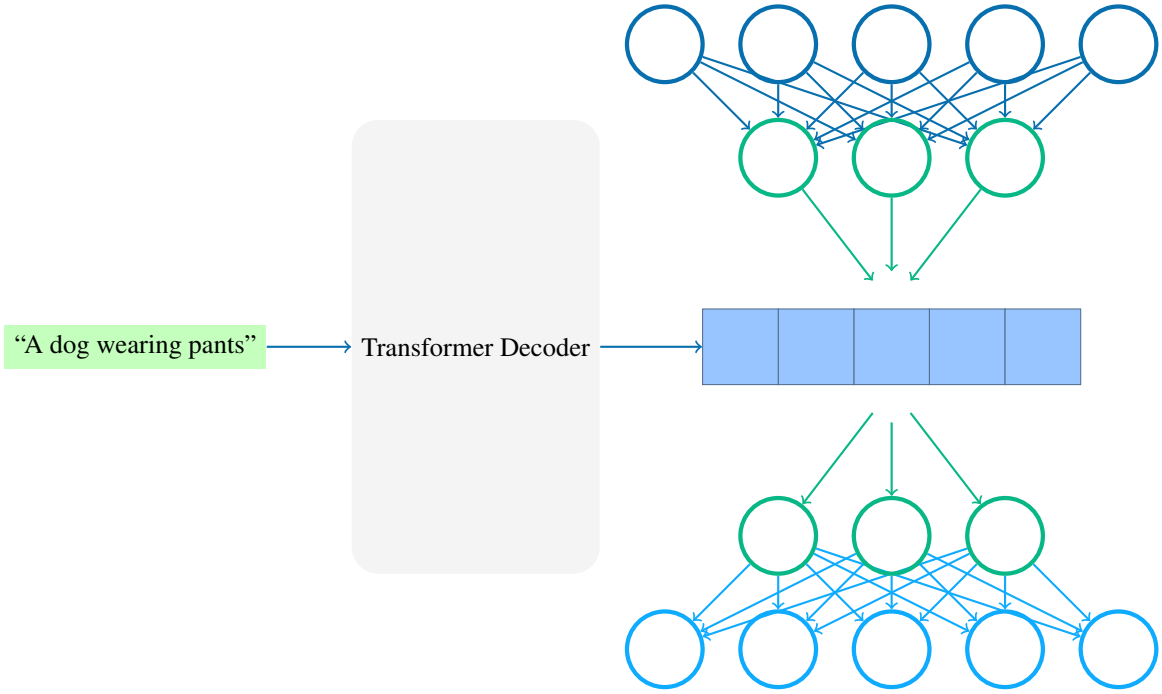


Figure 2.23: **Dall-E macro architecture** — With a green background, one can see textual operations. The complex on the right is a dVAE. Upon inference, the transformer’s output is used as the latent feature values of the dVAE.

The discrete variational autoencoder (dVAE) trained compresses images into a 32×32 grid of image tokens. Each token assumes one of 8192 possible values. This reduces the size of the image, improving the performance without significant degradation in visual quality.

At the second stage, the text is first encoded using byte pair encoding (BPE). BPE is a compression technique for representing text using a new, single symbol to replace frequent pairs of characters or symbols. The objective of BPE is to merge the most frequent pairs of consecutive symbols, such as letters or bytes, in the given text, until a specific number of merge operations is reached through an iterative process. As a result, this process generates a modified set of symbols that represents the original text but with a smaller vocabulary.

Applying BPE enables effective capture of repeating patterns and combinations within the text while achieving better compression than traditional character-level encoding. This encoding step is vital in preparing textual input for subsequent stages and helps enable more efficient processing and modeling.

After the encoding, the resulting symbols are passed as input to a transformer trained to generate the tokens corresponding to the image tokens.

To generate completely new images, the process would be as follows:

1. Encode the text into BPEs.
2. Pass the encoded text to the trained transformer encoder; This outputs image tokens.
3. Pass these tokens to the decoder of the dVAE trained in stage 1; This outputs the image.

2.1.5.3 Stable Diffusion

Stable Diffusion [106] rose from the idea of democratization high-resolution images. According to the original paper, high-end models demand hundreds of thousands of dollars to be trained solely due to their complexities. These models generally operate directly on the pixel space. The training and evaluation of such models necessitate significant computational resources, which are typically only available to the largest companies, thereby contributing to significant carbon footprints.

They apply the diffusion process (see Section 2.1.2.3) in pre-trained AEs' (Section 2.1.2.1) latent space to optimize the diffusion model practice instead of directly on the pixel space.

For this stable diffusion model, training is separated into two phases: the training of the AE and the training of the diffusion model.

One needs the text and Gaussian noise to infer new images given text. The text is embedded using a transformer (see Section 2.1.2.3). In order to increase the relation between the text and the generated image, the text embeddings for an empty string are also generated.

The model takes the initial noise, text and audio embeddings, and timestamp as inputs. Through inverse diffusion process, it generates two new images: one conditioned on the real embeddings, and another with no embeddings (empty string embedding). The image generated by the real embeddings is a synthesis conditioned on the given text and audio input. On the other hand, generating an image without any embeddings produces a random image unconditioned by the provided text or audio. At each step of diffusion process, the model compares the two images to evaluate their differences. By increasing this difference, the final output image becomes conditioned on the initial textual input to a greater extent.

The U-Nets in the diffusion process use cross-attention mechanisms for their respective embeddings (refer to Section 2.1.2.1). Cross-attention extends attention mechanisms used in neural networks. Self-attention captures relationships within a single sequence or embedding, while cross-attention captures dependencies between different sequences or embeddings.

In this study, the U-Nets use cross-attention to allow information exchange and interaction between audio features and other modalities, including text embeddings or image representations. Using cross-attention in the diffusion process improves the model's ability to consider local and global contextual cues from multiple sources during audio synthesis.

Using cross-attention allows different parts of the network to attend to each other's information, enabling better integration and coherence across various modalities involved in generating soundscapes from textual input. This comprehensive modeling improves the overall performance.

After the diffusion process is completed, the decoder takes the generated image embeddings and generates a new image.

This process can be seen in Figure 2.24.

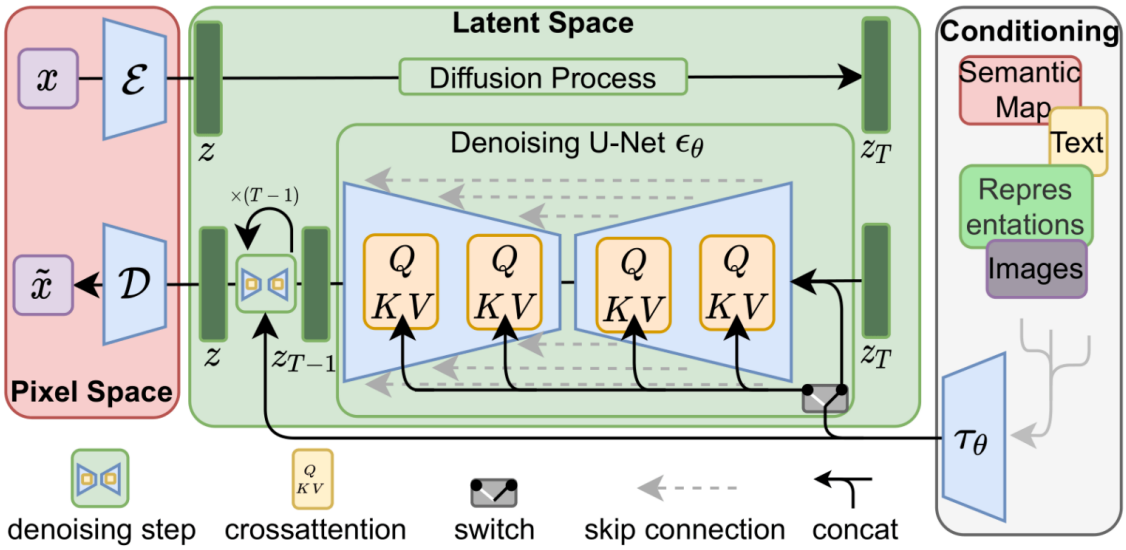


Figure 2.24: **Stable diffusion architecture** — The Figure was taken from the original paper. On the top left, the original image x is encoded with the AE, and the diffusion process happens with the encodings. The text (or another data kind) is encoded with a transformer τ_θ, and these encodings are applied with attention to the denoising U-Nets. This denoising U-Net is applied T times before the decoder transforms the encodings into an actual image in the pixel space again.

2.1.5.4 GLIDE

The Guided Language to Image Diffusion for Generation and Editing (GLIDE) model is a state-of-the-art approach for generating high-fidelity synthetic images from free-form natural-language text prompts. The model is based on a guided diffusion-based approach, which is the first attempt

by OpenAI at text-conditioned image generation using guided diffusion. The approach involves two types of guidance strategies during model training: classifier-free guidance [54], which relies solely on the model's knowledge, and Contrastive Language–Image Pre-training (CLIP) guidance which uses a pre-trained CLIP model [99] to provide guidance based on caption matching.

The experimental results of the GLIDE paper demonstrate several benefits of the proposed approach. Human evaluators preferred images generated with classifier-free guidance over CLIP guidance regarding photorealism and caption similarity. Samples from the 3.5 billion parameter GLIDE model were also found to outperform DALL-E (see Section 2.23) samples according to human evaluations. Additionally, the model can perform text-driven image editing tasks beyond zero-shot image generation from text prompts. Text-driven image editing refers to editing existing images according to text prompts, such as changing attributes or objects within an image as directed by the text.

Despite its success, the GLIDE model has some limitations. It fails to generate images for some complex or unusual text prompts. Moreover, the model's generation speed is slow, taking several seconds to generate one image on a flagship GPU. Possible solutions to address these limitations include improving the model architecture, optimization techniques, and combining GLIDE with faster GAN-based methods.

Contrastive Language–Image Pre-training (CLIP) refers to a model trained to determine if an image and text caption match. It consists of a transformer-based text encoder (see Section 2.1.2.3) and a convolutional neural network-based image encoder (see Section 2.1.2.1). The text encoder produces an embedding of the text, and the image encoder produces an embedding of the image. These embeddings are then compared, and during training, the model learns to produce similar embeddings for matching image-text pairs and dissimilar embeddings for mismatching pairs. This contrastive learning approach allowed CLIP to learn cross-modal understanding between text and images in an unsupervised manner. The pre-trained CLIP model can provide additional guidance to other text-to-image models, such as GLIDE, by scoring how well-generated images match given text prompts.

2.1.5.5 DALL-E 2

DALL-E 2 is a model proposed by researchers at OpenAI capable of generating images given a textual prompt [100]. This model can also modify given images, but this use case is not so interesting for the work present in this thesis.

The model consists of two blocks: the prior and the decoder. The prior converts captions into a lower-level representation, while the decoder turns this representation into an actual image.

They use CLIP [99] and GLIDE (see Section 2.1.5.4). For DALLE-2, the text is initially embedded using CLIP embeddings. Then, the role of the prior is to translate these embeddings into

embeddings related to an image and not the text itself. In other words, create an image representation with textual embeddings. For this, the researchers tried an AR and a diffusion model. The diffusion one yielded better results (see Sections 2.1.2.3 and 2.1.2.3). The decoder then takes the generated image representation and generates the image. The whole process can be seen in Figure 2.25.

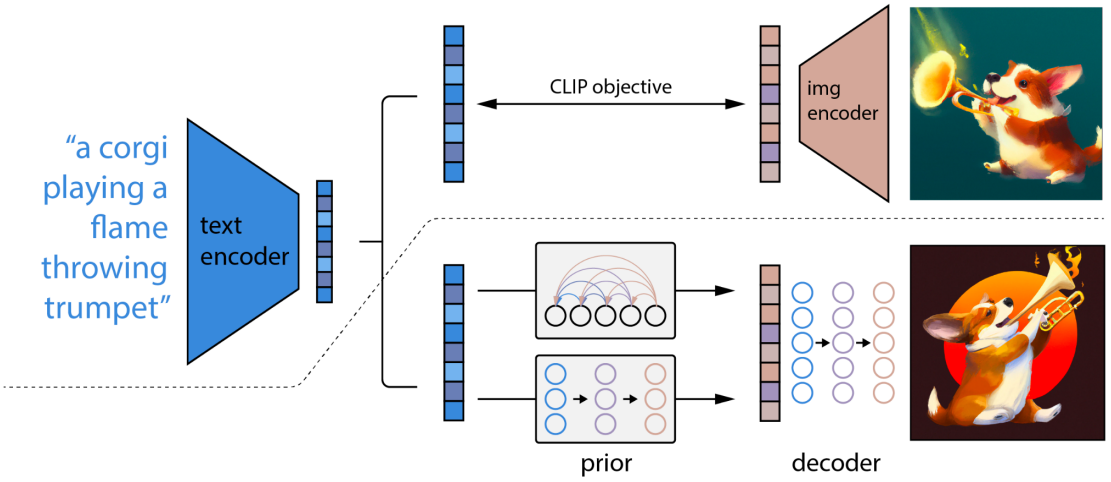


Figure 2.25: **DALL-E 2 architecture** — The image was taken from the original paper. Above the dotted line, the CLIP training process is depicted, where given textual and image embeddings, the CLIP learns to translate one into the other. Below the dotted line, a text-to-image generation process is represented: a text embedding is first fed to the model that produces the image embedding. Then, this embedding is used to condition the diffusion model GLIDE which produces a final image.

A model is also possible without the prior by passing the textual embeddings directly to the decoder. However, while the results were okay, they were way better with the generated image embeddings.

The decoder creates 64×64 images, but another network learns to upsample images until 1024×1024. Without this, generating high-resolution images with the decoder would make the whole operation incredibly heavy.

A significant problem of this model (and others presented here proposed by big companies) is that it needs hundreds of millions of images and an incredible amount of computation power to perform well. This highlights the importance of research toward openly accessible models such as stable diffusion (see Section 2.1.5.3).

2.2 Related Work

Traditionally, sound designers relied on manual labor to create audio, which involved recording and editing real-world sounds, mixing, and adding sound effects [118]. Creating high-quality sounds is challenging, costly, and time-consuming, requiring specialized skills and resources.

Hence, it engenders a notable impediment to the creation of soundscapes or any type of sound at scale [12, 120], namely in light of its growing popularity and consumption within podcasts, movies, and video games.

Consumer data reported in 2021 showed compelling evidence regarding the listening habits of individuals within the United States of America. The findings indicate a substantial growth in podcast listenership over the past decade, with 41% of Americans aged 12 or older having engaged with podcasts in the preceding month and 28% within the last week. Moreover, at the beginning of the same year, a notable 68% of Americans aged 12 and above had indulged in online audio consumption within the previous month, while 62% had done so within the preceding week [103].

To overcome the aforementioned limitations, algorithmic audio generation has emerged as a promising solution that streamlines its creation altogether. Focusing on soundscapes, preceding 2018, prevailing models for their generation primarily revolved around statistical methods, featuring prominent employment of ML techniques with feature engineering. For a comprehensive overview of techniques employed before the era of DL, reference can be made to the review papers by Alias et al.[8] and Kalonaris et al.[66]. Noteworthy efforts at the feature engineering level are exemplified by Fernandez et al. [37], who represent sounds as high-level features, such as musical sheets, as an approach to generating musical sounds.

DL models for audio generation aim to produce high-quality audio signals by learning from existing audio data. Typically, these models consist of three main components: translation of the sound signal into a compressed representation, generation of a new representation from previous data, and translation back into an audio signal.

The first component of the model involves transforming the original sound signal into a Mel-spectrogram (or other) representation (see Section 2.1.1), which is more compact and more accessible to process than the raw audio signal.

The second component involves generating new low-resolution representations from previous representations, such as spectrograms or feature vectors [71]. This is typically done using deep generative architectures. These models are trained on existing sound data to learn the target representation distribution and generate new, high-quality representations.

The final component of the model involves translating these representations back into an audio signal. These algorithms are called *vocoders* (for more, see Section 2.2.3). This component aims to produce high-quality audio signals that closely resemble the original sound data used to train the model.

The purpose of this section is to review the related work in this area. First, it examines the methods for generating traditional soundscapes, as described in Section 2.2.1. It then examines sound generation using machine learning. State-of-the-art models focus primarily on either using unsupervised sound generation techniques, as discussed in Section 2.2.2, or generating sounds from

internal representations, known as vocoders and discussed in Section 2.2.3, or creating an end-to-end system, as discussed in Section 2.2.4.

2.2.1 Traditional Soundscape Generation

Feature engineering methods for soundscape generation typically adopt a threefold strategy to resynthesize (and extend) a short soundscape recording provided by the user:

1. Segmentation,
2. Feature extraction and modeling, and
3. resynthesis of a given environmental sound.

Statistical models adopting stochastic processes or pattern recognition methods were commonly applied to model and recreate a given soundscape recording with a degree of variation while maintaining its structure. Generated soundscapes relied on the similarity among audio segments to create smooth transitions [58].

2.2.1.1 Scaper

Searching through the academic search engines, one finds that the most cited software for soundscape generation is *Scaper* [113].

Scaper is an open-source software library for soundscape generation designed to facilitate the creation of synthetic sound environments. It is a tool that allows users to simulate complex soundscapes, including urban, natural, and interior spaces, and investigate how various sound sources interact in these environments.

Scaper implements a modular soundscape generation framework based on basic sound-generating objects or "sound sources". These sound sources can represent simple sounds such as bird songs, human speech, or car horns, or more complex sounds like those produced by a crowd of people or a construction site. The user can specify the attributes of each sound source, such as its location, volume, and duration, and can adjust these parameters in real-time to create a dynamic soundscape.

One of the key features of Scaper is its ability to generate synthetic soundscapes that are diverse and statistically representative of real-world environments. To achieve this, the library implements various sound-generating algorithms that can be used to create sounds that are randomized yet realistic. For example, the library can generate sounds similar to real-world sources but with variations in volume, pitch, and timbre to avoid repetition and create a more diverse soundscape.

2.2.1.2 SEED

SEED is a system that addresses the formidable task of resynthesizing environmental sounds, such as city ambiances or nature scenes [12]. SEED aims to provide a solution that not only extends the

duration of environmental sounds but also provides precise control over the degree of variation in the output. This control over variation is critical in applications where maintaining the authenticity and coherence of the audio environment is essential.

SEED is built on a tri-partite architecture consisting of three main modules: segmentation, analysis, and generation.

In the segmentation module, SEED performs the task of dividing the input audio into segments. This segmentation process is based on detecting spectral stability between frames. Spectral stability is a measure of how similar the frequency spectrum is between consecutive frames. When this stability falls below a certain threshold, it signals a change in the underlying sound source or event, prompting the placement of a segment boundary. This approach ensures that the resynthesized audio remains cohesive and retains its natural flow.

The Analysis module has two main processes. First, it extracts several audio features that capture both the sonic and temporal characteristics of the segments. These features are then clustered into a discrete "dictionary" of audio classes, effectively reducing the feature space to a finite set. At the same time, the module builds a transition table that records the sequences of these audio classes. This table is used to determine the probability that one class follows another.

In addition, the analysis module computes a concatenation cost matrix that quantifies how smoothly two segments can transition from one to the other. This matrix is computed by comparing the features at the segment boundaries. A lower cost indicates a smoother transition, while a higher cost indicates a more abrupt change.

In the Generation module, SEED generates new audio by searching for segment sequences that meet certain criteria. To achieve this, SEED references the transition table to determine viable next classes based on the current class in the audio sequence. It then assembles segments belonging to these classes and selects the one with the lowest concatenation cost. Notably, SEED applies a temporary cost penalty to recently selected segments to encourage diversity in the generated audio.

2.2.1.3 Physics-Based Concatenative Sound Synthesis

In the current development of virtual environments, the generation of audio content has been the subject of extensive research. One prominent approach in this area is concatenative sound synthesis (CSS), a method that creates novel auditory experiences by assembling segments of pre-existing sounds from a given database, often referred to as "audio units".

A recent scientific paper by Magalhães et al. presents an innovative CSS framework based on physics-based principles for virtual reality [75]. This framework consists of two main components, namely the "Capture Component" and the "Synthesis Component".

The capture component of the framework is responsible for capturing essential data during interactions with virtual objects. This includes physics simulation data, haptic feedback data, position

sensor data, and audio data. In particular, the physics data includes critical information such as collision points, velocities, impulses, and normals, among other parameters. The haptic and audio data are derived from real-world interactions with a variety of materials. This capture process culminates in the creation of a multimodal corpus of annotated audio units, which serves as the foundational resource for subsequent synthesis efforts.

The synthesis component of the framework uses the captured data to orchestrate the synthesis of auditory and haptic feedback by concatenating audio units extracted from the corpus. This unique mapping between physics data and audio units ensures congruence between user interactions and the resulting sensory feedback. For example, when a user applies a certain force and angle to interact with a virtual metal object, the synthesis component selects an audio unit recorded from a similar interaction with a real-world metal object.

At runtime, the framework relies on the target physics vectors to guide the selection of audio units, thus generating congruent auditory and haptic experiences. An overlap-add phase vocoder is used to concatenate the audio segments, while temporal repetition penalties are incorporated to ensure smooth transitions between these segments.

2.2.2 Unsupervised Sound Generation

This Section focuses on models that address unsupervised or self-supervised training through learning sound features and their distributions without relying on explicit labels or annotations. In unsupervised sound generation, models learn from unlabeled audio data to capture underlying patterns and structures. This enables the generation of novel sound samples and the representation of latent features. This approach is particularly valuable when labeled datasets are scarce or expensive to acquire. Next, this discussion covers notable models in this area, selected based on their suitability for generating audio.

2.2.2.1 WaveGAN

GANs have a notable impact on generating coherent images at the local and global levels, as discussed in Section 2.1.2.3. A model based on GANs called WaveGAN [31] was proposed in 2019, to synthesize waveforms in an unsupervised manner. The model modifies the transposed convolution operation, used in deep convolutional generative adversarial networks (DCGANs) for image generation, to capture waveform structure at different timescales.

WaveGAN modifies the transposed convolution operation in DCGANs, expanding conventional GANs to encompass image generation tasks and precisely capture the structure of audio signals of varying timescales. This model uses lengthier, one-dimensional filters of 25 units in place of two-dimensional filters with dimensions of 5×5. The model also upsamples each layer by a factor of 4, as is done in traditional DCGANs. Despite these modifications, WaveGAN has the same number of parameters, numerical operations, and output dimensionality as DCGANs have.

The experiments conducted on WaveGAN show that it can synthesize one-second slices of audio waveforms with global coherence, which is suitable for sound effect generation. The model also learns to produce intelligible words when trained on a small-vocabulary speech dataset without labels.

The success of WaveGAN in generating coherent audio signals demonstrates that GANs can generate high-quality sounds. This work opens up new possibilities for unsupervised synthesis of raw-waveform audio, such as music and speech. It also suggests that GANs can learn to capture the structure of signals across various timescales, which is crucial for generating realistic audio.

2.2.2.2 Generative Transformer for Audio Synthesis

In this work, Verma and Chafe [129], proposed in 2022, explore an alternative architecture using transformer networks (see Section 2.1.2.3), which have shown great success in sequential modeling tasks such as language translation.

The authors develop a generative transformer model for raw audio waveforms. The model is trained to autoregressively predict the next audio sample by attending over previous context samples. Specifically, the input waveform is split into overlapping frames and embedded into a latent space. A series of multi-headed causal self-attention layers then learn to focus on relevant parts of the input context to predict the subsequent sample distribution.

To retain information about the relative sample positions, positional encodings are added. Training deeper models is facilitated by layer normalization and residual connections. In a neural network, residual connections (also known as skip connections) enable the direct flow of information from one layer to subsequent layers. The use of residual connections helps to mitigate the vanishing gradient problem and allows for more effective gradient propagation during training. The inclusion of these connections enables the model to learn new representations at each layer and retain useful features from previous layers, resulting in improved performance and faster convergence. Self-attention provides the model with flexibility and frees it from the fixed topology of convolutions in other models such as WaveNet (Section 2.2.3.1).

During training, previous samples are fed as input to the model to predict the next sample, optimized with cross-entropy loss (see Section 2.1.3.2). The authors quantitatively evaluate next-step sample accuracy and find that the transformer architecture can outperform WaveNet baselines substantially.

2.2.2.3 wav2vec 2.0

The wav2vec 2.0 model comprises three essential components: a convolutional feature encoder, a Transformer network, and a quantization module. This model was originally introduced in the 2020 paper by Baevski et al. [10] and designed for speech generation tasks. For further details on the convolutional layer and the Transformer network, refer to Sections 2.1.2.1 and 2.1.2.3.

Quantization is the process of discretizing continuous values into a finite set of discrete symbols or codes, particularly in the context of generative models. This method is comparable to the technique used in the VQ-VAE (refer to Section 2.1.2.3) In this technique, the input data is mapped to a limited number of discrete codebook entries.

For convolution, the feature encoder takes the raw audio waveform as input and generates a sequence of speech representations underlying it. This consists of several convolutional blocks, with each block including 1D temporal convolution and layer normalization. Wide kernels (*e.g.,* 10ms) are used in the convolutions and progressively reduce the resolution of the input to extract hierarchical features.

The output of the feature encoder is fed into a transformer network to build contextualized representations. For encoding positional information specific to speech generation tasks, we use a convolutional layer instead of absolute positional embeddings. The self-attention mechanism enables each time step to consider all other time steps, thus capturing long-range dependencies in the sequence. Several Transformer layers extract higher levels of contextual abstraction.

A quantization module is applied to the output of the feature encoder. It discretizes the continuous latent representations into a finite inventory of speech units. Multiple codebooks are maintained, and concatenating selections from each codebook construct discrete units.

After pre-training on unlabeled speech, the model is fine-tuned on transcribed speech for speech recognition by adding a randomly initialized output layer. Various augmentations are used during fine-tuning to improve robustness.

The key innovations are the joint training of discrete speech units and contextualized representations in a completely self-supervised fashion. Experiments demonstrate strong performance even with just minutes of labeled data, highlighting the benefits of pre-training on large unlabeled corpora.

2.2.2.4 SoundStream

SoundStream is a neural audio codec proposed in 2021 [136] that can efficiently compress speech, music, and general audio. A codec is software or hardware that compresses and decompresses audio signals. The model architecture consists of a fully convolutional encoder/decoder network and a residual vector quantizer.

The fully convolutional encoder receives a time-domain waveform as input. It produces a sequence of embeddings at a lower sampling rate, which is then quantized by the residual vector quantizer (RVQ). The fully convolutional decoder then receives the quantized embeddings and reconstructs an approximation of the original waveform. Both the encoder and decoder use only causal convolutions, so the overall architectural latency of the model is determined solely by the temporal resampling ratio between the original time-domain waveform and the embeddings.

While there are similarities between SoundStream and a standard AE (see Section 2.1.2.1) in terms of the encoder-decoder architecture, SoundStream includes additional components such as the RVQ and the use of structured dropout for variable bitrate compression.

A residual vector quantizer (RVQ) is a vector quantization method. It is a variant of the traditional vector quantization method present, for instance, in VQ-VAEs (see Section 2.1.2.3). In an RVQ, the input data is first transformed into a lower-dimensional space using a neural network encoder. The resulting embeddings are then quantized using a codebook of fixed-size vectors, where each input embedding is assigned to the nearest codebook vector. However, instead of encoding the input embedding directly as the index of the assigned codebook vector, an RVQ computes the difference between the input embedding and the assigned codebook vector, known as the residual. The residual is then quantized using a second codebook, and the indices of both codebook vectors are transmitted as the compressed representation.

Using residual vectors in RVQs allows for better compression performance than traditional vector quantization methods. It captures the fine details of the input data that may be lost during quantization. In SoundStream, the RVQ is used to quantify the embeddings produced by the fully convolutional encoder, enabling efficient audio compression at low bitrates while maintaining high audio quality.

2.2.3 Vocoders

DL vocoders are neural network models that have the ability to generate artificial audio [81]. These models employ deep learning networks to learn the mapping between the input and waveform data directly. There is no reliance on any predefined model or feature extraction method. This approach has the ability to capture complex nonlinear relationships between input and output representations that are difficult to be modeled analytically.

There are different types of DL vocoders, depending on the input and output representations they use. Some use Mel-spectrum features as conditioning inputs, while others do not require explicit features and directly generate raw waveform samples.

These models can achieve high quality and naturalness of audio synthesis, but they also face some challenges that limit their applicability. One challenge is the high computational cost of generating raw waveform samples at high sampling rates, which requires many computation and memory resources. This limits the scalability and efficiency of these models for real-time applications. Another challenge is the need for high-quality audio data with consistent annotations. This makes training these models with sufficient data diversity and coverage difficult. A third challenge is the generalization problem of these models, which tend to overfit the training data.

2.2.3.1 WaveNet

WaveNet is a generative neural network developed by DeepMind in 2016. It uses a unique architecture based on dilated causal convolutions to generate raw audio waveforms [87]. It implements the PixelCNN (see Section 2.1.5.1) model for sound and follows an AR architecture (see Section 2.1.2.3) with the predictive distribution for each audio sample being conditioned on a window of previous ones.

WaveNet's structure allows it to process input sequences in parallel, enabling it to model long context dependencies, even with thousands of timesteps. It uses a series of dilated convolutional layers, where the dilation rate is increased with each layer, which effectively increases the receptive field of the network without increasing the number of parameters.

A dilated convolution happens when the filter is applied over an area larger than its length by skipping input values with a specific step [87]. This architecture can be seen in Figure 2.26.

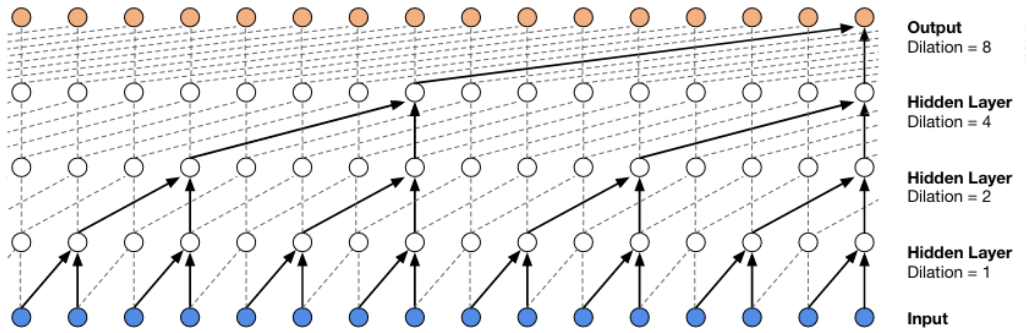


Figure 2.26: **WaveNet** — This illustration was taken from [87]. It shows the idea behind WaveNet, applying dilated convolutions to AR models.

This structure enables WaveNet to capture long-range dependencies in the input sequence, which is crucial for generating high-quality audio. If an RNN (see Section 2.1.2.1) sees only one input sample at each time step, WaveNet has direct access to multiple input samples [61]. For example, in speech generation, WaveNet can use its sizeable receptive field to model the relationship between a word spoken early in a sentence and its pronunciation later in the sentence.

WaveNet uses a softmax activation function at each output node to produce a probability distribution over the possible values at each time step. During training, the network is fed sequences of input data and their corresponding ground truth values. The model's parameters are adjusted so that its outputs match the ground truth as closely as possible.

WaveNet can use its trained parameters to generate new sequences by sampling from its output probability distribution during generation. This allows it to generate diverse and high-quality outputs, such as realistic human speech or written text, by combining its learned representations of the underlying data distribution with a small amount of randomness.

The input of WaveNet is usually a Mel-Spectrogram (or other representations), and the output is the sound signal.

WaveNet can be conditioned on, for instance, text for TTS settings by feeding extra information about the text itself (*e.g.* embeddings). If a model is not conditioned on text, it generates random sounds without any global structure behind it.

The results were astonishing. "A single WaveNet can capture the characteristics of many different speakers with equal fidelity, and can switch between them by conditioning on the speaker identity. When trained to model music, we find that it generates novel and often highly realistic musical fragments." [87].

Even though this model is good at learning the characteristics of sounds over brief periods, it struggles with global latent structure. They are also very slow for training and inferring [122].

2.2.3.2 WaveNet Variants

The WaveNet model has emerged as a powerful tool for generating high-quality audio waveforms, particularly for speech and music applications. However, its architecture, which employs dilated convolutions and deep residual networks, can be computationally intensive and challenging to train. To address these limitations, several WaveNet variants have been proposed in recent years that aim to reduce the complexity of the model while maintaining its effectiveness.

One such variant is **WaveRNN** [65], which employs a single RNN (see Section 2.1.2.1) to approximate the dilated convolutions in WaveNet. This approach significantly speeds up training time while maintaining the quality of the generated audio. Another variant, FloWaveNet [68], employs a flow-based generative model (Section 2.1.2.3) that allows for efficient training with only one training stage while producing high-quality audio. Additionally, Fast WaveNet [91] employs a caching mechanism to reduce the computational cost of the model while maintaining an AR structure.

These WaveNet variants are unique in their architectures and training procedures but share the goal of making audio generation more efficient and accessible. While these models are primarily focused on speech and music generation, they can be adapted to other types of audio data. Ongoing research in this area may explore further optimization of these models, integration with other models, and application to new domains.

2.2.3.3 MelGAN

According to Kumar et al. in 2019, in their *MelGAN* paper [73], audio generation with GANs is possible although a challenging task (see Section 2.1.2.3). Previous studies in this field have encountered difficulties generating coherent raw audio waveforms using GANs. Nonetheless, Kumar et al. demonstrate in their *MelGAN* paper that introducing certain architectural changes makes it feasible to train GANs to generate high-quality and coherent audio waveforms reliably.

The generator of MelGAN is a fully convolutional feed-forward network that takes a Mel-Spectrogram as input and generates a raw waveform as output. This approach allows for efficient and parallelized processing of audio data.

The decoder takes the waveform and decides whether it is a realistic sound. The decoder is not a single neural network but a multi-scale architecture with three discriminators (D1, D2, D3). These discriminators have identical network structures but operate on different audio scales. D1 operates on the scale of raw audio, while D2 and D3 operate on raw audio downsampled by a factor of 2 and 4, respectively. The use of multiple discriminators at different scales is motivated by the fact that audio has structure at different levels.

MelGAN proved itself way faster than other architectures such as WaveNet (see Section 2.2.3.1) with comparable results (for inference, roughly thirty-six thousand times faster than WaveNet), given its reduced number of parameters.

2.2.3.4 GANSynth

GanSynth, presented in 2019, [35] is a GAN (see Section 2.1.2.3) that uses log-magnitude spectrograms and phases to generate coherent waveforms. Compared to directly generating waveforms with stridden convolutions, the use of spectrograms and phases has been shown to produce better results.

The study focuses on the NSynth [36] dataset, a collection of 300 000 musical notes from 1 000 different instruments.

The model first samples a random vector z from a spherical Gaussian distribution. This vector is passed through a stack of transposed convolutions, which upsample and generate output data $x = G(z)$. This generated data is then fed into a discriminator network, which uses downsampling convolutions to estimate a divergence measure between the real and generated distributions.

The architecture of the discriminator network mirrors that of the generator, which allows for a more efficient training process. Optimizing the divergence measure allows the generator to produce spectrograms and phases that resemble actual musical notes more closely.

The study results demonstrate that GANs outperform WaveNet (see Section 2.2.3.1) baselines on automated and human evaluation metrics and can efficiently generate several audio orders of magnitude faster than their AR counterparts.

2.2.3.5 HiFi-GAN

Proposed in 2020, *HiFi-GAN* [71] is a GAN (see Section 2.1.2.3) model that combines efficiency and high-fidelity speech synthesis. HiFi-GAN achieves this by leveraging the periodic patterns

inherent in speech audio, demonstrating that modeling these patterns is crucial for enhancing sample quality. The model includes a generator and two discriminators, trained adversarially, and two additional losses for improving training stability and model performance.

The generator is a fully CNN (see Section 2.1.2.1) that takes Mel-Spectrograms as input and upsamples them through transposed convolutions, matching the temporal resolution of raw waveforms. The discriminators are the multi-scale discriminator (MSD) and a multi-period discrimintor (MPD). MSD evaluates the audio sequence on different scales using a mixture of three convolutional sub-discriminators with different average pools. At the same time, MPD consists of small sub-discriminators that capture different implicit structures of input audio by looking at different parts, accepting only equally spaced samples of input audio with different periods.

HiFi-GAN's performance is evaluated using a subjective human evaluation (mean opinion score (MOS)) on a single speaker dataset, which shows that the proposed method exhibits similarity to human quality. The model achieves a higher MOS score than WaveNet (see Section 2.2.3.1).

Importantly, HiFi-GAN achieves this high-quality synthesis efficiently. Specifically, the model generates 22.05 kHz high-fidelity audio 167.9 times faster than real-time on a single V100 GPU, demonstrating superior computational efficiency compared to AR and flow-based models. Moreover, a small-footprint version of HiFi-GAN generates samples 13.4 times faster than real-time on CPU with comparable quality to an AR counterpart.

2.2.4 End-to-End Models

Audio synthesis is the task of producing artificial audio from text or other kinds of data. Traditionally, audio synthesis systems consist of multiple stages, such as a data analysis frontend, a sound model, and an audio synthesis module. Building these components requires extensive domain expertise and may contain brittle design choices. Moreover, these components are usually trained separately on different objectives and datasets, which may introduce errors and inconsistencies in the final output. To overcome these limitations, end-to-end models have been proposed that directly learn the mapping between text (or other kinds of data) and audio waveform using deep neural networks. These models are presented in this Section.

Existing research establishes two main frameworks for end-to-end models: specialized models designed for a specific domain, and universal models aimed at broader applications. The table 2.6 shows some examples of these models based on their type, input, output, model architecture, and type of conditioning. Specialized models target either speech or music synthesis, such as Char2wav and Jukebox. Researchers have developed different subsets of technologies within speech and music synthesis models, such as neural codec speech models and discrete diffusion models. Universal models, such as SampleRNN and AudioGen, can generate audio from various inputs and domains, such as text or raw audio seeds.

Table 2.6: A comparison of different end-to-end generative models for audio.

Model	Type	Input	Output	Model Architecture
Char2wav [119]	Speech	Text prompt	Raw audio waveform	Encoder-decoder with attention and neural vocoder
VALL-E [131]	Speech	Text and acoustic prompt	Raw audio waveform	Neural codec language model and neural vocoder
Jukebox [29]	Music	Genre, artist, and lyrics	Raw audio waveform	Hierarchical VQ-VAE and autoregressive Transformer
Riffusion [40]	Music	Text prompt	Raw audio waveform	Neural codec language model based on discrete diffusion model and neural vocoder
MusicLM [4]	Music	Text prompt	Raw audio waveform	Neural codec language model and neural vocoder
SampleRNN [80]	General	None	Raw audio waveform	Hierarchical RNN and neural vocoder
AudioLM [14]	General	Text prompt	Raw audio waveform	Hybrid tokenization scheme with Transformer models and neural vocoder
DiffSound [135]	General	Text prompt	Mel-spectrogram and raw audio waveform	VQ-VAE, discrete diffusion model, and neural vocoder
AudioGen [72]	General	Text prompt	Mel-spectrogram and raw audio waveform	Transformer-based generative model and neural vocoder

2.2.4.1 Text-to-Speech

Text-to-speech (TTS) models are designed to convert written text into synthesized speech. These models use deep neural networks to directly learn the mapping between written text and audio waveform. Leveraging developments in NLP and speech synthesis techniques, TTS models have made significant progress in generating high-quality, human-like speech from text input. This Section examines some of the notable TTS models that have been developed recently.

Char2Wav The Char2Wav model, proposed in 2017 [102], serves as a speech synthesis model comprising two distinct components: a reader and a neural vocoder. The reader takes text as

inputs and produces a sequence of acoustic features as outputs. The neural vocoder then takes these acoustic features and generates raw waveform samples.

The reader is an attention-based recurrent sequence generator. It is a type of neural network that can generate a sequence of outputs based on a sequence of inputs. In this case, the inputs are text, and the outputs are acoustic features. The generator uses a bidirectional RNN (see Section 2.1.2.1) as an encoder and a RNN with attention as a decoder. The attention mechanism allows the model to focus on different parts of the input sequence as it generates the output.

Instead of using a traditional vocoder to generate the raw waveform samples, Char2Wav uses a learned parametric neural module. Specifically, it uses a conditional version of SampleRNN (see Section 2.2.4.3 to learn the mapping from vocoder features to audio samples. This allows *Char2Wav* to generate speech directly from the acoustic features without relying on a specific vocoder.

Although no formal proofs or analytical results are presented in this work, the proposed architecture is a significant breakthrough in speech synthesis. By demonstrating the effectiveness of using attention-based recurrent sequence generators and learned parametric neural modules, Char2Wav establishes a solid foundation for future research in this area.

VALL-E VALL-E is a language model developed by researchers at Microsoft for TTS that treats TTS as a conditional language modeling task [131]. It generates text based on a given context, where the context in VALL-E is the acoustic tokens and phoneme prompts. VALL-E conditions on these inputs to produce the acoustic token sequence for speech synthesis.

VALL-E comprises two components: an audio codec that generates discrete acoustic tokens from speech waveforms and a neural language model that conditions these tokens and phoneme prompts to generate speech for unseen speakers in a zero-shot setting. The high-level architecture can be seen in Figure 2.27.

The researchers trained VALL-E using the LibriLight dataset [64], which consists of 60,000 hours of English speech from over 7,000 unique speakers. The proposed approach is robust to noise and generalizes well by leveraging a large and diverse dataset. Previous TTS systems are typically trained with fewer data than VALL-E.

VALL-E's performance was evaluated on the LibriSpeech [92] dataset, where all test speakers are unseen during training. VALL-E delivers high performance for speaker-adaptive TTS in terms of speech naturalness and speaker similarity, as measured by comparative mean opinion score and similarity mean opinion score, respectively.

Qualitative analysis of VALL-E reveals several interesting findings. Firstly, VALL-E generates speech with diversity. As a result, the same input text can produce different speech outputs. This feature is important for downstream applications such as speech recognition, where diverse inputs

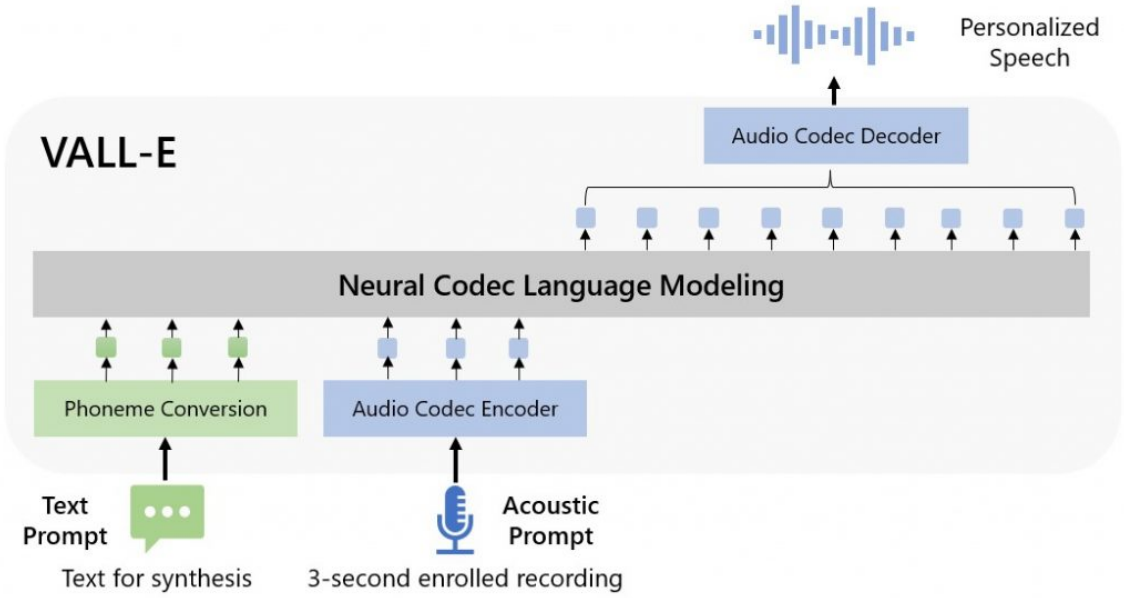


Figure 2.27: **VALL-E** — The image was extracted from the source publication. It illustrates that both encodings derived from a linguistic prompt and an auditory prompt - provided via the codec encoder - are fed into a language model such as a transformer, and the resulting outputs are passed to the codec decoder to generate audio.

with different speakers and acoustic environments are beneficial. The diversity of VALL-E makes it an ideal candidate for generating pseudo-data for speech recognition.

Another finding is that VALL-E maintains the acoustic environment of the prompt during speech synthesis. When the acoustic prompt has reverberation, VALL-E can synthesize speech with reverberation, whereas the baseline outputs clean speech. This can be attributed to VALL-E being trained on a large-scale dataset with diverse acoustic conditions, which allows it to learn acoustic consistency instead of only a clean environment during training.

Furthermore, VALL-E can preserve the emotion in the prompt during speech synthesis. The researchers selected acoustic prompts from the EmoV-DB dataset [3], which contains speech with five emotions. VALL-E kept the exact emotion of the prompt in speech synthesis, even without fine-tuning on an emotional TTS dataset.

VALL-E represents a significant advancement in TTS technology, with its language model approach and use of audio codec codes as intermediate representations.

In summary, VALL-E is a language model-based TTS system that utilizes audio codec codes as intermediate representations. It performs highly in speaker-adaptive TTS and demonstrates exciting features such as speech diversity, acoustic environment consistency, and emotion preservation.

2.2.4.2 Generative Music

Generative music is created using generative techniques. End-to-end generative music models enable the production of new musical compositions without using predefined templates or samples, directly from textual or other data inputs. These models use DL architectures to capture patterns and structures within different genres or styles of music, and can produce original pieces based on given prompts. This Section explores remarkable generative music models that demonstrate their ability to compose novel musical arrangements.

Jukebox Jukebox, a generative model for music that produces music with singing in the raw audio domain, was introduced by Dhariwal et al. in 2020 [29]. The model tackles the long context of raw audio using a multiscale VQ-VAE (see Section 2.1.2.3) to compress it to discrete codes and models those using AR Transformers (see Section 2.1.2.3).

The hierarchical VQ-VAE architecture compresses audio into a discrete space, retaining the maximum amount of musical information at increasing compression levels. The model uses residual networks consisting of WaveNet-style (see Section 2.2.3.1) noncausal 1-D dilated convolutions, interleaved with downsampling and upsampling 1-D convolutions to match different hop lengths. Separate autoencoders with varying hop lengths are trained to maximize the amount of information stored at each level.

After training the VQ-VAE, a prior $p(z)$ over the compressed space is learned to generate samples. The prior model is broken up as $p(z) = p(z_{top})p(z_{middle}|z_{top})p(z_{bottom}|z_{middle}, z_{top})$, and separate models are trained for the top-level prior $p(z_{top})$, and upsamplers $p(z_{middle}|z_{top})$ and $p(z_{bottom}|z_{middle}, z_{top})$. Autoregressive Transformers with sparse attention are used for modeling in the discrete token space produced by the VQ-VAE.

Jukebox can generate high-fidelity and diverse songs with coherence for up to multiple minutes. It can be conditioned on the artist and genre to steer the musical and vocal style and on unaligned lyrics to make the singing more controllable. The model's release includes thousands of non-cherry-picked samples, model weights, and code.

Riffusion Riffusion [40] is an open-source model presented in 2022 that generates music clips from text prompts. The model is based on Stable Diffusion (see Section 2.1.5.3). Riffusion fine-tunes Stable Diffusion to generate images of spectrograms, which can then be converted to music clips.

The authors use STFT (see Section 2.1.1.1) to compute the spectrogram from audio. The STFT is invertible so that the original audio can be reconstructed from a spectrogram. The authors use the Griffin-Lim algorithm [48] to approximate the phase when reconstructing the audio clip.

The authors use diffusion models (see Section 2.1.2.3) to condition the model's creations on a text prompt and other images, which is helpful for modifying sounds while preserving the structure of

an original clip. The authors also use the denoising strength parameter to control how much to deviate from the original clip and towards a new prompt.

So, for inference, the model takes a text prompt as input. Then, the text is encoded into a latent representation using a text encoder. The model generates an image of a spectrogram from the latent representation using a modified version of Stable Diffusion; this is, the Stable Diffusion model fine-tuned for spectrograms. Finally, the generated spectrogram image is converted into an audio clip using the Griffin-Lim algorithm.

MusicLM MusicLM is a generative model capable of synthesizing high-fidelity music characterized by realistic instrument timbres, accurate pitch, temporal patterns, and smooth transitions between notes based solely on textual descriptions of desired musical attributes (see Section 2.2.4.2). The model extends the AudioLM framework for audio generation (see Section 2.2.4.3) by incorporating text conditioning via the joint music-text model MuLan (see Section 2.1.3.3).

MusicLM employs a hierarchical modeling approach with two main stages: semantic modeling and acoustic modeling. The semantic modeling stage uses a Transformer decoder (see Section 2.1.2.3) to predict semantic tokens from the MuLan audio tokens. Using a separate Transformer decoder, the acoustic modeling stage then predicts acoustic tokens conditioned on both the MuLan audio tokens and predicted semantic tokens. This stage is subdivided into coarse and fine modeling substages to reduce the length of the token sequences, following the AudioLM approach.

Overall, MusicLM leverages pre-trained audio encoders (SoundStream, w2v-BERT, and MuLan) to obtain discrete acoustic and semantic tokens as input, after which hierarchical Transformer decoders first predict semantic tokens and then acoustic tokens when conditioned on MuLan text embeddings during synthesis. The hierarchical approach and use of semantic tokens aims to enable coherent long-term generation over extended durations.

MusicLM can generate coherent musical sequences up to 5 minutes in duration, constituting a notable achievement in the context of generative music models. The model captures various musical characteristics specified in textual prompts, including instrument timbre, melodic elements, and musical genre.

The model's performance was assessed using the MusicCaps dataset which comprises 5,500 pairs of music-texts that have been annotated by experts. The dataset includes diverse genres, instruments, and moods. The authors assert that this dataset thoroughly assesses the model's capability to generate various aspects of music from textual prompts. The evaluation metrics comprise human judgments of similarity between the generated outcomes and the prompts, as well as overall quality and others.

The paper discusses potential limitations, including a proclivity for mode collapse and difficulty generating fine-grained structures over long sequences. However, further investigation is needed to probe the model's limitations and failure modes in greater depth.

In summary, MusicLM represents a promising generative model capable of synthesizing high-fidelity music from textual descriptions via shared embedding spaces and learned associations between text and music encodings, enabling it to capture diverse musical characteristics specified in textual prompts. The MusicCaps dataset provides a valuable means of evaluating the model's performance across various musical styles and prompts.

2.2.4.3 General Text-to-Audio

Text-to-audio systems have a wide range of applications beyond speech synthesis or generative music tasks. End-to-end models convert different forms of textual input into corresponding audio outputs. The outputs have diverse purposes, including sound effects generation, voice transformation, and environmental sound synthesis. These models provide flexible solutions for transforming text into realistic auditory experiences by training on large-scale datasets containing paired text-audio examples across various domains. This Section presents several text-to-audio approaches that demonstrate innovative methods of audio synthesis based on specific textual cues.

SampleRNN *SampleRNN* is a neural audio generation model proposed in 2017 that can produce high-quality audio samples from scratch [80]. It uses a hierarchical structure of RNNs (see Section 2.1.2.1) to model the probability distribution of audio waveforms at different temporal resolutions. The lowest RNN operates on individual samples, while higher RNNs capture longer-term dependencies and structure. SampleRNN can learn from any audio data without any prior knowledge or labels.

The higher RNNs capture the longer-term dependencies by receiving inputs from lower RNNs at a lower sampling rate. This way, they can process longer audio sequences. The higher RNNs also use skip connections to directly access the outputs of lower RNNs, which helps to avoid vanishing gradients and preserve information across different levels of abstraction.

Each cell is a RNN variant, such as GRU (see Section 2.1.2.1) that takes as input a frame of audio samples from a lower RNN and outputs a hidden state vector that encodes the long-term context of the audio. This output is passed upwards in the hierarchy to other RNNs that take it. Multiple layers are possible, each operating at a different temporal resolution. All the outputs are then inputted in the final level RNN, whose output is the next audio sample based on the combined information from all hierarchy levels.

AudioLM The framework called AudioLM was introduced by Borsos et al. in 2022 as a means for high-quality audio generation with long-term consistency [14]. In this representation space, the framework maps input audio to a sequence of discrete tokens and treats audio generation as a language modeling task. AudioLM achieves high-quality synthesis and long-term structure through a hybrid tokenization scheme. This scheme combines the discretized activations of a masked language model pre-trained on audio (semantic tokens) and the discrete codes produced by a neural audio codec (acoustic tokens).

The AudioLM framework consists of three main components:

1. A *tokenizer model* that maps the input audio x into a sequence $y = \text{enc}(x)$ of discrete tokens from a finite vocabulary, with $T' < T$.

2. A *decoder-only Transformer language model* that operates on the discrete tokens y, trained to maximize the likelihood $\prod_{t=1}^{T'} p(y_t|y_{<t})$. The model predicts the token sequence $\hat{y}$ autoregressively at inference time.

3. A *detokenizer model* that maps the sequence of predicted tokens back to audio, producing the waveform $\hat{x} = \text{dec}(\hat{y})$.

The tokenizer and detokenizer models are pre-trained and frozen before training the language model, simplifying the training setup. The number of tokens T' is significantly smaller than T, allowing for increased temporal context size in the language model.

To reconcile the conflicting requirements of high-quality audio reconstruction and capturing long-term dependencies, AudioLM relies on a combination of acoustic and semantic tokens. Acoustic tokens are computed using SoundStream (see Section 2.2.2.4). Semantic tokens are computed using w2v-BERT [22], a model for learning self-supervised audio representations. The semantic tokens enable long-term structural coherence while modeling the acoustic tokens conditioned on the semantic tokens enables high-quality audio synthesis.

AudioLM adopts a hierarchical approach by first modeling the semantic tokens for the entire sequence and then using these as conditioning to predict the acoustic tokens. AudioLM generates syntactically and semantically plausible speech continuations while maintaining speaker identity and prosody for unseen speakers when trained on speech without any transcript or annotation. The approach also extends beyond speech, generating coherent piano music continuations despite being trained without any symbolic representation of music.

DiffSound *DiffSound* was presented in a paper, in 2022, that displays a novel text-to-sound generation framework that uses a text encoder, a VQ-VAE, a decoder, and a vocoder. The framework takes text as input and outputs synthesized audio corresponding to the input text. The decoder, in particular, is a critical component of the framework, and the paper focuses on designing a suitable decoder, calling it *DiffSound* [135].

DiffSound is a diffusion decoder (Section 2.1.2.3) based on the discrete diffusion model. DiffSound predicts all Mel-Spectrogram tokens in one step and then refines the predicted tokens in the next step, resulting in better-predicted results after several steps. It not only produces better text-to-sound generation results compared to an AR decoder, but it is also faster, with a generation speed five times faster than an AR decoder.

The entire framework acts as this: First, the text is encoded into embeddings using a model like a transformer (Section 2.1.2.3). Then, this representation conditions the generation of spectrogram

embeddings using diffusion (the DiffSound model). These embeddings are then passed through the pretrain VQ-VAE decoder to generate the spectrogram. The spectrogram runs through a vocoder (Section 2.2.3) to generate the waveform. In the original text, the vocoder used was the MelGAN (see Section 2.2.3.3). This process can be seen in Figure 2.28.

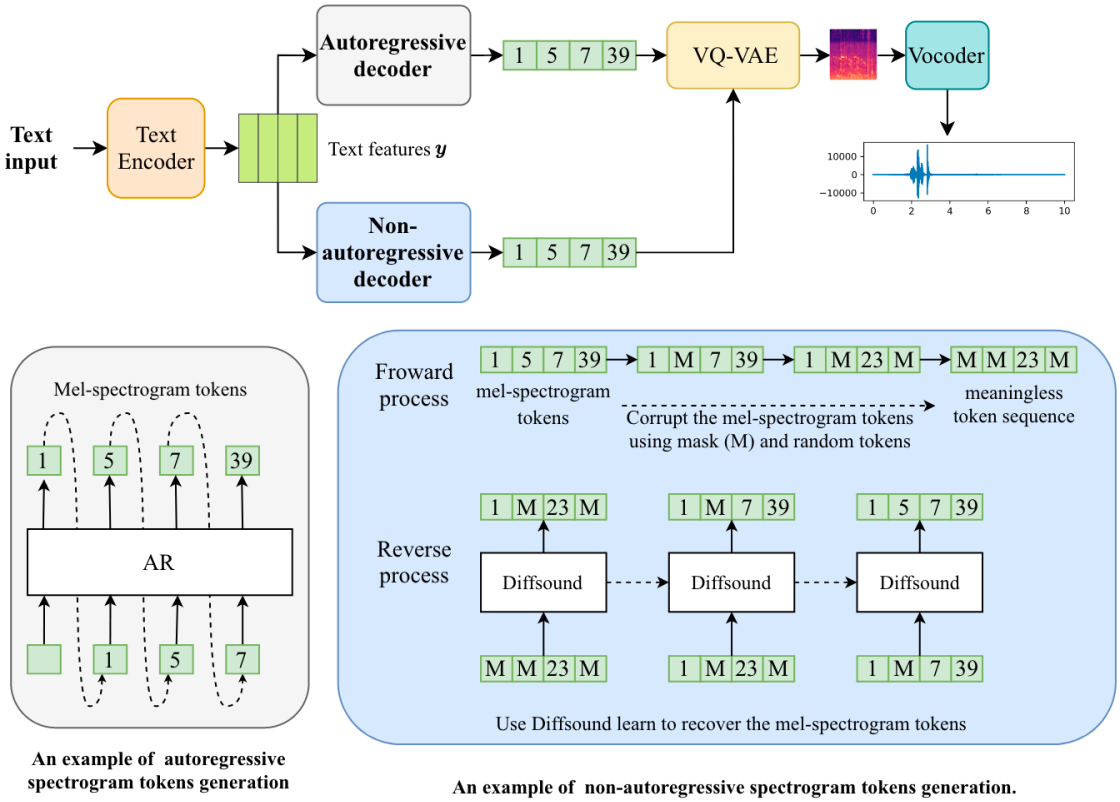


Figure 2.28: **DiffSound framework** — This illustration was taken from the original paper. At the top, the general framework is present. Two decoders are present, but only one of them is used. The decoder results in a set of latent features. These features are passed to the decoder of the VQ-VAE that generates a Mel-Spectrogram (the square with red and blue tones) that, through a vocoder, generates a sound. The two bottom images represent the two decoders. One is a DARN (see Section 2.1.2.3), the other works with diffusion.

AudioGen In 2023, Kreuk et al. [72] proposed AudioGen, an auto-regressive generative model that generates audio samples conditioned on text inputs. The model comprises two primary stages: (i) learning a discrete representation of the raw audio using an AE method and (ii) training a Transformer language model (see Section 2.1.2.3) over the learned codes obtained from the audio encoder, conditioned on textual features. During inference, the model samples from the language model generate a new set of audio tokens given text features, which can then be decoded into the waveform domain using the decoder component.

To address the challenge of text-to-audio generation, the authors propose an augmentation technique that mixes different audio samples to train the model to separate multiple sources inter-

nally. Furthermore, the authors explore the use of multi-stream modeling for faster inference, allowing the use of shorter sequences while maintaining a similar bitrate and perceptual quality. The proposed method outperforms evaluated baselines over both objective and subjective metrics. Additionally, the authors extend the proposed method to conditional and unconditional audio continuation, demonstrating its ability to generate complex audio compositions.

Chapter 3

The Synthesis Problem

Contents

This chapter explores the realm of audio synthesis via advanced generative AI models that operate on textual input. The goal is to gain a deeper understanding of audio generation.

Generative AI is fundamentally changing the landscape of audio synthesis. This chapter demonstrates the emerging capabilities of generative AI in producing complex audio.

The synthesis problem is examined in several crucial sections. The first section (3.1) presents the main challenges. Next, section 3.2 highlights the broader implications. Finally, section 3.3 analyzes the basic data. Section 3.4 details the limitations of the research, while section 3.5 dissects the fine-tuning of the model. Finally, section 3.6 explores the architectural complexities.

3.1 Problem Definition

The rapid advancement of artificial intelligence and machine learning techniques has paved the way for transformative developments in content creation across multiple disciplines. However, a significant gap remains in the field of audio synthesis, especially in the context of generative AI models. The challenge at hand is to bridge this gap and enable the transformation of textual input into high-fidelity and immersive soundscapes, thereby creating a new form of creative expression.

Recently, the landscape of audio generation has been undergoing a paradigm shift, catalyzed by significant investments and efforts from prominent technology conglomerates. As of 2023, major industry players are placing significant bets on the potential of AI-driven audio synthesis. This noteworthy development underscores the growing recognition of the transformative impact that sophisticated audio generation can have across sectors, including entertainment, education, virtual reality, and more.

3.1.1 Gap in the Literature

The landscape of audio synthesis through generative AI models presents a spectrum of challenges and limitations that impact various applications, including content creation, sound design, music production, and interactive media. These challenges include computational constraints, model architectural intricacies, quality/realism trade-offs, data-related hurdles, interpretability issues, and scalability concerns. Addressing these limitations has the potential to significantly improve the quality and utility of audio synthesis methods.

Significant hurdles in the use of generative AI models for audio synthesis are the computational cost and the training time. PixelCNN Decoders [88], Jukebox [29], and Hi-Fi GAN [71], which exemplify these models, require extensive computational resources and long training times. These limitations render these models inaccessible to researchers and developers and limit their applicability in real-world situations where efficient generation is critical. Developing more efficient training algorithms or model architectures could democratize access and broaden the reach of audio synthesis methods by overcoming these challenges.

Producing **high-quality and realistic audio** synthesis is a complex task. The lack of ability to capture minute details, as seen in models such as WaveNet [87] and MelGAN [73], causes audio outputs to be deficient in delicate nuances and textures. Limitations such as mode collapse, seen in GANSynth [35], and Jukebox [29] can cause a lack of variety, leading to repetitive and monotonous audio outputs. Additionally, maintaining coherence throughout extended audio sequences - as observed in PixelCNN [88], WaveNet [87], Jukebox [29], and Riffusion [40] - is a challenge that can affect the final output.

The versatility of synthesis models is impacted by data efficiency and generalization capabilities. Models such as DALL-E [101], VALL-E [131], and MusicLM [4] have suboptimal data efficiency,

which requires large datasets for effective training. It is crucial to address the challenges in handling unusual concepts, as observed in GLIDE [85], to broaden the creative potential of these models. Moreover, the practical utility and user-friendliness of models such as DALL-E [101] and VALL-E [131] are constrained due to their limited control and interpretability.

The challenges and limitations within audio synthesis through generative AI models align with the key objectives of this thesis. The aim to develop advanced systems that handle sound, classification, and end-to-end generation while accounting for hardware limitations aligns with the need to overcome computational constraints highlighted in the literature. The thesis aims to address the computational hurdles that hinder the accessibility and practicality of audio synthesis models by devising more efficient training algorithms or model architectures. This directly contributes to the broader objective of providing valuable contributions to deep learning and its applications.

The goal to create end-to-end systems that generate sound from any textual input is intricately connected to the challenges of quality, realism, and coherence in audio synthesis. The thesis aims to bridge the gap between textual input and high-fidelity, immersive soundscapes by exploring the intricacies of model architectural design and refining the generation process. Evaluating the accuracy of generated audio aligns with the broader pursuit of overcoming limitations in capturing fine-grained details, mode collapse, and long-term coherence. This pursuit ultimately enhances the authenticity and richness of the synthesized audio.

The ambitious mission to contribute to DL and its applications aligns well with the goal of addressing data-related issues, such as biases and limitations arising from training data. This thesis aims to enhance the adaptability of generative AI models by curating diverse and representative datasets and implementing techniques for bias detection and mitigation, to make them more responsive to various tasks and domains. This alignment highlights the research's importance in advancing audio synthesis and establishing a wider framework for responsible and inclusive development of AI.

To summarize, the literature's identified gap, marked by challenges ranging from computational constraints and quality/realism to data efficiency and interpretability issues, is intricately linked with this thesis's objectives. Advancing end-to-end audio synthesis systems while addressing these challenges not only deepens the field's scientific comprehension but also profoundly aligns with the broader goals of your research endeavor.

3.1.2 Formal Problem Definition

The goal of this study is to address the synthesis of audio based on textual prompts. The audio is not limited to a specific domain, such as speech or music. The audio produced by the model must be dependent on the inserted textual input. The generated audio must be produced in real-time, with minimal latency after the textual prompt is inserted into the model. This study considers a model that resolves this issue and refers to it as an end-to-end model. Furthermore, it is considered that the terms end-to-end and text-to-sound are interchangeable in this context.

These models are trained using sounds labeled by humans from openly available datasets. Furthermore, the evaluation involves comparing the generated sounds with their corresponding real sounds.

To formally define this problem, the elements in Figure 3.1 must be defined.

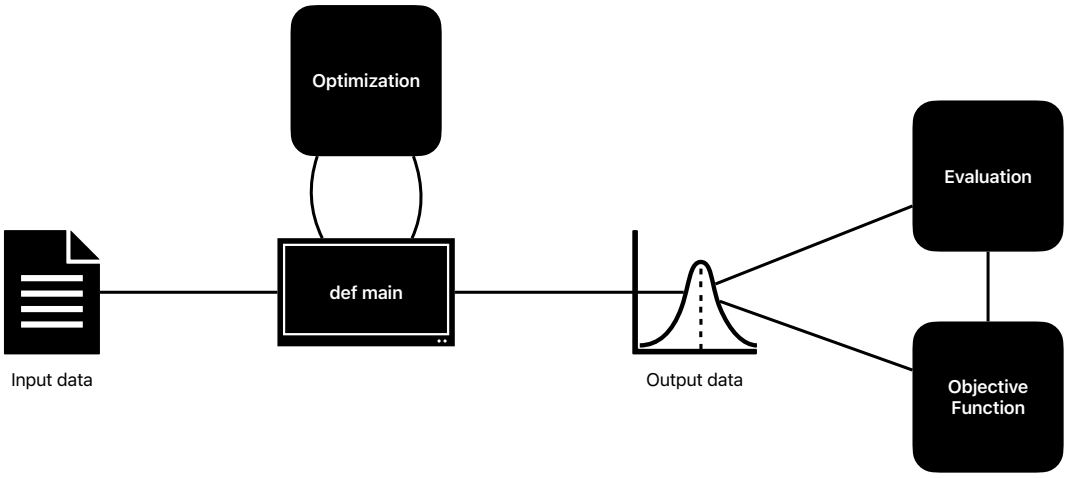


Figure 3.1: AI Problem Definition

3.1.2.1 Input Data

Synthesizing audio from textual prompts entails several challenges, especially when dealing with the diverse formats present in sound samples and text labels. As these formats converge, they add layers of intricacy that require innovative solutions and thoughtful consideration.

In the audio domain, dealing with diverse formats manifests itself in multiple sonic expressions, ranging from music and spoken dialogues to environmental noises and abstract compositions. These formats encompass a spectrum of acoustic textures and temporal dynamics, demanding a unique processing and analysis approach for each.

Text labels also introduce a variety of challenges as they take various forms from different sources. The range of text labels spans from brief categorical tags that capture high-level features to verbose textual descriptions that explore intricate details. This variation poses the model with the task of decoding and utilizing the diverse meanings, tones, and contexts embedded within these labels. Achieving balance between the weight of these different formats while incorporating them into the generative fabric poses its own challenge.

The intersection of different audio formats and textual labeling structures presents a challenge for the model to generate coherent and harmonious audio outputs. Each format has its implicit dimensions, nuances, and potential biases associated with it. Ensuring that the model avoids being biased towards a particular format or ignores others is a challenge that needs precise calibration.

Additionally, the model must handle the complex task of cross-modal learning, deciphering the interplay between sound and text to produce outputs that match seamlessly with textual prompts.

When embarking on the complex process of audio synthesis from text, it is important to acknowledge the intricacies that arise from combining sound samples and text labels. The difficulties that come with handling various formats serve as milestones of progress and discovery. These challenges put the model's true capabilities to the test as it navigates through the maze of formats, becoming a creative force capable of orchestrating sonic symphonies that blend seamlessly with the tapestry of human expression.

3.1.2.2 Output Data

The output produced by these models is in a particular audio format, resulting from the complex interplay between textual cues and sonic expression. An essential factor in this synthesis is real-time generation, which is critical for both user experience and practical applications.

Real-time generation has the potential to provide immediate results, smoothly converting textual input into audio output. In practical contexts, such as interactive media, virtual environments, or assistive technologies, the capability to respond promptly to user inputs with sonic output enhances the immersive quality of the experience.

The pursuit of real-time generation comes with its share of difficulties and trade-offs. The requirement for swift responsiveness can create limitations that may affect the precision and complexity of the produced audio. The delicate balance between speed and sonic complexity requires careful consideration.

Real-time generation plays a critical role in generative audio synthesis. Achieving real-time outputs requires a balance between immediacy and auditory finesse, guided by studies on human perception. During this pursuit, the generative model attunes itself not only to the cadence of textual cues, but also to the rhythm of human interaction and appreciation.

3.1.2.3 Objective Function

The development of an objective function is a crucial step in this work, where challenges, strategies, and insights come together to shape the core of creative synthesis.

The purpose of generative models is to uncover and encapsulate the complex distributions that form the basis of the data. This journey of comprehension accompanies the aspiration to create new data, which evokes familiarity with the original while testing the limits of artistic innovation. This essence, charming as it may be, emerges within a paradox that challenges us to bridge the gap between accuracy and realism by channeling the learned distributions into outputs that reflect authenticity.

The challenge in defining an objective function lies in extracting the core components of accuracy and realism, and quantifying them into a measurable metric. One approach that aligns with the generative pursuit involves reducing the divergence between the original and synthesized data.

Within the realm of generative literature, numerous strategies intertwine to inspire the creation of objective functions. Approaches presented in Section 2.1.3.2 should be taken into account.

The objective function emerges as a guiding principle in this interplay of creativity and computation. The objective function embodies the urge for realism and accuracy, reflecting the very essence of generative models in its mathematical grasp. The definition of the objective function is intertwined with the fabric of the generative narrative, uniting aspirations and strategies into a symphony that echoes across the domains of imagination and reality.

3.1.2.4 Optimization Algorithm

The optimization algorithm aims to shape the generative landscape based on the objectives defined within the objective function.

Selecting an optimization algorithm is a crucial decision in this field that needs careful consideration. Each generative model has its own intrinsic characteristics, strengths, and idiosyncrasies, and it is these traits that determine the choice of optimization algorithm. The algorithm guides the process towards optimal solutions and shapes the generative narrative with each iterative step.

Selecting an optimization algorithm is an arbitrated process where the properties of the generative model are tailored to the unique contours of problem landscape. It entails a delicate calibration to adapt the nuances of established algorithms, harmonizing them with the demands of the generative journey. Fine-tuning hyperparameters, sculpting convergence criteria, and managing trade-offs are integral facets of optimization.

The optimization landscape within the explored generative models consists of various algorithms that align with the specific requirements of each model.

Exploring optimization algorithms reveals a landscape that mirrors the fusion of disciplines within generative fields. Gaining insights from seminal works across various optimization methodologies, the optimization algorithm acts as a creative tool, ready to enhance the generative canvas with precision and convergence.

3.1.2.5 Evaluation Metrics

Evaluation metrics are necessary to guide exploration into the realm of generative models.

However, evaluating generative models involves navigating uncharted territories, especially when considering the intricate realm of subjective attributes. The concepts of audio realism, coherence, and emotive resonance are subjective, making it challenging to measure them objectively. Achieving sound realism requires balancing the timbral fidelity and temporal authenticity, which creates

a challenge requiring innovative solutions. Similarly, coherence in generative outputs, determined by the harmony among sequential elements, necessitates metrics that can reveal the intricate patterns.

Specific evaluation metrics emerge in this pursuit as guiding stars, each one poised to illuminate a unique facet of generative prowess. Perceptual audio quality metrics, which are rooted in the perceptual space of human hearing, transcend the realm of raw signal processing. They reflect the intricate tapestry of auditory perception. Subjective metrics such as MOS are used. Research delves into the visceral nature of human judgment and provides a medium on which subjective impressions are recorded.

Among the many metrics, their relevance, appropriateness, and coherence with the generative narrative are crucial. Metrics anchor the ephemeral expanse of creativity to the solid ground of quantitative analysis.

3.2 Application of Synthesizing Soundscapes with Generative AI

The development of AI technologies has enabled significant progress in sound synthesis, enabling the creation of sounds based on textual input. This technology has potential applications in various fields, including music production, film and game sound design, and therapeutic soundscapes. In this response, this Section explores possible applications for the AI models described in this document.

One potential application of such AI models is in the field of music composition. By generating sounds from textual descriptions, the model could assist composers in generating novel and unique sounds that match a piece's intended mood or atmosphere. For example, a composer might input the phrase "eerie forest at night", and the AI model could generate a soundscape incorporating the sounds of rustling leaves, distant animal calls, and other eerie sounds one might associate with a forest at night. This technology could help composers to create soundscapes more efficiently that match their creative vision, saving time and increasing their overall output. Besides, the real-time inference characteristic of this model may help a live performer blend the timbres of instruments with, for instance, natural sounds, creating soundscapes on the fly [61].

Another possible application of an AI model that generates sound from text is in the field of film and game sound design. Sound design plays a crucial role in creating immersive and engaging experiences in film and games, and the ability to generate custom soundscapes from textual input could enhance the creative potential of sound designers. For example, a sound designer might input the phrase "a bustling city street", and the AI model could generate a soundscape that includes the sounds of car horns, people talking, footsteps, and other city noises. Alternatively, they may want to generate sound cues such as explosions [61], and a simple query would do that. This technology could help sound designers create more realistic and immersive soundscapes, improving the overall quality of the final product.

A panoply of sounds is usually taken from expensive libraries in TV and film. One might imagine an infinite library with ML models, and one should imagine how that will enhance the power of sound producers for such endeavors, especially in indie and low-budget productions.

3.3 Datasets

For the proposed problem, ideal datasets would have entries of sound samples under multiple conditions with labels written in a NLP form.

With multiple conditions, one may think, for instance, of multiple sounds of a dog barking. An ideal dataset would have a dog barking in different places, under different atmospheric conditions, and with other possible variants. For example, this ideal dataset would have "A dog barking in a city", "A dog barking in the countryside where it is raining", and others.

Multiple sounds over or followed by each other would exist in the ideal dataset. For example, "A dog barking followed by a truck honking" or "A traffic jam while a woman sings" would be examples of the entries in the perfect dataset.

The perfect dataset would also have characteristics of the sounds themselves. For instance, not only "A dog barking", but rather "A dog barking aggressively", or "A dog barking with joy" would be examples of entries in the perfect dataset.

These datasets do not exist in the real world. Thus, some measures must be taken. This discussion is proposed in Section 4.2.

3.4 Scope, Limitations, and Technical Constraints

This study aims to advance the capabilities of audio generation from textual input, at the intersection of audio synthesis and generative AI. This research focuses on developing and refining end-to-end generative models that seamlessly translate textual prompts into immersive soundscapes, in the landscape of AI-driven audio synthesis.

3.4.1 Technical Constraints

This research in algorithmic audio generation is shaped by a series of pragmatic technical constraints that influence our study's course. These constraints include hardware and computational considerations that are intrinsic to the development and training of sophisticated AI models for audio synthesis.

Contemporary AI research requires significant resource allocation, profoundly impacting our efforts. Studies, such as the development of DALL-E 2 (see Section 2.1.5.5), vividly illustrate this resource-intensive trajectory.

The training process for this model requires an investment of 100,000 to 200,000 GPU hours, based on the architecture presented in Annex C of the paper. Assuming V100 GPUs are utilized at $3 per hour (in AWS [9]), the minimum financial commitment amounts to $300,000. This significant investment highlights the need for substantial resources in the ongoing pursuit of AI advancements. This substantial figure reflects not only the computational demands of AI but also the economic dimensions that drive progress within these domains.

When focusing on audio synthesis, a parallel relationship becomes evident. Generating audio snippets from textual cues poses similar challenges to those encountered in computer vision. Effective audio synthesis relies on robust generative AI models for which the training process demands a substantial share of computational resources. These computational resources are not available to individual scientists working alone. As larger models tend to perform better in the generative domain, it is increasingly challenging for scientists and developers to compete against big tech companies.

3.4.2 Ethical Considerations

Ethical considerations play a crucial role in the domain of audio synthesis driven by generative AI models. As the possibilities for creative expression continue to grow, it is important to address the ethical aspects that come with the transformative potential of these technologies. This discussion explores the potential ethical implications and highlights the utmost significance of responsible deployment in the field of generative audio synthesis.

One major concern arises in the potential creation of harmful or deceitful content. Generative models' ability to create various audio outputs from text inputs presents opportunities for innovation but also poses inherent risks of producing malicious or deceitful content. The ethical tightrope that creators must balance is underscored by the dual nature of generative prowess.

For example, if an ill-intentioned person utilizes generative audio synthesis to create authentic voice recordings of people, it may result in spreading false information or fabricating forged audio evidence. The remarkable precision of generative models in imitating voices could be misused to commit identity fraud, cause disharmony, or construct digital impersonations that deceive and manipulate unsuspecting listeners.

Moreover, ethical concerns apply to the various sources of textual inputs. Textual clues extracted from public texts, personal submissions, or online repositories are used as raw material for generative processes. These sources require rigorous methods to refine the generative process itself, ensuring that audio outputs are true and precise. Moreover, the responsibility also includes careful curation of data inputs to prevent the perpetuation of biases or prejudiced content. The handling of text inputs may perpetuate ethical implications based on cultural, social, or ideological connotations, which calls for robust mechanisms to prevent the unintended amplification of unethical content.

In studies like this, the researcher does not collect the dataset, but they are responsible for ensuring that the dataset used is unbiased and free of prejudices.

Considering these ethical dimensions, it is clear that the use of generative audio synthesis carries significant ethical responsibilities. While one explores this unexplored territory, it is essential that ethical considerations serve as steadfast sentinels, guiding the responsible use of generative AI models in audio synthesis.

3.5 Parametric Control

If one creates a generator model that generates sounds without any input, one would only generate random sounds without any meaning behind them. Without conditioning, WaveNet generates "babble" [61].

An end-to-end model must be able to take textual input and generate a sound from it. For this, the textual input must be somehow passed to the model during training and inference.

This is called parametric control, sound modeling, or model conditioning. "Sound modelling is [...] developing algorithms that generate sound under parametric control" [61]. This is the ability to manipulate and control the characteristics of the generated audio by adjusting the model parameters. For example, a generative model for musical audio synthesis may allow the user to control the pitch, timbre, or duration of the generated audio by adjusting the specific parameters of the model. The goal of parametric control in generative audio models is to allow the user to fine-tune the characteristics of the generated audio and achieve the desired results. Parametric control is essential in this work because the output must be tailored specifically for the user's input prompt.

This is a crucial job for these types of models and also one of the toughest obstacles in creating digital audio [61].

Luckily, this is a solved problem in the data generation realm. Given the quick advances in generative deep learning technologies, every generator production model relies on model conditioning. For instance, transformers receive an input text vector natively. Also, GANs can be conditioned on a specific range of inputs. These examples illustrate that it is more than possible to incorporate this kind of technology in an end-to-end model for sound generation.

Modern models for multimodal learning, which involve processing information from different media types such as images, sound, and text, have been based on a variation of VAEs as seen in sections 2.1.5 and 2.2. One approach to conditioning VAEs on text involves training different media types, such as images and text, to share a common latent space. This is achieved by optimizing a joint objective function that balances the reconstruction loss of each modality with the alignment of the respective latent spaces.

The text input is first encoded into a latent representation using an encoder network during inference. The image decoder network then decodes this latent representation to generate the output image that corresponds to the given text input.

This is lightly debated in Section 3.6, and more deeply argued in Section 4.3.

3.6 Model Requirements and Design

A model that solves the end-to-end problem aims to transform a textual prompt into a sound.

A model to solve this problem must be able to extract a representation of latent features of the given prompt. Then, it must condition a given generator on this representation. This generator must create the sound itself or create a lower-level sound representation. If the latter occurs, another piece must transform this representation into a raw sound, a vocoder.

The generator model needs to be trained with the text embeddings first. So, let us follow the example of training for a single sample: a pair of a sound and the respective natural language label. First, one must translate the sound into a lower-level feature representation. Second, one must have a model that translates the label into a latent features representation, this is, into a text embedding vector. Then, a model that generates a sound representation must be conditioned on these embeddings. Finally, the results of this model must be compared with the lower-level representation of the training sample. This process would train the generator of sound representations. A second process of training the vocoder could be done separately. Other architectures are possible. This simple architectural example would scale well because the two most difficult parts, the generator, and the vocoder, can be trained in parallel and do not depend on one another. Further discussion on this topic is presented in section 4.3.

For this architecture to be successful, at least three models need to be trained: the text embedding, the lower-level sound generation, and the vocoder. Vocoders already exist and are a field of relatively intense development, as shown in 2.2.3. Also, text embedding can be already done very well, as 2.1.3.3 shows. This simplifies the problem as, theoretically, only the generator needs to be developed and trained. The possibility of focusing on this generator is an asset because it eases the workload. However, for best results, one should develop all the models or at least fine-tune them to the specific data that one is working with.

Chapter 4

Development and Implementation

Contents

This thesis's development and implementation chapter corresponds to the crucial task of developing the solution. This chapter comprehensively explores the methodology, dataset analysis, generative model development, and research plan. This chapter aims to provide a detailed account of the steps taken to achieve the research objectives and contribute to the field of generative AI models for audio.

In Section 4.1, the author outlines the approach taken in developing this thesis. Section 4.2 details the various datasets considered in this research. A thorough description and analysis of these datasets is provided, highlighting their characteristics and potential challenges.

Section 4.3 forms the core of this research. There, the author presents the development of several generative models, including the central model, GANmix. The model architecture, training process, and evaluation are discussed, demonstrating their contributions to the field and their potential for generating audio samples.

In addition, Section 4.4 outlines the timeline for the development of this thesis. A roadmap of key milestones and stages is provided.

Finally, practical information and resources, such as the source code, are available at `https://ctrlmarcio.github.io/audio-gen-ai/`. This allows interested readers to access and explore the codebase, facilitating further research and collaboration.

4.1 Methodology and Approach

The development of this dissertation is divided into three main parts:

- The state-of-the-art research;
- The development;
- The writing of the document;

These areas were not necessarily exclusive. That is, they overlapped. Initially, work began with general research into the state of the art in modeling and audio processing. Early development began after establishing a basic understanding of what needed to be achieved. This development took place in parallel with the ongoing research. As issues arose during development, additional research was required to address them. Writing the document was an ongoing process that led to clearer thinking and planning. Nevertheless, writing was prioritized after the bulk of the development was complete.

The interaction between these phases is both dynamic and symbiotic. The state-of-the-art research phase is the foundation, providing information for future model development decisions. Research insights directly influence the design and direction of generative models. As development progressed, new challenges and possibilities emerged, triggering further exploration in the state-of-the-art literature. Similarly, the development phase provides empirical insights that contribute to refining the writing process. Model development findings and outcomes contribute to comprehensive and informative document content. This iterative interplay ensures that each phase informs, enriches, and refines the others, leading to a coherent and impactful dissertation.

4.1.1 State-of-the-Art Research

One of the goals of this thesis is to provide a comprehensive overview of the current state-of-the-art audio synthesis through textual input. A systematic and thorough approach was taken to search and review the relevant literature.

The process started with the collection of keywords related to the topic of audio synthesis and generative AI models. Through multiple combinations, these keywords were then used to search multiple online sources, including academic search engines. Searches were also conducted for specific papers given the author's knowledge. The results of these searches were analyzed to identify publications with significant citations and high relevance to the thesis topic.

These publications were then carefully read and analyzed, and their citations were further investigated to expand the search and deepen the understanding of the state-of-the-art in the field. If any aspect of a publication was unclear, additional research was conducted to clarify the concept and find additional relevant literature. This process of searching, reading, and analyzing was iterated multiple times, allowing the gathering of new keywords and publications. Notes and possible citations for each publication were stored in a private database, allowing easy access and organization.

4.1.2 Model Development

The development of audio synthesis models is a progressive approach that aims to achieve three primary objectives. Together, these goals guide the development of the models and ensure a comprehensive exploration of audio synthesis through text.

The primary objective is establishing fundamental models that facilitate an in-depth understanding of sound representation and DL principles. To achieve this, the initial focus was on tasks like audio classification. These fundamental models are developed using smaller datasets like Audio MNIST [11]. This step provides a foundational grasp of the models' capabilities and acts as a basis for subsequent progress.

The second goal is to develop generative models to synthesize audio from textual input. These generative models are designed to create audio content that aligns with provided text prompts by

building on the insights gained from fundamental models. The progressive approach provides the ability for the generative process's iterative refinement, optimization, and enhancement.

The third goal includes proposing theoretical models that establish the foundation for more advanced generative techniques. Although not all theoretical approaches are entirely implemented, these concepts add to the broader discussion of audio synthesis research. Exploring theoretical models boosts innovation and promotes the development of new strategies for generating audio from text.

The model development approach provides an exhaustive and iterative pathway to achieve an effective and innovative audio synthesis from textual input by pursuing these interconnected objectives.

4.1.3 Writing of the Dissertation

This dissertation was composed with a focus on clarity, conciseness, and informative content. A continuous approach was utilized to achieve this objective, consisting of three key phases for each section: initial drafting, incorporation of researched references, and ongoing refinement. This iterative process required multiple re-readings and revisions. This dynamic method aligns well with the principles of agile development [30], promoting flexibility and continuous progress during the writing process.

Furthermore, this document adheres to established academic norms, encompassing writing style, citational methodologies, and the lucid and consistent presentation of results and figures. In addition to the individual author's contributions, the insights of the thesis coordinators have strengthened this project. They have diligently reviewed and enhanced the document to ensure its precision and authenticity.

This dissertation, crafted with meticulous attention to detail, linguistic technologies, and rigor and scrutiny, is the culminating embodiment of the author's resolute research pursuits and the zenith of scholarly achievement.

4.2 Dataset Selection and Analysis

This section is crucial to synthesizing audio from text inputs, as it reveals the foundation on which the generative models are developed and improved. Datasets are the foundation of this research. These datasets serve as the learning material for DL algorithms and a means to evaluate the perceptual quality, coherence, and creativity of the resulting audio outputs.

It presents the datasets researched and used to implement the models. It is worth noting that datasets that are well-known within the field but are neither used nor relevant to the specific problem at hand, are excluded from citation. For example, this discussion does not include datasets specialized in speech or music. In addition, although the data size is significant for DL algorithms,

all the datasets discussed in this section include more than 1000 sound samples. Nonetheless, it is important to recognize that this number is relatively small compared to image recognition datasets with millions of entries, like CLIP [99], commonly used in the field.

Two types of datasets are cataloged, differentiated mostly by the nature of the adopted label – *categorical* and *descriptive*. A categorical-label dataset comprises sounds that are associated with a single, distinct label, such as "music", "piano", or "singing". On the other hand, descriptive-label datasets comprise natural language sentences or textual descriptions of soundscapes, such as "boy singing while playing the piano". In the realm of DL, descriptive-labeled datasets are deemed more conducive. Nevertheless, datasets containing categorical labels can still be valuable when undergoing augmentation techniques. To the author's knowledge, Table 4.1 provides a comprehensive list of audio datasets that meet the aforementioned criteria to date.

Table 4.1: Comparison of datasets for soundscapes

Name	**Type**	**# Samples**	**Duration**	**Labels**
Acoustic Event Dataset [123]	Categorical labeled	5223	Average 8.8s	One of 28 labels
AudioCaps [67]	Descriptive labeled	39597	10s each	9 words per caption
AudioSet [43]	Categorical labeled	2084320	Average 10s	One or more of 527 labels
Audio MNIST [11]	Categorical labeled	30000	Average 0.6s	One of 10 labels
Clotho [33]	Descriptive labeled	4981	15 to 30s	24 905 captions (5 per audio). 8 to 20 words long each
FSDKaggle2018 [38]	Categorical labeled	11073	From 300ms to 30s	One or more of 41 labels
UrbanSound8K [112]	Categorical labeled	8732	Less or equal to 4s	One of 10 labels
YouTube-8M Segments [2]	Categorical labeled	237000	5s	One or more of 1000 labels

4.2.1 Categorical Labeled Datasets

The datasets represented here are the ones where the audio samples are tagged with simple labels. An example of one of these labels might be "Dog". This labeling contrasts complete with descriptive labeling as seen in Section 4.2.2. Even though these datasets are not perfect for this use case, they might be augmented or enhanced (see Section 2.1.3.1). One such example would be taping different sounds. For instance, taping a "dog" "bark" with a "truck" "honk" could generate the label "a dog barking followed by a truck honking".

The datasets presented here contain audio samples tagged with simple labels such as “Dog”, in contrast to the descriptive labeling discussed in Section 4.2.2. While these datasets may not be ideal for the purposes of this thesis, they can be augmented or enhanced (as explained in Section 2.1.3.1) through means such as taping difference sounds. For example, joining a “do barking” and a “truck honking” could produce the label “a dog barking followed by a truck honking.”

4.2.1.1 Acoustic Event Dataset

Information regarding this dataset is not much, and the classes have rather specific names such as “hammer” or “mouse_click”. Nevertheless, it has 5223 samples, completing 768.4 minutes, representing 28 different classes [123].

4.2.1.2 Audio MNIST

Audio MNIST [11] is a sonic take on the famous MNIST dataset [26]. It has a series of 60 speakers saying different numbers from 0 to 9. These samples add up to 30000 samples.

This dataset is not used in the model but rather as a first take for every baseline model for its simplicity and small size.

4.2.1.3 AudioSet

AudioSet is the most comprehensive dataset existent for audio [43].

It is a large-scale, high-quality dataset of annotated audio clips. It was created by Google Research and released in 2017 as part of the ongoing effort to advance state-of-the-art audio understanding. The dataset consists of over 2 million 10-second audio clips, each annotated with one or more labels from a set of 527 classes, covering a wide range of sounds such as human speech, animal vocalizations, musical instruments, environmental sounds, and more. The audio clips are sourced from YouTube videos and cover diverse content, including news broadcasts, movie trailers, music videos, and everyday videos.

The annotations in AudioSet are created using a hierarchical ontology, where the classes are organized into a tree-like structure based on their semantic relationship. This allows for a more fine-grained representation of audio events and makes it easier for machine learning models to learn the relationship between different sound categories.

4.2.1.4 FSDKaggle2018

Freesound Dataset Kaggle 2018 [38] is an audio dataset containing 11,073 audio files annotated with 41 labels following the same ontology as AudioSet. The sounds were taken from Freesound which is an online database maintained by the community. So, the quality may vary.

This dataset was used for competitions on Kaggle regarding sound classification.

4.2.1.5 UrbanSound8K

The UrbanSound8K [112] is an available dataset focusing on urban sounds. It is one of the largest and most diverse datasets of its kind.

The UrbanSound8K dataset consists of 8732 audio clips, each lasting approximately 4 seconds, collected from various urban environments. The audio clips are annotated with one of 10 classes. The classes in the dataset represent a wide range of typical urban sounds. The audio clips in the UrbanSound8K dataset are collected from various sources, including YouTube and Freesound, and are carefully selected to ensure a diverse and representative sample of urban sounds. The dataset includes audio recordings from different cities, captured in different seasons and weather conditions, to provide a wide range of sound characteristics and environments.

4.2.1.6 YouTube-8M Segments

The YouTube-8M Dataset is a significant video dataset [2] that contains millions of YouTube video IDs with high-quality machine-generated annotations from a diverse vocabulary of over 3,800 visual entities. This dataset aims to accelerate research on large-scale video understanding, representation learning, noisy data modeling, transfer learning, and domain adaptation approaches for video.

Upon reviewing the YouTube-8M dataset, it is evident that the minimum video length constraint of 120 seconds makes it unsuitable for this research as it should focus on snippets shorter than this length.

An alternative dataset that may suit this research's needs is the YouTube-8M Segments Dataset. This dataset is an extension of the YouTube-8M dataset and comes with human-verified segment annotations that temporally localize entities in videos. With about 237K human-verified segment labels on 1000 classes, the dataset enables the training of models for temporal localization using a large amount of noisy video-level labels in the training set.

The vocabulary of the segment-level dataset is a subset of the YouTube-8M dataset vocabulary, excluding entities that are not temporally localizable, such as movies or TV series that usually occur in the whole video. The dataset also provides time-localized frame-level features, enabling classifier predictions at the segment-level granularity.

While the YouTube-8M Segments Dataset provides temporal annotations for video entities, this dataset could still be valuable if one extracts the audio from each YouTube video, as it includes time-localized frame-level features that could be used for audio analysis. Thus, while the video content may not be essential for this research project, the YouTube-8M Segments Dataset could still be a valuable audio-analysis resource.

4.2.2 Descriptive Labeled Datasets

The datasets presented here present way richer textual information regarding the sound. For instance, instead of a sound being connected with the label "engine", datasets in this section connect it with a sentence such as "a vehicle engine revving". This allows for training the natural language input, one of the main assumed problems.

4.2.2.1 AudioCaps

AudioCaps [67] is a large-scale dataset of 46 thousand audio clips with human-written text pairs collected via crowdsourcing on the AudioSet dataset.

4.2.2.2 Clotho Dataset

Clotho [33] is an audio dataset that consists of 4981 audio samples. Each audio sample has five captions (a total of 24 905 captions). Audio samples are 15 to 30 seconds long, and captions are 8 to 20 words long.

A practical example is a sound that is described both by "a car honks from the midst of a rainstorm", "rain from a rainstorm is hitting surfaces and a car honks", and others.

4.3 Generative Model Development

Audio synthesis is the process of creating artificial sounds from scratch. It is a challenging task that requires a deep understanding of the nature and structure of sound, as well as the ability to generate realistic and diverse audio samples. Generative models have been widely used in image synthesis, but their application to audio synthesis is relatively new and less explored.

This section presents the development of a novel generative model for audio synthesis, called GANmix, which combines elements of both a GAN and a VAE. GANmix exploits the strengths of both models to produce high-quality and diverse audio samples in a computationally efficient manner. GANmix also allows for latent space manipulation, which can be used to control various aspects of the generated audio, such as pitch, timbre, and style.

Before introducing GANmix, this section also describes a series of exploratory experiments that provide a valuable learning phase for the development of the main model. The experiments cover different aspects of audio synthesis, such as audio representation, model architecture, training process, dataset, and evaluation metrics. The experiments aim to provide insight into the performance, limitations, and potential of different generative audio models.

The section is organized as follows: Section 4.3.1 presents the exploratory experiments that inform the development of GANmix. Section 4.3.2 introduces GANmix as a novel technique for audio synthesis.

4.3.1 Exploratory Experiments

In the development of generative AI models for audio synthesis, it is crucial to conduct extensive experiments that explore various aspects of model architecture, training processes, datasets, and evaluation metrics. This section presents a series of experiments that provide a valuable learning phase before developing the primary model, GANmix. The experiments aim to gain insight into the performance, limitations, and potential of different generative audio models. The author analyzes empirical findings to inform the development of GANmix, aiming for a robust and efficient solution.

The initial experiment focuses on a classification model to uncover the fundamental principles of sound. Training the model on labeled audio data, the author aims to grasp the connections among different audio features and their corresponding categories. This experiment lays the foundation for further research on audio representation.

Based on the results of the classification experiment, the author proceeds to develop a GAN for audio synthesis. The aim of this study is to assess the effectiveness of GANs in generating high-quality and realistic audio samples.

Furthermore, the author explores the efficacy of AEs and VAEs in audio generation. The experiments investigate the reconstruction and generation capabilities of these models, respectively.

The purpose of these experiments is to achieve a comprehensive understanding of both audio and different generative models for audio synthesis. These empirical findings provide a crucial basis for developing GANmix, ensuring a robust and effective solution for audio generation.

4.3.1.1 Preliminary Classification

To develop a comprehensive understanding of sound and its representation, a preliminary classification model was developed as a starting point.

The Audio MNIST dataset (see Section 4.2.1.2) was selected due to its simplicity and appropriateness. The dataset consists of spoken digit recordings divided into ten categories (numbers from 0 to 9). This classification was developed model using both TensorFlow and PyTorch frameworks (see Section 2.1.4).

This model's central premise is the possibility of achieving satisfactory outcomes using a simple CNN. The aim is to surpass 90% accuracy.

Conventional CNNs usually have convolutional and pooling layers (see Section 2.1.2.1), which are versatile and can handle inputs of different dimensions. However, utilizing these layers with inputs of varying sizes leads to different output dimensions. It's essential to conduct necessary pre-processing steps to impose uniform dimensions for all input data to obtain a more accurate assessment of the network's outputs.

To reduce variations in sample size, a combination of techniques, such as padding and random cropping, was used. To address padding, an approach called edge padding was adopted. This technique includes extending the beginning and end of smaller audio samples to match a fixed size. The main objective of edge padding is to preserve the audio content's integrity and ensure consistency in the dataset.

In contrast, the random crop technique has a unique operation within the confines of this study because the dataset mainly includes recordings of spoken digits that possess inherent characteristics. The spoken digit samples' nature is leveraged to systematically select discrete segments from the audio samples using random crop. This process produces 1.5-second segments.

The classification model is based on an architecture of CNN, with a sequence of five blocks, its general architecture is present in Figure 2.4. Each block included a convolutional layer followed by max pooling that systematically extracted hierarchical features. Finally, the network ended with an average pooling layer, which connects to three linear layers for final classification. The output layer, consisting of ten units, corresponds to the ten distinct classes in the dataset.

Average pooling is a down-sampling technique that partitions the input feature map into distinct regions and computes the average value for each partition, resulting in a down-sampled feature map that retains essential information about the input while decreasing spatial dimensions. As a result, the pooled feature map preserves essential information about the input while reducing the spatial dimensions. This technique helps control parameter count and reduce overfitting, thereby improving the generalization ability of the model.

In the present classification model, the average pooling layer effectively reduces dimensionality. The pooled features maintain vital high-level information gathered from previous convolutional layers, allowing the ensuing linear layers to concentrate on extracting more sophisticated semantic features for classification purposes. The decision to use this architecture was purposefully made to make the model lighter. Flattening the convolutional output would result in a higher number of parameters.

Model training involved using SGD optimization and Cross-Entropy loss. The training process lasted 20 epochs and involved batch-wise backpropagation to update parameters. It is noteworthy that the training occurred on Tesla P100 with 8GB.

The code for the model and the training loop can be see in Annex A.

4.3.1.2 Audio Generation with GAN

This section presents the methodology and results of applying a GAN model to generate audio samples from raw sound waves without spectrograms.

A GAN model, as explained in Section 2.1.2.3, consists of two neural networks: the generator and the discriminator. The generator creates new data samples that mimic the input data distribution while the discriminator evaluates how realistic the created data samples are. The goal is to train the generator to create samples that are indistinguishable from the real data, according to the discriminator.

However, training GAN models poses significant computational cost and time challenges. This is because a GAN model involves two neural networks that must be trained simultaneously. The generator creates new data samples, and the discriminator evaluates their realism. Therefore, each network has to wait for the output of the other network to update its parameters. This process can be time-consuming, especially for larger datasets or more complex architectures. Moreover, since GAN models are computationally intensive, they require powerful GPUs and can take days or weeks to train.

In this section, raw sound waves were used as input data instead of spectrograms. This was done to eliminate the need for a vocoder (see Section 2.2.3). The audio input size of the Audio MNIST dataset varied, so the first step was to pad the audio samples with zeros to a fixed size that could accommodate all samples. This was achieved by applying the same preprocessing techniques applied in Section 4.3.1.1.

A normalization process was applied to the audio data to confine the audio signal values within a well-defined range, mitigating the potential occurrence of gradient explosion. This facilitates the convergence of the learning process for the model.

The TensorFlow framework was utilized to implement the GAN model.

The generator network architecture consists of a series of layered operations that include dense, reshape, and transposed convolutional transformations. The layers are comprehensively depicted in Annex B.

The input layer takes a noise vector (z) of size 100 as input and passes it through a dense layer with 256 units. Then, six transposed convolution layers are applied with a kernel of 25 and stride set to 4. The number of filters goes from 32 to 1, generating an output with only one channel. Each transposed convolution layer is followed by a ReLU activation function (see Section 2.1.2.2), except for the last one, which uses a tanh activation function to bound the last values between -1 and 1, emulating a normalized raw sound. The output of the last layer is a 16384×1 vector, which represents the generated audio sample ($G(z)$).

The generator network can be formally defined as follows:

$$G(z) = \tanh(\text{Conv1DTranspose}_6(\text{ReLU}(\text{Conv1DTranspose}_5(\cdots \\ \text{ReLU}(\text{Conv1DTranspose}_1(\text{Reshape}(\text{Dense}(z)))))))) \tag{4.1}$$

where Dense, Reshape, Conv1DTranspose, ReLU, and tanh denote the corresponding operations with their parameters omitted for brevity.

The discriminator network is composed of several layers of convolution and dense operations. These can also be seen in Annex B

The input layer takes either a real audio sample (x) or a fake audio sample ($G(z)$) of size 16384×1 as input. Its architecture mirrors that of the generator. It passes through six convolution layers with a kernel of 25 and a stride of 4, starting from one filter until 32. Each convolution layer is followed by a leaky ReLU activation function with an alpha parameter of 0.2 to avoid the dying ReLU problem (see Section 2.1.2.2). The output of the last layer is flattened into a 32-dimensional vector. Then, a dense layer with one unit is applied to produce a scalar value, which represents the probability ($D(x)$ or $D(G(z))$) of the input being real.

The discriminator network can be formally defined as follows:

$$D(x) = \text{Dense}(\text{Flatten}(\text{Conv1D}_6(\text{LeakyReLU}(\text{Conv1D}_5(\cdots \text{LeakyReLU}(\text{Conv1D}_1(x))))))) \tag{4.2}$$

where Conv1D, Flatten, Dense, and LeakyReLU denote the corresponding operations with their parameters omitted for brevity.

The training process alternates between updating the parameters of the generator and the discriminator using gradient descent. Both networks use the binary cross-entropy (BCE) loss function, which is a variation of the cross-entropy loss function explained in Section 2.1.3.2. Cross-entropy measures how accurately the networks predict the input labels. BCE is designed for discriminating between two categories — real and fake samples, in this case. The generator aims to reduce BCE loss by encouraging the discriminator to assign high scores to fake samples. On the other hand, the discriminator aims to minimize BCE by producing low output values for fake samples and high output values for real samples. The interaction between generator and discriminator, which is guided by binary cross-entropy, helps in the training and enhancement of the generative model.

The training algorithm can be summarized as presented in Algorithm 1.

The implementation code for this architecture is presented in Appendix B, its general architecture is present in the Figure 2.18.

Algorithm 1 GAN Training Algorithm

Initialize generator (G) and discriminator (D) parameters randomly
Set number of epochs (E) and batch size (B)
for $e = 1, \ldots, E$ **do**
    Shuffle the real audio samples (X)
    for $b = 1, \ldots, \frac{|X|}{B}$ **do**
        Sample a batch of noise vectors (Z) from a normal distribution
        Generate a batch of fake audio samples ($G(Z)$) using G
        Compute the discriminator outputs for real ($D(X)$) and fake ($D(G(Z))$) samples using D
        Compute the generator loss (L_G) using BCE and $D(G(Z))$
        Compute the discriminator loss (L_D) using BCE and $D(X)$ and $D(G(Z))$
        Update G parameters by descending the gradients of L_G
        Update D parameters by descending the gradients of L_D
    end for
    Generate and save a sample audio using G
end for

The GAN model was trained on a small dataset of audio samples for 20 epochs. This was done to reduce training time since it was developed on a personal computer. Despite limited training time, results were promising. Generated audio samples were based on random noise but resembled real ones in terms of wave amplitude.

These results demonstrated that it was possible to generate realistic audio samples using raw sound waves as input data.

However, developing this model was challenging. The author used TensorFlow but faced difficulties with code complexity and memory usage. This led to frustration as the model needed scaling up.

To overcome this challenge, the author had to learn PyTorch, known for simplicity and flexibility, as seen in section 2.1.4.2. From now on, assume every implementation was done in PyTorch.

4.3.1.3 Simple Autoencoder for Audio Data Compression

This section outlines the procedure for building a basic AE (refer to Section 2.1.2.1), which is an essential step in making more complex versions like the VAE (refer to Section 2.1.2.3) in future research. The PyTorch-based implementation of the AE is available in the given code repository (refer to Annex C).

The same preprocessing applied to previous models (padding and random cropping) is applied in this one.

The architecture of the AE is reminiscent of a U-Net (see Section 2.1.2.1), incorporating four types of layers: convolutional, max pooling, upsampling, and transposed convolutional layers (see Section 2.1.2.1). The activation function used in convolutional and transposed convolutional layers is the tanh function (see Section 2.1.2.2). The tanh activation function ensures that the output of the AE remains within the range of -1 to 1, which is crucial for the subsequent denormalization process. Code for defining and training the model, as well as a result example can be seen in Annex C, an abstract representation of this architecture is in Figure 2.10.

The encoder is constructed as a sequence of convolutional layers, followed by batch normalization, activation functions, and max-pooling operations. The input to the encoder is a 1D audio waveform with a single channel. The first convolutional layer has 32 filters, a kernel size of 9, a stride of 1, and a padding of 4. It is followed by batch normalization and the tanh activation function. A max-pooling layer with kernel size 2 and stride 2 is then applied.

There were four convolutional layers. For each one, the number of filters is doubled from the previous layer, and the exact configuration of convolution, batch normalization, activation, and max-pooling operations is repeated. The kernel size, stride, and padding remain consistent for all convolutional layers.

The decoder is constructed as a sequence of upsampling (by a factor of 2), transposed convolutional layers, batch normalization, and activation functions. The decoder's architecture is symmetric to the encoder, with the number of filters halving at each layer until reaching the original number of channels (1). The transposed convolutional layers have the same kernel size, stride, and padding as the corresponding encoder convolutional layers. The last layer of the decoder applies batch normalization, a tanh activation function, and produces the reconstructed audio waveform.

The ReLU activation function was tested during experimentation, but the results proved unsatisfactory, as every sound frame was above 0. Consequently, the decision was made to continue using the tanh activation function for the simple AE.

A series of steps is carried out during the training loop to train the model. First, each batch of audio samples undergoes a normalization process to ensure consistent data representation. The normalized data is then fed through the model, resulting in two crucial outputs: an encoded representation of the audio and a reconstructed audio waveform.

To assess the quality of the reconstructed audio, a loss function is employed, which quantifies the dissimilarity between the reconstructed waveform and the original input using MSE (see Section 2.1.3.2). This loss value serves as a measure of how well the AE can capture and reproduce the essential characteristics of the audio data.

The calculated backpropagation gradients are subsequently used to update the model's parameters using the Adam optimizer (see Section 2.1.2.2), iteratively refining the AE's ability to encode and decode the audio samples.

The training process follows an iterative approach, where the model is trained for multiple epochs. The training data is iterated over in each epoch, and the model's parameters are updated based on the computed gradients. The best model (with the lowest loss) obtained during training is saved for future use.

By constructing this simple AE, valuable insights into the underlying mechanisms of AEs are gained, which is instrumental in developing more sophisticated techniques. Moreover, the provided code implementation in PyTorch (refer to Annex C) facilitates a deeper understanding and exploration of the AE architecture, enabling improvements and extensions to audio data compression.

4.3.1.4 Simple Variational Autoencoder

This section focuses on the development and experimentation of the VAE, a generative model that, as explained in Section 2.1.2.3, with a representation in Figure 2.17, aims to learn latent representations of data while enabling controlled generation. The VAE was developed as a means of further exploring generative techniques, expanding on the groundwork established by earlier models and adjusting it to suit the specific characteristics of audio data.

Initial implementation The developed VAE utilized the identical dataset as the previous models, covering unprocessed audio waveforms. Comparable preprocessing transformations were performed, such as random cropping and padding.

Three primary components constituted the VAE's structure: an encoder, a bottleneck layer, and a decoder. The encoder utilized a variable number of convolutional layers, but for practical reasons, the number was restricted to two. Hardware restrictions played a vital role in imposing this constraint since more layers would result in excessively extended convergence times. Despite its limitations, the bottleneck layer, consisting of two separate linear layers for mean and variance, effectively encapsulated latent features.

The decoder mirrored the encoder's structure, using deconvolutional layers instead of convolutional ones and upsampling instead of max-pooling. This architecture ensured the generation of audio samples preserving the input data's characteristics.

The training process utilized a loss function that combined BCE and KL divergence, which is in line with standard VAE practice. The training emphasized the reconstruction of input samples and learning the latent space, similar to the previousAE model.

Fine-Tuning Given Resource Constraints As the project progressed, resource limitations played a significant role in shaping the trajectory of the model. Achieving optimal performance of the VAE within these constraints became a major challenge. To overcome this challenge, the author underwent fine-tuning of multiple elements such as convolutional layer count and bottle neck layers dimension size. However, the computational demands of these fine-tuning steps proved to be prohibitive, leading the author to explore a different approach.

Focusing on the learning rate, a critical parameter that affects training speed and convergence, was crucial. A structured exploration of learning rates was conducted, considering a range of values spanning several orders of magnitude. This approach facilitated meaningful experimentation within a limited timeframe. The exploration of learning rates uncovered the trade-offs between excessively small and overly large values. Values in the range of 1×10^{-4} to 1×10^{-3} have shown promising results, indicating a need for further investigation.

The process of training and fine-tuning the VAE has highlighted the significant impact of computational resources on model development. Balancing model complexity with resource limitations

has proven to be a complex task, resulting in innovative approaches to optimizing training while retaining high-quality generative capabilities.

After training for just five epochs, the loss amounted to 124261.2969. A visual representation of the reconstructed sample can be found in the annex D.

4.3.2 GANmix

This section introduces GANmix as a novel technique for audio generation that fuses components of both a GAN and a VAE. GANmix addresses the problem of generating high-quality audio under resource constraints.

Audio generation is computationally expensive, especially when constrained by hardware capabilities. GANmix offers a potential solution by combining the strengths of GANs and VAEs to improve audio synthesis in limited computational environments through latent space manipulation.

Model Architecture Plan The GANmix architecture consists of a generator and a discriminator that operate in latent space rather than in the space of the sound or soundscape. This approach was inspired by stable diffusion (see Section 2.1.5.3), which generates samples through its latent space instead of working on the sample space itself.

Traditionally, in GANs, the generator produces samples to be evaluated by the discriminator. However, GANmix takes a different approach: its generator produces values in an embedding space defined by the pre-trained AudioLDM VAE encoder [74]. The discriminator, on the other hand, objectively verifies latent features by comparing the generated features with those obtained by the VAE encoder. Samples are generated posteriorly by passing the generated features through the decoder of the VAE.

The decision to incorporate the AudioLDM VAE model within the GANmix architecture was based on thoughtful reasons.

1. **VAE Training:** Training VAEs poses a challenge since it typically requires significant computational resources and extensive datasets to achieve effective convergence. Previous experiences with model development (in Section 4.3.1.4) for this thesis highlighted the intricacies involved in VAE training.

2. **AudioLDM's High-Performance:** At the time of GANmix's development, the AudioLDM model was one of the top models for audio generation.

3. **Accessibility and Open Source Nature of AudioLDM:** One key benefit of integrating the AudioLDM VAE model into GANmix architecture was its openness and accessibility through platforms like Hugging Face's model hub through `https://huggingface.co/cvssp/audioldm`. This accessibility streamlined the incorporation of the high-performing AudioLDM.

The generator is designed to convert random Gaussian noise vectors into latent features that resemble the encodings of the AudioLDM VAE model. Initially, a convolutional model was designed. It used four convolutional transpose 2D layer blocks, with Leaky ReLU activation functions following three of them. Each block comprised several convolutional layers. Furthermore, after the

deconvolutional layer, two convolutional layers are employed to preserve the data's shape. The final block, responsible for producing the transformed audio samples, did not use an activation function. This design decision enabled the generator to generate unrestricted values, ensuring the accuracy of the produced audio samples.

The decision to utilize 2D convolutions arised from the innate characteristics of the latent features generated by the AudioLDM VAE model. These latent features have various dimensions, and integrating 2D convolutional layers enabled GANmix to efficiently grasp complicated patterns across these dimensions. By utilizing 2D convolutions, the architecture could more effectively utilize the multidimensional information in the latent representations, resulting in an improved quality of the generated audio output.

The discriminator architecture mimicked that of the generator with three blocks of convolutional 2D layers using leaky ReLU activation. Each block contained multiple convolutional layers (three in the proposed architecture). The final layer of the discriminator applied a sigmoid activation function to generate a probability score indicating the authenticity of the input audio sample.

Experimental Results Preliminary experiments with GANmix and the Audio MNIST dataset produced promising but flawed results. Objective analysis reveals that the generator's performance lagged behind the discriminator's, resulting in suboptimal sample quality. Therefore, the training process requires further refinement.

In order to bridge the performance gap between generator and discriminator, a range of refinements were explored. These efforts included exploring different optimizers, such as Adam, RMSProp, and SGD (see Section 2.1.2.2), each with varying hyperparameters. In addition, the author adjusted model sizes, experimented with diverse loss functions, and introduced noise to training samples. However, the speed of training and comprehensive experimentation were still hindered by hardware resource limitations.

More information regarding these explorations can be found in Chapter 5.

With access to a more powerful computing environment, GANmix was further developed using the Clotho dataset (see Section 4.2.2.2), leading to a significant improvement in the quality of generated audio samples. Nonetheless, there were still issues with achieving equilibrium between the generator and discriminator, even with the upgraded dataset.

Final Model After conducting a series of experiments, the GANmix model reached its culmination. This architecture incorporates the Clotho dataset and includes significant improvements to both the generator and discriminator.

The GANmix model is the final model used in this thesis, corresponding to the model used in Experiment 10. Unlike most state-of-the-art generative models that use CNNs, the GANmix model

uses fully connected neural networks for both the generator and the discriminator. The rationale for this was empirical and is given in section 5.2.10.

The generator takes as input a random Gaussian noise vector of size NZ and passes it through a fully connected hidden layer of NH neurons. The output of the hidden layer is then fully connected to another layer of size $EW \times EH \times ED$, where EW, EH, and ED are the width, height, and number of dimensions of the embeddings generated by the VAE, respectively. The output layer is reshaped to match the shape of the VAE embeddings.

The discriminator takes as input an embedding of shape $EW \times EH \times ED$ and flattens it to a vector. The vector is then passed through a fully connected hidden layer of NH neurons, and then fully connected to a single output neuron that applies a tanh activation function.

The VAE encoder and decoder are pre-trained by a state-of-the-art network, AudioLDM. However, the GANmix model can work with any VAE network, as long as the size of the last layer of the generator (and the first layer of the discriminator) is adjusted accordingly.

The loss function used to train the GANmix model is BCE. The generator is optimized using Adam with a learning rate of 1×10^{-3}, while the discriminator is optimized using Adam with a learning rate of 1×10^{-4}. A scheduler updates the learning rate every 10 epochs.

The implementation of GANmix can be found in Appendix E. More about the configuration and parameters of the GANmix model can be found in Appendix F. The model's architecture is shown in Figure 4.1.

4.4 Research Plan

This section presents a brief overview of the research tasks, complete with their corresponding timelines and milestones. One should be aware that as the activities and tasks become more defined, the research plan may be altered and enhanced over time.

The primary phase of this research entails conducting an exhaustive examination of existing generative AI models for audio. This task requires acquiring a fundamental understanding of the topic, exploring the history of AI, and gaining basic knowledge of sound. Furthermore, an extensive study was conducted on the latest advancements in generative models, specifically for audio. The findings of this analysis are presented in the Chapter 2. The interim checkpoint in February 2023, ending of the Dissertation Planning course, mandated the completion of this task.

After conducting a thorough literature review, the next step is to craft the introduction section of the thesis, which involves understanding the research objectives and scope and providing a comprehensive overview. The introduction sets the context, motivation, and objectives of the study. The deadline for completing the introduction is February 2023.

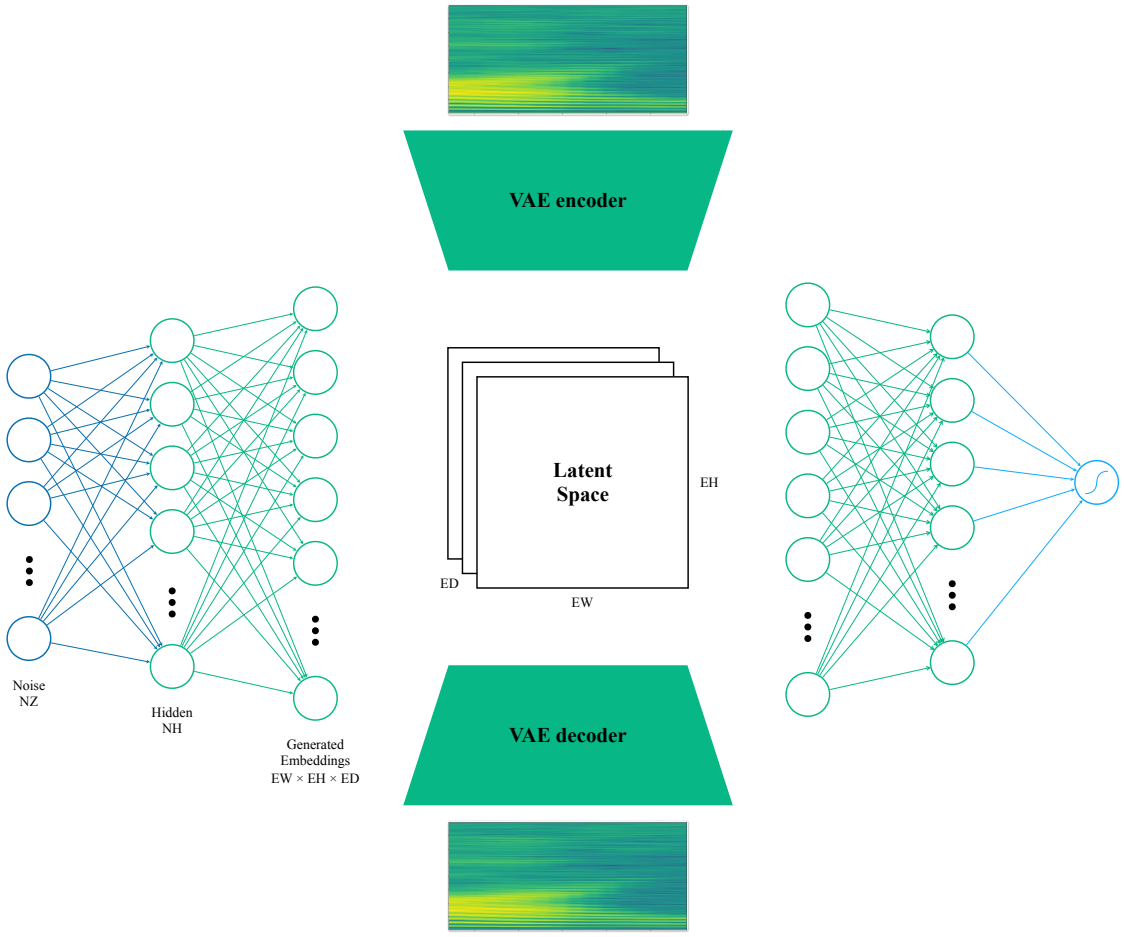


Figure 4.1: **GANmix architecture** — On the left is the GANmix generator with three layers: the initial noise layer with NZ neurons, the hidden layer with NH neurons, and the layer corresponding to the embedding space with $EW \times EH \times ED$ neurons. The outputs of this layer are transformed into the form of the outputs of the VAE. On the right is the discriminator that mirrors the generator. The last layer has a single neuron with a sigmoid activation function. At the top is a spectrogram that passes through the pre-trained VAE generator to generate the latent space. At the bottom, the opposite happens: embeddings in the latent space go through the VAE decoder to generate the audio sample in the form of a spectrogram.

To aid in the experimentation and development of generative AI models for audio, a comprehensive exploration and analysis of prevalent audio datasets was conducted. This included identifying relevant datasets that align with the research objectives and gaining permission and access to these datasets.

Following the assembly of the datasets, the subsequent step involves outlining the problem to be addressed in the thesis. This involves clearly stating the research questions and goals that will direct the development and assessment of these models.

In anticipation of future assignments, a rudimentary dataset to generate a straightforward classification model was utilized. This entailed constructing and training the model, evaluating its

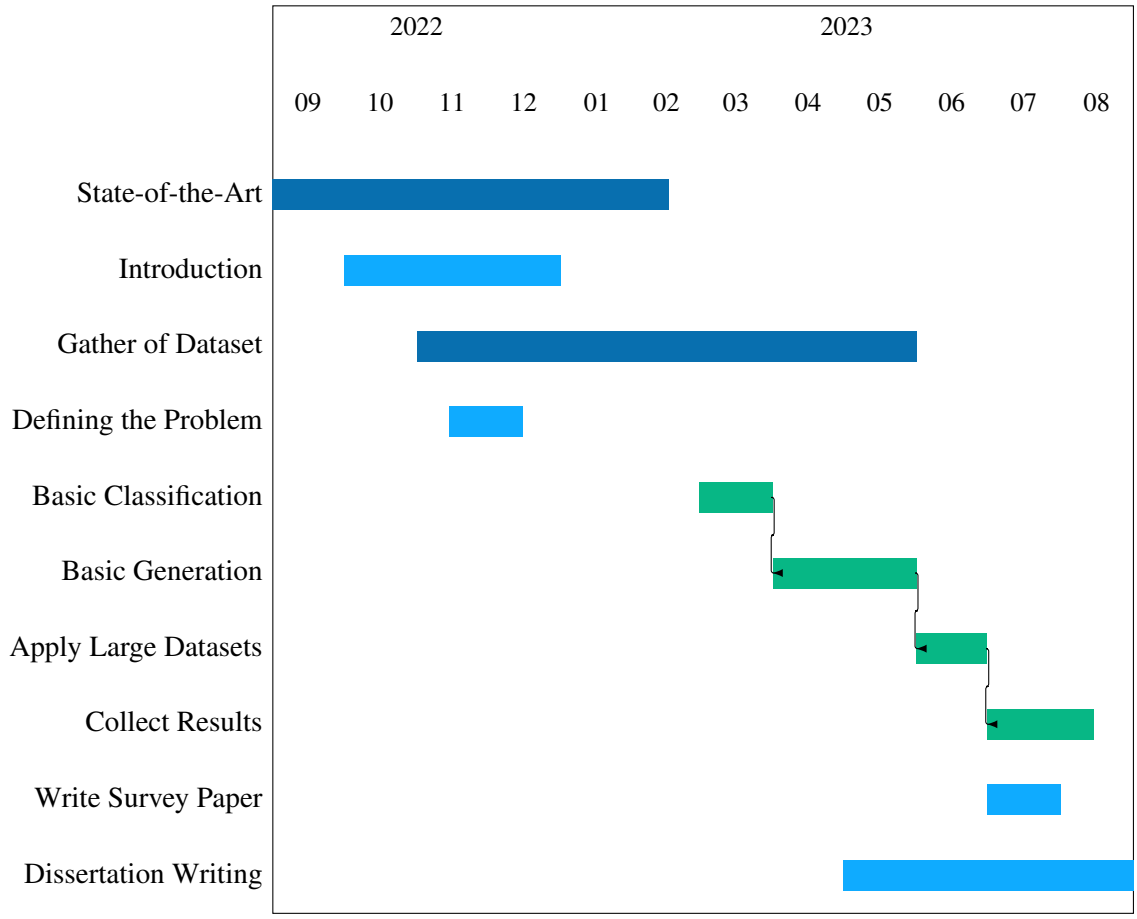


Figure 4.2: **Work Plan Gantt Chart** where the dark blue represents researching, light blue represents writing, and green represents developing.

effectiveness, and recording the outcomes.

Having established the classification model, the author proceeded to build a foundational audio generation model. This task involved implementing and training the model on the same basic dataset, evaluating its performance, and documenting the findings.

To improve the developed generative models, significant datasets were employed. The models were modified and updated to account for the limitations and intricacies of these larger datasets. The model's performance was assessed, and the outcomes were analyzed and documented.

During this phase, a range of extensive experiments were conducted to evaluate the effectiveness of the developed models. Different techniques were examined, and the models were improved based on the findings. The outcomes of these experiments were carefully examined and recorded.

Subsequently, after concluding the research and experimentation phases, the focus shifted to drafting the thesis. This involved consolidating the research results, analyzing their significance, and producing an organized paper that neutrally presents the research methodology, findings, and outcomes.

In addition to the dissertation, a survey paper was composed that outlines the latest developments in generative AI models for soundscapes. The paper provides an extensive synopsis of advancements in the domain and contributes significantly to the scholarly audience.

It is important to note that the deadline for completing these tasks has been extended until September 1st due to the challenges in obtaining permission to access the Artificial Intelligence and Computer Science Laboratory (LIACC) servers. As the research progressed, the schedule was reviewed and updated to ensure accuracy and importance.

Chapter 5

Evaluation and Discussion

Contents

The assessment of generative AI models for audio is pivotal in advancing the field of audio generation and synthesis. It is imperative to comprehend the capabilities and limitations of these models for their effective development and implementation. This chapter presents the outcomes and analysis of experiments performed with the GANmix architecture, utilizing objective metrics based on loss functions. The aim of this study is to assess the efficacy of generative AI models in producing audio, and to draw insights from their performance.

The experiments were conducted utilizing different hardware configurations, For GPUs, it consisted of Kaggle's Tesla V100, LIACC's GeForce GTX 1080, and LIACC's GPU Quadro RTX 8000. These configurations, along with the utilization of the PyTorch deep learning framework, allowed for proficient training and assessment of the generative AI models.

In this chapter, the author describes the experimental setup in Section 5.1, comprising both hardware and software configurations, alongside the GANmix architecture. The experiments' outcomes are presented in Section 5.2 with line plots exhibiting the generator and discriminator's loss evolution per epoch. Additionally, the resulting spectrograms are presented to offer a graphical depiction of the produced audio.

Following the presentation of results, the author analyzes and interprets the findings in Section 5.3, identifying trends, patterns, and significant insights that emerged from the experiments. Comparisons are conducted among various models and variations in the GANmix framework for evaluating their respective performances.

Furthermore, in Section 3.4, the author addresses any limitations or challenges encountered during the evaluation process. These factors may include hardware limitations or other constraints that impact the performance of the models.

5.1 Experimental Setup

This section outlines the experimental setup utilized to assess the audio generation capabilities of the GANmix model.

The GANmix model was trained using the BCE loss, which was implemented through pytorch's `BCEWithLogitsLoss` module. Although BCE is commonly employed for binary classification, it can also effectively train the generator in GANs. In the context of GANmix, the generator's goal is to create authentic embeddings that can fool the discriminator into classifying them as authentic. To achieve this, the generator's loss is calculated using the discriminator's output for the fake embeddings generated by the generator and the true label — either real or fake — for a real embedding.

By using BCE, the generator calculates its loss by measuring the difference between the discriminator's classification of genuine and false embeddings. This loss directs the generator to generate

more realistic embeddings over time by minimizing this difference. Therefore, despite being a binary classification loss function, BCE is a suitable approach for training the generator in GANs.

Two datasets, the Audio MNIST dataset [11] and the Clotho dataset [33], were utilized in these experiments. These datasets were previously described in detail in Section 4.2.

The Audio MNIST dataset contains short audio clips where each clip represents a spoken digit. This dataset sets a standard for evaluating audio classification tasks.

In contrast, the Clotho dataset is more extensive and diverse. It offers a variety of audio samples including environmental sounds and speech from various sources. This dataset offers a diverse and comprehensive range of audio data, allowing the GANmix model to learn and generate audio samples that capture the complexity and diversity present in real-world audio recordings.

For a thorough understanding of the datasets, including their characteristics, preprocessing steps, and data augmentation techniques, please refer to Section 4.2.

The experiments were conducted on three distinct hardware configurations, referred to as *Kaggle*, *LIACC 1*, and *LIACC 2*, each selected based on the available resources during the development process.

Kaggle, the initial configuration implemented for the GANmix model development phase, utilized a Tesla V100 GPU equipped with 12 gigabyte (GB) of memory, 73.1 GB of disk space, and 13 GB of Random Access Memory (RAM).

As development progressed, the LIACC 1 became available, which offered around the same computational power, LIACC 1 used a GeForce GTX 1080 GPU with 8 GB of memory, 50 GB of disk space, and 32 GB of RAM.

In the final stages of the project, the third configuration, LIACC 2, was made available. LIACC 2 employed a GPU Quadro RTX 8000 with 48 GB of memory, 50 GB of disk space, and 128 GB of RAM, allowing for extensive training and evaluation of the GANmix model.

These hardware configurations were chosen to facilitate GANmix model training and evaluation throughout the developmental stages.

The GANmix model was implemented using the Python programming language and the PyTorch deep learning framework. There is a further discussion about this framework in Section 2.1.4.

Before training, a preprocessing step was applied to the audio data by randomly cropping the dataset sample to the duration of 5 seconds. Random cropping is discussed further in Section 4.3.1. This approach enabled a more diverse dataset since the majority of its samples had longer durations. Random cropping aided in capturing diverse segments of the audio and improved the model's ability to generate realistic audio samples.

The GANmix model underwent training using a set of specific hyperparameters. The batch size ranged between 1 and 32, providing varying trade-offs between computational efficiency and model convergence. The training epochs varied from a few tens to a few hundreds, depending on the dataset and model complexity.

The learning rates utilized to train the GANmix model ranged from 1×10^{-5} to 1×10^{-2}.

The training process was stopped based on evidence of convergence before reaching any predetermined number of epochs. This choice was made to optimize computational resources while still obtaining satisfactory results. In the analysis presented in section 5.2, the graphs are set to plot a maximum of 100 epochs.

5.2 Presentation of Results

This section details the outcomes of experiments that aimed to develop and evaluate generative AI models for audio production. The chief objectives of these experiments were to explore the potential of these models to generate audio with exceptional quality and to assess their performance in various circumstances. The experiments addressed specific research questions and hypotheses.

This section presents a thorough summary of the experiments conducted in this research. Each experiment was intentionally designed to evaluate and test the effectiveness of the generated AI models for audio.

Each experiment begins with an introduction that highlights the motivation and specific objectives, thereby establishing the context and rationale for the experiment. In addition, the text provides a detailed description of the hardware, software, and architecture used in each experiment to ensure transparency and reproducibility.

The performance of the generative AI models is assessed by presenting line plots showing the evolving losses throughout the training process. In addition, spectrograms are displayed to visually represent the generated audio, allowing a qualitative evaluation of the model's performance.

By systematically organizing the experimental details, a thorough understanding of the experimental procedure and results is established. This section provides the basis for analyzing and discussing the effectiveness of the generative AI models. These results can also be seen in table format in Appendix H.

5.2.1 Experiment 1: Initial Model Evaluation

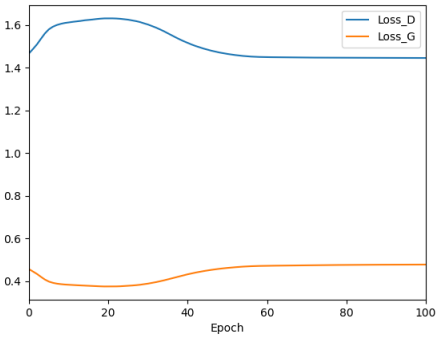
The objective of Experiment 1 was to assess the initial model's performance and to establish a baseline for future development in generative model research. The goals were to appraise the model's ability to produce realistic audio and to analyze its training process.

The initial model comprised about 4 million parameters, evenly distributed between the generator and the discriminator. The model was trained using the BCE loss function, and no regularization techniques were applied in this experiment.

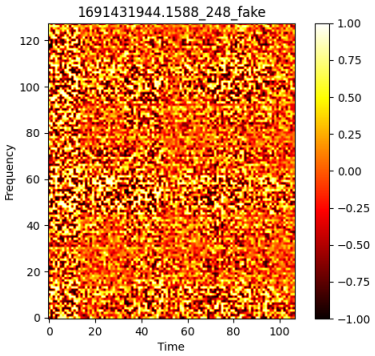
The Audio MNIST dataset was used for both training and evaluation. The Adam optimizer with a learning rate of 1×10^{-4} was used for the generator and discriminator. Throughout the training, the generator loss was calculated as 0.487, while the discriminator loss was measured to be 1.440. The total loss, which is the sum of the generator and discriminator losses, was calculated to be 1.927.

Upon evaluation, it was determined that the initial model's spectrogram did not resemble actual audio. However, the training process quickly stabilized.

Figure 5.1a presents a line plot of the continual losses during the training process for Experiment 1, providing insight into the model's learning progress and convergence. Additionally, the spectrogram generated by the initial model is depicted in Figure 5.1b, providing a visual assessment of the audio created.



(a) Evolving losses throughout the training process for Experiment 1.



(b) Spectrogram generated in Experiment 1.

Figure 5.1: Results of Experiment 1.

5.2.2 Experiment 2: Accelerating Convergence through Elevated Learning Rates

Given the extended convergence observed during the initial experiment, the author deliberated on a strategic approach to hasten the convergence rate. This led to the contemplation of raising the learning rates as a feasible method of accelerating the convergence process.

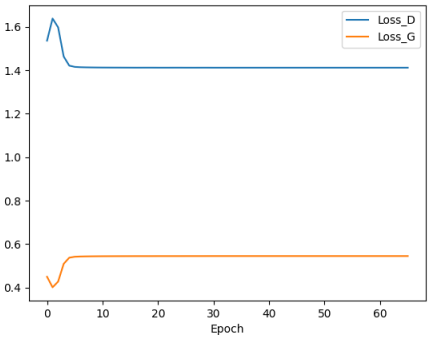
This model shared the same number of parameters (about 4 million), loss function (BCE), dataset, optimizer, and hardware setup as the former one.

The learning rate, for both the generator and the discriminator, was elevated to 1×10^{-2}. At the conclusion of the training process, the generator's loss was calculated at 0.545, and the discriminator's loss was measured at 1.412. The total loss, represented as the cumulative sum of the

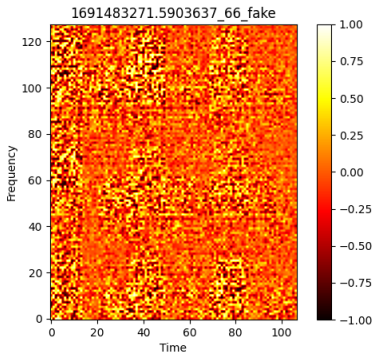
generator and discriminator losses, was calculated at 1.957.

Upon evaluation, the spectrogram of the initial model was determined to be similar to the first one. The point at which the models began learning at a slower rate was reached more quickly, but the outcomes remained unsatisfactory.

Figure 5.2 shows the loss and final spectrogram produced.



(a) Evolving losses throughout the training process for Experiment 2.



(b) Spectrogram generated in Experiment 2.

Figure 5.2: Results of Experiment 2.

5.2.3 Experiment 3: Enhancing Performance through Learning Rate and Model Complexity

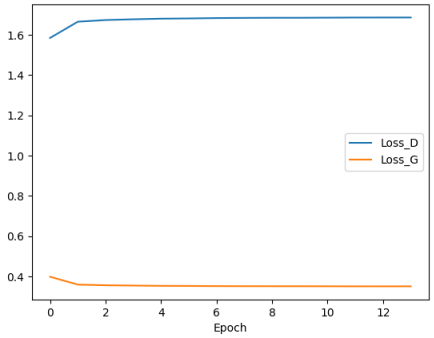
Experiment 3 was designed to test two hypotheses aimed at improving the generative model's performance. The first hypothesis assumes that setting a higher learning rate for the generator than for the discriminator at the outset could enhance performance. The second hypothesis suggests that improving the generator model's complexity could lead to better results. These hypotheses stem from the assumption that the generator task is more challenging than the discriminator task.

For this experiment, the discriminator model remained unchanged while the generator model had significantly more trainable parameters, totaling about 25 million. This was done by increasing the number of filters and the number of convolutional layers per deconvolutional block. The experiment utilized the same loss function (BCE), dataset, optimizer, and hardware configuration as the previous iterations.

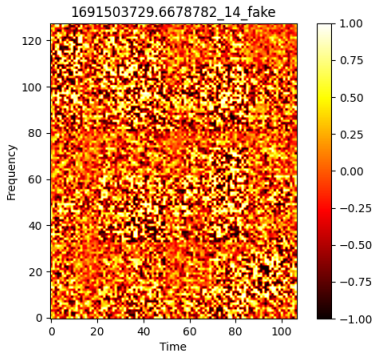
To test the hypotheses, the author derived the learning rates from the initial experiment. The generator learning rate was increased to 1×10^{-3}, while the discriminator learning rate remained at 1×10^{-4}. Training for this model stopped after 14 epochs because the results were comparable to those of the first iteration.

The total loss was 2.037, calculated by adding the generator loss of 0.351 and the discriminator loss of 1.686.

Figure 5.3 presents the loss and final spectrogram generated in this experiment.



(a) Evolving losses throughout the training process for Experiment 3.



(b) Spectrogram generated in Experiment 3.

Figure 5.3: Results of Experiment 3.

5.2.4 Experiment 4: Comparing Optimization Algorithms: RMSProp and SGD

Experiment 4 tested two optimization algorithms on the generative models.

The experiment examined the performance of the RMSProp and SGD optimizers, in contrast to the previous experiments that relied on the Adam optimizer.

This experiment utilized identical model architecture, loss function (BCE), and dataset as Experiment 3. The code was transferred to LIACC 1 hardware configuration, with minimal alterations.

The initial experiment in this set utilized the RMSProp optimizer's default settings. The comprehensive loss was 1.966, comprising a generator loss of 0.558 and a discriminator loss of 1.407.

Figure 5.4 offers insight into the loss and last spectrogram produced during this trial.

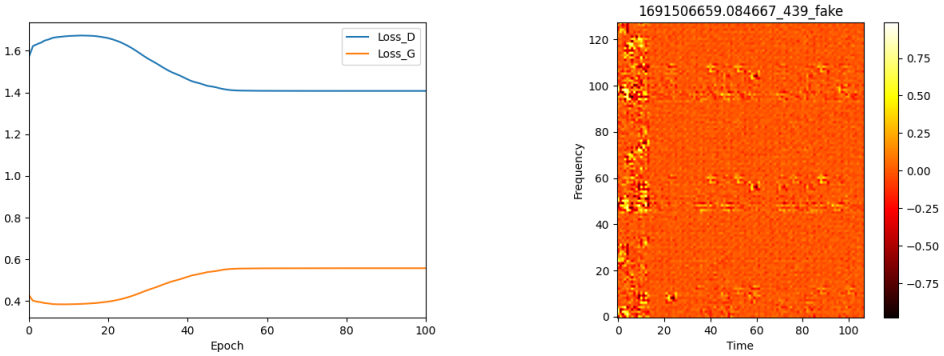
The next experiment in the sequence employed the standard configuration of the SGD optimizer (for specifics, see Section 2.1.2.2). The total loss was calculated to be 1.945, with a generator loss of 0.516 and a discriminator loss of 1.429.

Figure 5.5 presents the loss and the final spectrogram generated in this experiment.

The obtained results were comparable to those obtained with the Adam optimizer in previous iterations.

5.2.5 Experiment 5: Enhancing Performance with Regularization Techniques

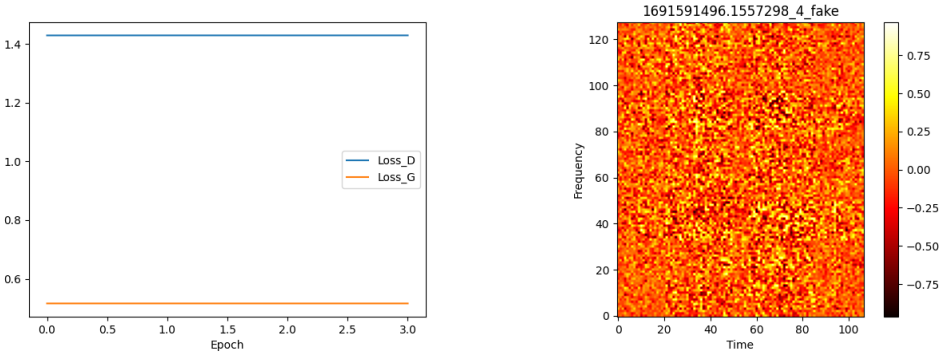
In this experiment, the author aimed to test the hypothesis that incorporating regularization techniques can enhance model performance. As such, several regularization techniques were utilized



(a) Evolving losses throughout the training process for Experiment 4 with RMSprop.

(b) Spectrogram generated in Experiment 4 with RMSprop.

Figure 5.4: Results of Experiment 4 with RMSprop.



(a) Evolving losses throughout the training process for Experiment 4 with SGD.

(b) Spectrogram generated in Experiment 4 with SGD.

Figure 5.5: Results of Experiment 4 with SGD.

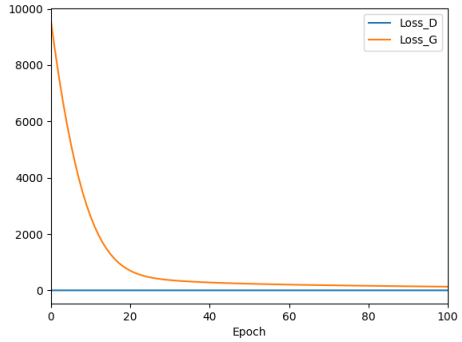
in the model. Initially, dropout layers were added, which randomly deactivate a percentage of neurons during training. This technique strengthens the model's ability to generalize. Batch normalization was included to normalize layer activations and decrease internal covariate shifts, speeding up training.

Additionally, the author introduced noise to the discriminator input to enhance model robustness, making it less sensitive to small perturbations and improving overall stability. Elastic network regularization, which combines both L1 and L2 penalties, was also utilized [137].

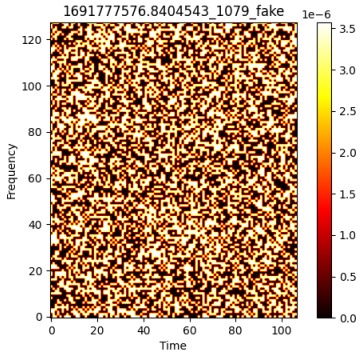
It should be noted that the experimental setup was identical to Experiment 4, except for the use of the Adam optimizer. The total loss for this experiment was 4.791, calculated by summing the generator loss of 3.362 and the discriminator loss of 1.428. However, it is important to note that comparing the current generator loss to previous results is not appropriate due to the incorporation of the elastic net regularization. This regularization method substantially elevates the generator loss but is crucial for training goals.

Regrettably, a flaw in the elastic net implementation was detected in the code, but not until after a few subsequent experiments. Although this error has a negative impact on the overall absolute loss and results, it does not affect the comparisons between experiments.

Refer to Figure 5.6 for a visual representation of the results, displaying the loss and final spectrogram generated in this experiment.



(a) Evolving losses throughout the training process for Experiment 5.



(b) Spectrogram generated in Experiment 5.

Figure 5.6: Results of Experiment 5.

5.2.6 Experiment 6: Optimizing Model Balance

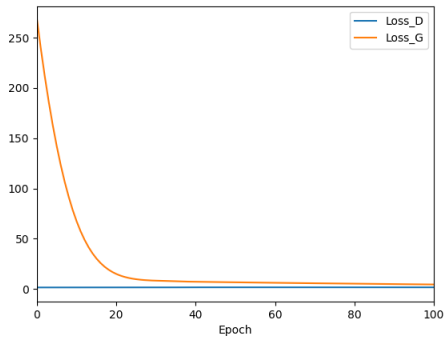
In this experiment, the author aimed to leverage the upgraded hardware configuration of the LIACC 2 system. After updating the model to include the Clotho dataset, which demanded more significant GPU and RAM resources, the results were analyzed.

Nevertheless, the author found them unsatisfactory, possibly due to the imbalance of the generator and discriminator models, with the former having about 25 million parameters and the latter having only about 2 million. Typically, GAN models are designed as a pair of similar or mirrored models.

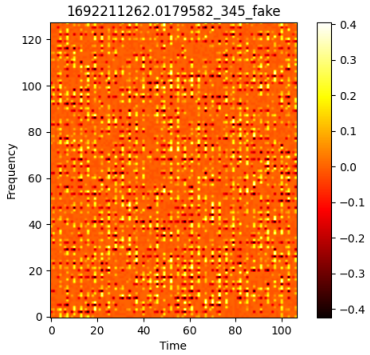
To mitigate this problem, the author reversed the model difference that was introduced in Experiment 3. The generator and discriminator models were both adjusted to have around 2 million parameters each.

The loss function, regularization techniques, and optimizer remained unchanged. The total loss for this experiment was 2.045, calculated by adding the generator loss of 0.587 and the discriminator loss of 1.458.

For a visual representation of the results, which includes the loss and the final spectrogram generated in this experiment, please see Figure 5.7.



(a) Evolving losses throughout the training process for Experiment 6.



(b) Spectrogram generated in Experiment 6.

Figure 5.7: Results of Experiment 6.

5.2.7 Experiment 7: Scaling Complexity

Building upon the success of the previous experiment, the purpose of this study was to replicate the same approach utilizing larger models. This was achieved by amplifying the number of convolutional layers in each block and the number of filters in each convolutional layer.

This experiment tested two sets of models. The first set comprised of models, each with 10 million parameters, resulting in a total of 20 million parameters. The second set includes models with a total of 50 million parameters, each having 25 million parameters.

The other configuration aspects, such as the loss function, regularization techniques, data set, optimizer, and hardware configuration, remained the same.

In the initial test, the generator loss of 1.132 and the discriminator loss of 1.389 resulted in a total loss of 2.512.

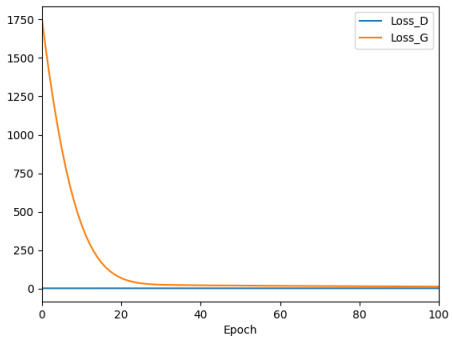
For a visual representation of the experiment results, including the loss and final spectrogram, please see Figure 5.8.

The second experiment resulted in a total loss of 3.177, comprising a generator loss of 1.738 and a discriminator loss of 1.439.

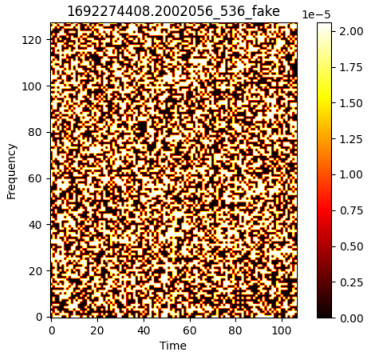
To view the experiment results, including the loss and final spectrogram, refer to Figure 5.9.

5.2.8 Experiment 8: Unregularized and Removal of Elastic Net

In Experiment 5, a bug was identified and the author had to eliminate the elastic net implementation, which necessitated a complete rewrite of the models and training loop. Due to this, no regularization was enforced in this experiment.

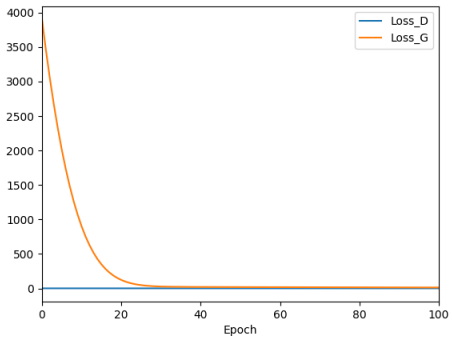


(a) Evolving losses throughout the training process for Experiment 7 with 20 million parameters.

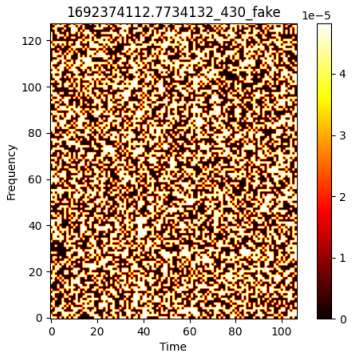


(b) Spectrogram generated in Experiment 7 with 20 million parameters.

Figure 5.8: Results of Experiment 7 with 20 million parameters.



(a) Evolving losses throughout the training process for Experiment 7 with 50 million parameters.



(b) Spectrogram generated in Experiment 7 with 50 million parameters.

Figure 5.9: Results of Experiment 7 with 50 million parameters.

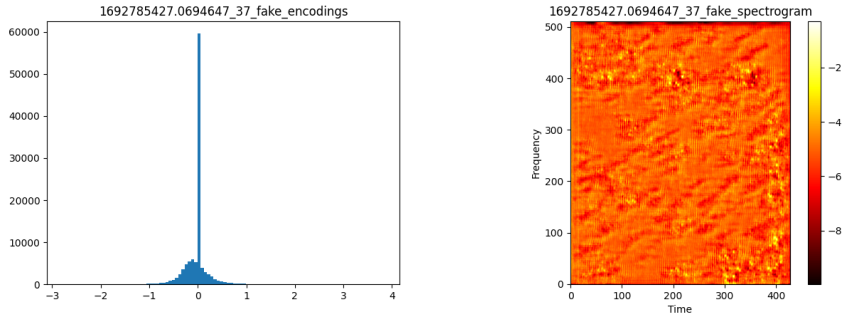
However, aside from the absence of regularization and elastic net, the experimental setup is identical to the larger test performed in Experiment 7, which featured a model containing roughly 50 million parameters.

In epoch 37, the experiment incurred a total loss of 1.832, consisting of the generator loss of 0.693 and the discriminator loss of 1.139.

Training of the model resulted in a sudden collapse at epoch 37, with inexplicable loss values plummeting to as low as 0 thereafter.

To gain insight into the training progress, two spectrograms were generated: one for epoch 37 (see Figure 5.10) and another for epoch 49, which marked the end of training (see Figure 5.11).

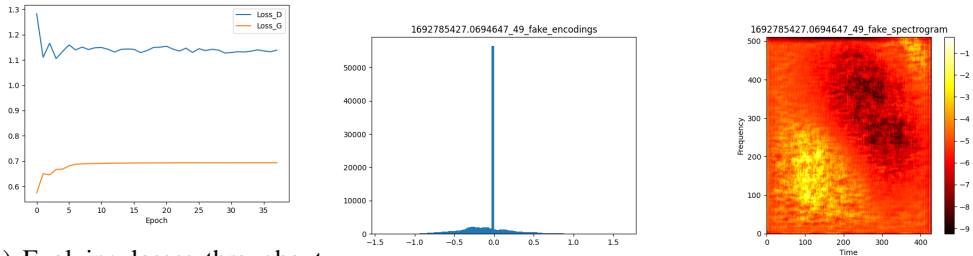
Additionally, a histogram depicting the values present in the generated encodings has been included in this experiment.



(a) Histogram of the generated embeddings for Experiment 8 in epoch 37.

(b) Spectrogram generated in Experiment 8 in epoch 37.

Figure 5.10: Results of Experiment 8 in epoch 37.



(a) Evolving losses throughout the training process for Experiment 8.

(b) Histogram of the generated embeddings for Experiment 8.

(c) Spectrogram generated in Experiment 8.

Figure 5.11: Results of Experiment 8 at the end of training.

5.2.9 Experiment 9: Regularization Techniques and Training Progress for Rebuilt Model

In this study, the author employed regularization techniques while holding all other parameters constant to build upon the previous experiment.

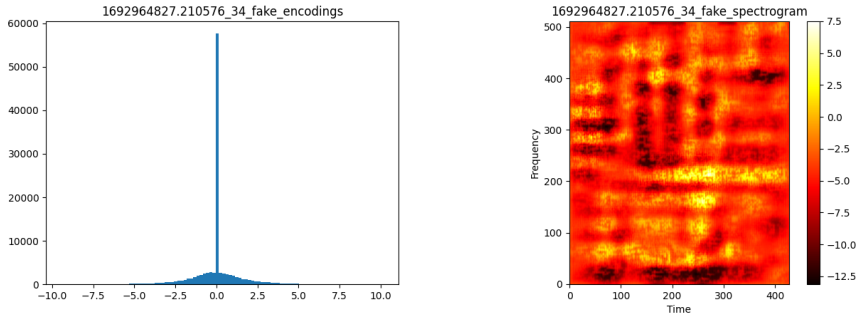
Nonetheless, an unforeseen crash transpired at epoch 34, resulting in a substantial reduction in loss values that ultimately converged to zero. It is crucial to note that all presented values are objective and exclusively quantitative.

The experiment incurred a total loss of 1.700, with 0.693 pertaining to the generator loss and 1.007 to the discriminator loss.

To improve comprehension of the training progress, two spectrograms were produced: one for epoch 34 (refer to Figure 5.12) and the other for the final epoch, 49 (refer to Figure 5.13).

5.2.10 Experiment 10

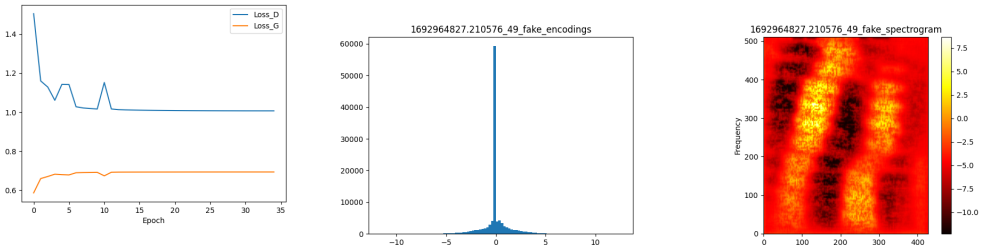
The GANmix model is made possible by the use of a pre-trained VAE. In other words, the GANmix generator tries to generate encodings that are similar to those produced by the VAE. The VAE



(a) Histogram of the generated embeddings for Experiment 9 in epoch 34.

(b) Spectrogram generated in Experiment 9 in epoch 34.

Figure 5.12: Results of Experiment 9 in epoch 34.



(a) Evolving losses throughout the training process for Experiment 9.

(b) Histogram of the generated embeddings for Experiment 9.

(c) Spectrogram generated in Experiment 9.

Figure 5.13: Results of Experiment 9 at the end of training.

uses CNNs, which explains why the GAN models also use encodings.

However, what the GANmix model actually produces are embeddings that have no significant spatial relationship.

This experiment builds on the notion that fully connected neural networks may provide better results.

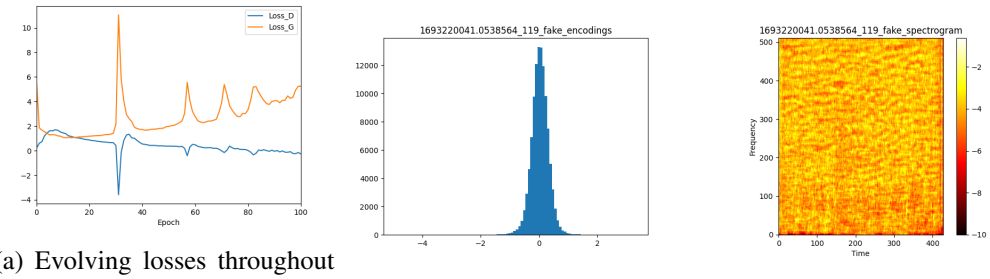
To achieve this, the models were modified so that the generator has a connection (fully connected linear layer) from the input to the hidden layer, and another connection from the hidden layer to the output (the embeddings). The discriminator follows a similar model architecture.

Other configuration aspects such as loss function, regularization techniques, data set, optimizer, and hardware configuration remained unchanged.

The experiment incurred a total loss of 6.425, with $6 - 987$ pertaining to the generator loss and -0.562 to the discriminator loss.

For a visual representation of the results, which includes the loss, the histogram of the generated embeddings and the final spectrogram generated in this experiment, please see Figure 5.14.

The data shows a curious pattern: the discriminator loss steadily decreases as the generator loss continuously increases. This trend could be due to insufficient epochs, problems with the optimizer, or the lack of a suitable scheduler. However, based on previous experiments, the criterion seems unlikely to be the primary cause.



(a) Evolving losses throughout the training process for Experiment 10.

(b) Histogram of the generated embeddings for Experiment 10.

(c) Spectrogram generated in Experiment 10.

Figure 5.14: Results of Experiment 10.

5.3 Analysis and Interpretation

This section analyzes and assesses the results of experiments conducted on GANmix. The evaluation identifies trends, investigates potential future directions for research, and interprets the results in the context of the research goals and the broader field.

5.3.1 Identifying Trends

Examination of the experimental results reveals several patterns and trends. There is an inverse correlation between generator and discriminator losses; as one increases, the other decreases. However, in certain cases both losses decrease together, which is beneficial. Also, convergence tends to plateau after a certain number of epochs.

This can be seen in Figure 5.1a. There, it can be seen that the loss of the generator decreases as the loss of the discriminator increases, and then the opposite occurs. Eventually both losses level off and decrease slightly.

The study found that the speed of convergence was affected by the learning rate. Although higher learning rates led to a faster convergence plateau, they did not necessarily lead to better results. This was demonstrated in Experiment 2 (Section 5.2.2), where the learning rates were increased and a loss plot is observed that is similar to Experiment 1 (5.2.1), but with a significantly faster plateau.

In Experiment 4 (Section 5.2.4), it appears that SGD initially outperforms RMSprop, although it learns at a significantly slower rate. The results seem to be similar to Adam. Given the limited

training time, one can only make assumptions, but it seems that using SGD as the optimization algorithm leads to better performance than alternative methods such as RMSprop and Adam, despite the slower convergence.

Regularization methods such as dropout, batch normalization, and Gaussian noise have been shown to improve results and extend convergence, as shown in Experiment 5 (5.2.5). Although the spectrogram was suboptimal, the loss curves showed a healthy trend, with both losses decreasing consistently over time. Although the application of the elastic net regularization presented some challenges, it showed promise. The model performed better overall and achieved faster convergence when the generator and discriminator had similar parameter sets.

It was determined that larger models resulted in more rapid and resilient convergence. However, it should be noted that achieving satisfactory results was critically impacted by the size of the dataset. Typically, larger datasets and models produced better outcomes.

The embeddings generated by AudioLDM's VAE show similarities characterized by a normal-like distribution with a significantly low standard deviation. Figure 5.15 shows a histogram of these values, which extend to hundreds on the x-axis due to the residuals. It can be seen that the distribution is predominantly concentrated in an area close to zero.

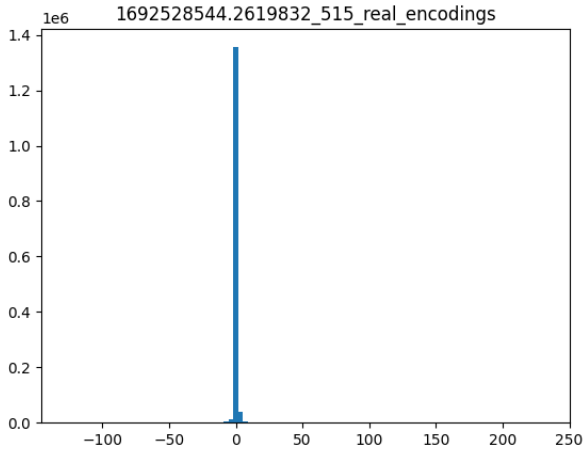


Figure 5.15: An histogram that represents the latent values created by AudioLDM's VAE.

Experience 10 (5.2.10) showed an interesting trend. However, it is worth noting that although the generated spectrograms showed improvement, the generator loss increased while the discriminator loss consistently decreased after a few epochs, as can be seen in Figure 5.14a. The use of linear layers instead of convolutions proved to be advantageous in generating more robust spectrograms. Further investigation is needed to determine if there are problems with the optimizer.

This last experiment, which used linear layers, produced the most encouraging results and was chosen as the primary study for this dissertation.

5.3.2 Results for Future Investigation

In some experiments, a decline in performance after a few epochs was observed, resulting in losses becoming not a number (NaN). This is seen in Experiments 8 and 9 (Sections 5.2.8, 5.2.9). Further research is required to determine the underlying cause and prevent this occurrence in future experiments.

The impact of elastic network regularization on model performance requires further investigation. Despite the implementation challenges encountered, the initial results suggest that this regularization method has improved and shows potential effectiveness.

In addition, the issue of the continuously increasing generator loss in the last experiment requires further investigation in the future.

5.3.3 Interpretation of Results

The study's findings did not meet the expectations set forth in the thesis as the generated sounds are not state-of-the-art for generative models; however, they are encouraging. With sufficient data and time, the model has the potential to generate high-quality sound. This suggests that generative AI models, especially GANs, have significant capabilities in audio generation.

Furthermore, exploring the model's latent space is seen as a promising strategy for achieving better results. The latent space method is a simpler approach that can potentially yield more favorable outcomes for future models.

The limitations of the small datasets used in this analysis are evident. The bigger one, Clotho, comprised sounds lasting from 15 to 20 seconds, but with 5 seconds removed from each sound, certain sounds were too short to produce satisfactory results. Additionally, the total sound count in the data set was less than 5,000, which isn't enough to properly train a generative model.

Thus, a significant concern uncovered in this study is the lack of an appropriate dataset. To achieve more precise results, it is critical to have access to comprehensive datasets. Nevertheless, it is imperative to recognize that the process of training models on large data sets involves significant computational resources that may not be readily available.

5.3.4 Conclusion

In summary, the analysis and interpretation of the data show trends and patterns observed in the experiments. The results did not meet initial expectations; however, they show potential for future advances in generative AI models for audio synthesis. It is important to note that comparisons with state-of-the-art audio generation models are not discussed in this study due to the unsatisfactory practical results obtained.

5.4 Constraints and Challenges

This section discusses the major constraints and challenges faced in developing the proposed solution. It also describes the strategies and solutions adopted or proposed to address these issues, and the potential impacts, tradeoffs, and opportunities that result.

5.4.1 Hardware Resources

One of the biggest challenges was the scarcity of hardware resources for training and evaluating these models. Generative models require large amounts of computing power and memory to process high-dimensional data and learn complex patterns. However, such resources are often limited or expensive to access, especially for individual researchers or small teams. This is a significant barrier to achieving state-of-the-art results in audio synthesis, as few companies and labs have the necessary hardware capabilities.

To overcome this challenge, several strategies have been employed to optimize the use of available hardware resources. First, the neurons of the models were parallelized across the available GPUs (two in the final configuration — LIACC 2). This allowed to distribute the workload and speed up the training process. Second, checkpoints were implemented to save and load the model state at each epoch. This allowed the training to be resumed from where it was stopped in case of interruptions or failures. Third, the audio files were dynamically read and translated into spectrograms during training. This reduced memory consumption and disk space requirements. Fourth, gradient scaling and autocast (mixed precision training) techniques were used. These techniques involve performing some computationally expensive operations in 16-bit, while performing other numerically sensitive operations, such as accumulations, in 32-bit. This improved the performance and accuracy of the models while reducing memory consumption.

By applying these strategies, the models were trained and evaluated more efficiently and effectively. However, it is also recognized that these strategies have some limitations and trade-offs. For example, parallelizing neurons across GPUs can introduce communication overhead and synchronization issues. Checkpoints may not capture the full state of the model or optimizer. Dynamic data processing can increase pipeline latency and complexity. Mixed-precision training can introduce numerical errors or instability.

It is important to note that the hardware configuration with some computational power (LIACC 2) was only available about a month before the submission of this thesis, so most of the work was done with really scarce resources. This means that the results presented in this thesis may not reflect the full potential or optimal performance of the proposed solution, as more experiments and improvements could be done with more hardware resources.

5.4.2 Data Quality and Quantity

Another challenge was the quality and quantity of available data. Generative models require a large amount of high-quality and diverse data to learn the underlying patterns and distributions of the data domain. However, such data is often scarce or difficult to obtain, especially for audio synthesis from textual input. Existing datasets for this task are either too small, focused on a specific domain, or lack descriptive labels. This limits the generalization and robustness of the models, as they may overfit to the training data or fail to capture the variability and richness of natural language and sound.

To mitigate this challenge, some data augmentation techniques were applied to increase the size and diversity of the data. For example, random cropping was used to generate different segments of audio from the same file. This increased the number of samples and introduced some variation in the data. However, it is also recognized that these techniques are not sufficient to solve the problem of data quality and quantity. Data augmentation may not produce realistic or novel samples but may introduce noise or artifacts into the data.

5.4.3 Hyperparameter Tuning

A final challenge was the time required for the hyperparameter tuning of the generative models. Hyperparameters are parameters that are not learned by the model, but are set by the user prior to training. They include learning rate, batch size, number of layers, number of neurons, activation functions, regularization methods, etc. Hyperparameters significantly impact the performance and behavior of the model, as they determine how the model learns from the data and adapts to different situations. However, finding the optimal values for these hyperparameters is often a tedious and time-consuming process involving trial-and-error experiments with different combinations of values.

Due to time constraints and deadlines, there was insufficient time to fine-tune our hyperparameters for these generative models. All models presented in this thesis are vanilla versions with default or arbitrary values for their hyperparameters. This means they may not reach their full potential or optimal performance in audio synthesis from textual input. Therefore, it is suggested that future work should devote more time and effort to the hyperparameter tuning of our generative models using methods such as grid search, random search, Bayesian optimization, etc.

5.5 Conclusion

In this section, the author discusses the major limitations and challenges they faced in developing the proposed solution. The author has also described how they addressed these issues and the possible implications, trade-offs, and opportunities that arose. Despite these challenges and limitations, the author believes that the proposed solution has several strengths and advantages that advance the state of the art in generative AI models for audio synthesis.

Chapter 6

Conclusions

Contents

The field of DL and audio has witnessed the emergence of generative AI models for audio synthesis, a challenging and fascinating research topic. This thesis has investigated the evolution and evaluation of these models, with the main objectives of understanding the current state of the art and creating a novel model using a generative approach. To achieve these goals, this thesis has followed a comprehensive methodology that included conducting a literature review and developing and evaluating a generative AI model for audio synthesis.

This chapter concludes the thesis by presenting the main results and contributions of the research. It also reflects on the research process and methodology, and suggests future directions for further research.

The first section, Section 6.1, of this chapter provides an overview of the research goals that guided the investigation into the development and evaluation of generative AI models for audio synthesis. Section 6.2 reflects on the research process and methodology employed throughout the research, highlighting the strengths and limitations of the chosen approach, and discussing the challenges and lessons learned during the development of the AI models. The third section, 6.3, outlines potential areas of future work that can help advance the field of generative AI models for audio synthesis, addressing some of the open questions and limitations identified in the research.

6.1 Overview of Research Goals

This section presents an overview of the research goals that directed the examination into the evolution and evaluation of generative AI models for audio synthesis.

The two research goals can be summarized as follows: to understand the current state of the art in generative DL and audio, and to create one of these models.

6.1.1 Comprehensive Study of State-of-the-Art Deep Learning Architectures and Models for Audio Synthesis

Throughout the thesis, a comprehensive study of the current state-of-the-art deep learning architectures for audio synthesis was conducted, including GANs, VAEs, diffusion models, and other related architectures. This study included an in-depth analysis of the strengths, limitations, and potential applications of these architectures in the context of audio synthesis.

In addition, various models that use these deep learning architectures were examined, such as VALL-E, AudioGen, AudioLM, and others. Each model was thoroughly analyzed to understand

their unique approaches, techniques, and contributions to audio synthesis. The results of this study provided valuable insights into the different models and their effectiveness in producing high-quality audio.

Extensive research was also conducted to examine previous algorithms used for sound processing, including techniques for augmentation, feature extraction, and other purposes. This research involved a thorough review of existing algorithms and their applicability to audio generative models. The insights gained from this research were used in the development of practical systems and contributed to the overall understanding of sound processing techniques in the context of generative AI models.

This work culminated in a thorough state-of-the-art chapter (Chapter 2), which provides a comprehensive overview of current advances in deep learning architectures and models for audio synthesis. The chapter presents a detailed analysis of the studied architectures and models, highlighting their strengths, limitations, and potential applications.

In addition, the findings and insights from this research have been summarized in a review paper that has been submitted to a prestigious journal for peer review. This review paper aims to provide a comprehensive and up-to-date overview of the state of the art in deep learning architectures and models for audio synthesis. It is currently awaiting approval and publication in the journal.

6.1.2 Developing End-to-End Systems for Sound Synthesis and Evaluation

The goal of developing end-to-end systems for sound synthesis from text input has been partially achieved. While initial prototypes have been created, the results have not been satisfactory due to limitations in the available datasets and the lack of hyperparameter fine-tuning in the models. However, the potential for improvement is significant, and future work proposed in the conclusion suggests new models that could yield better results.

The goal of evaluating the ability of systems to generate sound from textual input remains an ongoing challenge. Finding and testing robust evaluation functions is a complex task that requires further research and dedicated effort. Due to time constraints, this objective has not been fully completed, and only a comparison using the loss function of GAN has been performed.

In summary, the research objectives outlined in this dissertation have been addressed to varying degrees. A comprehensive study of DL architectures and prior sound processing algorithms has been conducted. The development of end-to-end systems for sound synthesis from text input has shown promising progress, while the evaluation of their accuracy remains an ongoing challenge. These achievements contribute to the existing knowledge and understanding of audio generative models and pave the way for future research and development in this area.

6.2 Reflection on the Research Process

Throughout this research, a comprehensive methodology was employed to ensure the successful achievement of the research objectives. The chosen methodology is presented in Section 4.1 and involved conducting a thorough review of the state of the art while simultaneously initiating the writing process. This iterative and agile approach allowed for continuous refinement of ideas and incorporation of the latest developments in the field.

Reflecting on the effectiveness of the chosen methodology, it can be concluded that it successfully guided the research process. The methodology provided a structured framework for conducting the research and ensured that the objectives were addressed in a systematic and efficient manner. The challenges encountered during the research process were not related to the methodology itself, but rather to the complexity of the chosen research topic.

One challenge encountered during the research process was the sometimes tedious and frustrating nature of developing AI models. Troubleshooting problems during training required additional time and effort, often requiring extended training periods to evaluate potential solutions. However, these challenges provided valuable lessons in patience, problem solving, and the importance of careful experimentation.

6.3 Future Directions

The study and development of generative AI models for audio synthesis have shown promising results in producing realistic and diverse audio output. However, there are still several avenues for further exploration and improvement. This section outlines potential areas of future work that can help advance the field of generative AI models for audio synthesis.

6.3.1 Exploring Novel Architectures

The objective of this section is to recommend new frameworks that expand upon the groundwork established in this thesis. Developing these suggested frameworks is deferred to future research due to limited time and resources.

The proposed theoretical methods presented in this Section are motivated by the necessity of increasing the quality, diversity, and efficacy of the produced audio.

This section outlines the objectives, design principles, and possible applications of each suggested architecture. Furthermore, it discusses the methodological considerations and potential obstacles that may arise during their development and implementation.

It is essential to note that the proposed theoretical approaches presented here are intended to serve as a basis for future research. Researchers and practitioners are encouraged to explore, refine, and contribute to the advancement of generative AI models for audio synthesis.

This section provides detailed explanations of each proposed architecture, including insights into its design principles, implementation considerations, and potential applications. This approach aims to stimulate further exploration and innovation in the field.

6.3.1.1 VAMOS - Variable Audio Model for Sound Synthesis

In the field of audio and sound processing, VAMOS (Variable Audio Model for Sound Synthesis) is a novel implementation inspired by the well-known DALL-E 2 framework (see Section 2.1.5.5). While DALL-E 2's lies in its ability to generate images from textual descriptions, VAMOS extends this concept to the auditory domain, generating audio outputs based on corresponding textual inputs. This section describes the architecture, components, and underlying mechanisms that make up the VAMOS model.

Indeed, the VAMOS model goes beyond mere conceptualization, as its development was diligently initiated from the ground up. The foundation of this innovative model has been established through craftsmanship, with each component designed. The VAMOS architecture is available in open-source. Appendix G shows and explains the significant code portions.

Model Composition VAMOS is comprised of four distinct yet interconnected models, each contributing to a comprehensive audio generation process. These models — CLAP, Text Encoder, Audio Encoder (ResNet), and VamGen — work together to synthesize audio content that aligns with given textual cues. It is crucial to note that these models were thoughtfully constructed with functions designed for training and inference, without requiring the time-intensive process of fine-tuning.

CLAP The foundation of VAMOS's cross-modal capability is CLAP. Similar to OpenAI's CLIP [99], but specifically designed for audio, CLAP brings text and audio together by projecting them onto a common dimensional space. In practice, CLAP uses independent feature extraction techniques for audio and text inputs, enabling flexibility for differing sources and types of information. These extraction methods, whether state-of-the-art or custom-built, produce latent feature vectors that allow for comparability between textual and auditory data.

Linear layers connecting audio and text features to a predefined embedding space are central to the operation of CLAP. This embedding space is achieved through distinct linear transformations for audio and text inputs. The fundamental principle of CLAP is to align the output of these linear layers for corresponding text and audio inputs. This alignment is accomplished by pairwise similarity calculations, which serve as the foundation for CLAP's loss function. The loss function guides the convergence of both audio and text features.

Text Encoder: Unveiling Semantic Context The Text Encoder plays a crucial role in translating textual inputs into semantically-rich representations in VAMOS. Complementary to CLAP, it

harnesses BERT's transformative capabilities [28], a language model renowned for its contextual understanding.

To accomplish this, the Text Encoder follows a multi-step process guided by the principles of BERT. It begins by breaking down the input text, dividing the words and phrases into tokens that are then mapped onto a sequence of input IDs. Through the encoder transformer, the tokenized sequence is converted into an encoding that captures the deep semantic context embedded in the text.

During the development of VAMOS's Text Encoder, a deliberate decision was made to adopt a pre-existing model rather than create a new one tailored for this specific project. This decision was influenced by the specialized nature of the Text Encoder, which exclusively concentrates on processing textual input. Since the central focus of this thesis focuses on audio, adopting a verified text encoding model facilitated a more efficient development process, allocating resources to the innovative challenges presented by audio synthesis.

It is important to mention that the decision to utilize BERT is motivated by its dual quality of being a cutting-edge language processing model as well as an open-source tool. The HuggingFace library allows for BERT's capabilities to be integrated into VAMOS's architecture.

Audio Encoder: ResNet-based Auditory Embedding To address the auditory aspect of VAMOS, the Audio Encoder utilizes the ResNet architecture [51] to generate informative embeddings from audio inputs. Unlike the Text Encoder, which depends on pre-existing implementations, the Audio Encoder is a customized solution designed specifically for audio-related tasks. A defining feature of the ResNet design is the integration of residual connections, otherwise known as ResBlocks, which enhances the network's capability to process complex audio features. These ResBlocks neatly divide the layers, permitting specific computations and bolstering the network's flexibility.

The fundamental principle of the Audio Encoder rests on leveraging residual connections to optimize audio feature extraction. By selectively including or excluding layers, the network achieves a flexible architecture. This allows it to capture both complex and simple audio features. Although the ResNet architecture is complex, the implementation presented in this thesis is customizable, giving users the ability to create ResBlocks specific to their use cases.

VamGen: Auditory Diffusion At the top of the VAMOS architecture is VamGen, a model rooted in DALL-E 2 but adapted for audio synthesis. VamGen exploits the potential of text-audio alignment to generate auditory output from textual prompts.

A key tenet of VamGen is a diffusion decoding process (see Section 2.1.2.3). The textual input is encoded into latent features, which undergo a diffusion process to transform them into audio latent features. These audio latent features serve as the basis for the subsequent generation of spectrograms, simpler representations of audio suitable for further processing.

Generative Stack Components VamGen's generative stack comprises two key components: the prior and the decoder. The prior generates image embeddings based on captions, thereby aligning textual cues with image-like representations. The decoder utilizes these image embeddings and captions to produce spectrogram outputs. The decoder, modeled as a diffusion process, provides a dynamic framework in which audio embeddings transform into coherent spectrograms gradually.

The diffusion-based approach to generating audio embeddings emphasizes the stochastic nature of the process, allowing for creative freedom within a structured framework.

Implementation Though time constraints limited the realization of the full VamGen model, significant progress was made in its development. The foundation of VamGen lies in the U-net architecture, custom-built to align with the diffusion process.

In essence, the U-net architecture, celebrated for its skill in image segmentation tasks (see Section 2.1.2.1), inherently facilitates the diffusion process - a valuable feature that aligns effortlessly with VamGen's aim to synthesize audio from text input.

It consists of two interrelated parts: an encoder and a decoder. The encoder component skillfully converts raw audio inputs into intermediate representations. Simultaneously, the decoder resamples these intermediate representations into a format that resembles the original one.

The U-net integrates seamlessly into the diffusion process thanks to its innate ability to maintain the input's original dimensions. This attribute is critical in preserving the fidelity of audio representations during the diffusion process. As a result, maintaining these dimensions ensures a faithful reconstruction of audio content, preserving the essence of the original input. This preservation and subsequent fusion with the new dimensions introduced by VamGen's diffusion process come together to shape an audio synthesis narrative that resembles segmentation - a clear distinction between noise and authentic input.

The U-net architecture has been fully developed and tested. In addition to the public repository, its code is available in the annex G.

Concluding Remarks It is important to acknowledge that the VAMOS model presented here is a prototype, designed to serve as a foundation for future advancements. Although these models have not undergone training and some elements remain incomplete, the investigation of cross-modal alignment and audio synthesis highlights the possibility of merging textual and auditory domains.

In summary, VAMOS represents a significant advancement in harnessing the creative synergy between text and audio, fostering a rich auditory experience through the intersection of innovative models and cross-modal alignment.

6.3.1.2 Fine-Tune Stable Diffusion for Spectrograms

A promising avenue of research in the DL community is the fine-tuning of existing generative models to improve their capabilities and produce high-quality audio output. This section introduces the concept of fine-tuning generative models for audio synthesis, and proposes fine-tuning the stable diffusion model to generate spectrograms.

The stable diffusion model uses the principles of diffusion processes to generate high-fidelity and diverse image samples (see Section 2.1.5.3). One of the key advantages of the stable diffusion model is its ability to capture complex dependencies and generate realistic output.

It is worth noting that the stable diffusion model is an open source model, meaning that its code and implementation details are publicly available. This accessibility allows researchers and practitioners to study, modify, and build upon the foundations of the model. The availability of open source code for the stable diffusion model facilitates its fine-tuning for specific tasks, such as audio synthesis.

The fine-tuning of the stable diffusion model to spectrograms provides an exciting opportunity to explore the generation of high-quality audio output based on this visual representation.

One of the key advantages of the stable diffusion model is its ability to capture complex dependencies and generate realistic images. By fine-tuning the stable diffusion model on spectrograms, one can leverage its prior knowledge and adapt it to the specific characteristics of audio signals represented in the frequency and time domains.

The potential benefits of using the prior knowledge of the stable diffusion model for audio synthesis are manifold. First, the stable diffusion model has already shown impressive results in other domains, in this case image synthesis. By building on this foundation, one can exploit the model's ability to generate high-fidelity and diverse outputs, which can greatly improve the quality of the synthesized audio.

Furthermore, fine-tuning the stable diffusion model to spectrograms can provide a unique perspective on audio synthesis. By treating spectrograms as images and applying the stable diffusion model, one can explore the potential of generating audio based on this visual representation. This approach opens up new possibilities for manipulating and synthesizing audio in innovative ways.

To prove the usefulness of this method, Riffusion (see Section 2.2.4.2), which already performs a similar task of generating audio from visual representations, has shown considerable results. By considering the insights and techniques used in Riffusion, one can build on its foundations and adapt the stable diffusion model accordingly.

It is important to note that the success of the proposed approach depends on the availability of a sufficiently large dataset for fine-tuning.

6.3.1.3 Theoretical General Audio Transformer

Transformers have become a prevalent architecture for several sequence modeling tasks in NLP [49]. Their success has also expanded to the generative modeling of mediums such as images and videos. This section proposes adapting the transformer architecture for generative modeling and audio waveform synthesis.

A novel encoder-decoder transformer specifically designed for audio generation is introduced, referred to as the Audio Transformer (AT). The AT incorporates convolutional layers and custom attention mechanisms tailored to handle audio data.

The primary motivation is to enable high-quality, flexible audio generation for various applications. While transformers have proven effective in capturing long-range dependencies in sequences, the proposed AT aims to optimize these models specifically for audio data.

The use of a transformer architecture for audio generation, propelled by textual prompts, is based on the encoder-decoder framework of the transformer, modified for audio synthesis. This configuration permits the effortless incorporation of textual data to aid in the synthesis of consistent and context-appropriate audio waveforms.

The process for generating audio using transformers includes two components: an encoder and a decoder. Each is composed of stacked layers that work together to process the textual input and create the corresponding audio output. The encoder's main function is to map the textual input to continuous representations that capture semantic information. For a deeper understanding on the encoder's functionality, as well as on transformers, in general, please refer to Section 2.1.2.3.

Upon receiving the continuous representations from the encoder, the decoder begins to generate the audio waveform. The decoder's functioning is dependent on the encoder's representations, which guarantees that the synthesized audio is aligned with the intended meaning of the input text.

The synthesis process unfolds sequentially, with the decoder generating the output audio waveform sample-by-sample. The incorporation of multi-head attention mechanisms at the local and global levels enables the model to generate a wide range of natural-sounding audio outputs that correspond with the semantic context of the input.

The hyperparameters such as model dimensions, number of layers, attention heads, and hidden sizes can be tuned to balance performance and computational constraints. In summary, this architecture leverages the strengths of transformers for sequence modeling while optimizing the components for conditional audio generation.

Sound Representation Transformers have demonstrated power in autoregressive sequence modeling and generation. However, directly applying them to raw audio waveforms poses challenges due to the high sampling rates and lack of inherent discretization. As a potential alternative, spectrograms offer a 2D time-frequency audio representation (see section 2.1.1). This section analyzes

the trade-offs between raw audio versus spectrograms as inputs to transformer-based audio generation models.

Challenges of Using Spectrograms Spectrograms explicitly encode frequency information, providing interpretable intermediate representations. However, transformers generate outputs one step at a time, and the notion of "next" becomes ambiguous on 2D spectrograms, as there are multiple values for each moment.

Benefits of Raw Audio Raw audio waveforms allow direct modeling of the time-domain signal to be generated. The sequential structure matches the autoregressive nature of transformers without modification. Transformers can learn to model dependencies in the continuous waveform samples directly. Raw audio represents the most natural fit for a sequential generation. Parameterizing waveform samples also avoids imposing and inverting a fixed spectrogram transformation that may discard information. Raw audio can capture nuances and exhibit fidelity beyond what prescribed spectrogram representations encode.

Latent Feature Translation To fix these problems and bridge the gap between spectrograms and transformers, a novel approach is introduced. Instead of directly using the 2D spectrogram as input, a neural network is employed to translate each column of the spectrogram into a single continuous latent feature. This neural network, known from now on as the Latent Feature Translator, is shared across all columns and is conditioned on the corresponding timestamp of the column. The resulting continuous latent features capture the essential characteristics of the audio content while simplifying the representation for the subsequent transformer-based modeling.

This translation process effectively compresses the complex spectral information into a more manageable and informative format that still maintains temporal information, facilitating the transformer's ability to capture dependencies and generate coherent audio output.

This innovative approach leverages the strengths of both spectrogram representations and transformers, enabling efficient and effective generative audio modeling.

Training The AT is first pretrained in an unsupervised manner to learn effective representations of audio data. During unsupervised pretraining, the model input is a segment of continuous latent features, and the target output is the next latent feature.

The model trains on audio-only data to predict the next latent feature. This relies on no text conditioning or labels. The goal is to learn generalized audio representations that capture dependencies across long sequences.

During training, audio clips are split into fixed-length chunks, 1-10 seconds. Batching multiple chunks together allows more efficient GPU processing. Given all previous samples, the model is trained to predict the next latent feature at each step.

An error loss between the predicted and target audio samples is calculated. Over many training iterations, the model parameters are updated to minimize the loss function. Additional regularization techniques like dropout are used to prevent overfitting. The learning rate is gradually decayed for training as the loss function converges.

Validation audio clips not used in training monitor overfitting and determine early stopping. The model with the lowest validation loss is selected.

During supervised training, text conditioning is added to the pre-trained model. The loss function now optimizes conditional generation quality given input text.

This leverages the representations learned during unsupervised pretraining. Text conditioning trains the model to generate relevant audio for textual descriptions.

Pretraining provides an effective regularization technique to prime the model and prevent overfitting the paired text-audio data. This semi-supervised approach with unsupervised pretraining enables learning robust generative audio models.

Why Transformers Transformers are a natural choice for generative audio modeling compared to alternatives like GANs or diffusion models (see Sections 2.1.2.3, 2.1.2.3) due to their strength in sequential modeling. Audio signals inherently exhibit strong temporal consistency and context.

Transformers leverage a self-attention mechanism to model long-range dependencies in sequences effectively. This allows transformers to capture structures spanning longer time scales than recurrent models like LSTMs (see Section 2.1.2.1). The global receptive field enables coherently modeling whole utterances, phrases, and even entire passages.

Additionally, as seen in Section 2.1.2.3, transformers have demonstrated state-of-the-art performance across various sequence transduction tasks in language, speech, and other domains. Leveraging their proven modeling capabilities for a new modality in audio is a natural progression.

The large model capacity of transformers is also advantageous, allowing sufficient expressivity to represent the complexity and nuance of audio data. With solid scalability to leverage large datasets through efficient parallel training, transformers are uniquely positioned as cutting-edge architecture for generative audio.

Limitations and Future Work While the AT shows promise for generative audio modeling, there are several limitations and areas for improvement through future work.

First, the model requires large, diverse datasets of high-quality audio examples to train effectively. Collecting such datasets can be challenging, particularly for specialized domains like musical composition.

Data efficiency could be improved through transfer learning and unsupervised pretraining approaches. Leveraging models pretrained on other transformer tasks might provide valuable initializations for audio generation.

Training complex transformer models can also incur high computational costs and time requirements. Optimization for faster training and inference will be necessary for practical deployment. Architectural modifications to reduce model size should be explored.

Rigorously evaluating generated audio samples' coherence, naturalness, and creativity remains difficult. Developing quantitative and qualitative evaluation protocols to measure these attributes is an open research question.

There are many potential extensions to the base AT architecture proposed here. For example, alternative conditioning mechanisms, sparser architectures optimized for audio, and adversarial training could improve results. Multi-task learning objectives that combine reconstruction, prediction, and discrimination may also help.

Multimodal integration of audio, text, and other modalities is also an exciting future direction. Jointly modeling text and audio could improve text-to-speech synthesis and transcription tasks. Exploring these multimodal representations with transformers is promising.

Innovative Approach Previous work has explored adapting transformers for raw audio generation, but these models have had limitations. Some, such as AudioGen 2.2.4.3, have proposed transformer architectures for next-step audio sample prediction, demonstrating solid results for short-term modeling. However, these models do not allow conditional generation.

The proposed AT builds on these predecessors to directly enable unconditional and conditional synthesis from the time-domain audio. By implementing an autoencoder, working on continuous latent features, and leveraging standard encoder-decoder transformers with pretraining, the AT offers a novel approach to generative audio synthesis.

6.3.2 Dataset Expansion

Expanding the available datasets plays a critical role in training generative AI models for audio synthesis. Larger and more diverse datasets provide models with a broader understanding of audio patterns, leading to more realistic and varied audio synthesis. In this subsection, we explore various strategies for dataset expansion, including the incorporation of new data augmentation techniques and the creation of new datasets through real-world recordings and crowdsourcing.

Data augmentation techniques are essential for increasing the diversity and size of the dataset. In the context of audio synthesis, several state-of-the-art data augmentation techniques have been proposed. These techniques can be seen in Section 2.1.3.1 and include time stretching, pitch shifting, noise injection, and others. By applying these techniques, one can artificially introduce variation into the dataset, allowing the models to learn from a wider range of audio patterns.

In future work, it is important to investigate the effectiveness of these data augmentation techniques in the context of audio synthesis. Specifically, the impact of each technique on the performance and generalization of generative models should be evaluated. This evaluation can be done by conducting systematic experiments that compare the performance of models trained with and without data augmentation. In addition, it would be valuable to investigate the combination of multiple data augmentation techniques to further increase the diversity and quality of the dataset.

Furthermore, the exploration of novel data augmentation techniques specifically tailored to audio data should be considered. This may involve exploring and adapting techniques from other domains, such as image or text data augmentation, and tailoring them to the unique characteristics of audio data. The development of domain-specific data augmentation techniques can potentially open up new possibilities for improving the realism and variety of audio synthesis.

Expanding the dataset may also involve creating new datasets that capture a broader range of audio characteristics. One approach is to include real-world recordings, such as live performances, field recordings, or professional studio recordings. These recordings provide a more realistic training environment for the models because they capture the nuances and complexities of real-world audio. Collaborating with musicians, audio engineers, or other experts in the field can ensure the acquisition of high-quality and diverse audio recordings.

Crowdsourcing provides an opportunity to expand the dataset by incorporating user-generated content from online platforms or social media. This approach allows for the inclusion of a diverse and constantly evolving dataset that reflects current trends and preferences in audio production. However, crowdsourced datasets come with their own challenges, such as ensuring data quality and addressing potential biases. Careful curation and validation processes should be implemented to maintain the integrity of the dataset.

6.3.3 Evaluation Metrics

Accurately evaluating the performance of generative AI models' performance is challenging. Existing evaluation metrics often fail to capture the quality, variety, and realism of the generated audio. Future work should focus on developing robust evaluation metrics that align with human perception and subjective audio quality. By establishing reliable evaluation metrics, one can objectively evaluate and compare the performance of different models, facilitating advancements in the field.

As the dataset expands, it becomes necessary to develop new evaluation metrics that can assess the quality and diversity of the dataset. Existing evaluation metrics used in the field may have limitations in capturing the richness and diversity of the expanded datasets. Therefore, exploring and proposing new evaluation metrics that can effectively measure the performance and capabilities of generative models trained on the expanded datasets is important. These metrics should consider factors such as audio realism, diversity of generated outputs, and alignment with ground truth data.

By addressing these areas of future work, the scientific community can further advance the capabilities of generative AI models for audio synthesis. Exploring novel architectures, expanding datasets, and developing improved evaluation metrics will contribute to the development of more powerful and reliable models, enabling applications in diverse domains such as music production, sound design, and interactive audio experiences.

6.4 Conclusion

In summary, this chapter has presented the conclusive results of an extensive investigation into the study and advancement of generative AI models. The primary objectives of this research have been largely achieved, with the identification and discussion of the obstacles that prevented the full achievement of all objectives.

In addition, this study has significantly contributed to the existing body of knowledge on generative DL. While developing end-to-end systems for sound synthesis from textual input remains arduous, particularly for researchers not affiliated with prominent technology companies, this thesis has made notable progress in this area. Initial prototypes have been developed, albeit with unsatisfactory results due to limitations in available datasets and the need for meticulous fine-tuning of hyper-parameters. Nevertheless, the potential for further improvement is recognized. Evaluating the ability of systems to generate sound from textual input also remains an ongoing challenge that warrants further investigation and refinement.

In conclusion, this research has provided invaluable insights into the study and development of generative AI, focusing on audio synthesis. It has successfully achieved significant milestones, conducted a comprehensive investigation of DL architectures and models, and made notable progress in creating end-to-end systems for sound synthesis.

References

[1] Olusola O. Abayomi-Alli, Robertas Damaševičius, Atika Qazi, Mariam Adedoyin-Olowe, and Sanjay Misra. Data Augmentation and Deep Learning Methods in Sound Classification: A Systematic Review. *Electronics*, 11(22):3795, January 2022. Number: 22 Publisher: Multidisciplinary Digital Publishing Institute. Cited on pages 50 and 51.

[2] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. YouTube-8M: A Large-Scale Video Classification Benchmark, September 2016. arXiv:1609.08675 [cs]. Cited on pages 109 and 111.

[3] Adaeze Adigwe, Noé Tits, Kevin El Haddad, Sarah Ostadabbas, and Thierry Dutoit. The Emotional Voices Database: Towards Controlling the Emotion Dimension in Voice Generation Systems, June 2018. arXiv:1806.09514 [cs, eess]. Cited on page 87.

[4] Andrea Agostinelli, Timo I. Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, Matt Sharifi, Neil Zeghidour, and Christian Frank. MusicLM: Generating Music From Text, 2023. Cited on pages 85 and 95.

[5] Hadeer Ahmed, Issa Traore, Mohammad Mamun, and Sherif Saad. Text augmentation using a graph-based approach and clonal selection algorithm. *Machine Learning with Applications*, 11:100452, March 2023. Cited on page 55.

[6] Natsuki Akaishi, Kohei Yatabe, and Yasuhiro Oikawa. Improving Phase-Vocoder-Based Time Stretching by Time-Directional Spectrogram Squeezing. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, June 2023. Cited on page 53.

[7] Ethem Alpaydin. *Introduction to machine learning*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, fourth edition edition, 2020. Cited on page 2.

[8] Francesc Alías, Joan Claudi Socoró, and Xavier Sevillano. A Review of Physical and Perceptual Feature Extraction Techniques for Speech, Music and Environmental Sounds. *Applied Sciences*, 6(5):143, May 2016. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute. Cited on page 74.

[9] Amazon Web Services, Inc. Amazon EC2 P3 – Ideal for Machine Learning and HPC - AWS. Cited on page 102.

[10] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations, 2020. _eprint: 2006.11477. Cited on page 78.

[11] Sören Becker, Marcel Ackermann, Sebastian Lapuschkin, Klaus-Robert Müller, and Wojciech Samek. Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals. *CoRR*, abs/1807.03418, 2018. _eprint: 1807.03418. Cited on pages 107, 109, 110, and 131.

[12] Gilberto Bernardes, Luis Aly, and Matthew Davies. Seed: Resynthesizing environmental sounds from examples. In *Proceedings of the Sound and Music Computing Conference*, pages 55–62, September 2016. Cited on pages 74 and 75.

[13] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational Inference: A Review for Statisticians. *Journal of the American Statistical Association*, 112(518):859–877, April 2017. arXiv:1601.00670 [cs, stat]. Cited on page 60.

[14] Zalán Borsos, Raphaël Marinier, Damien Vincent, Eugene Kharitonov, Olivier Pietquin, Matt Sharifi, Olivier Teboul, David Grangier, Marco Tagliasacchi, and Neil Zeghidour. AudioLM: a Language Modeling Approach to Audio Generation, September 2022. arXiv:2209.03143 [cs, eess]. Cited on pages 85 and 90.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, July 2020. arXiv:2005.14165 [cs]. Cited on page 4.

[16] Tilanka Chandrasekera, So-Yeon Yoon, and Newton D'Souza. Virtual environments with soundscapes: a study on immersion and effects of spatial abilities. *Environment and Planning B: Planning and Design*, 42(6):1003–1019, November 2015. Publisher: SAGE Publications Ltd STM. Cited on page 13.

[17] Tsz-Him Cheung and Dit-Yan Yeung. {MODALS}: Modality-agnostic Automated Data Augmentation in the Latent Space. In *International Conference on Learning Representations*, 2021. Cited on page 55.

[18] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, October 2014. arXiv:1409.1259 [cs, stat]. Cited on page 25.

[19] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, September 2014. arXiv:1406.1078 [cs, stat]. Cited on page 27.

[20] Francois Chollet and others. Keras, 2015. Publisher: GitHub. Cited on page 65.

[21] Christopher Olah. Understanding LSTM Networks, August 2015. Cited on page 26.

[22] Yu-An Chung, Yu Zhang, Wei Han, Chung-Cheng Chiu, James Qin, Ruoming Pang, and Yonghui Wu. W2v-BERT: Combining Contrastive Learning and Masked Language Modeling for Self-Supervised Speech Pre-Training, September 2021. arXiv:2108.06209 [cs, eess]. Cited on page 91.

[23] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989. Publisher: Springer London. Cited on page 17.

[24] Jeffrey Dean. A Golden Decade of Deep Learning: Computing Systems & Applications. *Daedalus*, 151(2):58–74, May 2022. Cited on page 4.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004. Cited on page 3.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. Publisher: IEEE. Cited on page 110.

[27] Li Deng and Dong Yu. Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, June 2014. Publisher: Now Publishers, Inc. Cited on page 15.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805, 2018. arXiv: 1810.04805. Cited on pages 63 and 152.

[29] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A Generative Model for Music, April 2020. arXiv:2005.00341 [cs, eess, stat]. Cited on pages 85, 88, and 95.

[30] Torgeir Dingsøyr, Sridhar Nerur, VenuGopal Balijepally, and Nils Brede Moe. A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, 85(6):1213–1221, June 2012. Cited on page 108.

[31] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial Audio Synthesis, February 2019. arXiv:1802.04208 [cs]. Cited on page 77.

[32] Constance Douwes, Philippe Esling, and Jean-Pierre Briot. Energy Consumption of Deep Generative Audio Models, October 2021. arXiv:2107.02621 [cs, eess]. Cited on page 62.

[33] Konstantinos Drossos, Samuel Lipping, and Tuomas Virtanen. Clotho: An Audio Captioning Dataset. *CoRR*, abs/1910.09387, 2019. arXiv: 1910.09387. Cited on pages 109, 112, and 131.

[34] Peter Elsea. Basics of Digital Recording, 1996. Cited on page 12.

[35] Jesse Engel, Kumar Krishna Agrawal, Shuo Chen, Ishaan Gulrajani, Chris Donahue, and Adam Roberts. GANSynth: Adversarial Neural Audio Synthesis, April 2019. arXiv:1902.08710 [cs, eess, stat]. Cited on pages 83 and 95.

[36] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi. Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders, April 2017. arXiv:1704.01279 [cs]. Cited on page 83.

[37] J. D. Fernandez and F. Vico. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research*, 48:513–582, November 2013. Cited on page 74.

[38] Eduardo Fonseca, Manoj Plakal, Frederic Font, Daniel P. W. Ellis, Xavier Favory, Jordi Pons, and Xavier Serra. General-purpose Tagging of Freesound Audio with AudioSet Labels: Task Description, Dataset, and Baseline, 2018. Cited on pages 109 and 110.

[39] Forero, J., Bernardes, G., and Mendes, M. Are words enough? *AIMC 2023*, August 2023. Cited on page 4.

[40] Seth* Forsgren and Hayk* Martiros. Riffusion - Stable diffusion for real-time music generation, 2022. Cited on pages 85, 88, and 95.

[41] Alexander L. Fradkov. Early History of Machine Learning. *IFAC-PapersOnLine*, 53(2):1385–1390, January 2020. Cited on page 3.

[42] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980. Cited on page 18.

[43] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. Audio Set: An ontology and human-labeled dataset for audio events. In *Proc. IEEE ICASSP 2017*, New Orleans, LA, 2017. Cited on pages 109 and 110.

[44] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, May 2015. Number: 7553 Publisher: Nature Publishing Group. Cited on page 3.

[45] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. Cited on page 4.

[46] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, June 2014. arXiv:1406.2661 [cs, stat]. Cited on page 40.

[47] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. Deep AutoRegressive Networks, May 2014. arXiv:1310.8499 [cs, stat]. Cited on page 37.

[48] D. Griffin and Jae Lim. Signal estimation from modified short-time Fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(2):236–243, April 1984. Conference Name: IEEE Transactions on Acoustics, Speech, and Signal Processing. Cited on page 88.

[49] Ross Gruetzemacher and David Paradice. Deep Transfer Learning & Beyond: Transformer Language Models in Information Systems Research. *ACM Computing Surveys*, 54(10s):204:1–204:35, September 2022. Cited on page 155.

[50] Hongyu Guo, Yongyi Mao, and Richong Zhang. Augmenting Data with Mixup for Sentence Classification: An Empirical Study, May 2019. arXiv:1905.08941 [cs]. Cited on page 55.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. arXiv:1512.03385 [cs]. Cited on page 152.

[52] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949. Published: Hardcover. Cited on page 15.

[53] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models, December 2020. arXiv:2006.11239 [cs, stat]. Cited on page 43.

[54] Jonathan Ho and Tim Salimans. Classifier-Free Diffusion Guidance, July 2022. arXiv:2207.12598 [cs]. Cited on page 72.

[55] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997. Cited on pages 24 and 25.

[56] Timothy O. Hodson, Thomas M. Over, and Sydney S. Foks. Mean Squared Error, Deconstructed. *Journal of Advances in Modeling Earth Systems*, 13(12):e2021MS002681, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2021MS002681. Cited on page 58.

[57] Constance Holden. The Origin of Speech. *Science*, 303(5662):1316–1319, 2004. _eprint: https://www.science.org/doi/pdf/10.1126/science.303.5662.1316. Cited on page 4.

[58] Reynald Hoskinson and Dinesh K. Pai. Manipulation and Resynthesis with Natural Grains. In *Proceedings of the 2001 International Computer Music Conference, ICMC 2001, Havana, Cuba, September 17-22, 2001*. Michigan Publishing, 2001. Cited on page 75.

[59] Qingqing Huang, Aren Jansen, Joonseok Lee, Ravi Ganti, Judith Yue Li, and Daniel P. W. Ellis. MuLan: A Joint Embedding of Music Audio and Natural Language, August 2022. arXiv:2208.12415 [cs, eess, stat]. Cited on page 63.

[60] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 148(3):574–591, October 1959. Cited on page 17.

[61] Muhammad Huzaifah and Lonce Wyse. Deep Generative Models for Musical Audio Synthesis. In Eduardo Reck Miranda, editor, *Handbook of Artificial Intelligence for Music: Foundations, Advanced Approaches, and Developments for Creativity*, pages 639–678. Springer International Publishing, Cham, 2021. Cited on pages 4, 5, 18, 37, 38, 39, 41, 42, 60, 81, 100, and 103.

[62] H. Hyötyniemi. Turing machines are recurrent neural networks. In J. Alander and T. Honkela, editors, *STeP '96 - Genes, Nets and Symbols; Finnish Artificial Intelligence Conference, Vaasa, Finland, 20-23 August 1996*, pages 13–24. University of Vaasa, Finnish Artificial Intelligence Society (FAIS), 1996. Cited on page 24.

[63] International Organization for Standardization. ISO 12913-1:2014(en), Acoustics — Soundscape, 2014. Cited on pages 5 and 13.

[64] Jacob Kahn, Morgane Riviere, Weiyi Zheng, Evgeny Kharitonov, Qiantong Xu, Pierre-Emmanuel Mazaré, Julien Karadayi, Vitaliy Liptchinsky, Ronan Collobert, Christian Fuegen, and others. Libri-light: A benchmark for asr with limited or no supervision. In *ICASSP*

2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 7669–7673. IEEE, 2020. Cited on page 86.

[65] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient Neural Audio Synthesis, June 2018. arXiv:1802.08435 [cs, eess]. Cited on page 82.

[66] Stefano Kalonaris and Anna Jordanous. Computational Music Aesthetics: a survey and some thoughts. Dublin, Ireland, August 2018. Accepted: 2018-07-09. Cited on page 74.

[67] Chris Dongjoo Kim, Byeongchang Kim, Hyunmin Lee, and Gunhee Kim. AudioCaps: Generating Captions for Audios in The Wild. In *NAACL-HLT*, 2019. Cited on pages 109 and 112.

[68] Sungwon Kim, Sang-gil Lee, Jongyoon Song, and Sungroh Yoon. FloWaveNet : A Generative Flow for Raw Audio. *CoRR*, abs/1811.02155, 2018. arXiv: 1811.02155. Cited on page 82.

[69] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs]. Cited on page 36.

[70] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, December 2022. arXiv:1312.6114 [cs, stat]. Cited on page 38.

[71] Jungil Kong, Jaehyeon Kim, and Jaekyoung Bae. HiFi-GAN: Generative Adversarial Networks for Efficient and High Fidelity Speech Synthesis. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17022–17033. Curran Associates, Inc., 2020. Cited on pages 74, 83, and 95.

[72] Felix Kreuk, Gabriel Synnaeve, Adam Polyak, Uriel Singer, Alexandre Défossez, Jade Copet, Devi Parikh, Yaniv Taigman, and Yossi Adi. AudioGen: Textually Guided Audio Generation, March 2023. arXiv:2209.15352 [cs, eess]. Cited on pages 85 and 92.

[73] Kundan Kumar, Rithesh Kumar, Thibault de Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre de Brébisson, Yoshua Bengio, and Aaron C Courville. MelGAN: Generative Adversarial Networks for Conditional Waveform Synthesis. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. Cited on pages 82 and 95.

[74] Haohe Liu, Zehua Chen, Yi Yuan, Xinhao Mei, Xubo Liu, Danilo Mandic, Wenwu Wang, and Mark D. Plumbley. AudioLDM: Text-to-Audio Generation with Latent Diffusion Models, February 2023. arXiv:2301.12503 [cs, eess]. Cited on pages 123 and 192.

[75] E Magalhães, J Jacob, N Nilsson, R Nordahl, and G Bernardes. Physics-based Concatenative Sound Synthesis of Photogrammetric models for Aural and Haptic Feedback in Virtual Environments. In *2020 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 376–379, March 2020. Cited on page 76.

[76] Gianluca Maguolo, Michelangelo Paci, Loris Nanni, and Ludovico Bonan. Audiogmenter: a MATLAB Toolbox for Audio Data Augmentation, January 2022. arXiv:1912.05472 [cs, eess]. Cited on page 53.

[77] Dave Martinez, Nick Malyska, Bill Streilein, Rajmonda Caceres, William Campbell, Charlie Dagli, Vijay Gadepally, Kara Greenfield, Robert Hall, Andre King, and others. Artificial intelligence: Short history, present developments, and future outlook, 2019. Cited on page 3.

[78] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Cited on page 64.

[79] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. 1969. Cited on pages 3, 17, and 23.

[80] Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron Courville, and Yoshua Bengio. SampleRNN: An Unconditional End-to-End Neural Audio Generation Model, February 2017. arXiv:1612.07837 [cs]. Cited on pages 85 and 90.

[81] Ambuj Mehrish, Navonil Majumder, Rishabh Bhardwaj, Rada Mihalcea, and Soujanya Poria. A Review of Deep Learning Techniques for Speech Processing, May 2023. arXiv:2305.00359 [eess]. Cited on page 80.

[82] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, 2013. Cited on page 63.

[83] Tom M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. McGraw-Hill, New York, 1997. Cited on page 2.

[84] Zohaib Mushtaq and Shun-Feng Su. Environmental sound classification using a regularized deep convolutional neural network with data augmentation. *Applied Acoustics*, 167:107389, October 2020. Cited on pages 51 and 52.

[85] Alex Nichol, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob McGrew, Ilya Sutskever, and Mark Chen. GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models, 2021. Cited on pages 68 and 96.

[86] Ondřej Novotný, Oldřich Plchot, Ondřej Glembek, Jan Honza Cernocký, and Lukáš Burget. Analysis of DNN Speech Signal Enhancement for Robust Speaker Recognition. *Computer Speech & Language*, 58:403–421, November 2019. Cited on page 51.

[87] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A Generative Model for Raw Audio, September 2016. arXiv:1609.03499 [cs]. Cited on pages 12, 81, 82, and 95.

[88] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional Image Generation with PixelCNN Decoders, June 2016. arXiv:1606.05328 [cs]. Cited on pages 67, 68, and 95.

[89] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural Discrete Representation Learning, May 2018. arXiv:1711.00937 [cs]. Cited on page 46.

[90] Oxford English Dictionary. music, n. & adj., July 2023. Cited on page 4.

[91] Tom Le Paine, Pooya Khorrami, Shiyu Chang, Yang Zhang, Prajit Ramachandran, Mark A. Hasegawa-Johnson, and Thomas S. Huang. Fast Wavenet Generation Algorithm, November 2016. arXiv:1611.09482 [cs]. Cited on page 82.

[92] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, April 2015. ISSN: 2379-190X. Cited on page 86.

[93] Aniruddha Parvat, Jai Chavan, Siddhesh Kadam, Souradeep Dev, and Vidhi Pathak. A survey of deep-learning frameworks. In *2017 International Conference on Inventive Systems and Control (ICISC)*, pages 1–7, January 2017. Cited on page 64.

[94] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. Cited on page 65.

[95] Geoffroy Peeters and Gaël Richard. Deep Learning for Audio and Music. In Jenny Benois-Pineau and Akka Zemmari, editors, *Multi-faceted Deep Learning*, pages 231–266. Springer International Publishing, Cham, 2021. Cited on pages 12, 13, and 14.

[96] Hieu Pham, Xinyi Wang, Yiming Yang, and Graham Neubig. Meta Back-Translation. In *International Conference on Learning Representations*, 2021. Cited on page 55.

[97] Domagoj Pluščec and Jan Šnajder. Data Augmentation for Neural NLP, February 2023. arXiv:2302.11412 [cs]. Cited on page 50.

[98] Yanmin Qian, Hu Hu, and Tian Tan. Data augmentation using generative adversarial networks for robust speech recognition. *Speech Communication*, 114:1–9, November 2019. Cited on page 52.

[99] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision, 2021. Cited on pages 63, 72, 109, and 151.

[100] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical Text-Conditional Image Generation with CLIP Latents, April 2022. arXiv:2204.06125 [cs]. Cited on pages 4, 68, and 72.

[101] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-Shot Text-to-Image Generation, February 2021. arXiv:2102.12092 [cs]. Cited on pages 4, 49, 67, 68, 95, and 96.

[102] Kanishka Rao, Fuchun Peng, Hasim Sak, and Francoise Beaufays. Grapheme-to-phoneme conversion using Long Short-Term Memory recurrent neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4225–4229, South Brisbane, Queensland, Australia, April 2015. IEEE. Cited on page 85.

[103] Edison Research. The Infinite Dial 2021, March 2021. Section: Featured. Cited on page 74.

[104] Danilo Jimenez Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows, June 2016. arXiv:1505.05770 [cs, stat]. Cited on page 42.

[105] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2(11):1019–1025, November 1999. Number: 11 Publisher: Nature Publishing Group. Cited on page 21.

[106] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. *CoRR*, abs/2112.10752, 2021. arXiv: 2112.10752. Cited on pages 68 and 70.

[107] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015. arXiv:1505.04597 [cs]. Cited on page 28.

[108] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. Cited on pages 2 and 16.

[109] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, 1986. Cited on page 23.

[110] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986. Cited on page 3.

[111] David E. Rumelhart and James L. McClelland. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pages 318–362. MIT Press, 1987. Conference Name: Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations. Cited on page 23.

[112] J. Salamon, C. Jacoby, and J. P. Bello. A Dataset and Taxonomy for Urban Sound Research. In *22nd ACM International Conference on Multimedia (ACM-MM'14)*, pages 1041–1044, Orlando, FL, USA, November 2014. Cited on pages 109 and 111.

[113] Justin Salamon, Duncan MacConnell, Mark Cartwright, Peter Li, and Juan Pablo Bello. Scaper: A library for soundscape synthesis and augmentation. In *2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 344–348, October 2017. ISSN: 1947-1629. Cited on page 75.

[114] R. Murray Schafer. *The Tuning of the World*. Knopf, 1977. Google-Books-ID: SIufAAAAMAAJ. Cited on pages 5 and 13.

[115] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997. Publisher: Ieee. Cited on page 27.

[116] Connor Shorten, Taghi M. Khoshgoftaar, and Borko Furht. Text Data Augmentation for Deep Learning. *Journal of Big Data*, 8(1):101, July 2021. Cited on page 54.

[117] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics, November 2015. arXiv:1503.03585 [cond-mat, q-bio, stat]. Cited on page 43.

[118] David Sonnenschein. *Sound Design: The Expressive Power of Music, Voice, and Sound Effects in Cinema*. Michael Wiese Productions, 2001. Cited on page 73.

[119] Jose Sotelo, Soroush Mehri, Kundan Kumar, Joao Felipe Santos, Kyle Kastner, Aaron Courville, and Yoshua Bengio. Char2wav: End-to-end speech synthesis. 2017. Cited on page 85.

[120] Gerda Strobl, Gerhard Eckel, and Davide Rocchesso. Sound Texture Modeling: A Survey,. In *Proceedings of the Sound and Music Computing Conference*, pages 61–65, 2006. Cited on page 74.

[121] Eric Strong, Alicia DiGiammarino, Yingjie Weng, Andre Kumar, Poonam Hosamani, Jason Hom, and Jonathan H. Chen. Chatbot vs Medical Student Performance on Free-Response Clinical Reasoning Examinations. *JAMA Internal Medicine*, July 2023. Cited on page 5.

[122] Koray Tahiroğlu, Miranda Kastemaa, and Oskar Koli. Al-terity: Non-Rigid Musical Instrument with Artificial Intelligence Applied to Real-Time Audio Synthesis. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Proceedings of the International Conference on New Interfaces for Musical Expression, pages 337–342. International Conference on New Interfaces for Musical Expression, July 2020. Cited on pages 6, 10, 41, and 82.

[123] Naoya Takahashi, Michael Gygli, Beat Pfister, and Luc Van Gool. Deep Convolutional Neural Networks and Data Augmentation for Acoustic Event Recognition. In *Proc. Interspeech 2016*, pages 2982–2986, 2016. Cited on pages 109 and 110.

[124] Andros Tjandra, Berrak Sisman, Mingyang Zhang, Sakriani Sakti, Haizhou Li, and Satoshi Nakamura. VQVAE Unsupervised Unit Discovery and Multi-scale Code2Spec Inverter for Zerospeech Challenge 2019, May 2019. arXiv:1905.11449 [cs, eess]. Cited on page 49.

[125] Lutz Trautmann and Rudolf Rabenstein. Classical Synthesis Methods Based on Physical Models. In Lutz Trautmann and Rudolf Rabenstein, editors, *Digital Sound Synthesis by Physical Modeling Using the Functional Transformation Method*, pages 63–93. Springer US, Boston, MA, 2003. Cited on page 5.

[126] University of York. What is Computer Science? Cited on page 2.

[127] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel Recurrent Neural Networks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1747–1756, New York, New York, USA, June 2016. PMLR. Cited on page 67.

[128] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, December 2017. arXiv:1706.03762 [cs]. Cited on pages 44 and 47.

[129] Prateek Verma and Chris Chafe. A Generative Model for Raw Audio Using Transformer Architectures, July 2021. arXiv:2106.16036 [cs, eess]. Cited on page 78.

[130] Ashvala Vinay and Alexander Lerch. Evaluating generative audio systems and their metrics, August 2022. arXiv:2209.00130 [cs, eess]. Cited on page 57.

[131] Chengyi Wang, Sanyuan Chen, Yu Wu, Ziqiang Zhang, Long Zhou, Shujie Liu, Zhuo Chen, Yanqing Liu, Huaming Wang, Jinyu Li, Lei He, Sheng Zhao, and Furu Wei. Neural Codec Language Models are Zero-Shot Text to Speech Synthesizers, January 2023. arXiv:2301.02111 [cs, eess]. Cited on pages 85, 86, 95, and 96.

[132] Jason Wei and Kai Zou. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6382–6388, Hong Kong, China, November 2019. Association for Computational Linguistics. Cited on page 55.

[133] Lilian Weng. Flow-based Deep Generative Models. *lilianweng.github.io*, 2018. Cited on page 42.

[134] Cort J. Willmott and Kenji Matsuura. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research*, 30(1):79–82, December 2005. Cited on page 57.

[135] Dongchao Yang, Jianwei Yu, Helin Wang, Wen Wang, Chao Weng, Yuexian Zou, and Dong Yu. Diffsound: Discrete Diffusion Model for Text-to-sound Generation, July 2022. arXiv:2207.09983 [cs, eess]. Cited on pages 49, 85, and 91.

[136] Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. SoundStream: An End-to-End Neural Audio Codec, July 2021. arXiv:2107.03312 [cs, eess]. Cited on page 79.

[137] Hui Zou and Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 67(2):301–320, 2005. Publisher: [Royal Statistical Society, Wiley]. Cited on page 136.

Appendices

Appendix A

Classification Model

This appendix presents the architecture and training process of the developed audio classification model.

A.1 Model Architecture

The ConvClassifier class defines the architecture of the one-dimensional CNN. The following code snippet provides an overview of the ConvClassifier class:

```
class ConvClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv1d(
            in_channels=1, out_channels=32, kernel_size=9, stride=1, padding=1)
        self.pool1 = nn.MaxPool1d(2, stride=2)

        self.conv2 = nn.Conv1d(32, 64, 9, stride=1, padding=1)
        self.pool2 = nn.MaxPool1d(2, stride=2)

        self.conv3 = nn.Conv1d(64, 128, 9, stride=1, padding=1)
        self.pool3 = nn.MaxPool1d(2, stride=2)

        self.conv4 = nn.Conv1d(128, 256, 9, stride=1, padding=1)
        self.pool4 = nn.MaxPool1d(2, stride=2)

        self.conv5 = nn.Conv1d(256, 512, 9, stride=1, padding=1)
        self.pool5 = nn.MaxPool1d(2, stride=2)

        self.gap = nn.AdaptiveAvgPool1d(1)

        self.linear1 = nn.Linear(512, 256)
```

```
        self.linear2 = nn.Linear(256, 128)
        self.linear3 = nn.Linear(128, 10)

        self.relu = nn.ReLU()

    def forward(self, x):
        # pass to float
        x = x.float()

        x = self.pool1(self.relu(self.conv1(x)))
        x = self.pool2(self.relu(self.conv2(x)))
        x = self.pool3(self.relu(self.conv3(x)))
        x = self.pool4(self.relu(self.conv4(x)))
        x = self.pool5(self.relu(self.conv5(x)))

        x = self.gap(x)

        x = x.view(-1, 512)

        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        x = self.linear3(x)

        return x
```

Listing A.1: ConvClassifier class for sound classification

A.2 Training Process

The model is trained using the provided `train` function, which implements SGD optimization and batch-wise backpropagation. The following code snippet outlines the training process:

```
def train(data_loader, model, loss_fn, optimizer):
    model.train()

    for batch, (sample, label, _) in enumerate(data_loader):
        sample, label = sample.to(device), label.to(device)

        # Compute prediction error
        pred = model(sample)
        loss = loss_fn(pred, label)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

```
16          if batch % 100 == 0:
17              loss, current = loss.item(), batch * len(sample)
18              print(f"loss: {loss:>7f}  [{current:>5d}/{len(data_loader.dataset)
                    :>5d}]")
19
20  def test(data_loader, model, loss_fn):
21      model.eval()
22
23      test_loss, correct = 0, 0
24
25      with torch.no_grad():
26          for sample, label, _ in data_loader:
27              sample, label = sample.to(device), label.to(device)
28
29              pred = model(sample)
30              test_loss += loss_fn(pred, label).item()
31
32              correct += (pred.argmax(1) == label).type(
33                  torch.float).sum().item()
34
35      test_loss /= len(data_loader)
36      correct /= len(data_loader.dataset)
37
38      print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {
            test_loss:>8f} \n")
39
40  epochs = 20
41
42  for epoch in range(epochs):
43      print(f"Epoch {epoch + 1}\n-------------------------------")
44
45      train(dataloader_train, model, loss_fn, optimizer)
46      test(dataloader_test, model, loss_fn)
47      print()
48
49  print("Done!")
```

Listing A.2: Training process for sound classification

Appendix B

GAN Implementation Details

B.1 Models

B.1.1 Generator

The generator model is constructed as a sequential neural network. Its goal is to take a noise input and generate audio samples that resemble the desired output.

```
def _make_generator_model(self, input_shape=(100,)):
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Dense(
        256, use_bias=False, input_shape=input_shape))
    model.add(tf.keras.layers.Reshape((16, 16)))
    model.add(tf.keras.layers.ReLU())
    model.add(tf.keras.layers.Conv1DTranspose(
        32, 25, strides=4, padding='same'))
    model.add(tf.keras.layers.ReLU())
    model.add(tf.keras.layers.Conv1DTranspose(
        16, 25, strides=4, padding='same'))
    model.add(tf.keras.layers.ReLU())
    model.add(tf.keras.layers.Conv1DTranspose(
        8, 25, strides=4, padding='same'))
    model.add(tf.keras.layers.ReLU())
    model.add(tf.keras.layers.Conv1DTranspose(
        4, 25, strides=4, padding='same'))
    model.add(tf.keras.layers.ReLU())
    model.add(tf.keras.layers.Conv1DTranspose(
        2, 25, strides=4, padding='same'))
    model.add(tf.keras.layers.ReLU())
    model.add(tf.keras.layers.Conv1DTranspose(
        1, 25, strides=4, padding='same'))
    model.add(tf.keras.layers.Activation('tanh'))

```

```
26      return model
```

Listing B.1: Generator initialization

The generator consists of 6 blocks, each of which consists of a transposed convolution layer followed by a ReLU activation function (see Sections 2.1.2.1 and 2.1.2.2). The exception is the last block, which uses a tanh activation function (explained in Sections 2.1.2.2) to ensure that the generated audio values fall within the range of -1 to 1.

B.1.2 Discriminator

The discriminator model is designed to distinguish between real and generated audio samples. Like the generator, it's implemented as a sequential neural network.

```
1  def _make_discriminator_model(self, input_shape=(INPUT_SIZE, 1)):
2      model = tf.keras.Sequential()
3      model.add(tf.keras.layers.Conv1D(1, 25, strides=4,
4                                       input_shape=input_shape, padding='same'))
5      model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
6
7      model.add(tf.keras.layers.Conv1D(2, 25, strides=4, padding='same'))
8      model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
9
10     model.add(tf.keras.layers.Conv1D(4, 25, strides=4, padding='same'))
11     model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
12
13     model.add(tf.keras.layers.Conv1D(8, 25, strides=4, padding='same'))
14     model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
15
16     model.add(tf.keras.layers.Conv1D(16, 25, strides=4, padding='same'))
17     model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
18
19     model.add(tf.keras.layers.Conv1D(32, 25, strides=4, padding='same'))
20     model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
21
22     model.add(tf.keras.layers.Flatten())
23
24     model.add(tf.keras.layers.Dense(1))
25
26     return model
```

Listing B.2: Discriminator initialization

The discriminator consists of 6 convolutional blocks, each followed by a leaky ReLU activation function. After these blocks, the model is flattened to be fed into a dense layer with a single output unit to provide a binary classification indicating whether the input is real or generated.

B.2 Training

The GAN training procedure involves a series of steps to iteratively optimize the generator and discriminator networks.

```
def train(self, dataset, epochs):
    # ... (checkpoint loading operations)

    # run the epochs
    for epoch in tqdm(range(epochs)):
        print("Epoch {}/{}".format(epoch+1, epochs))

        # run the batches
        for audios in tqdm(dataset):
            self._train_step(audios)

        # ... (display and checkpoint operations)

@tf.function
def _train_step(self, audios):
    noise = tf.random.normal([BATCH_SIZE, self._fake_input_shape[0]])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_audios = self._generator(noise, training=True)

        real_output = self._discriminator(audios, training=True)
        fake_output = self._discriminator(generated_audios, training=True)

        gen_loss = self._generator_loss(fake_output)
        disc_loss = self._discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(
        gen_loss, self._generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(
        disc_loss, self._discriminator.trainable_variables)

    self._generator_optimizer.apply_gradients(
        zip(gradients_of_generator, self._generator.trainable_variables))
    self._discriminator_optimizer.apply_gradients(
        zip(gradients_of_discriminator, self._discriminator.trainable_variables
            ))

def _generator_loss(self, fake_output):
    return tf.keras.losses.BinaryCrossentropy(from_logits=True)(tf.ones_like(
        fake_output), fake_output)

def _discriminator_loss(self, real_output, fake_output):
    real_loss = tf.keras.losses.BinaryCrossentropy(
```

```
42          from_logits=True)(tf.ones_like(real_output), real_output)
43      fake_loss = tf.keras.losses.BinaryCrossentropy(
44          from_logits=True)(tf.zeros_like(fake_output), fake_output)
45      total_loss = real_loss + fake_loss
46      return total_loss
```

Listing B.3: Training operations

For each epoch, the generator produces a set of generated audio samples using random noise. Both the real and generated audio samples are then passed through the discriminator network to calculate their respective outputs.

The generator and discriminator losses are calculated based on the discriminator outputs. The generator aims to minimize the BCE loss associated with the output of the fake samples, thereby encouraging the discriminator to classify them as real. Conversely, the discriminator seeks to minimize the loss by discriminating between real and false samples.

The gradients of the generator and discriminator losses are computed using a gradient band, and the model parameters are updated accordingly using the Adam optimizer (see Section 2.1.2.2).

Appendix C

Autoencoder Implementation Details

C.1 Generative Model Code

Below is the implementation of the AE model architecture used for generating images. This code defines the architecture of the encoder and decoder components, as well as the training and testing functions.

```
class AutoEncoder(nn.Module):
    # ... (other class operations)

    def forward(self, x):
        # passes the input through the encoder and then through the decoder
        x = x.float()
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    def _build_encoder(self):
        layers = []

        layers.append(nn.Conv1d(1, 32, kernel_size=9, stride=1, padding=4))
        layers.append(nn.BatchNorm1d(32))
        layers.append(nn.Tanh())
        layers.append(nn.MaxPool1d(kernel_size=2, stride=2))

        for i in range(self.convolutional_layers - 1):
            layers.append(nn.Conv1d(2**i*32, 2**(i+1)*32,
                          kernel_size=9, stride=1, padding=4))

            layers.append(nn.BatchNorm1d(2**(i+1)*32))
            layers.append(nn.Tanh())

```

```
            layers.append(nn.MaxPool1d(kernel_size=2, stride=2))

        return nn.Sequential(*layers)

    def _build_decoder(self):
        layers = []

        for i in range(self.convolutional_layers - 2, -1, -1):
            layers.append(nn.Upsample(scale_factor=2))
            layers.append(nn.ConvTranspose1d(2**(i+1)*32, 2**i *
                          32, kernel_size=9, stride=1, padding=4))
            layers.append(nn.BatchNorm1d(2**i*32))
            layers.append(nn.Tanh())

        layers.append(nn.Upsample(scale_factor=2))
        layers.append(nn.ConvTranspose1d(
            32, 1, kernel_size=9, stride=1, padding=4))
        layers.append(nn.BatchNorm1d(1))
        layers.append(nn.Tanh())

        return nn.Sequential(*layers)
```

Listing C.1: Model architecture

C.2 Training Code

The following code snippet showcases the training operations for the AE model. It includes data loading, computation of prediction error, backpropagation, and optimization.

```
def train(data_loader, model, loss_fn, optimizer):
    model.train()

    for batch, (sample, label, _) in enumerate(data_loader):
        # normalize the audio wav sample
        sample = sample / 32768

        # ... (other input operations)

        # compute prediction error
        pred = model(sample)
        loss = loss_fn(pred, sample)

        # backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

```
        # ... (printing and memory management operations)


def test(data_loader, model, loss_fn, name):
    model.eval()

    test_loss, average_mse = 0, 0
    showed = False

    with torch.no_grad():
        for sample, label, _ in data_loader:
            sample = sample / 32768

            # ... (other input operations)

            pred = model(sample)
            test_loss += loss_fn(pred, sample).item()

            average_mse += torch.mean((pred - sample) ** 2)

            # ... (input operations)

    test_loss /= len(data_loader)
    average_mse /= len(data_loader)

    return test_loss, average_mse

def run():
    # model initialization
    model = AutoEncoder(convolutional_layers=4).to(device)
    loss_fn = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    # start training
    # ... (display and early stopping initializations)
    epoch = 0

    while True: # actual code checks for early stopping
        train(dataloader_train, model, loss_fn, optimizer)
        loss, mse = test(dataloader_test, model, loss_fn, f"{conv_layers}_{
            epoch}")

        # ... (early stopping settings)

        epoch += 1

    # ... (printing and storing settings)
```

Listing C.2: Training operations

C.3 Results

This section presents a comparison of an original sound, represented by a soundwave, and its recreation.

It is important to note that the recreated image is the result of the original sound passing through an AE with the specified threshold, as detailed in the accompanying annex.

This result was achieved after only 20 epochs, and the recreated sound is nearly as good as the original.

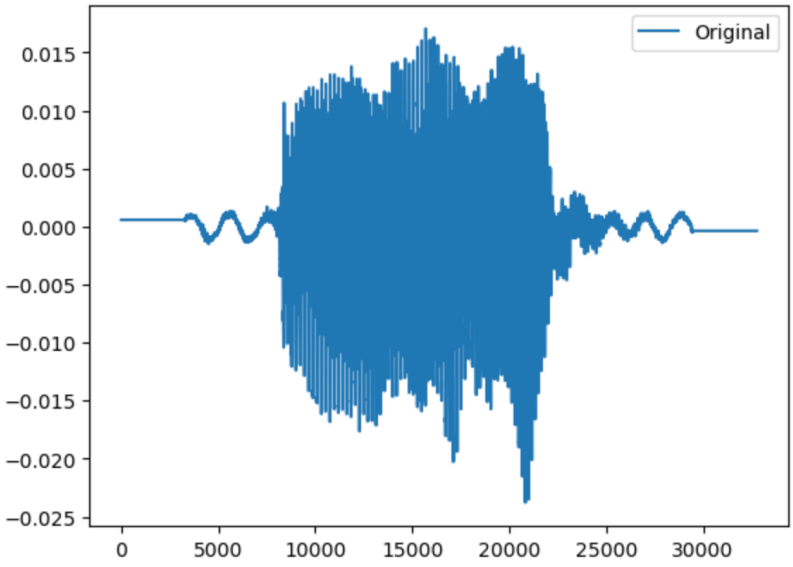


Figure C.1: Original Sample

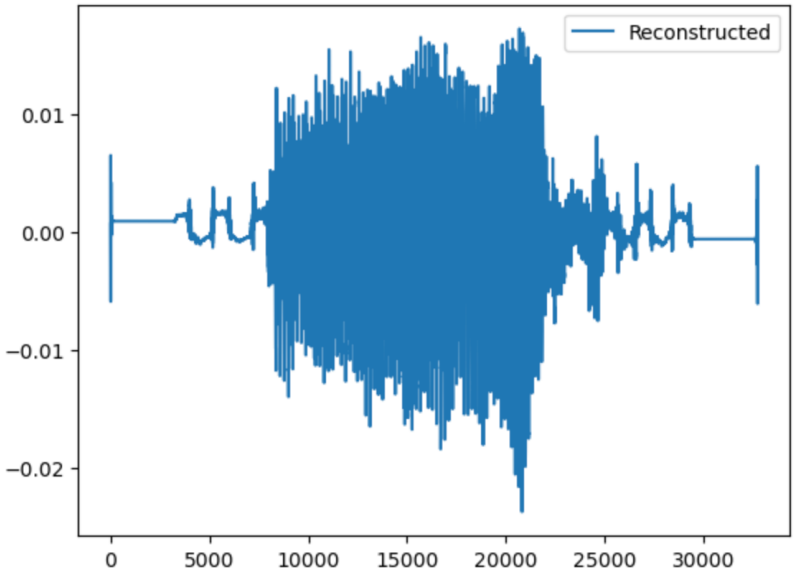


Figure C.2: Reconstructed Sample

Appendix D

Variational Autoencoder Implementation Details

In this annex, code implementation of the VAE is provided. The VAE is a generative model that is specifically designed to learn latent representations of data and enable controlled generation. The following sections elucidate the VAE model's structure and its training process.

D.1 Model Implementation

The subsequent Python code exemplifies the execution of the simple VAE model.

```
class VAE(nn.Module):
    # ... (class operations)

    def forward(self, x):
        x = x.float()

        y = self.encoder(x)

        mean = self.meaner(y)
        logvar = self.varer(y)

        z = self._sample(mean, logvar)

        decoded = self.decoder(z)
        return decoded, mean, logvar

    def _build_encoder(self):
        layers = []

        layers.append(nn.Conv1d(1, 32, kernel_size=9, stride=1, padding=4))
```

```
21          layers.append(nn.BatchNorm1d(32))
22          layers.append(nn.Tanh())
23          layers.append(nn.MaxPool1d(kernel_size=2, stride=2))
24
25          for i in range(self.convolutional_layers - 1):
26              layer = nn.Conv1d(2**i*32, 2**(i+1)*32,
27                                kernel_size=9, stride=1, padding=4)
28              layers.append(layer)
29              layers.append(nn.BatchNorm1d(2**(i+1)*32))
30              layers.append(nn.Tanh())
31
32              layers.append(nn.MaxPool1d(kernel_size=2, stride=2))
33
34          mean_layer = nn.Sequential(
35              nn.AdaptiveAvgPool1d(1),
36              nn.Flatten(),
37              nn.Linear(2**(self.convolutional_layers-1)*32, self.latent_dim)
38          )
39
40          var_layer = nn.Sequential(
41              nn.AdaptiveAvgPool1d(1),
42              nn.Flatten(),
43              nn.Linear(2**(self.convolutional_layers-1)*32, self.latent_dim)
44          )
45
46          return nn.Sequential(*layers), mean_layer, var_layer
47
48      def _build_decoder(self):
49          layers = []
50
51          layers.append(
52              nn.Linear(self.latent_dim, 2**(self.convolutional_layers-1)
53                        * 32 * (self.input_size // 2**self.convolutional_layers))
54          )
55
56          layers.append(
57              nn.Unflatten(1, (2**(self.convolutional_layers-1)*32,
58                               self.input_size // 2**self.convolutional_layers))
59          )
60
61          for i in range(self.convolutional_layers - 2, -1, -1):
62              layers.append(nn.Upsample(scale_factor=2))
63              layers.append(nn.ConvTranspose1d(2**(i+1)*32, 2**i *
64                                               32, kernel_size=9, stride=1,
                                                     padding=4))
65              layers.append(nn.BatchNorm1d(2**i*32))
66              layers.append(nn.Tanh())
67
68          layers.append(nn.Upsample(scale_factor=2))
```

```
        layers.append(nn.ConvTranspose1d(
            32, 1, kernel_size=9, stride=1, padding=4))
        layers.append(nn.BatchNorm1d(1))
        layers.append(nn.Tanh())

        return nn.Sequential(*layers)

    def _sample(self, mean, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return eps * std + mean

    def loss_function(self, recon_x, x, mean, logvar):
        recon_x = torch.clamp(recon_x, 0, 1) # normalize output
        x = torch.clamp(x, 0, 1) # normalize input
        BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
        KLD = -0.5 * torch.sum(1 + logvar - mean.pow(2) - logvar.exp())

        return BCE + KLD
```

The VAE model comprises an encoder, a bottleneck layer, and a decoder. The encoder analyzes the input data to extract significant characteristics, whereas the decoder produces reconstructions from covert representations. The latent space is sampled by employing mean and variance parameters from the encoder.

D.2 Training Process

The VAE is trained with the following code.

```
def train_step(data_loader, model, optimizer):
    model.train()

    train_loss = 0

    for batch, (sample, label, _) in enumerate(data_loader):
        # ... (input preprocessing)

        recon_batch, mean, logvar = model(sample)
        loss = model.loss_function(recon_batch, sample, mean, logvar)

        optimizer.zero_grad()
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

        # ... (display and memory management operations)
```

```
18
19      train_loss /= len(data_loader.dataset)
20      return train_loss
21
22
23  def test(data_loader, model, name):
24      model.eval()
25
26      test_loss = 0
27
28      with torch.no_grad():
29          for sample, label, _ in data_loader:
30              # ... (input preprocessing)
31
32              pred, mean, logvar = model(sample)
33              loss = model.loss_function(pred, sample, mean, logvar)
34
35              test_loss += loss
36
37              # ... (display operations)
38
39      test_loss /= len(data_loader)
40
41      return test_loss
42
43  def run():
44      epoch = 0
45
46      # ... (display and early stopping initializations)
47
48      while True: # actual code had early stopping checking
49          # ... (display operations)
50          train_loss = train_step(dataloader_train, model, optimizer)
51          loss = test(dataloader_test, model, f"{CONV_LAYERS}_{epoch}")
52
53          epoch += 1
54
55      # ... (display and memory management operations)
```

The training process iteratively updates the model with data batches. In each training step, the model is optimized with a loss function that combines BCE and KL divergence terms. The function encourages the model to produce accurate reconstructions and learn a significant latent space.

D.3 Results

As an illustration of the VAE's potential, this section exhibits two images: one featuring an initial audio waveform and the other displaying its reconstructed version produced by the trained VAE.

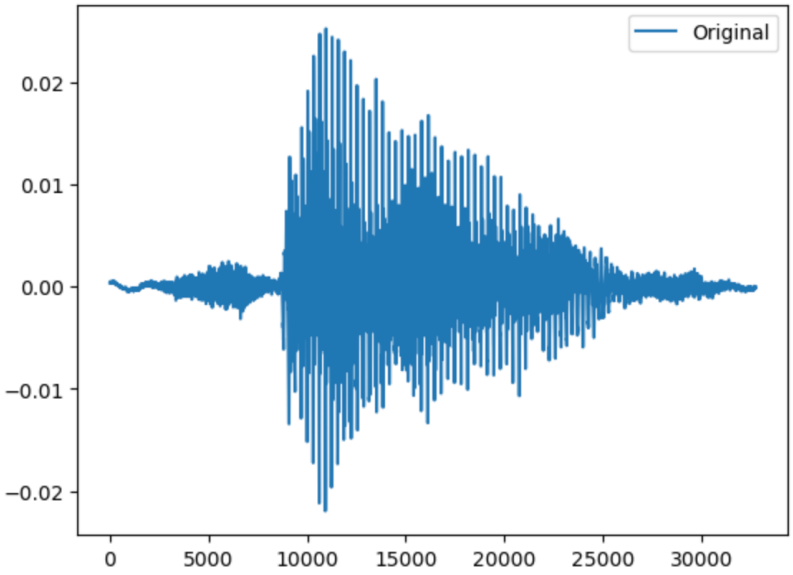


Figure D.1: Original Sample

These images represent the best reconstruction achievable by the author, considering the hardware and time limitations.

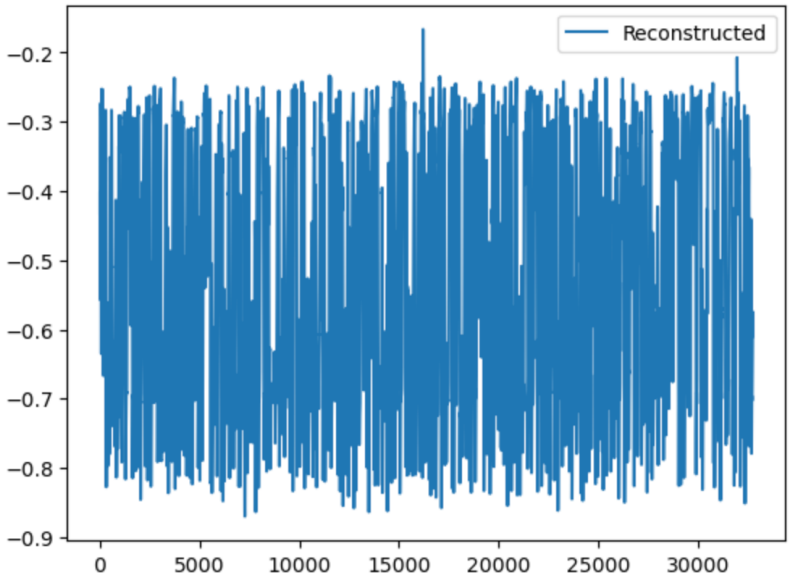


Figure D.2: Reconstructed Sample

Appendix E

GANmix Implementation Details

E.1 Model Implementation

```
class GANMix():

    # ... (initialization code)

    @classmethod
    def build(cls):
        latents_shape = (8, 128, 107)

        num_workers = torch.cuda.device_count()

        vae = models.VAE()

        netG = Generator(config.GENERATOR_INPUT_SIZE, 200, latents_shape)
        netD = Discriminator(latents_shape, 200, config.GAUSSIAN_NOISE)

        # ... (parallelization and GPU)

        criterion = BCEWithLogitsLoss()

        optimizerD = optim.Adam(netD.parameters(
        ), lr=config.LEARNING_RATE_DISCRIMINATOR, betas=(config.BETA1, 0.999))
        optimizerG = optim.Adam(netG.parameters(
        ), lr=config.LEARNING_RATE_GENERATOR, betas=(config.BETA1, 0.999))

        return cls(vae, netG, netD, optimizerG, optimizerD, criterion,
            num_workers)

class Generator(nn.Module):

    # ... (initialization)
```

```
30
31      def forward(self, x):
32          x = x.to(config.DEVICE)
33
34          x = self.network(x)
35
36          return x
37
38      def _build_network(self):
39          network = nn.Sequential(
40              nn.Linear(self.input_size, self.hidden_size),
41              nn.LeakyReLU(0.2),
42              nn.Linear(self.hidden_size, np.prod(self.output_shape)),
43          )
44          return network
45
46      def forward(self, x):
47          x = x.to(config.DEVICE)
48
49          x = self.network(x)
50          x = x.view(-1, *self.output_shape)
51
52          return x
53
54  class Discriminator(nn.Module):
55
56      # ... (initialization)
57
58      def forward(self, x):
59          x = x + torch.randn_like(x) * self.gaussian_noise_std
60
61          x = self.network(x)
62
63          return x
64
65      def _build_network(self):
66          network = nn.Sequential(
67              nn.Flatten(),
68              nn.Linear(np.prod(self.input_shape), self.hidden_size),
69              nn.LeakyReLU(0.2),
70              nn.Linear(self.hidden_size, 1),
71              nn.Tanh()
72          )
73          return network
```

Listing E.1: Implementation of the GANmix model.

The GANmix model is a GAN that combines a VAE to generate realistic audio from latent vectors.

E.1.1 VAE

The VAE is a neural network that encodes an input spectrogram into a latent vector and then decodes it back into an output spectrogram.

The VAE used in this model is AudioLDM's [74] VAE, which is already trained. It has an encoder and a decoder, both of which are CNNs with residue blocks. The encoder outputs two vectors: the latent vector's mean and logarithmic variance. The decoder takes a latent vector as input and outputs an image.

E.1.2 Generator

The generator is a neural network that inputs a latent vector and outputs a set of embeddings corresponding to the VAE embedding space. The generator is trained to fool the discriminator into believing that the generated embeddings are real.

The generator used in GANmix is implemented by the class `Generator`. The generator has three attributes: `input_size`, `hidden_size`, and `output_shape`. The `input_size` is the dimension of the latent vector, which is stored in the config file and is 100 (this and all values can be seen in the appendix F. The `hidden_size` is the dimension of the hidden layer, which in this case is 200. The `output_shape` is the shape of the embeddings, in this case $(8, 128, 107)$.

The generator has a method called `_build_network()` that returns a sequential network consisting of three layers: a linear layer, a leaky ReLU activation function, and another linear layer. The first linear layer maps the input vector to the hidden layer. The second linear layer maps the hidden layer to the output vector, which has the same size as the product of the output form.

The generator also has a `forward()` method that takes an input vector `x` and passes it through the network. It then transforms the output vector into the shape of the VAE embedding space.

E.1.3 Discriminator

The discriminator is a neural network that takes a set of embeddings as input and outputs a value indicating how real or fake the image is. The discriminator is trained to distinguish between real embeddings from the VAE and fake embeddings from the generator. The discriminator can also be used to evaluate how realistic the generated embeddings are by calculating their scores.

The discriminator used in this thesis is implemented using the class `Discriminator`. The discriminator has three attributes: `input_shape`, `hidden_size`, and `gaussian_noise_std`. The `input_shape` is the shape of the input embeddings. The `hidden_size` is the size of the hidden layer. The `gaussian_noise_std` is the standard deviation of the Gaussian noise added to the input image, which in this case is 0.1, no fine-tuning was performed.

The discriminator has a method called `_build_network()` that returns a sequential network consisting of five layers: a flatten layer, a linear layer, a leaky ReLU activation function, another

linear layer, and a tanh activation function. The flatten layer flattens the input image into a vector. The first linear layer maps the input vector to the hidden layer. The second linear layer maps the hidden layer to the output scalar.

The discriminator also has a method called `forward()`, which takes an input image `x` and adds Gaussian noise with standard deviation `gaussian_noise_std` to it. Then, it passes it through the network and returns the output scalar.

E.1.4 GANMix

The GANMix class is a wrapper class that contains all of these models. Plus the loss criterion and optimizers

E.2 Training Implementation

```
# ... (initialization)

def _gen_fake_samples(generator, num_samples):
    noise = torch.randn(num_samples, config.GENERATOR_INPUT_SIZE, device=config
        .DEVICE)
    fake_samples = generator(noise)
    return fake_samples

def _embed_samples(vae, samples):
    embeddings = vae.encode(samples)
    embeddings = embeddings.latent_dist.mode()
    embeddings = torch.nan_to_num(embeddings, nan=0)
    return embeddings

def _train_discriminator(data, ganmix, settings):
    with autocast():
        real_data = data.to(config.DEVICE)
        batch_size = real_data.size(0)

        real_embeddings = _embed_samples(ganmix.vae, real_data)

        fake_embeddings = _gen_fake_samples(ganmix.generator, batch_size)

        ganmix.discriminator_optimizer.zero_grad()

        prediction_real = ganmix.discriminator(real_embeddings)
        prediction_fake = ganmix.discriminator(fake_embeddings.detach())

        real_label = torch.ones(batch_size, 1, device=config.DEVICE)
        fake_label = -torch.ones(batch_size, 1, device=config.DEVICE)

```

```
31          loss_real = ganmix.criterion(prediction_real, real_label)
32          loss_fake = ganmix.criterion(prediction_fake, fake_label)
33
34          loss_discriminator = loss_real + loss_fake
35
36      settings.scaler.scale(loss_discriminator).backward()
37      settings.scaler.step(ganmix.discriminator_optimizer)
38      settings.scaler.update()
39
40      return loss_discriminator
41
42  def _train_generator(ganmix, settings):
43      with autocast():
44
45          fake_data = _gen_fake_samples(ganmix.generator, config.BATCH_SIZE)
46
47          ganmix.generator_optimizer.zero_grad()
48
49          prediction_fake = ganmix.discriminator(fake_data)
50
51          fake_label = torch.ones(config.BATCH_SIZE, 1, device=config.DEVICE)
52
53          loss_generator = ganmix.criterion(prediction_fake, fake_label)
54
55      settings.scaler.scale(loss_generator).backward()
56      settings.scaler.step(ganmix.generator_optimizer)
57      settings.scaler.update()
58
59      return loss_generator
60
61
62  # ... (output and display)
63
64
65  def run_train():
66
67      # ... (initialization)
68
69      with open(settings.stats_file_path, 'a', newline='') as csvfile:
70
71          # ... (display options)
72
73          for epoch in range(config.NUM_EPOCHS):
74              loss_discriminator_list = []
75              loss_generator_list = []
76
77              for data, quote in tqdm(settings.dataloader):
78                  loss_discriminator = _train_discriminator(data, ganmix,
                        settings)
```

```
79                  loss_discriminator_list.append(loss_discriminator.item())
80
81                  loss_generator = _train_generator(ganmix, settings)
82                  loss_generator_list.append(loss_generator.item())
83
84              _output_epoch_results(settings.start_time, epoch, ganmix.generator,
                      ganmix.vae, loss_discriminator_list, loss_generator_list,
                     csv_writer, csvfile)
85
86  def main():
87      run_train()
```

Listing E.2: Implementation of the training loop for the GANmix.

Training of the GANmix model is implemented using the `run_train()` and `main()` functions. Training involves several steps: data loading, model building, loss calculation, optimization, and output display.

E.2.1 Data Loading

The data loading step is responsible for loading the dataset of audios and quotes and creating a data loader that iterates over the data batches. The data set and the data loader are stored in the `settings` object, which is an instance of the `Settings` class. The `Settings` class is defined in another module and contains various parameters and paths for the training.

The data loader is created using PyTorch's `torch.utils.data.DataLoader` class. The data loader takes the data set as an argument and returns batches of data with a specified batch size. The batch size used in this model was 8. The data loader also shuffles the data and supports multiprocessing.

E.2.2 Model Building

The model build step is responsible for creating an instance of the `GANMix` model and moving it to the GPU, if available.

E.2.3 Loss Calculation

The loss computation step is responsible for computing the losses for the generator and the discriminator using the criterion and the predictions from the model.

The loss calculation consists of two sub-steps: generating and embedding dummy samples.

E.2.3.1 Generate Dummy Samples

The generate dummy samples substep generates dummy embeddings from random latent vectors using the generator.

This sub-step is implemented using the `_gen_fake_samples()` function, which takes the generator and the number of samples to generate as arguments and returns the dummy images as a tensor.

The `_gen_fake_samples()` function performs the following steps:

1. Sample random latent vectors from a standard normal distribution using PyTorch's `torch.rand()` function.
2. Generate dummy images from the latent vectors using the generator's `forward()` method.

E.2.3.2 Embedding Samples

The embedding samples sub-step is responsible for embedding real spectrograms into latent vectors using the VAE's encoder.

This sub-step is implemented using the `_embed_samples()` function, which takes the VAE and the samples to embed as arguments and returns the latent vectors as a tensor.

The `_embed_samples()` function performs the following steps:

1. Encodes the samples using the `encode()` method of the VAE.
2. Extract the mode of the latent vector from the `latent_dist` attribute.

The loss computation step also consists of two main steps: training the discriminator and training the generator.

E.2.3.3 Discriminator Training

The discriminator training step is responsible for updating the discriminator parameters using the discriminator optimizer and the discriminator loss. The discriminator loss is calculated by comparing the discriminator predictions for the real and fake emails with the real and fake labels.

The discriminator training step is implemented using the `_train_discriminator()` function, which takes the data, the ganmix model, and the settings object as arguments and returns the discriminator loss as a scalar.

The `_train_discriminator()` function performs the following steps:

1. Move the real data to the GPU using the `to()` method.
2. Get the batch size from the real data using the `size()` method.
3. Embed the real data into latent vectors using the `_embed_samples()` function.
4. Generate fake embeddings from random latent vectors using the `_gen_fake_samples()` function.

5. Compute the discriminator predictions for the real and fake embeddings using the discriminator's `forward()` method.
6. Compute the discriminator losses for the real and fake predictions using the criterion's `forward()` method.
7. Compute the total discriminator loss by adding the real and fake losses.

E.2.3.4 Training the Generator

The generator training step updates the generator parameters using the generator optimizer and the generator loss. The generator loss is calculated by comparing the discriminator predictions for the fake images with the real labels.

The generator training step is implemented using the `_train_generator()` function, which takes the ganmix model and the settings object as arguments and returns the generator loss.

The `_train_generator()` function performs the following steps:

1. Generate fake data from random latent vectors using the `_gen_fake_samples()` function.
2. Compute discriminator predictions for the fake data using the discriminator's `forward()` method.
3. Generate the real labels using PyTorch's `torch.ones()` function.
4. Compute the generator loss by comparing the fake predictions and the real labels using the criterion's `forward()` method.

E.2.4 Training Loop

For the specified number of epochs, the program analyzes the entire data set, first training the discriminator and then training the generator. The corresponding losses for both are noted and displayed by the program.

Appendix F

GANmix Model Configuration and Parameters

This annex contains the details of the final GANmix model. The GANmix model is a GAN that uses fully connected neural networks for both the generator and the discriminator, and leverages a pretrained VAE for generating and discriminating audio embeddings.

Table F.1: GANmix model parameters

Parameter	Value
Number of trainable parameters (generator)	22043368
Number of trainable parameters (discriminator)	21914001
Total number of trainable parameters	43957369
Total number of parameters	99334242
Size of the noise vector (NZ)	100
Size of the hidden layer (NH)	200
Generator learning rate	0.001
Discriminator learning rate	0.0001
Generator scheduler step size	10
Generator scheduler gamma	0.1
Discriminator scheduler step size	10
Discriminator scheduler gamma	0.1
Batch size	8

Table F.2: AudioLDM's VAE model encodings

Encoding dimension	Value
Number of dimensions (ED)	8
Width (EW)	128
Height (EH)	107

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                    [8, 200]          20,200
         LeakyReLU-2                    [8, 200]               0
            Linear-3                 [8, 109568]      22,023,168
================================================================
Total params: 22,043,368
Trainable params: 22,043,368
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 6.71
Params size (MB): 84.09
Estimated Total Size (MB): 90.80
----------------------------------------------------------------
```

Listing F.1: Generator summary

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Flatten-1                 [8, 109568]               0
            Linear-2                    [8, 200]      21,913,800
         LeakyReLU-3                    [8, 200]               0
            Linear-4                      [8, 1]             201
              Tanh-5                      [8, 1]               0
================================================================
Total params: 21,914,001
Trainable params: 21,914,001
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 3.34
Forward/backward pass size (MB): 6.71
Params size (MB): 83.60
Estimated Total Size (MB): 93.65
----------------------------------------------------------------
```

Listing F.2: Discriminator summary

This annex provides the configuration and parameters of the GANmix model that was used to generate audio embeddings from random noise and discriminate them from real audio embeddings.

Appendix G

VAMOS

Contained in this appendix are important insights into the architectural foundations of the VAMOS model, along with three key components that are central to its effectiveness. These components include the *Text Encoder*, which uses BERT's encoder for transforming textual data, the *ResNet (Audio Encoder)* designed for audio processing tasks, and the *CLAP* model for creating joint audio-text embeddings. Furthermore, this text presents an implementation and analysis of the *U-Net* architecture, which is known for its superior segmentation capabilities. All components of the model are accompanied by their code and academic explanation to highlight their importance in the overall framework.

G.1 Text Encoder

```
def encode(text):
    # Pass text through the BERT tokenizer
    input_ids = tokenizer(text, add_special_tokens=True)["input_ids"]
    input_tensor = torch.tensor([input_ids])

    # Pass the input tensor through the BERT encoder
    with torch.no_grad():
        encoded_output = self.model(input_tensor)[0]
        flatten_output = torch.flatten(encoded_output, start_dim=1)

    return flatten_output
```

Listing G.1: Text encoding function utilizing BERT tokenizer and encoder for semantic representation extraction.

The provided code excerpt describes the `encode` function, which plays an essential role in the developed model. This function utilizes a BERT model's encoder to convert input textual data into a structured numerical representation that encapsulates its semantic essence.

The function begins by calling a BERT tokenizer that translates human-readable text into an ordered sequence of discrete numerical identifiers. This conversion aligns with the established protocol for facilitating computational analysis and interpretation.

Afterward, the input tensor passes through the encoder module of the BERT model, which captures contextual dependencies within the text data. This process fosters a dynamic and enriched representation at the token level.

G.2 ResNet (Audio Encoder)

```
class ResNet(nn.Module):

    # ... (other initialization code)

    def build_resnet(self):
        if self.useBottleneck:
            filters = [64, 256, 512, 1024, 2048]
        else:
            filters = [64, 64, 128, 256, 512]

        self.layer1 = nn.Sequential()
        self.layer1.add_module('conv2_1', resblock(
            filters[0], filters[1], downsample=False))
        for i in range(1, self.repeat[0]):
            self.layer1.add_module('conv2_%d' % (
                i+1,), resblock(filters[1], filters[1], downsample=False))

        self.layer2 = nn.Sequential()
        self.layer2.add_module('conv3_1', resblock(
            filters[1], filters[2], downsample=True))
        for i in range(1, self.repeat[1]):
            self.layer2.add_module('conv3_%d' % (
                i+1,), resblock(filters[2], filters[2], downsample=False))

        self.layer3 = nn.Sequential()
        self.layer3.add_module('conv4_1', resblock(
            filters[2], filters[3], downsample=True))
        for i in range(1, self.repeat[2]):
            self.layer3.add_module('conv2_%d' % (
                i+1,), resblock(filters[3], filters[3], downsample=False))

        self.layer4 = nn.Sequential()
```

```
        self.layer4.add_module('conv5_1', resblock(
            filters[3], filters[4], downsample=True))
        for i in range(1, self.repeat[3]):
            self.layer4.add_module('conv3_%d' % (
                i+1,), resblock(filters[4], filters[4], downsample=False))

        self.gap = torch.nn.AdaptiveAvgPool2d(1)
        self.fc = torch.nn.Linear(filters[4], outputs)

    def forward(self, input):
        input = self.layer0(input)
        input = self.layer1(input)
        input = self.layer2(input)
        input = self.layer3(input)
        input = self.layer4(input).detach()
        input = self.gap(input)
        input = torch.flatten(input, start_dim=1)
        input = self.fc(input)

        return input

class ResBlock(nn.Module):

    def __init__(self, in_channels, out_channels, downsample):

        # ... (other initialization code)

        # Define convolutional layers and batch normalization
        if downsample:
            # If downsample is True
            # apply convolution that reduces the size and apply a shortcut
            self.conv1 = nn.Conv2d(
                in_channels, out_channels, kernel_size=3, stride=2, padding=1)
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2),
                nn.BatchNorm2d(out_channels)
            )
        else:
            # If downsample is False, apply convolution with stride 1 and no
            # shortcut path
            self.conv1 = nn.Conv2d(
                in_channels, out_channels, kernel_size=3, stride=1, padding=1)
            self.shortcut = nn.Identity()

        # Define additional convolutional layers and batch normalization
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
```

```
82
83      def forward(self, input_tensor):
84          # Apply the shortcut path to the input
85          shortcut = self.shortcut(input_tensor)
86          shortcut = shortcut.detach()
87
88          # Apply the first convolutional layer and ReLU activation, followed by
89          # batch normalization
90          x = nn.ReLU()(self.bn1(self.conv1(input_tensor)))
91          x = x.detach()
92
93          # Apply the second convolutional layer and ReLU activation, followed by
94          # batch normalization
95          x = nn.ReLU()(self.bn2(self.conv2(x)))
96          x = x.detach()
97
98          # Add the shortcut to the output of the second convolutional layer
99          output_tensor = x + shortcut
100
101         # Apply ReLU activation to the output and return it
102         return nn.ReLU()(output_tensor)
```

Listing G.2: Residual Network (ResNet) architecture for audio processing with custom residual block implementation.

The provided code segment introduces a customized neural network architecture based on the ResNet framework, specifically designed for audio processing. The architecture includes a variety of key components, including the use of residual blocks that are critical to constructing the ResNet layers. The explanation then provides a concise overview of the overall structure and operation of the code.

At the core of this demonstration is the `ResNet` class. This class includes methods for assembling the architecture and managing data flow through the network. This class includes methods for assembling the architecture and managing data flow through the network. A key method is `build_resnet`, which creates the network by stacking residual blocks. This assembly follows the architectural configuration principles outlined by the specified guidelines.

Additionally, the `ResBlock` class serves as a fundamental construct that houses a single residual block- revered for its role as a building block within the larger ResNet architecture. Within this enclosed domain, the architectural formulation of the residual block is contextualized.

G.3 CLAP

```
1  class Clap(nn.Module):
2      def __init__(self,
```

```
                audio_feature_dim,
                text_feature_dim,
                shared_embedding_dim
                ):
        # ... (other initialization code)

        # Initialize the weights that will connect the input audio and text
        # features to the shared embedding space.
        self._audio_projection = torch.nn.Linear(
            audio_feature_dim, shared_embedding_dim)
        self._text_projection = torch.nn.Linear(
            text_feature_dim, shared_embedding_dim)

        # Initialize the temperature parameter used for scaling the pairwise
        # cosine similarities between image and text embeddings.
        self._learned_temperature = torch.nn.Parameter(torch.tensor([1.0]))

    def encode_audio(self, audio_features):
        # Project the audio features into the shared embedding space and
        # normalize the resulting embedding vectors.
        audio_embeddings = F.normalize(
            self._audio_projection(audio_features), p=2, dim=1)
        return audio_embeddings

    def encode_text(self, text_features):
        # Project the text features into the shared embedding space and
        # normalize the resulting embedding vectors.
        text_embeddings = F.normalize(
            self._text_projection(text_features), p=2, dim=1)
        return text_embeddings

    def forward(self, audio_features, text_features):
        # Encode the audio and text features into their respective embedding
        # spaces.
        audio_embeddings = self.encode_audio(audio_features)
        text_embeddings = self.encode_text(text_features)

        # Compute the pairwise cosine similarities between the image and text
        # embeddings, scaled by the learned temperature parameter.
        pairwise_similarities = torch.matmul(
            audio_embeddings, text_embeddings.T) * torch.exp(
            self._learned_temperature)

        # Compute the symmetric cross-entropy loss between the predicted
        # pairwise similarities and the true pairwise similarities.
        labels = torch.arange(audio_features.size(0))
        loss_i = F.cross_entropy(
            pairwise_similarities, labels, reduction='mean')
        loss_t = F.cross_entropy(
```

```
            pairwise_similarities.T, labels, reduction='mean')
        loss = (loss_i + loss_t) / 2

        return loss
```

Listing G.3: CLAP model architecture for joint audio-text embeddings with pairwise cosine similarity-based alignment.

The presented code introduces a class called `Clap`. It is a neural network model designed to acquire shared embeddings for both audio and text input data. The model aims to align audio and text features in a shared embedding space by utilizing pairwise cosine similarities.

The essence of the code is explained in further detail below. The initialization of the model includes parameterizing the primary dimensions: audio and text feature dimensions, coupled with the dimensionality of the embedding space for their alignment.

The model's architecture incorporates linear projection layers for audio and text input streams, as well as a temperature parameter. The learned temperature parameter is useful in calibrating pairwise cosine similarities by incorporating a scaling factor to the magnitudes.

The `forward` method represents the procedural logic for the model's forward pass. This includes incorporating audio and text input features into their respective embedding spaces, accomplished through the use of the `encode_audio` and `encode_text` methods. An important step is the subsequent calculation of pairwise cosine similarities using matrix multiplication, which reflects the underlying relationships between audio and text embeddings. The use of the acquired temperature parameter adjusts the importance of these pairwise similarities. Based on this foundation, the algorithm's optimization is enhanced by calculating a symmetric cross-entropy loss that takes into account audio-to-text as well as text-to-audio relationships. The final loss metric directing the model's training trajectory is achieved by averaging these dual losses.

G.4 U-Net

```
class UNet(nn.Module):

    # ... (other initialization code)

    def forward(self, x):
        x = x.to(self.device)

        # implements the forward pass with concatenations
        skip_connections = []

        # contracting path
        for block in self._contracting_path:
```

```
            # create a sequential block from the list of layers
            net = nn.Sequential(*block).to(self.device)

            # apply
            x = net(x)

            # save the skip connection
            skip_connections.append(x)

        # bottleneck
        x = self._bottleneck(x)

        # expanding path
        for layer_idx in range(self.depth - 1):
            # the first layer is a transposed convolutional layer
            transposed_conv = self._expanding_path_layers[layer_idx * 2].to(
                self.device)
            x = transposed_conv(x)

            # concatenate the skip connection
            skip_connection = skip_connections.pop()
            # make sure the shapes match
            if x.shape != skip_connection.shape:
                # resize the skip connection
                skip_connection = nn.functional.interpolate(skip_connection,
                                                            size=x.shape[2:],
                                                            mode='nearest')

            # concatenate the skip connection
            x = torch.cat((x, skip_connection), dim=1)

            # the second layer is a sequential block of convolutional layers
            conv_block = self._expanding_path_layers[layer_idx * 2 + 1].to(
                self.device)
            x = conv_block(x)

        # final convolutional layer
        x = self._expanding_path_layers[-1].to(self.device)(x)

        return x

    def _make_contracting_path(self):
        """
        Create the contracting path of the U-Net.
        """
        layers = []

        # configs the used convolutional layers
        in_channels = self.in_channels
```

```
62          out_channels = 64
63
64          # create a convolutional block for each number in depth
65          for _ in range(self.depth - 1):
66              # create a convolutional block
67              block = []
68
69              # create the number of convolutional layers specified by
70              # conv_layers_per_block
71              for _ in range(self.conv_layers_per_block):
72                  block.append(nn.Conv2d(in_channels=in_channels,
73                               out_channels=out_channels, kernel_size=3, padding=
                                     "same"))
74                  # batch normalization
75                  block.append(nn.BatchNorm2d(out_channels))
76                  # ReLU activation
77                  block.append(nn.ReLU())
78
79                  # update the in_channels
80                  in_channels = out_channels
81
82              # add a max pooling layer
83              block.append(nn.MaxPool2d(kernel_size=2))
84
85              # add the block to the layers
86              layers.append(block)
87
88              # double the number of channels
89              out_channels *= 2
90
91          # create a network from the layers
92          return layers
93
94      def _make_bottleneck(self):
95          """
96          Create the bottleneck of the U-Net.
97          """
98          # build the bottleneck
99          layers = []
100
101         # config the bottleneck channels
102         in_channels = 64 * (2 ** (self.depth - 2))
103         out_channels = in_channels * 2
104
105         # create the number of convolutional layers specified by
106         # conv_layers_per_block
107         for _ in range(self.conv_layers_per_block):
108             layers.append(nn.Conv2d(in_channels=in_channels,
109                                     out_channels=out_channels,
```

```
                                    kernel_size=3,
                                    padding="same"))
            # batch normalization
            layers.append(nn.BatchNorm2d(out_channels))
            # ReLU activation
            layers.append(nn.ReLU())

            # update the in_channels
            in_channels = out_channels

        # add all the layers in block in the layers list
        return nn.Sequential(*layers).to(self.device)

    def _make_expanding_path(self):
        """
        Create the expanding path of the U-Net.
        This returns a list of layers, which will be used in the forward pass.
        Some layers are transposed convolutional layers, others are sequential
            blocks of convolutional layers.
        """
        layers = []

        # configs the used convolutional layers
        # the number of in channels is the number of out channels from the last
        # block in the contracting path
        in_channels = 64 * (2 ** (self.depth - 1))
        out_channels = in_channels // 2

        # create a convolutional block for each number in depth
        for _ in range(self.depth - 1):
            # add an up conv 2x2
            layers.append(nn.ConvTranspose2d(in_channels=in_channels,
                                             out_channels=out_channels,
                                             kernel_size=2,
                                             stride=2))

            block = []

            # create the number of convolutional layers specified by
            # conv_layers_per_block
            for _ in range(self.conv_layers_per_block):
                block.append(nn.Conv2d(in_channels=in_channels,
                             out_channels=out_channels, kernel_size=3, padding=
                                 "same"))
                # batch normalization
                block.append(nn.BatchNorm2d(out_channels))
                # ReLU activation
                block.append(nn.ReLU())

```

```
158                 # update the in_channels
159                 in_channels = out_channels
160
161             # add the block to the layers
162             layers.append(nn.Sequential(*block))
163
164             # double the number of channels
165             out_channels //= 2
166
167         # final convolutional layer
168         final_layer = []
169         final_layer.append(nn.Conv2d(in_channels=in_channels,
170                                      out_channels=self.out_channels,
171                                      kernel_size=1,
172                                      padding="same"))
173
174         final_layer.append(nn.Softmax(dim=1))
175
176         layers.append(nn.Sequential(*final_layer))
177
178         # create a network from the layers
179         return layers
```

Listing G.4: U-Net architecture for semantic segmentation

The code provides insight into the U-Net architecture, renowned for its effectiveness in segmentation tasks. It includes a class called `UNet`, which embodies the architectural blueprint of the U-Net model. Its structure is composed of three primary components - the contracting path, the bottleneck layer, and the expanding path - which are crucial to the model's implementation. The forward pass follows the characteristic U-shaped configuration. This architectural layout is used to create the favorable integration of skip connections, which enhances the capacity for precise segmentation.

Appendix H

GANmix Results Table

This appendix presents the results of experiments conducted with the GANmix model.

The experiments were designed to evaluate the performance of the GANmix model on two datasets: Audio MNIST and Clotho. The experiments also varied the number of parameters, the loss function, the regularization scheme, the dataset, and the optimizer used to train the GANmix model. The results are summarized in two tables: Table H.1 shows the experimental settings and H.2 shows the corresponding losses for the generator, the discriminator, and the total loss. Losses are calculated using BCE.

Table H.1: Experimental GANmix results (Part 1).

Exp.	#Params	Loss	Regularization	Dataset	Optimizer
1	∼4M even	BCE	FALSE	Audio MNIST	Adam
2	∼4M even	BCE	FALSE	Audio MNIST	Adam
3	∼2M discriminator, ∼25M generator	BCE	FALSE	Audio MNIST	Adam
4	∼2M discriminator, ∼25M generator	BCE	FALSE	Audio MNIST	SGD
5	∼2M discriminator, ∼25M generator	BCE + Elastic Net	TRUE	Audio MNIST	Adam
6	∼4M even	BCE + Elastic Net	TRUE	Clotho	Adam
7	∼50M even	BCE + Elastic Net	TRUE	Clotho	Adam
8	∼50M even	BCE	FALSE	Clotho	Adam
9	∼50M even	BCE	TRUE	Clotho	Adam
10	∼50M even (linear)	BCE	TRUE	Clotho	Adam

Table H.2: Experimental GANmix results (Part 2).

Exp.	**HW**	**LR Generator**	**LR Discriminator**	**Gen Loss**	**Dis Loss**	**Total Loss**
1	Kaggle	1.00E-04	1.00E-04	0.487	1.440	1.927
2	Kaggle	1.00E-02	1.00E-02	0.545	1.412	1.957
3	Kaggle	1.00E-03	1.00E-04	0.351	1.686	2.037
4	LIACC 1	1.00E-03	1.00E-04	0.516	1.429	1.945
5	LIACC 1	1.00E-03	1.00E-04	3.362	1.428	4.791
6	LIACC 2	1.00E-03	1.00E-04	0.587	1.458	2.045
7	LIACC 2	1.00E-03	1.00E-04	1.738	1.439	3.177
8	LIACC 2	1.00E-03	1.00E-04	0.693	1.139	1.832
9	LIACC 2	1.00E-03	1.00E-04	0.693	1.007	1.700
10	LIACC 2	1.00E-03	1.00E-04	6.987	-0.562	6.425