FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Synthesizing Soundscapes from Textual Input

Development and Comparison of Generative AI Models

Márcio Duarte

WORKING VERSION

Mestrado em Engenharia Informática e Computação

Supervisor: Luís Paulo Reis

Second Supervisor: Gilberto Bernardes

March 7, 2023

Resumo

Hoje em dia, o áudio é um elemento essencial na maioria do conteúdo produzido. Normalmente, é trabalho manual que está por detrás dos terabytes de áudio publicados diariamente. Se algum editor desejar um som específico, tem de o pesquisar em bases de dados online, sintetizá-lo ou até gravá-lo ele mesmo. 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 — cujos modelos de geração estão a tornar-se comuns — apresentam três dimensões de dados. Independentemente disso, as dependências a longo prazo 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 estuda modelos estado da arte de inteligência artificial generativa e propõe e acompanha a implementação de um sistema capaz de sintetizar trechos de áudio a partir de entradas textuais . A avaliação é feita através, entre outras técnicas, da divergência KL. O sistema proposto não limita estas amostras de áudio a um domínio como a música ou a fala, mas aceita qualquer solicitação textual. A dissertação centra-se no desenvolvimento e na comparação de várias arquitecturas de modelos com base nos sistemas mais avançados.

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 constitutes a central part of most generated content. Usually, manual labor is behind the terabytes of audio published daily. If some creator desires a specific sound, they must research it under online databases, synthesize it, or even record it themselves. This amount of work is a barrier to content creation, mainly if such sounds are elaborate or too specific.

Audio files present solely one dimension, *i.e.*, the amplitude of its sound wave collected at a designated rate interval. In comparison, image files — whose generator models are becoming mainstream — present three data dimensions. Regardless, long-term dependencies convey a challenge in sound, as they are more complex and intricate than those of images. For instance, there is the expectation that the timbre of a given instrument or the background noise of a given sonic environment remains over time.

This dissertation studies state-of-the-art generative AI models and proposes and follows the implementation of a system capable of synthesizing audio snippets with quality through textual input. The quality is measured through, amidst other techniques, KL divergence. The proposed system does not confine these audio samples to a domain like music or speech but accepts any textual prompt. The dissertation focuses on developing and comparing various model architectures based on state-of-the-art systems.

Given that meaningful work on models that create images or perform text-to-speech already exists, the expectation is that an end-to-end audio generative model produces satisfactory outcomes. This work's results will impact the speed of the production process for content creators, sound engineers, and everyone interested in creating audio outputs.

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

Fyodor Mikhailovich Dostoyevsky

Todo list

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AE autoencoder

AI artificial intelligence

API Application Programming Interface

AR autoregressive

BCE binary cross-entropy

BPE byte pair encoding

CNN convolutional neural network

DARN deep autoregressive network

DCGAN deep convolutional generative adversarial network

DCNN deconvolutional neural network

DFT Discrete-Fourier-Transform

DL deep learning

DNN deep neural network

dVAE discrete variational autoencoder

ELBO evidence lower bound

GAN generative adversarial network

GLIDE Guided Language to Image Diffusion for Generation and Editing

GPU graphics processing unit

GRU gated recurrent unit

Hz Hertz

KL Kullback–Leibler

KLD Kullback–Leibler divergence

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

NLP natural language processing

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

In the digital age, audio is crucial in shaping human experiences and interactions. From podcasts and music to sound effects and immersive environments, audio content enriches our lives and enhances multimedia experiences. However, generating high-quality, diverse, and contextually relevant audio remains a time-consuming and labor-intensive task. As the demand for audio content continues to grow, there is a pressing need for innovative solutions to streamline the process and empower creators to produce a wide range of sounds with minimal effort.

The current advancements in artificial intelligence (AI) and deep learning (DL) are revolutionizing various domains, including image generation and text-to-speech synthesis. These state-of-the-art models demonstrate remarkable capabilities in generating high-quality content from simple textual inputs. Inspired by these successes, this thesis explores the potential of AI-driven generative models for synthesizing audio snippets based on textual descriptions. The proposed system is not limited to specific domains, such as music or speech, but is designed to accommodate a diverse 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 is present in 1.1. This section discusses the context of the research study. Its focus is to provide the background and context and explain the research's importance. The section 1.2 explores the motivation for the study. The objective of this section is to explain why the research is necessary and what gap in the knowledge it aims to fill. Then, in 1.3, the objectives of the research study are outlined. The focus of this section is to explain the specific goals and outcomes the research aims to achieve. Finally, in section 1.4, the structure of the dissertation is presented. This section provides an organized overview of the remaining chapters.

1.1 Context

"Computer Science is the study of computation and information" [100]. In practice, most engineers and computer scientists write programs that machines execute. Traditionally, these programs are not more than a series of instructions for the computer to follow.

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 [64]. ML is a subset of computer science that uses algorithms and statistical models to enable computers to learn and make decisions or predictions based on data. Modern artificial intelligence was born in 1958 when F. Rosenblatt [86], 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 [32]. However, in 1969, Minksy and Papert [61] published a book demonstrating the limitations of perceptrons. Namely, the authors showed that perceptrons could only represent linear functions, and simple 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* [32].

During the 1980s, studies of learning under multilayer neural networks went underway [32]. In 1986, Rumelhart et al. [88] 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. [32] 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*.

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., 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 [18]) 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 [35]. 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. 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 2010s saw the rise of DL in everyday applications. Everyone depends on DL for every computer-made task nowadays. Some examples include text translation, recommender systems, fake news detection or spam, image captioning, and self-driving cars.

To create new data, one uses generative models. Generative models are DL models that generate new synthetic data similar to a training dataset. These models learn the underlying distribution of the training data and can then generate new data points from that distribution. "Unlike more common learning objectives that try to discriminate labelled inputs (*i.e.* classification) or estimate a mapping (*i.e.* regression), generative models instead learn to replicate the hidden statistics behind observed data." [47]

The scientific community turned its heads to generative models from a few years prior. These models allow a variety of new applications and products. For instance, DALL-E [80] and DALL-E 2 [79] allow the generation of images given any textual input. GPT-3 is an extensive language model [11] that powers applications such as ChatGPT, capable of generating text. Modern text-to-speech applications also rely on these technologies.

Given the relevance and efficacy of these new generative technologies, DL gained a mainstream status, with the general public aware of the capabilities of these algorithms. If the 2010s saw an increase in artificial intelligence with predictions and classifications, it is not far-fetched that the 2020s will be a decade of generative applications. Enhancing human creativity by automating manual work has never been closer.

Sounds create the sonic fabric of the world. Broadly, sounds can be categorized as music, speech, or soundscapes. Music consists of organized tones and rhythms created by humans or other living entities since they require patterns . Speech encompasses all human vocalizations used for communication and expression . Soundscapes refer to an acoustic environment that includes natural and human-made sounds, as perceived, experienced, and understood by individuals, in context [1, 92]. These three categories cover many of the sounds encountered.

find citation

find citation

There are two types of soundscapes, the ones that exist in the real world and, thus, can be recorded, and the ones that can be generated artificially through, for instance, mathematical equations. Taking the latter, one can imagine sounds such as white or brown noise; they are no more than a repeated mathematical pattern and are easy to generate. To the best of the author's knowledge, the rest of the sounds are either recorded or created through sampling or some other operation to pre-recorded sounds. Computer-wise, generating new soundscapes has not been achieved without DL.

Deep generative models utilize neural networks to generate audio from parameters. These models discover latent data structures to generate new samples with the same distribution as the training

data [47]. They allow for the generation of more realistic audio than traditional generative models.

However, despite the vast interest in generative technologies, audio research still needs to catch up. While image generation has reached a dangerous level, and text generation can pass medicine exams, one can still quickly notice when a DL process generates a given sound. Most sound research is put into text-to-speech, and these technologies still need improvement. Apart from this, there is no tool for a general sound synthesis that resembles what state-of-the-art models do for images.

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.

However, despite the recent advancements in ML-based sound synthesis, there still needs to be comprehensive studies that explore the potential and limitations of these techniques.

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 already shaping our relationship with music and sound by enabling new possibilities [97]. This work aims to explore and expand this by providing a new tool that further enhances human capacity toward sound creation. Moreover, this study can serve as a valuable resource for researchers, developers, and practitioners in audio processing. The findings of this research help guide future research and development efforts in sound generation technologies and provide insights into the best practices and techniques for using ML for 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 make light of these motivation principles, this dissertation proposes implementing a system capable of taking any textual input and generating a sound from it.

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. Make a study of previous algorithms capable of handling sound for augmentation, feature extraction, or others.
3. Make a study of the current state-of-the-art architectures used to develop sounds artificially.
4. Develop multiple systems that handle sound, classification systems, vocoders, and others.
5. Develop end-to-end systems capable of taking any textual input and generating a sound from it.
6. Evaluate the systems' ability to generate a sound from the given textual input accurately.
7. Develop a practical application that uses the end-to-end sound generation system.
8. Provide a valuable contribution to deep learning and its applications.

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 next aspect of the dissertation deals with formulating the problem at hand and defining it mathematically. The practical applications of the developed technologies are then demonstrated, and the various decisions and consequences that arise in developing such a system are explored.

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 broadly reviews the relevant literature on DL and audio handling topics. It aims to give the reader a thorough understanding of the current state of knowledge in the field and its evolution. Current developments of tools for sound comprehension and generation are advancing existing models further, leading to the modeling of new sounds more efficiently and faster [97]. In a more precise way, the presence of this chapter serves several essential functions:

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 work places itself in a sea of research regarding DL. Although it proposes a new technology, it nevertheless relies on other researchers' work. This section will place this dissertation in this background. It is not responsible for explaining other technologies that resolve this and similar problems but instead explaining other technologies that are useful to solve this problem.

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.

With this, it will start by explaining how sound can be digitally processed in subsection 2.1.1. Then it will display general deep learning architectures and techniques in subsection 2.1.2. Subsection 2.1.3 shows essential techniques for the developed models. Then, for contextualization, subsection 2.1.5 will discuss some state-of-the-art models for data generation unrelated to audio — for example, image generators. Subsections are used to provide a hierarchical structure within sections.

2.1.1 Sound

An audio signal, usually known as a wave, represents the evolution of the sound pressure over time. Sound is a continuous phenomenon but can be discretized by collecting samples at a given time rate. The number of samples taken in one second is called the "sampling rate". The most common value for it is 44100 Hertz (Hz). The sampling rate directly affects the fidelity of the signal [26].

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 Hz needs to accommodate 1,411,200 bits per second [26].

2.1.1.1 Short-Time Fourier Transform

Even though sound media is technically 1D data [67], it can be transformed. By applying a Discrete-Fourier-Transform (DFT), the array can be represented in the frequency domain [74]. Given that 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 STFT equation is given by:

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

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 $X(m,\omega)$ 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, denoted by $w(n-mR)$, is used to isolate a particular 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 successive 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).

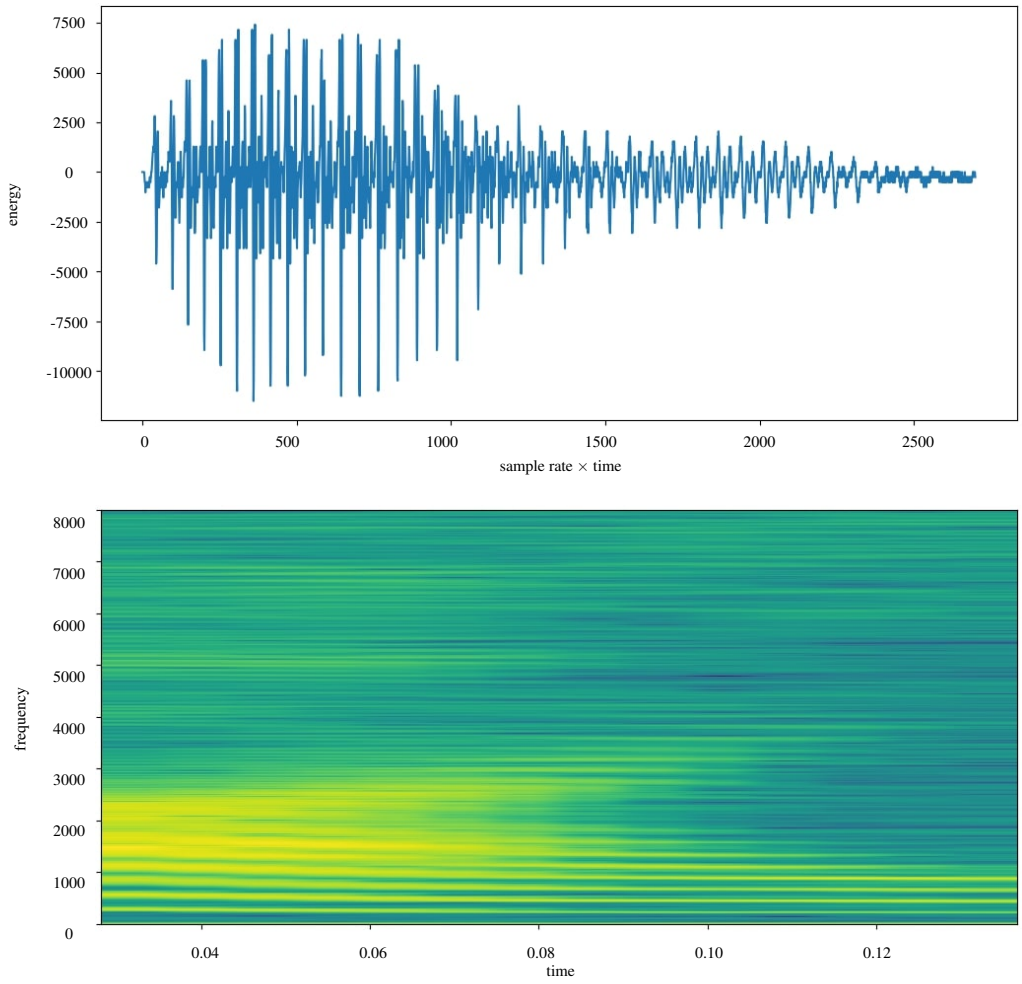

Figure 2.1: **Raw Wave vs. Spectrogram** — A comparative analysis of a sound sample from the Audio MNIST dataset (showcased in section 4.2.1.2), specifically entry number 9 of speaker Nicolas uttering the digit "five". The top plot illustrates the raw waveform with time (sample rate × time) on the X-axis 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.

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 [74]. 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 [74].

2.1.1.3 Soundscapes

The digitalization of sound has allowed for multiple applications and use cases. Music is the most popular form of media. 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 [1, 92]. 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 [12].

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 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 [74].

2.1.2 Deep Learning

As stated before, deep learning (DL) grew at the turn of the millennium with the need to handle large amounts of data. Furthermore, at its core, DL is precisely ML applied to large amounts of data. Accurately, DL is a sub-field of ML that uses many levels of information processing and abstraction for feature learning and representation [20]. These features are then used for classification, regression, and other models. Traditionally, these would be hand-picked features 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.

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

This section will describe breakthrough architectures for DL. It is important to notice that the date on which they were theorized differs from the date on which they became widely used. This

is a popular trend in ML as software theories have more rapid development than their hardware counterparts.

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 [39] 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 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 [86]. 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 a 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* [61], hence the need for more complex structures.

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 [17] 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.

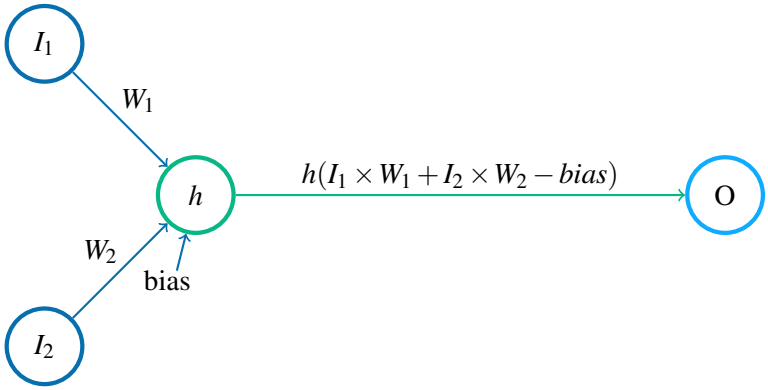

Figure 2.2: **Perceptron** — The perceptron receives multiple inputs associated with a weight. It also holds a given bias. Its output is the application of a given function (*e.g.* step function) to the weighted average of the inputs minus the bias.

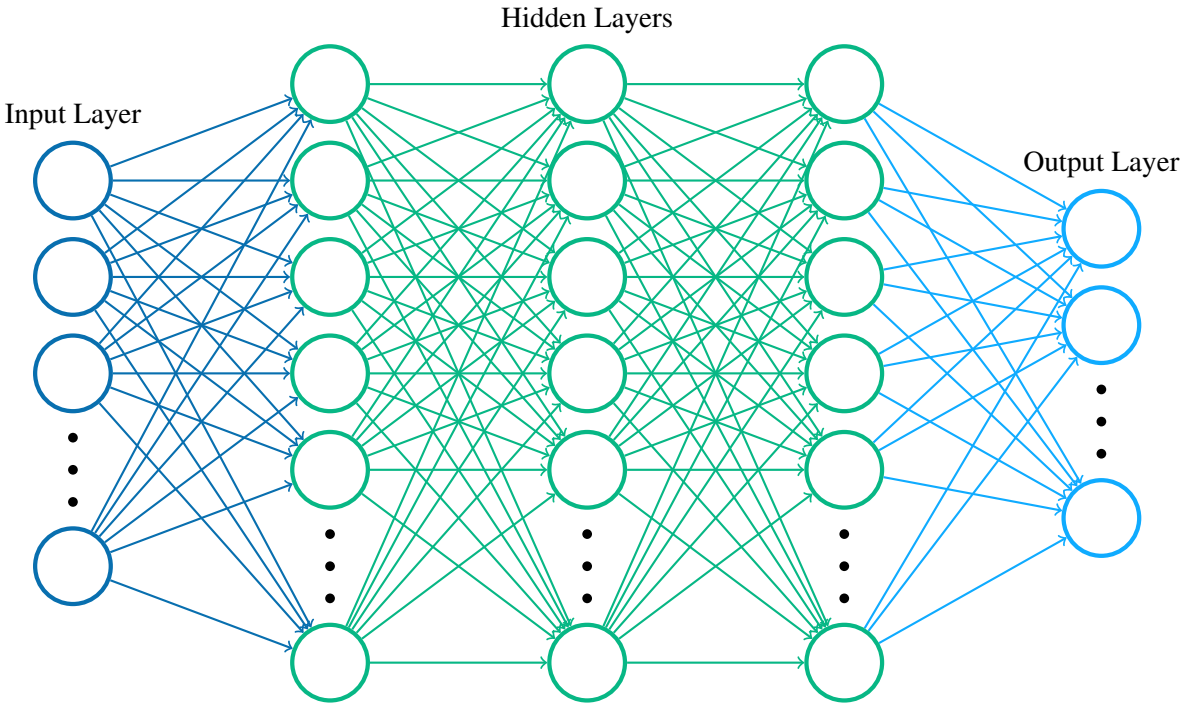

Figure 2.3: **Feedforward neural network** — The size of the input is constant. The flow of information goes through nonlinear functions on hidden layers. Both weights between nodes on different layers and the nodes' bias are trained with backpropagation.

In a sense, and given that the perceptron is the simplest feedforward neural network (a perceptron is a network with only one neuron), these networks are not necessarily synonyms of DL. Small networks exist and have been used for decades. However, most DL architectures derive from this; thus, its acknowledgment is fundamental.

Convolutional Neural Network (CNN) (1980) In the fifties and sixties, Hubel and Wiesel [46], neurophysiologists, showed that cat eyes had neurons that mapped multiple regions of what they

were seeing. These regions were called receptive fields. They also found that there are two basic cells in the brain regarding vision: simple cells whose output increases on edges, and complex cells, with larger receptive fields and the input does not care about the exact position of the edges but instead about their organization.

The first CNN was called *neocognitron* [33] and was presented by Kinihiko Fukushima in 1980.

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.

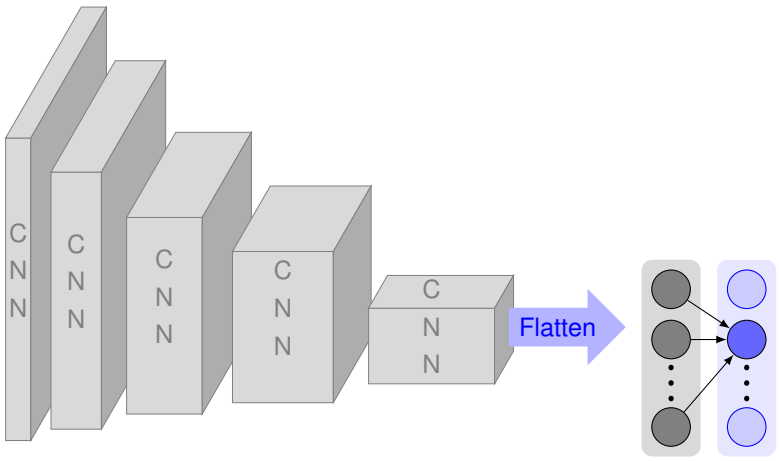

Figure 2.4: **Convolutional neural network (CNN)** — The network present in the figure deals with 2D data, such as images. The networks' first step consists of convolutional (and pooling) layers. This first step is responsible for finding features. The deeper the convolutional layer, the bigger the receptive 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.

A significant advantage of CNNs over other types of networks is their capacity for parallelism, where a long input sequence can be processed quickly [47]. This can significantly speed up training since the entire output can be processed in one forward pass. The shared weights and local connectivity between neurons in the network allow for efficient computation, and using pooling layers reduces 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 (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.

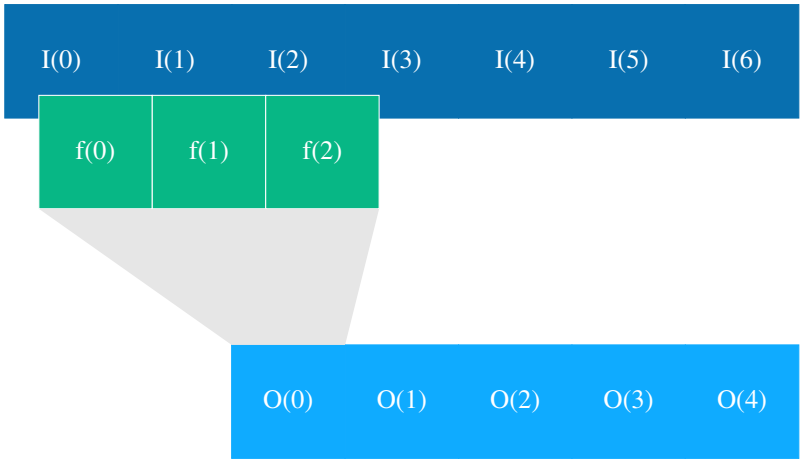

Figure 2.5: **Convolutional layer** — The presented diagram depicts a 1-dimensional convolutional layer designed to process an input signal consisting of a single channel and a length of 7. The layer employs a single filter with a size of 3, resulting in an output signal of length 5 and a single channel. 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.

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:

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

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

Max pooling has several benefits for CNNs:

- 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.
- Max pooling 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.
- Max pooling 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

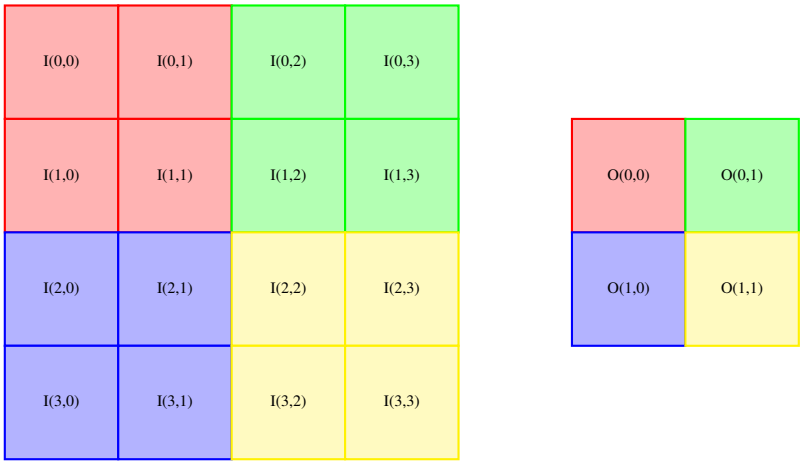

Figure 2.6: **Pooling layer** — The figure illustrates a 4×4 input tensor with a single channel. Each colored square in the tensor denotes a single element. The pooling layer partitions the input tensor into non-overlapping 2×2 regions and applies a pooling function to each region, resulting in a 2×2 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.

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.

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.

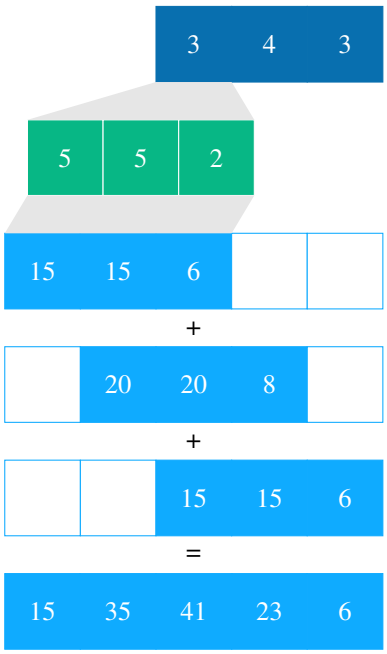

Figure 2.7: **Transposed convolution** — This illustration shows how a transposed convolution operation works. The input array has three elements $(3,4,3)$ and is shown by the dark blue rectangles at the top. The filter array also has three elements $(5,5,2)$ and is shown by the green rectangles in the 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.

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.

add figure, it doesn't need to be here

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 [87] where, between others, they took the declarations made in Perceptrons [61], 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 [89]. 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, 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 recurrent neural networks could perform any operation that a regular computer can, meaning that they are Turing complete [48].

RNN have the problem of forgetting terms seen long ago as new terms come along. 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.

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.

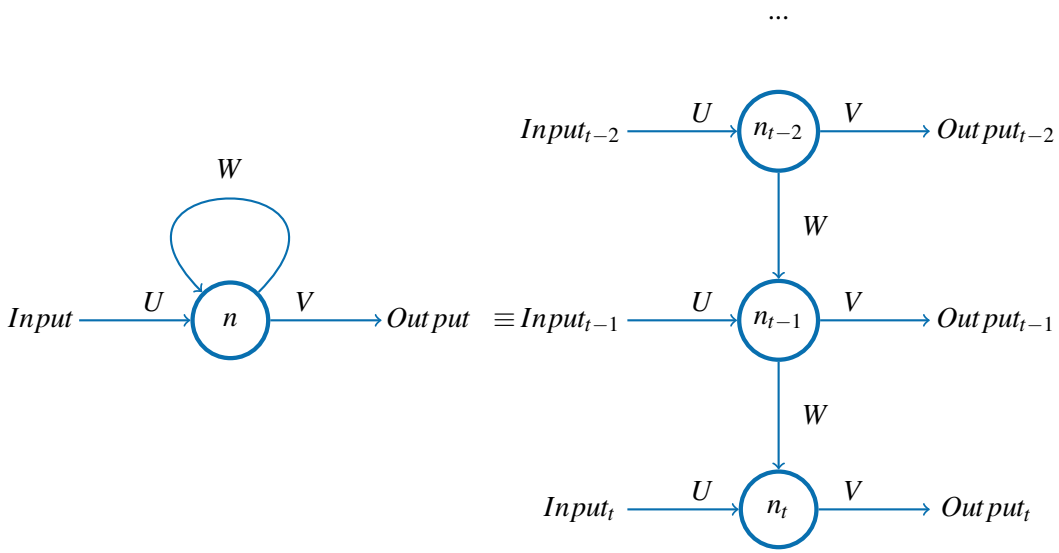

Figure 2.8: **Simple recurrent neural network** — The left side of the image displays the most straightforward representation of a RNN because it only has one recurrent neuron. Apart from the bias, the neuron has an input weight, a weight for the recursive connection, and a weight for the output. The 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.

The most prominent examples are the Gated recurrent units (GRUs) [14], and the Long short-term memorys (LSTMs) [42].

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, why would one want different architectures from the vanilla one? The truth is that the vanilla RNN presents problems. The most significant of these problems is the *vanishing gradient problem*.

RNNs can suffer from the vanishing gradient problem because of how they update their internal state. In an RNN, the internal state at time step t is computed based on the input at time step t and the previous state at time step $t-1$. This procedure creates a chain of dependencies between the current and previous states, resulting in repeated matrix multiplications during backpropagation. As the gradient becomes smaller and smaller with each multiplication, it can eventually become so small that the network cannot learn long-term dependencies in the input sequence.

The prime example of an architecture that fights the vanishing gradient problem is the **long short-term memory (LSTM)** [42]. 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.

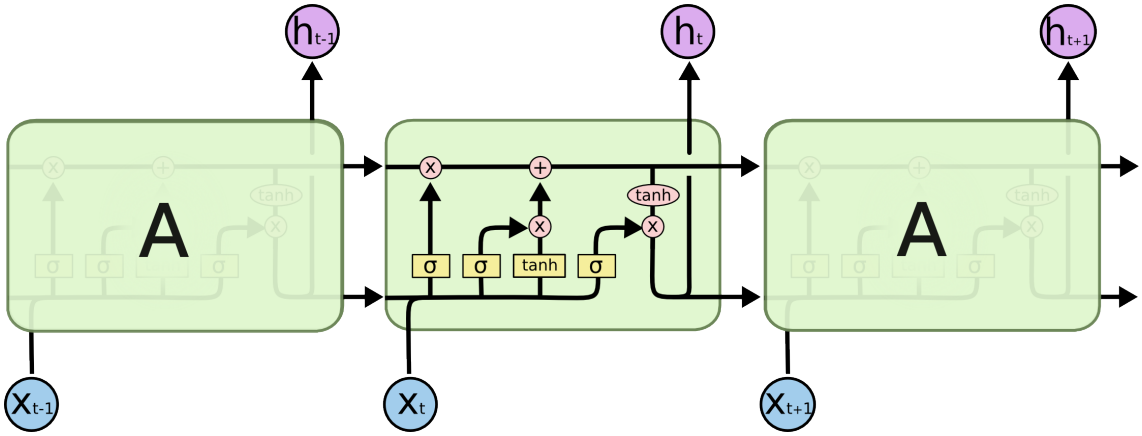

Figure 2.9: **Long short-term memory (LSTM)** — The network was taken from [16]. Each green rectangle is an LSTM cell. The blue circles represent an input at a given time, such that X_t is the input at time t. The pink circles represent the output at a given time h_t. Each LSTM cell takes three inputs: 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.

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 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)**. 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**. 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) (late 1980s) 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.

U-Net (2015) U-Net is a deep learning architecture introduced by Olaf Ronneberger et al. in 2015 for biomedical image segmentation tasks [85]. The name *U-Net* comes from the shape of the network, which resembles the letter *U*.

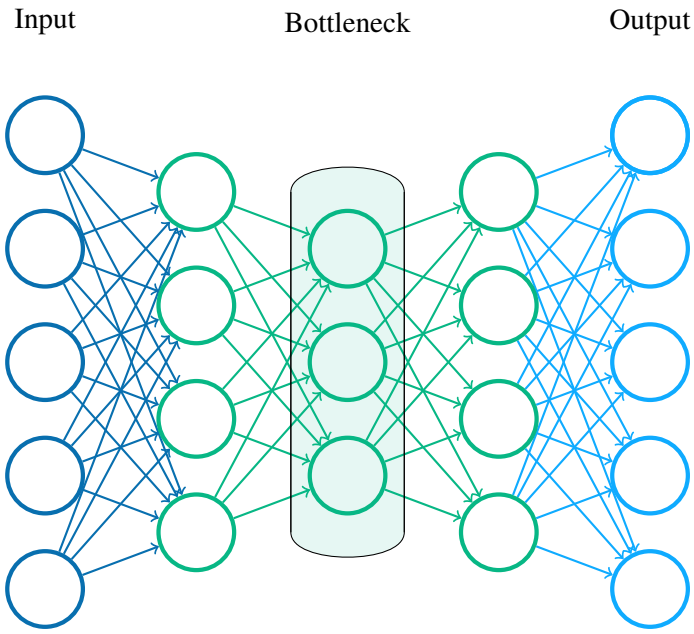

Figure 2.10: **Autoencoder (AE)** — This AE receives an input of size five and tries to learn a way to transform it into itself by passing through a bottleneck of size three. In the initial part, from the input to the bottleneck, an encoder is present, while the second part displays a decoder.

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.

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 discusses the foundations of deep learning, including activation functions, back-propagation algorithm for training neural networks, optimization with stochastic gradient descent,

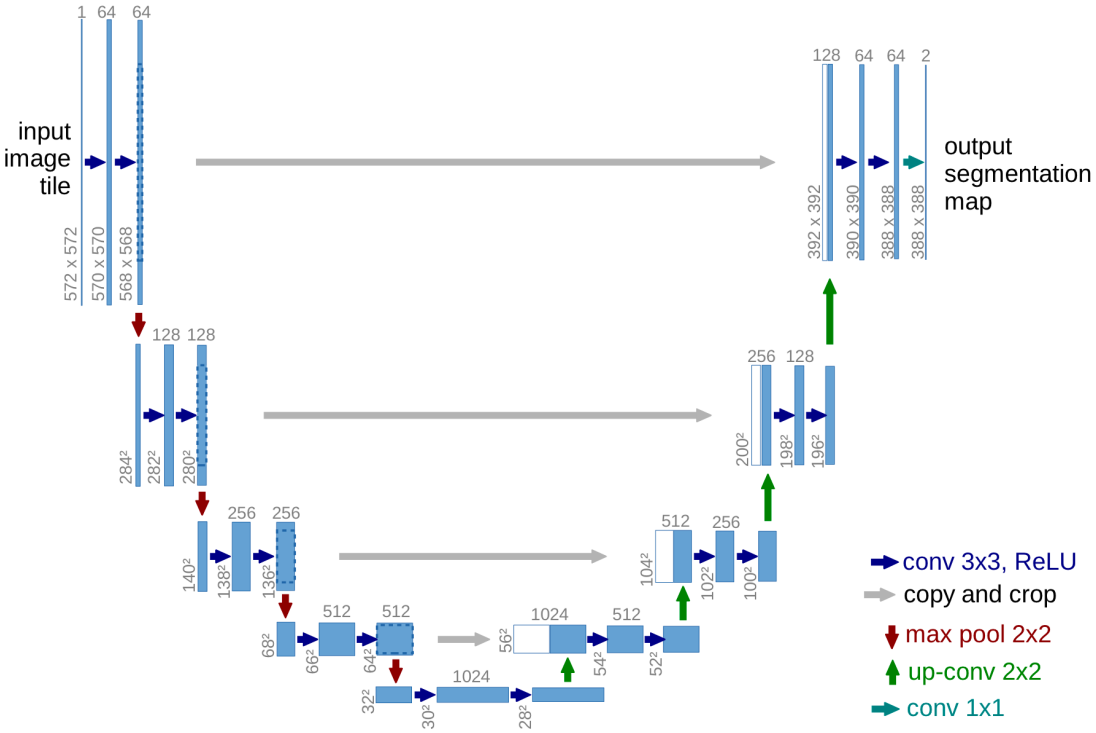

Figure 2.11: **U-Net** — This figure was taken from the original paper and follows an example for images with 572×572 pixels. Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White 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).

and optimization with the Adam optimizer. Activation functions introduce non-linearity into the model, and this section covers commonly used activation functions, including sigmoid, tanh, and ReLU. Backpropagation is an algorithm that trains feedforward neural networks by computing the gradient of the loss function concerning the network weights. Stochastic gradient descent is a widely used optimization algorithm in deep learning. 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. The Adam optimizer is a variant of SGD that adapts the learning rate for each parameter based on the estimates of the first and second moments of the gradients.

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.

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:

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

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.

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

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

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.

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. 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.

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 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.

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 [54].

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.

These do so by discovering latent data structures and learning to replicate the hidden statistics behind observed data. To do this, the model aims to approximate an underlying probability distribution p_{data} when given a set of samples from this distribution. So, training a generative model entails choosing the best parameters that minimize some notion of distance/loss/error between the model and the actual distribution. As Huzaifah [47] puts it: "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 iteratively changing model parameters θ". This is translated into the problem present 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) (2013) First introduced in 2013, deep autoregressive network (DARN) is an architecture for generative models that are used to generate data by using an autoregressive approach [37].

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.

Concretely AR models define a tractable density model by decomposing the probability distribution over n time steps via the chain rule of probability [47], shown in equation

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

From this approach, it is notable that data is assumed to have a canonical sequential order—the current term in the sequence (x_i) is only conditioned on a recency window of previous 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. Indeed an RNN can be cast as an autoregressive model that compresses the prior terms into a hidden state instead of providing them explicitly as input to a layer [47]. Besides, autoregressive models are more straightforward and faster to train. Indeed, an autoregressive model is a feedforward model 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 2.12.

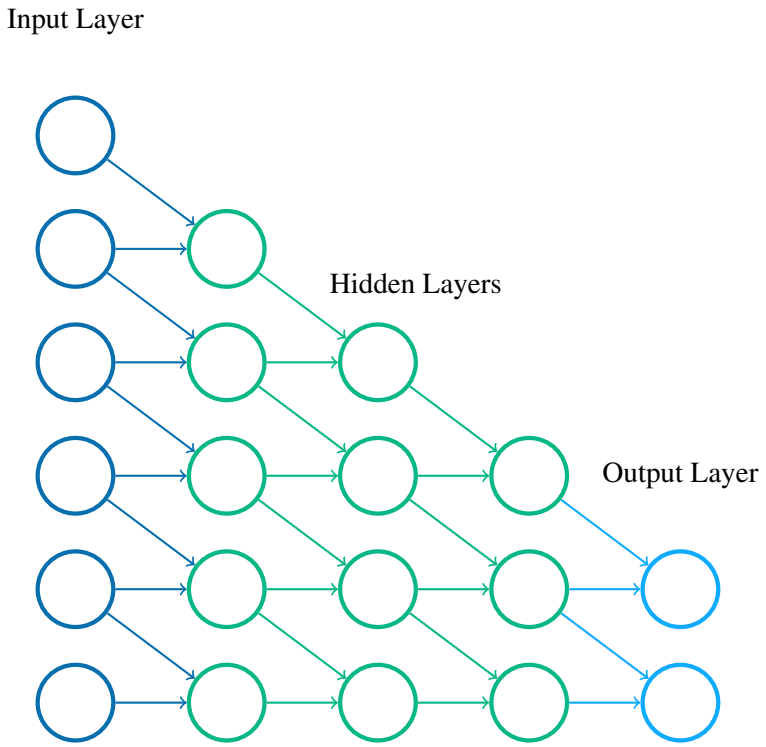

Figure 2.12: **Deep autoregressive network (DARN)** — The input of each neuron in a given layer is conditioned by the output of the 2 previous neurons in the previous output.

These models have been proven to accomplish various tasks, such as image, text, and audio generation.

Variational Autoencoder (VAE) (2013) Kingma and Welling proposed the concept of variational autoencoders (VAEs) in 2013 [55]. 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.

As seen in section 2.1.2.1, traditional autoencoders try to map the identity function by having an encoder and a decoder network. The encoder would encode the input into a vector, and the decoder would decode it into a sample as close as possible to the input. The difference in VAEs is that the encoder models the input data as a probability distribution over possible representations. The decoder processes samples from this distribution to generate the output. This restricts the embedded vector of the VAE to a given number of points in a hyperplane. The idea is that one can discard the encoder. Given a new data distribution in a continuous latent space, the decoder would generate new diverse samples similar to the training data. This can be seen in figure 2.13.

This was promising, as distributions close to each other would generate similar outputs. This is,

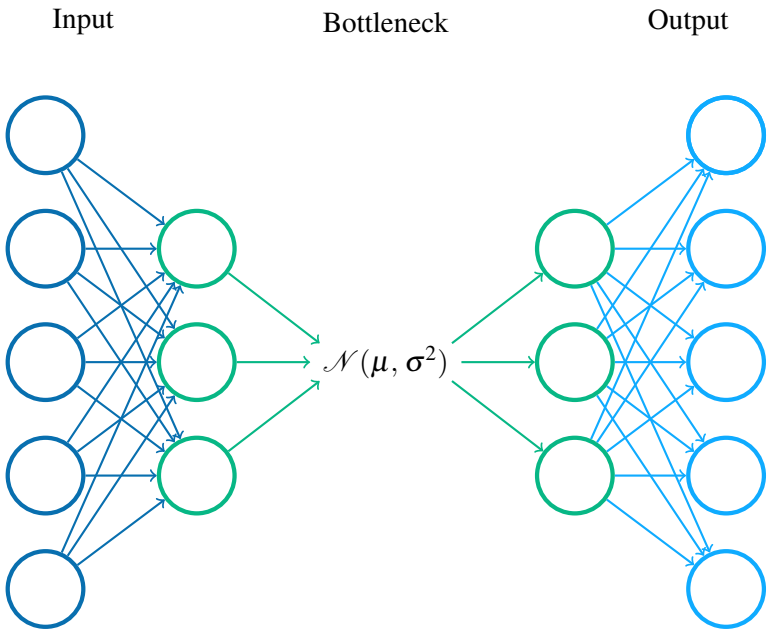

Figure 2.13: **Variational autoencoder (VAE)** — Very similar to the traditional AE, but the VAE encoder translates the input into a probability distribution (in this case, normal), and the decoder learns to upsample from the probability distribution.

it generated smooth transitions between data points. This happens because VAEs discover low-dimensional parametric representations of the data [47].

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 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) (2014) The paper "Generative Adversarial Networks" by Goodfellow et al. [36] 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.

The generator network G generates new samples by transforming a random noise vector z into a target distribution in some data space $\hat{X}$ (*e.g.* spectrograms). At the same time, the discriminator D tries to distinguish between synthetic and real data; this is, D maps the input data, be it X or $\hat{X}$, to a categorical label according to whether it thinks the input came from the actual data distribution $p(X)$ or the model distribution $p(z)$. This process is displayed in figure 2.14.

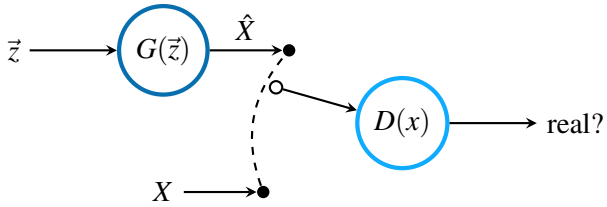

Figure 2.14: **Generative adversarial network (GAN)** — A random noise vector $\vec{z}$ is passed through the generator in $G(\vec{z})$ to create the synthetic sample $\hat{X}$. Both this and the real sample X are passed to the discriminator D that predicts which of the samples is the real one. It is important to notice that in this illustration, the circles represent entire neural networks and not simply neurons.

The networks are trained adversarially using a minimax optimization framework. The idea is that G creates a synthetic sample $\hat{X}$ passed in conjunction with a real one X to D. This network then has to tell which is real and which is generated. D is trained to maximize the probability of distinguishing the real from the synthesized data. While G is trained in parallel for the antagonistic objective, that is, to fool D by minimizing $\log(1 - D(G(z)))$ [47]. G and D are trained alternately until a Nash equilibrium is reached.

A Nash equilibrium happens when G produces perfectly synthetic data indistinguishable from real data. D cannot distinguish between the synthetic and the real data and merely guesses the input label. In this state, G's performance is at its maximum, and D's is at its minimum. The models cannot improve further than this. This is an ideal setting that takes work to achieve in practice. It is important to note that the generator is never exposed to the training samples, only to the feedback passed by the discriminator [47].

After training, the discriminator is discarded, and the generator can be used to sample from the learned distribution of the actual data. The generator has learned to map random vectors to data samples in the target domain. Usually, these vectors tend to be some feature representation. With that, they tend to cluster similar output data to neighborhoods of input values, providing a natural means of navigating among output data with different characteristics. This means that similar input vectors will generate similar outputs [47].

Although this technique has seen great success in producing high-resolution images, it still needs to improve in the audio domain. 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 [97], showing promising features for audio generation.

Normalizing Flow Models (2015) Normalizing flow models provide a flexible and robust framework for generative modeling and were first introduced in 2015 [83].

The key idea is to map simple distributions to more complex ones using the change of variables in the probability distributions technique. This technique involves applying a transformation to a distribution that transforms it into another, more complex, distribution. The whole concept starts with a simple distribution (*e.g.* Gaussian) for a set of latent variables z. The aim is to transform this distribution into a complex one representing an output X. A single transformation is given by a smooth and invertible function f that can map between z and X, such that $X = f(z)$ and $z = f^{-1}(X)$. Given the complexity of X, one of these transformations may not yield a complex enough distribution. Therefore, multiple invertible transformations are composed one after the other, constructing a "flow". Neural network layers parameterize each mapping function in the flow [47]. This process can be seen in figure 2.15.

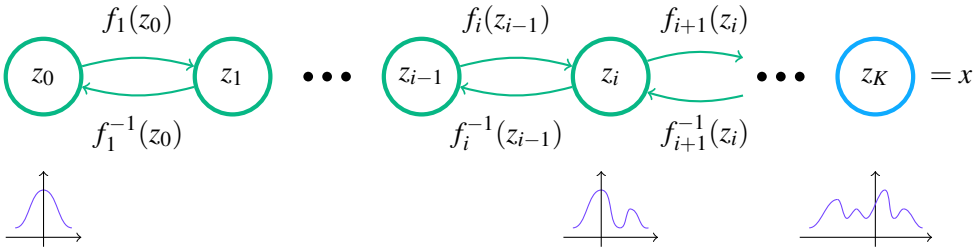

Figure 2.15: **Normalizing flows network** — This illustration was based on [106] and shows the application of multiple invertible functions f_k composed one after the other in order to build the complex output $z_K = x$ from a simple Gaussian distribution.

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 [47].

Diffusion Models (2015) 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 [94] 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 2.16

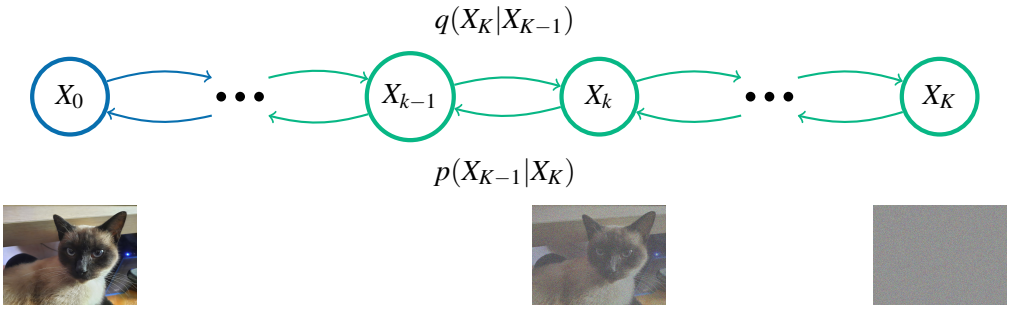

Figure 2.16: **Diffusion model** — This illustration was based on [40] and shows the process of applying Gaussian noise to an image sample through multiple steps $q(X_t|X_{t-1})$. The model will then learn the operation p that transforms X_t into X_{t-1} with $p(X_{t-1}|X_t)$ and so on until X_0. At 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 [94].

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 (2017) DL has revolutionized audio generation in recent years. In 2017, the introduction of the "Attention is All You Need" [102] 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. 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.

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.

This architecture change allows for way faster training and inference, allowing for larger datasets, drastically improving the generation results, and revolutionizing the field.

The original structure can be seen in figure 2.17.

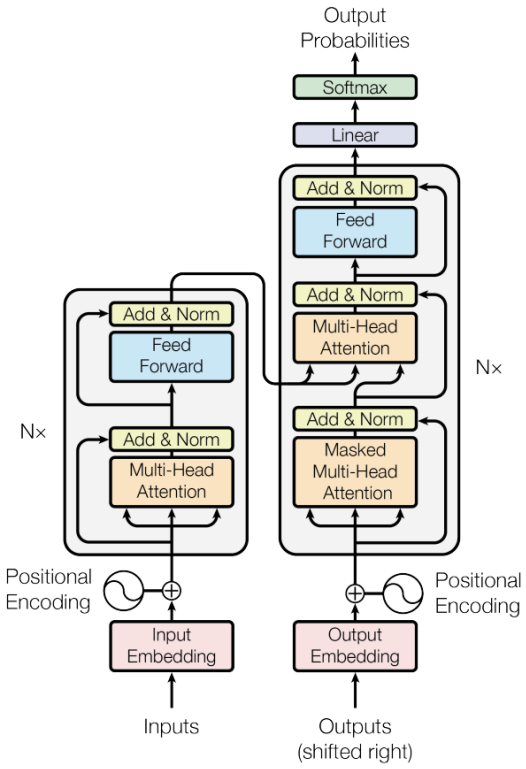

Figure 2.17: **Transformer** — This illustration was taken from [102] and shows the general architecture of the base transformer. One can see that the constituents are pretty simple. These are simple word embeddings (which are not covered in this study), self-attention, and feedforward layers. The left side of the structure is called the encoder, while the right side is called the decoder.

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 [69], concentrates on discrete representations, a natural fit for complex reasoning, planning, and predictive learning.

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

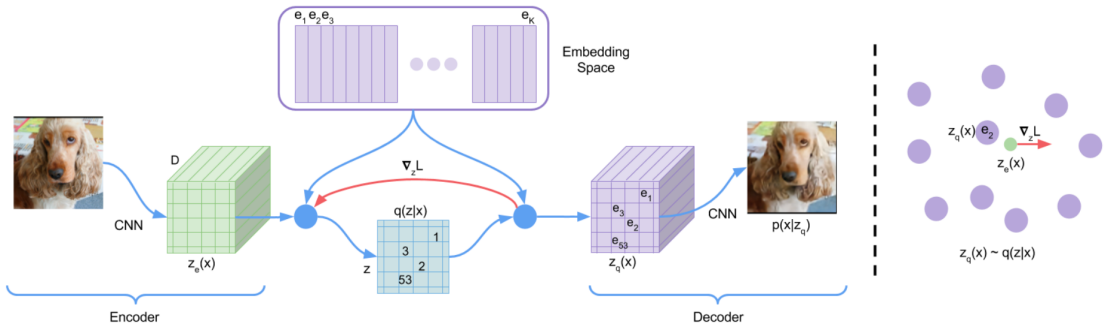

Figure 2.18: **VQ-VAE** — Taken from the original paper, this figure presents two distinct illustrations. On the left side, a detailed diagram of the VQ-VAE architecture is provided, showcasing the flow of information through the encoder, the discrete latent space, and the decoder. On the right side, 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.

avoids the "posterior collapse". As illustrated in Fig 2.18, 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. 2.27):

$$q(z=n|X) = \begin{cases} 1 & \text{for } n = \text{argmin}_j ||z_e(X) - e_j||_2, \\ 0 & \text{otherwise.} \end{cases} \tag{2.27}$$

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:

$$L = \log p(X|z_q(X)) \tag{2.28}$$
$$+ ||\text{sg}[z_e(X)] - e||_2^2 \tag{2.29}$$
$$+ \beta||z_e(X) - \text{sg}[e]||_2^2 \tag{2.30}$$

This equation combines the three following terms:

1. **Reconstruction loss** (Equation 2.28): 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 2.29): 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 2.30): 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 [80] and sound generation [108].

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 [99] 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 $e^s \in R^{K_s D_s}$, 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_i^s. There are K_s embedding vectors $e_i^s \in R^{D_s}, i \in 1, 2, ..., K_s$. The model takes an input x, encoded into outputs $z_e^s(x)$ 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_k^s. 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. 2.31), analogous to Eq. 2.27 in section 2.1.2.3, but with an additional scale index:

$$q(z^s = k|x) = \begin{cases} 1 & \text{for } k = \text{argmin}_j ||z_e^s(x) - e_j^s||_2, \\ 0 & \text{otherwise.} \end{cases} \tag{2.31}$$

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. 2.32), analogous to Eq. 2.28 in section 2.1.2.3, but with a summation over scales:

$$L = \sum_{s=1}^{S} (\log p(x|z_q^s(x)) + ||\text{sg}[z_e^s(x)] - e^s||_2^2 + \beta ||z_e^s(x) - \text{sg}[e^s]||_2^2) \tag{2.32}$$

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 More on Generative Models

Developing generative models for audio requires addressing several aspects that affect their performance and quality. This thesis focuses on 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 section 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 sound-based 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 [76].

Acoustic Data Augmentation According to Abayomi-Alli et al. [2], 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.

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 [66], such as white noise, babble noise, static noise, factory noise, jet cockpit, shouting, background noise, and others, [2]. 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 [65].
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.

To implement pitch shifting, one shifts the pitch of an audio signal in a positive or negative direction. For instance, plus or minus two [65]. This results in audio signals with a different pitch and is helpful for training models to recognize sound patterns invariant to pitch changes.

GAN Based Methods Utilizing GANs 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 [77].

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.

Sound Concatenation Mixing up sounds, or sound concatenation, is a data augmentation 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 known as sound mixing or audio blending, is a data augmentation technique for audio signals that involves combining two or more audio signals to create a new composite audio signal. This method can help the model recognize sound patterns in the presence of multiple simultaneous sounds, a common scenario in real-world applications.

The process of sound overlapping can be achieved through several steps:

1. Select multiple audio signals: Choose two or more audio signals that must be combined. These signals can be from the same or different sources, depending on the desired outcome and the problem.
2. Adjust the amplitude of each audio signal: It is essential to balance the amplitude levels of the audio signals combined to avoid one signal overpowering the others. This can be achieved by normalizing the amplitude of each signal or scaling them based on a predetermined factor.
3. Mix the audio signals: Combine the selected audio signals by adding them element-wise. This process creates a new, composite audio signal containing the overlapping sounds from the original signals.
4. Normalize the output: Normalize the resulting composite audio signal to prevent clipping or overloading.

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.

Sound overlapping has been successfully applied in various audio ML tasks, such as sound event detection, speech recognition, and music classification. For example, Maguolo et al. [59] introduced Audiogmenter, a MATLAB toolbox for audio data augmentation, includes sound overlapping as one of its core features.

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 [93].

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 [105].

- 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" [38].

- Feature-based augmentation applies transformations to word embeddings, *e.g.* adding noise to the embedding space [13].

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" [75].

- Generative augmentation employs generative language models to synthesize novel text, *e.g.* using an 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 2.2 categorizes several common text augmentation strategies according to their transparency, complexity, dependence on labels, and paradigm.

Table 2.2: 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

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[103]. 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 [107]. It is computed as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.33}$$

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 note that the limitations of the MAE should be considered when evaluating generative models. While it can be used to compare different generative models or optimize their parameters, it should be complemented with other metrics and human evaluation to assess the generated audio's quality comprehensively. Moreover, the MAE is unsuitable for comparing predictions using different scales and does not directly capture the perceptual quality of the generated audio samples.

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 [43].

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.

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.

By using KL divergence as a loss function, the deep learning system can be trained to generate sounds close to the target sounds in terms of their probability distribution. The idea is that the sound needs not to be similar to the input, but only its distribution. Minimizing the KL divergence can be considered as maximizing the log-likelihood between the generated output and the given input [47].

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 [9].

The ELBO is defined as follows:

$$ELBO = E_{Z\sim q}\left[\log \frac{p(X,Z;\theta)}{q(Z)}\right] \tag{2.34}$$

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. The ELBO for a VAE can be written as:

$$ELBO = E_{Z\sim q_\phi(Z|X)}[\log p_\theta(X|Z)] - D_{KL}(q_\phi(Z|X)||p(Z)) \tag{2.35}$$

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 VAE 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

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 [24]. 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 [63]. 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.

However, for the problem at hand, one cannot simply embed the words. If a user inserts a textual input, one should embed it as well; this is the whole textual input. For instance, a naive approximation would be to average the latent factors for each input.

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 [21] 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 [78] 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 (2022) MuLan [45] 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 deep learning networks. The most recent is from 2017 [72].

2.1.4.1 TensorFlow

Developed by Google, TensorFlow [60] is one of the most widely used deep learning frameworks. It supports both CPU and GPU computations and provides a variety of 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' Application Programming Interfaces (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 [73]. 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 high-level neural network library written in Python that provides a user-friendly interface for building and training deep learning models [15]. Keras is built on top of lower-level libraries like TensorFlow, and it abstracts away many of the low-level details, making it easy to create and train neural networks. Keras allows one to quickly create neural network models by assembling layers of pre-built building blocks. These building blocks include layers for input, CNNs, RNNs, fully connected layers, activation functions, and others.

Keras is often considered more straightforward than TensorFlow because it provides a higher-level interface that abstracts away many low-level details.

Keras provides a simplified API for building and training DL models, allowing one to quickly create models using pre-built building blocks like layers, activation functions, and optimizers. Keras also includes a range of utilities for data preprocessing, data augmentation, and visualization, making it easier to work with data.

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 2.3. 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 2.3: 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 (2016)

PixelCNNs were introduced in 2016 by Van den Oord et al. [68, 101] alongside pixel recurrent neural network. These networks model high-dimensional discrete data, such as images.

These networks are based on the AE architecture (section 2.1.2.3), meaning that they generate one pixel at a time, sequentially, conditioning on the previously generated 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. 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.

$$p(x) = \prod_{i=1}^{h \times w} p(x_i | x_1, \ldots, x_{i-1}) \tag{2.36}$$

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 (2021)

The first iteration of OpenAI's zero-shot text-to-image generation was published in 2021 [80], and it can generate images from textual descriptions. This problem is very similar to the task tackled by the present work.

The general architecture is easy to understand. It consists of mainly two stages. The first stage learns image representations or features using a discrete variational autoencoder (dVAE). In contrast, the second stage generates said image representations from textual descriptions using a transformer (see section 2.1.2.3). This macro-architecture can be seen in figure 2.19.

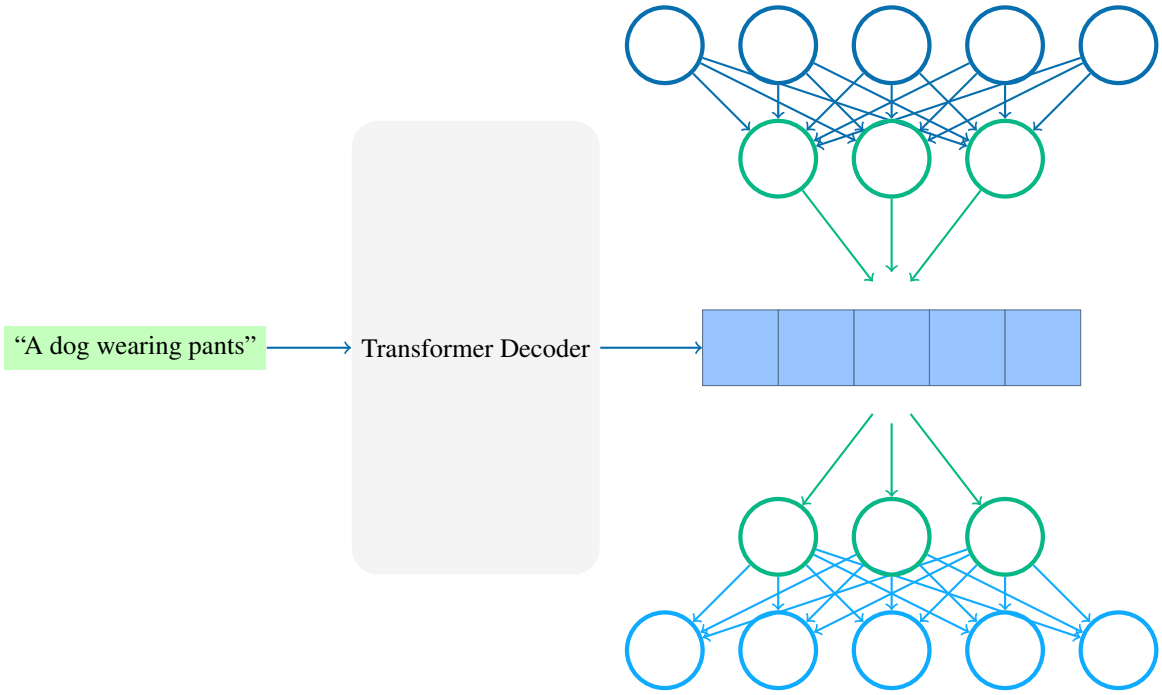

Figure 2.19: **Dall-E macro architecture** — With a green background, one can see textual operations. The complex on the right is a dVAE. Upon inference, the transformer’s output is used as the latent feature values of the dVAE.

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.

For the second stage, the text is initially byte pair encoding (BPE)-encoded. BPEs replace the most frequent character pairs in the text with a new, single symbol. BPE works by iteratively merging the most frequent pairs of consecutive symbols, such as letters or bytes, in the text until the desired number of merge operations is reached. This results in a new set of symbols representing the original text but with a smaller vocabulary.

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 (2021)

Stable Diffusion [84] rose from the idea of democratization high-resolution images. State-of-the-Art models require hundreds of thousands of dollars to train because of their complexity. These models typically operated directly in pixel space. Training and evaluating such models require massive computational resources only available to the biggest companies, leaving huge 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.

Then, taking the initial noise, both embeddings, and the timestamp, through the inverse diffusion process, the model generates two new images. The empty string embedding creates a random image. The idea is that at the end of each diffusion process, a comparison is made between the image generated by the real embeddings and the image generated using no embeddings. Increasing the difference between these two allows conditioning the image even more on the initial text.

The U-Nets (section 2.1.2.1) in the diffusion process have cross-attention for the respective embeddings.

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 2.20.

2.1.5.4 GLIDE (2022)

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 [41], which relies

refer guided diffusion

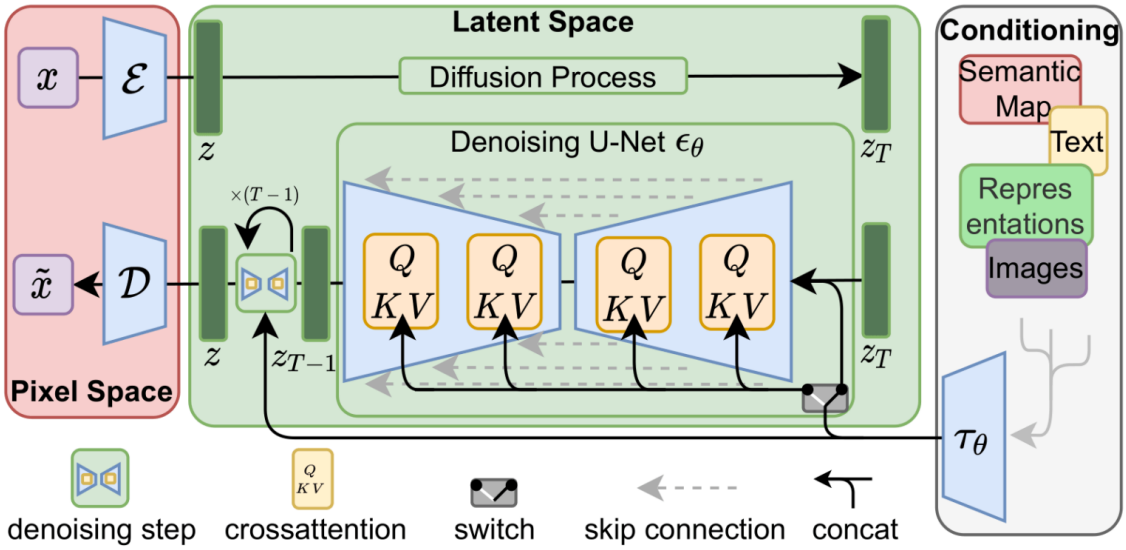

Figure 2.20: **Stable diffusion architecture** — The figure was taken from the original paper. On the top left, the original image x is encoded with the AE, and the diffusion process happens with the encodings. The text (or another data kind) is encoded with a transformer τ_θ, and these encodings are 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.

solely on the model's knowledge, and CLIP guidance which uses a pre-trained CLIP model [78] to provide guidance based on caption matching.

The GLIDE model consists of a diffusion process conditioned on a text prompt and random noise. This process generates a sequence of intermediate images, which are then projected onto the image manifold using a generator network. The generator network is trained to produce high-quality images that match the intermediate images produced by the diffusion process. The model is trained using a combination of adversarial and diffusion-based losses, ensuring that the generated images are realistic and diverse.

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 2.19) 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 graphics processing unit (GPU), which is slower

than other state-of-the-art GAN-based methods (see Section 2.1.2.3). Possible solutions to address these limitations include improving the model architecture, optimization techniques, and combining GLIDE with faster GAN-based methods.

CLIP stands for Contrastive Language-Image Pretraining and 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 (2022)

DALL-E 2 is a model proposed by researchers at OpenAI capable of generating images given a textual prompt [79]. 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 [78] 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 autoregressive (see section 2.1.2.3) and a diffusion model. The diffusion one yielded better results. The decoder then takes the generated image representation and generates the image. The whole process can be seen in Figure 2.21.

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.

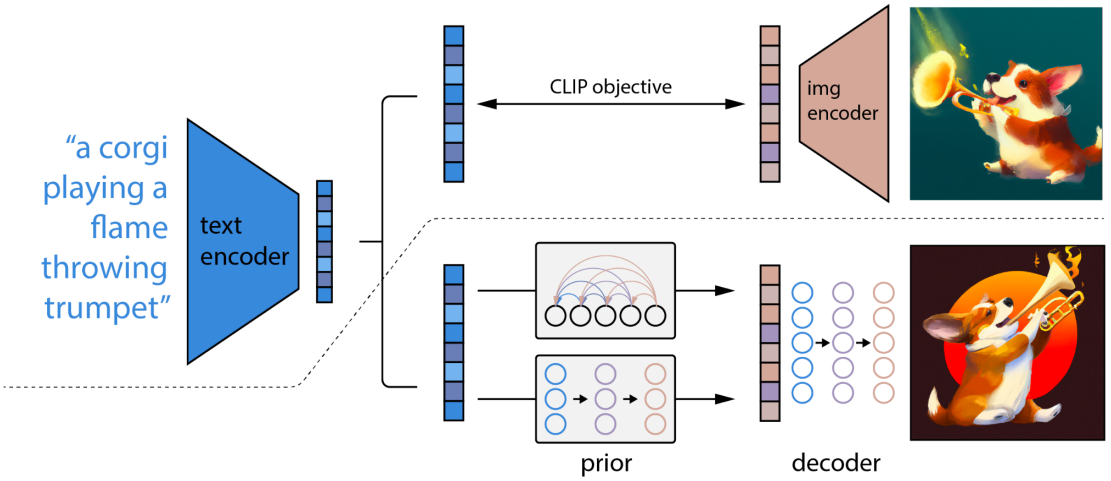

Figure 2.21: **DALL-E 2 architecture** — The image was taken from the original paper. Above the dotted line, the CLIP training process is depicted, where given textual and image embeddings, the CLIP learns to translate one into the other. Below the dotted line, a text-to-image generation process is represented: 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.

2.2 Related Work

Traditionally, sound designers relied on manual labor to create soundscapes, which involved recording and editing real-world sounds, mixing, and adding sound effects [95]. 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 [8, 96], 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 [82].

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 machine learning techniques with feature engineering. For a comprehensive overview of techniques employed before the era of deep learning, reference can be made to the review papers by Alias et al.[6] and Kalonaris et al.[51]. Noteworthy efforts at the feature engineering level are exemplified by Fernandez et al. [29], 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 sound 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.

One common representation in these models is the mel-spectrogram, a time-frequency representation of audio signals. The first component of the model involves transforming the original sound signal into a mel-spectrogram (or other) representation, 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 [56]. 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*. This component aims to produce high-quality audio signals that closely resemble the original sound data used to train the model.

This section of the thesis will review the related work in this field. First, a view on traditional soundscape generation methods is studied in section 2.2.1. Then, one will look to machine learning sound generation. State-of-the-art models either focus on generating the sound from internal representation (vocoders), in section 2.2.3, or on building an end-to-end-system (*e.g.* a full text-to-speech (TTS)) in section 2.2.4. These will be the focus of this section.

2.2.1 Traditional Soundscape Sound 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 [44].

2.2.1.1 Scaper

Searching through the academic search engines, one finds that the most cited software for soundscape generation is *Scaper* [91].

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.

2.2.2 Unsupervised Sound Generation

This section focuses on models that tackle unsupervised (or self-supervised) training, which involves acquiring knowledge of sound features and their distribution. This approach facilitates the generation of novel samples and enables latent feature representation. The following discussion covers notable models in this area. The selection of models in this section is based on their suitability for audio generation.

2.2.2.1 WaveGAN

GANs have been highly influential in generating images that exhibit local and global coherence. In 2019, *WaveGAN* [23] has been proposed as a GAN-based model for the unsupervised synthesis of raw-waveform audio. The model modifies the transposed convolution operation used in deep convolutional generative adversarial networks (DCGANs) to capture the structure of audio signals across various timescales. This modification includes using longer one-dimensional filters of length 25 instead of two-dimensional filters of size 5×5 and upsampling by a factor of 4 instead of 2 at each layer. Despite these changes, WaveGAN has the same number of parameters, numerical operations, and output dimensionality as DCGAN.

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 SoundStream

SoundStream is a neural audio codec proposed in 2021 [109] 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, trained jointly end-to-end using both reconstruction and adversarial losses.

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

Deep learning vocoders are neural network models that can generate artificial audio. Deep learning vocoders can overcome some of the limitations of traditional vocoders, such as low quality, unnaturalness, and lack of flexibility.

Deep learning vocoders use deep neural networks to directly learn the mapping between the input and waveform from data. They do not rely on any predefined model or feature extraction method. This approach has some advantages over traditional settings. For example, the model can capture complex nonlinear relationships between input and output representations that are difficult to model analytically.

There are different types of deep learning 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 speech 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 [67]. It implements the PixelCNN (see section 2.1.5.1) model for sound and follows an autoregressive 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 [67]. This architecture can be seen in figure 2.22.

This structure enables WaveNet to capture long-range dependencies in the input sequence, which is crucial for generating high-quality audio and text. 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 [47]. 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.

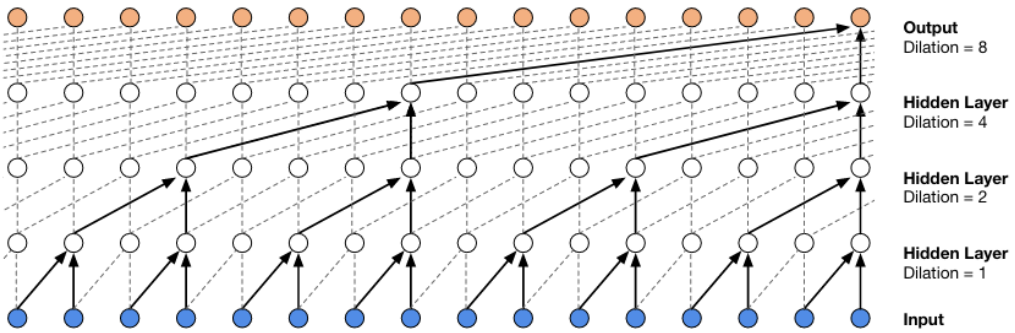

Figure 2.22: **WaveNet** — This illustration was taken from [67]. It shows the idea behind WaveNet, applying dilated convolutions to AR models.

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." [67].

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 [97].

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 [50], 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 [53], 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 [70] employs a caching mechanism to reduce the computational cost of the model while maintaining an autoregressive 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

The 2019's *MelGAN* paper [58] proposes that although difficult, it is possible to approach audio generation with GANs (see section 2.1.2.3). Previous works in this area have faced the challenge of generating coherent raw audio waveforms with GANs. However, the authors of MelGAN show that it is possible to train GANs reliably to generate high-quality and coherent waveforms by introducing some architectural changes.

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, [27] 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 [28] 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 strong 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 autoregressive counterparts.

2.2.3.5 HiFi-GAN

Proposed in 2020, *HiFi-GAN* [56] 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 convolutional neural network 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 autoregressive 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.n 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.

The extant research establishes two principal frameworks for end-to-end models: specialized models designed for a specific domain and universal models aimed at broader applications. Specialized models target either speech or music synthesis. Researchers have developed distinct subsets of technologies within speech and music synthesis models.

2.2.4.1 Text-to-Speech

Char2Wav (2017) The Char2Wav model, proposed in 2017 [81], 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 (2023) VALL-E is a language model developed by researchers at Microsoft for TTS that treats TTS as a conditional language modeling task [104]. 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 2.23.

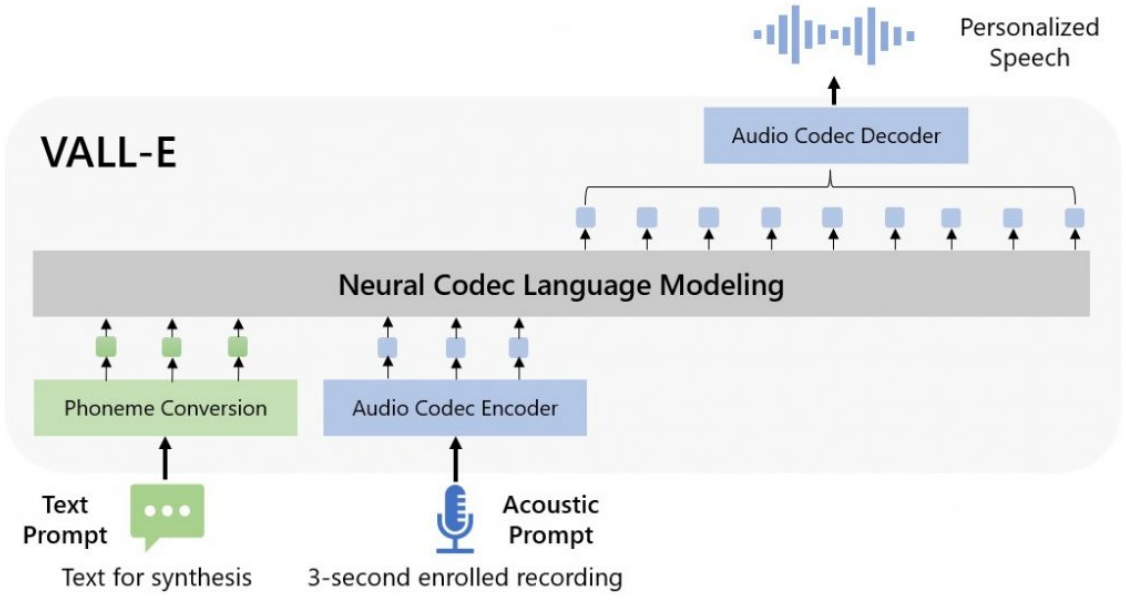

Figure 2.23: **VALL-E** — The image was extracted from the source publication. It illustrates that both encodings derived from a linguistic prompt and an auditory prompt - provided via the codec encoder - are fed into a language model such as a transformer, and the resulting outputs are passed to the codec decoder to generate audio.

The researchers trained VALL-E using the LibriLight dataset [49], 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 [71] 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 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 [4], 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. This finding makes VALL-E a promising option for emotional TTS applications.

VALL-E represents a significant advancement in TTS technology, with its language model approach and use of audio codec codes as intermediate representations. However, there are still several areas for improvement and future work.

Considering the broader impacts of VALL-E and similar TTS technologies is essential. TTS has the potential to benefit individuals with speech or hearing disabilities, as well as those who require synthetic speech for various applications. However, as with any AI technology, risks, and ethical concerns exist, such as the potential for misuse in deepfake and impersonation applications. Researchers and developers must consider these issues and develop safeguards to prevent potential harm.

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. However, there are still areas for improvement and ethical considerations to be addressed in developing and deploying these TTS technologies.

2.2.4.2 Generative Music

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 [22]. 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 autoregressive 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 [31] is an open-source model presented in 2022 that generates audio 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 audio 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 to approximate the phase when reconstructing the audio clip.

talk about griffin-lim

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 (2023) 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 /todoref audiolm)by incorporating text conditioning via the joint music-text model MuLan (see Section).

cite mulan

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 [citation].

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 evaluated using the MusicCaps dataset of 5,500 music-text pairs annotated by experts, covering various genres, instruments, and moods. The authors claim this dataset rigorously evaluates the model's ability to synthesize different aspects of music from textual prompts. Evaluation metrics include the perceptual evaluation of audio quality (PEAQ) score and human judgments of generation similarity to prompts and overall quality.

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 represented 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

SampleRNN *SampleRNN* is a neural audio generation model proposed in 2017 that can produce high-quality audio samples from scratch [62]. 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 AudioLM, proposed in 2022, is a framework Borsos et al. [10] introduced for high-quality audio generation with long-term consistency. The framework maps input audio to a sequence of discrete tokens and treats audio generation as a language modeling task in this representation space. AudioLM achieves both high-quality synthesis and long-term structure by utilizing a hybrid tokenization scheme, which combines discretized activations of a masked language model pre-trained on audio (semantic tokens) and 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 2.2.2.2). Semantic tokens are computed using w2v-BERT (see section), 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.

ref w2v bert

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* [108].

A vector quantized variational autoencoder (VQ-VAE) is a type of variational autoencoder (VAE) (see section 2.1.2.3) where the encoder neural network emits discrete values by mapping the encoder's embedding values to a fixed number of codebook values. It differs from VAEs in two ways: the encoder network outputs discrete, rather than continuous, codes, and the prior is learned rather than static. In a VQ-VAE, the prior refers to the distribution of the latent codes. In a standard VAE, the prior is usually assumed to be a fixed distribution, such as a standard normal distribution. However, in a VQ-VAE, the prior is learned from the data rather than being fixed. This means that the model can adapt the prior to better represent the data distribution.

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 2.24.

AudioGen In 2023, Kreuk et al. [57] 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 auto-encoding method and (ii) training a Transformer language model 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 internally. 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.

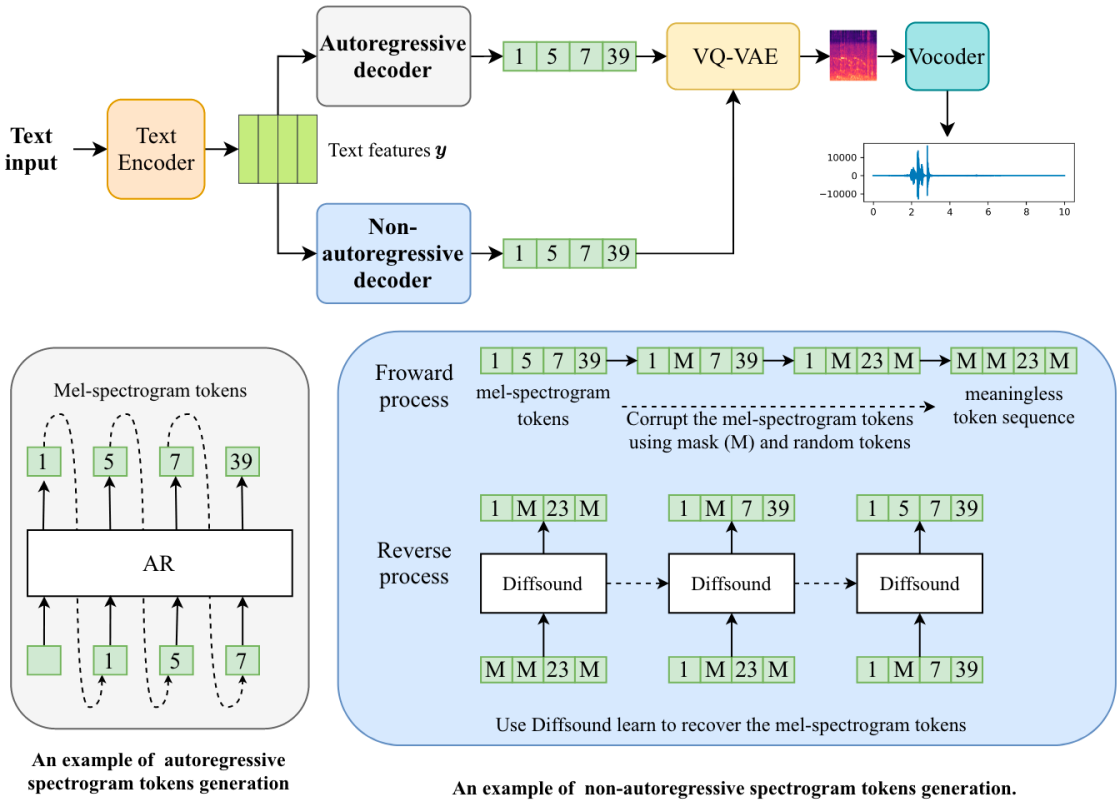

Figure 2.24: **DiffSound framework** — This illustration was taken from the original paper. At the top, the general framework is present. Two decoders are present, but only one of them is used. The decoder results in a set of latent features. These features are passed to the decoder of the VQ-VAE that 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.

Chapter 3

The Synthesis Problem

Contents

This research proposal aims to explore and implement advanced generative AI models for synthesizing audio from textual input. The goal is to create a system that can produce high-fidelity audio snippets, meaning that the generated sound is indistinguishable from real-world sounds by humans. The system should not be limited to any particular domain, such as music or speech, but instead should be able to accept any textual prompt and synthesize the corresponding audio. The focus of this research will be on developing and testing various model architectures based on state-of-the-art systems, with the ultimate aim of achieving optimal results. Examples of prompts to be used in the experiment range from simple, such as a truck horn, to more complex sounds, such as a bird whistling on a rainy day.

First, in section 3.1, the problem is defined following input and output data, objective function, minimization algorithm, and evaluation metrics. Then, in section 3.2, to demonstrate that this research is not only theoretical, some practical use cases are displayed and described. Section 3.3 discusses the datasets used. Furthermore, sections 3.4 and 3.5 discuss possible problems that might arise with the development of the model.

3.1 Problem Definition

This study aims to solve the generation of sounds given a textual prompt. These sounds are not restricted to a given area, such as speech or music. The sounds generated by the model must depend on the textual input inserted. This generation has to be in real-time; after the insertion of the textual prompt, the time it takes for the model to generate the given sound must be minimal. In this study, a model that solves this problem is called an end-to-end model. Moreover, "end-to-end" and "text-to-sound" are to be taken as synonymous.

These models are to be trained with human-labeled sounds from open-source datasets. Moreover, the evaluation means to compare the generated sounds with their real counterpart.

However, to define this problem more formally, one needs to specify the following elements:

- input data
- output data
- objective function
- optimization algorithm
- evaluation metrics

3.1.1 Input Data

Usually, the input data in machine learning is similar when the model is training and in production. For instance, a classification model may take several parameters and a label for each sample for the fitting. When one wants to classify, one may receive the parameters and try to predict the label.

However, for these generative models, the input data is slightly different. For training, one still receives the samples and their labels — for instance, sound files and their respective titles. But in production, one only gets the labels instead of the samples.

For this problem, in particular, one receives sound samples — that may be in a variety of formats, but following every time the main structure defined in section 2.1.1. And apart from the sound samples, one also receives labels that may be in the form of tags (*e.g.* coming from datasets made for classification) or regular text (*e.g.* coming from places like YouTube).

3.1.2 Output Data

These models' output data is in a particular audio format. This output is to be generated in real-time. This is, the time it takes for the model to generate a given output must not be substantial. A hard constraint such as X seconds must be set. Studies regarding human perception of slowness must back this constraint.

3.1.3 Objective Function

This is the goal of the problem. For example, in a supervised learning problem, the objective function might be to minimize the difference between the predicted output and the proper output.

It isn't easy to define objective functions in generative models. Generative models try to learn and understand the underlying distributions of the data, to use that knowledge to generate new data similar to the original one. As stated in chapter 3, one wants to create accurate data as realistically as possible. For this, one may want to find a function that minimizes the difference between the original data to the synthesized one.

3.1.4 Optimization Algorithm

This is the algorithm that will be used to optimize the objective function.

This will depend on the used model. Check section 4.3 for more details.

3.1.5 Evaluation Metrics

These are the metrics that this dissertation will use to measure the performance of the model.

3.2 Applications

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 [47].

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 [47], 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 machine learning models, and one should imagine how that will enhance the power of sound producers for such endeavors, especially in indie and low-budget productions.

One can imagine multiple user interfaces where these applications are helpful. This work will not discuss that. However, an API enabling such interfaces and a simple website where anyone can access it are in development. A further discussion is present in section 4.5.

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 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" [47].

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” [47]. 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 critical task for these kinds of models and also one of the most difficult challenges in digital audio production [47].

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 3.5, and more deeply argued in section 4.3.

3.5 Model

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 chapter presents the development and implementation of a system for synthesizing audio from textual input. The goal is to design a robust method to generate high-quality audio from any text.

The chapter begins with a discussion of the approach to tackling the completion of the present thesis, including indications about the research, writing, and code development in section 4.1. Next, in section 4.2 the datasets used for training and validating the models are delved into, explaining their significance and how they contribute to the system's performance.

Subsequently, in section 4.3 a detailed description of the developed models is provided, focusing on their architecture, components, and the rationale behind their design. This is followed by evaluating the solutions, in section 4.4, where its performance, strengths, and limitations are analyzed.

The system's user interface is also discussed in section 4.5, allowing users to input text and generate corresponding audio output. The interface's design considerations, functionality, and user experience aspects are covered in the discussion.

Lastly, in section 4.6 the work plan is outlined, which includes the milestones, timelines, and resources required to bring the solution to fruition. Through this comprehensive exploration of the development and implementation process, valuable insights into creating a state-of-the-art sound generation system are hoped to be provided.

4.1 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 sections are not necessarily done exclusively. This implies that they should overlap each other. More specifically, the work begins with general research on state-of-the-art models and audio handling. After obtaining a basic idea of what should be done, early development should begin. This development will be conducted in parallel with continuous research. More research is needed to address these questions if problems arise regarding development. Writing the document must be ongoing work, leading to clear thoughts and planning. However, writing should be prioritized when most of the development is made.

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. These keywords were then used, through multiple combinations, to search multiple online sources, including academic search engines such as Google Scholar and the Universidade do Porto's. Searches were also conducted for specific papers given one'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 approach to developing the audio synthesis model is to be taken progressively, starting with simple tasks and gradually increasing the models' complexity and the dataset's size. This approach allows for a thorough evaluation of the model's capabilities and provides an opportunity for continuous improvement and refinement.

Initially, the focus is on simple tasks, such as classification and generation without text conditioning, using smaller datasets such as Audio MNIST. This permits a basic understanding of the model's capabilities and a foundation for further development. The ideas and techniques that produce better results must be documented and used to develop more complex models.

4.1.3 Writing of the Dissertation

In terms of the content, the goal was to ensure that each section was clear, concise, and informative. To achieve this, the writing of this dissertation was approached continuously, with each section going through three main phases: the drafting stage, the integration of researched references, and the refinement stage. The refinement process is ongoing, with multiple re-readings and continual improvements to ensure the highest level of quality. This approach was in line with an agile development method, allowing for flexible and iterative writing progress.

In addition, this document adheres to the academic standards of writing and formatting, including the use of proper citation styles and clear and consistent presentation of results and figures. Academic peers and advisers also reviewed the document to ensure its quality and accuracy.

The writing process was approached with rigor and attention to detail, ensuring that this thesis document represents the culmination of the author's research efforts and the best possible representation of their work.

4.2 Datasets

This section contains the datasets researched and used for the models' implementation. It is important to note that datasets that might be well-known in the area but were neither used nor helpful for this specific problem are not cited here. For instance, datasets that are specific to speech or music are not taken into account. Apart from this, given that the data size is essential for deep learning algorithms, all the present datasets have more than 1000 sound samples. However, this is still a relatively small number compared to the datasets with millions of entries used for image recognition tasks, such as CLIP [78].

Two types of datasets are listed. Their main difference relies on the type of label (i.e., the textual description) adopted. This work distinguish two labels: *categorical* and *descriptive*. Datasets featuring categorical labels are composed of sounds associated with a unique label, such as "music", "piano", or "singing." Datasets featuring descriptive labels include natural language sentences or descriptions of the soundscape, such as "boy singing while playing the piano." In the context of deep generative learning, descriptive labeled datasets are more suitable. However, datasets with discrete labels can also be beneficial when augmented. To the author's knowledge, Table 4.1 lists all soundscape datasets to date that fulfill the aforementioned criteria.

Table 4.1: Comparison of datasets for soundscapes

Name	**Type**	**# Samples**	**Duration**	**Labels**
Acoustic Event Dataset [98]	Categorical labeled	5223	Average 8.8s	One of 28 labels
AudioCaps [52]	Descriptive labeled	39597	10s each	9 words per caption
AudioSet [34]	Categorical labeled	2084320	Average 10s	One or more of 527 labels
Audio MNIST [7]	Categorical labeled	30000	Average 0.6s	One of 10 labels
Clotho [25]	Descriptive labeled	4981	15 to 30s	24 905 captions (5 per audio). 8 to 20 words long each
FSDKaggle2018 [30]	Categorical labeled	11073	From 300ms to 30s	One or more of 41 labels
UrbanSound8K [90]	Categorical labeled	8732	Less or equal to 4s	One of 10 labels
YouTube-8M Segments [3]	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 . 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”.

reference text augmentation

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 [98].

4.2.1.2 Audio MNIST

Audio MNIST [7] is a sonic take on the famous MNIST dataset [19]. 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 [34].

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 [30] 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 [90] 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 [3] 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 [52] 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 [25] 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 Developed Models

As shown above, in Section 4.6, the development of the models followed a given structure. First, one is to develop a basic classification model on a simple dataset; then, basic generative models; then, the application and study of these models to larger datasets; and finally, the application of conditioned generation as seen in Section 3.4. The following sections describe and study this process.

4.3.1 Preliminary Classification

The work at hand requires a solid grasp of the fundamental concepts of sound and its representation. Therefore, the first step was to develop a preliminary classification model to distinguish valuable data into discrete categories. Sound classification has various practical applications in speech recognition, music analysis, and environmental monitoring. This section describes the methodology and results of building a basic classification model using the Audio MNIST dataset (see section 4.2.1.2).

The Audio MNIST dataset was selected for its simplicity and suitability. This dataset consists of recordings of spoken digits labeled into ten different classes. The model was implemented using both TensorFlow and PyTorch.

The model pipeline consisted of the following steps: First, a spectrogram was extracted from the sound signal (see section 2.1.1). Then, a 2D CNN (see section 2.1.2.1) was applied to the spectrogram, which consisted of three convolutional and max pooling layers, followed by two fully connected layers. The output layer had ten units corresponding to the ten classes in the dataset.

The code for the model in PyTorch is provided in the appendices . The model achieved an accuracy of 95.17% on the test set.

reference appendix

The high accuracy of the model indicates its efficacy in classifying audio data. However, this is only a preliminary model that can be further improved. For example, tuning the hyperparameters of the model could enhance its performance. Nevertheless, this model serves as a baseline and an introduction to the main topic of this paper, which is to explore more advanced and challenging sound classification tasks.

4.3.2 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 (see Section 4.2.1.2) 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 resizing the audio to a size that was a power of 2 (more specifically, 65536). Resizing to a power of 2 helps ensure that the convolutions and pooling operations in the network, which keep the same size or split by powers of 2, will be divisible by 2.

To prevent gradient explosion (see Section 2.1.2.2), the audio was normalized by dividing by 32768. This ensured the audio values stayed within a fixed range and facilitated the model's learning.

The GAN model was implemented in TensorFlow (see Section 2.1.4.1).

The generator network comprises several layers of dense, reshape, and transposed convolution operations. Figure **??** shows the architecture of the generator network. figure

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. Figure **??** shows the architecture of the discriminator network.

discriminat figure

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 generator and the discriminator parameters using gradient descent. The loss function used for both networks is binary cross-entropy (BCE) , which measures how well the networks predict the correct input labels. The generator tries to minimize loss by making the discriminator output high values for fake samples. In contrast, the discriminator tries to minimize loss by making its output low for fake samples and high for real samples.

reference BCE

The training algorithm can be summarized as presented in Algorithm 1.

The code for implementing this architecture is presented in annex 123

annex 123

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,

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

results were promising. Generated audio samples were based on random noise but resembled real ones in terms of wave amplitude.

show results images

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.

4.3.3 Simple Autoencoder for Audio Data Compression

This section presents the process of constructing a simple AE (see Section 2.1.2.1), which serves as a foundational step toward developing more advanced variations, such as the VAE (see Section 2.1.2.3), in future work. The implementation of the AE, written in PyTorch, can be found in the provided code repository (see Annex).

reference to code repository

A random cropping approach (see Section 2.1.3.1) is employed instead of padding to ensure consistent audio sample lengths. By cropping the audio samples to half the padding size mentioned in a previous section (32,768), smaller samples are obtained, enabling a more compact model architecture and faster training times. Since the beginning and end of audio samples usually contain silence or minimal noise , this cropping method is not expected to affect the data significantly.

point to figure

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.

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 (‘encoded‘) and a reconstructed audio waveform (‘decoded‘).

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 autoencoder can capture and reproduce the essential characteristics of the audio data.

Backpropagation is employed, enabling the computation of gradients that reflect the impact of each parameter on the overall loss. These 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.

Throughout the training process, the progress is monitored and logged, providing information such as the current loss value and the index of the processed batch. This allows researchers to track the convergence of the model and gain insights into its learning dynamics.

Upon concluding the training phase, an evaluation of the trained model is conducted. This evaluation entails calculating the test loss. The dataset was initially partitioned into two random and distinct subsets, namely the training set and the test set, in an 80-20 ratio to perform this evaluation. The test split, reserved explicitly for assessment purposes, allows for an unbiased evaluation of the model's performance on unseen data. By quantifying the deviation between the reconstructed waveforms and their original counterparts using the test loss, researchers can gauge the efficacy of the trained model in faithfully reproducing the audio data.

Plots that compare the original and reconstructed versions of a randomly selected audio waveform are generated to make a more intuitive comparison between the original and the generated samples possible. These plots provide a visual representation of the AE's performance in capturing the intricate details of the audio data.

point to figure

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 will be instrumental in developing more sophisticated techniques like the VAE in future research. Moreover, the provided code implementation in PyTorch (refer to Annex) facilitates a deeper understanding and exploration of the AE architecture, enabling improvements and extensions to audio data compression.

reference to aannex

4.3.4 Simple Variational Autoencoder

The next step involves constructing a crucial component of the complete generator model: the variational autoencoder (VAE) (see section 2.1.2.3). The VAE differs from the AE of the previous section in that it is reduced to a fixed number of latent dimensions. Both models have an encoder and decoder structure, but the VAE has an intermediate step where it learns the means and variances of distributions. The initial idea was to create a basic VAE to simulate the dataset.

The encoder network used here is the same as the one used in the previous chapter's AE, albeit with fewer layers. The encoder is flattened and divided into two linear, fully connected layers with the same number of nodes, representing the latent dimension. One layer is for the means, and the other is for the variances of the Gaussian distributions.

During training, early stopping was applied, which halts the model after a specified number of iterations without test improvement. However, the model had difficulties converging. The loss function was modified by weighting differently BCE and Kullback–Leibler divergence (KLD). These were set as hyperparameters and finetuned to a given number of epochs, but the results were unsatisfactory.

The Adam optimizer (see section 2.1.2.2) was used, but the initial learning rate range was adjusted to address convergence issues. The number of convolutional and deconvolutional layers and the number of latent dimensions were experimented with and finetuned to find the optimal values. Despite not overfitting, the model could not learn the data's features and did not converge. The model was too large for the GPU to handle, even after reducing the batch size. It was unclear if the model would converge with further adjustments to the training rate.

It is worth noting that none of the VAE models produced satisfactory results, and comparing poor results with one another is not helpful. Herefore, a more detailed comparison between the models developed would not be helpful.

4.3.5 Stable Diffusion VAE for Spectrograms

Training a VAE (see Section 2.1.2.3) from scratch can be a time-consuming process. To address this challenge, researchers have turned to pre-trained models that have already achieved satisfactory results on similar tasks. In the case of image processing, the Stable Diffusion model (see Section 2.1.5.3) has become a staple.

Stable Diffusion involves encoding the data using a VAE, applying the diffusion process in the latent features, and decoding with the VAE decoder. The diffusion process can be conditioned on text, requiring a text encoder/tokenizer to encode the text. The model consists of three components: a VAE for encoding and decoding, a text encoder/tokenizer, and a U-Net (see Section 2.1.2.1) for diffusion. The models are fine-tuned to the specific problem of the thesis.

To employ Stable Diffusion for the current task, it was necessary to convert sound data into a format the model could comprehend. This was accomplished by generating mel-spectrograms from the sound data, which are essentially 2D representations of the sound data that can be treated as images. The mel-spectrograms were fed to the VAE encoder to generate the latent features.

Spectrograms, unlike images, have only one channel, while images typically have three channels (red, green, and blue). Spectrograms, like images, have width and height dimensions. To make the spectrograms compatible with Stable Diffusion, they were tripled, resulting in two additional channels identical to the first. The loss function only considers the first channel and disregards the other two. For the initial implementation, the loss function was the absolute difference between the initial spectrogram and the first channel of the generated image.

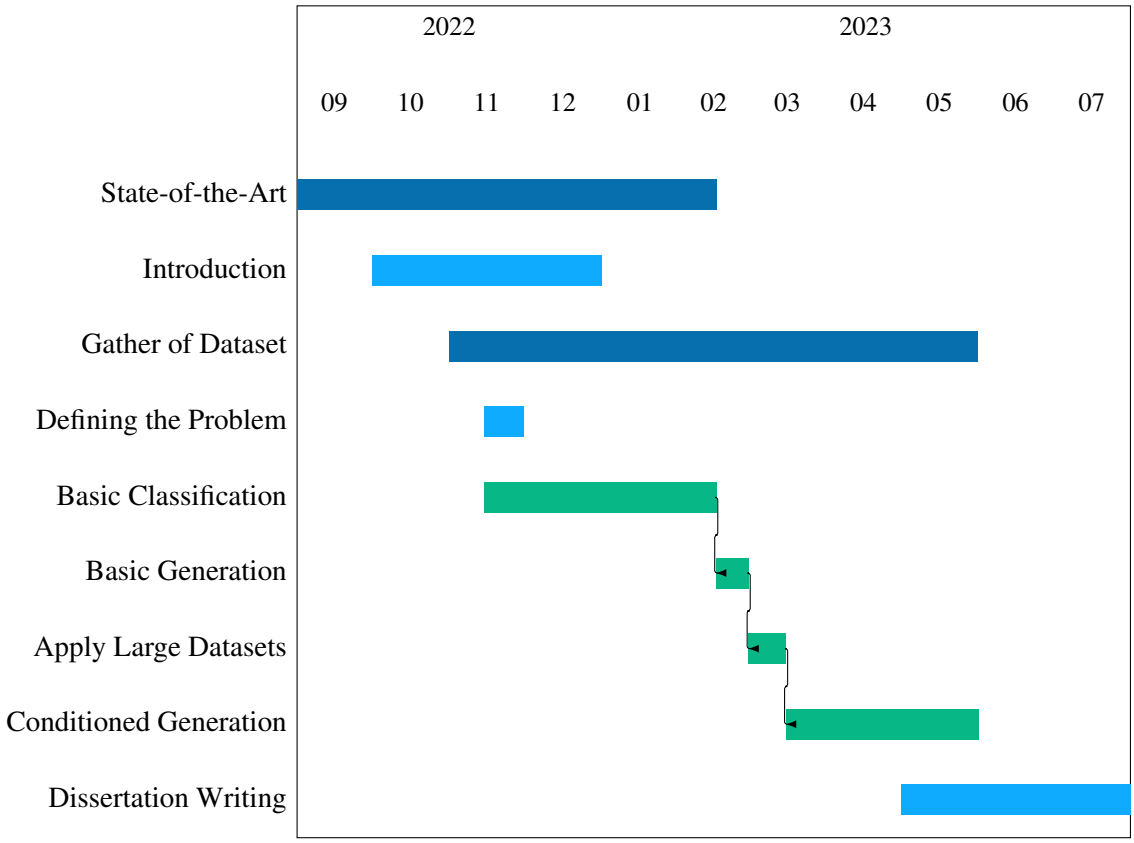

Figure 4.1: **Work Plan Gantt Chart** where the dark blue represents researching, light blue represents writing, and green represents developing.

4.4 Evaluation

4.5 User Interface

4.6 Work Plan

As stated in the course's official calendar, the State-of-the-Art is to be carried out until February 2023. While this is being done task a parallel task of searching for datasets is required. This one might go further than the rest of the State-of-the-Art research as it is not essential for understanding the models. Also, there is no reason not to write the problem definition and the introduction with the context, motivation, and objectives. When a vague idea of what to do was set, it was time to start developing simple things. Developing is a small and incremental process. The idea described by the green lines is the same mentioned in section 4.1.2; this is, one may start with a simple problem (classification) on a simple dataset (AMNIST). And then slowly increase the complexity of the task and the dataset. Finally, when the process is done, the writer dedicates full time to the writing of this document.

Chapter 5

Conclusions

Given that the dissertation process is already halfway, giving a state of affairs of the objectives one proposed to accomplish is vital. The reader can revise these objectives in section 1.3.

The work in this first phase of the dissertation accomplished the first three objectives. The sea of machine learning research is vast, and a complete outline of the panorama would be a quasi-impossible task. However, the work in this document is more than enough if the reader desires a profound knowledge of current developments of audio generative models and the whole deep learning field.

Developing practical systems is still in the very early stages, but positive results when handling Mel-Spectrograms for classification were already achieved, showing promising signs for future implementations.

At last, this state-of-the-art research can, in the future, be compiled into a survey. Students and researchers alike could base their study on the field with the developed survey, proving that the work present here could already positively impact the deep learning community.

References

[1] ISO 12913-1:2014(en), Acoustics — Soundscape — Part 1: Definition and conceptual framework, 2014.

[2] 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.

[3] 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].

[4] 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].

[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.

[6] 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.

[7] 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.

[8] 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.

[9] 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].

[10] 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].

[11] 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].

[12] 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.

[13] Tsz-Him Cheung and Dit-Yan Yeung. {MODALS}: Modality-agnostic Automated Data Augmentation in the Latent Space. In *International Conference on Learning Representations*, 2021.

[14] 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].

[15] Francois Chollet and others. Keras, 2015. Publisher: GitHub.

[16] Christopher Olah. Understanding LSTM Networks, August 2015.

[17] 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.

[18] 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.

[19] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. Publisher: IEEE.

[20] 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.

[21] 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.

[22] 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].

[23] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial Audio Synthesis, February 2019. arXiv:1802.04208 [cs].

[24] Constance Douwes, Philippe Esling, and Jean-Pierre Briot. Energy Consumption of Deep Generative Audio Models, October 2021. arXiv:2107.02621 [cs, eess].

[25] Konstantinos Drossos, Samuel Lipping, and Tuomas Virtanen. Clotho: An Audio Captioning Dataset. *CoRR*, abs/1910.09387, 2019. arXiv: 1910.09387.

[26] Peter Elsea. Basics of Digital Recording, 1996.

[27] 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].

[28] 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].

[29] J. D. Fernandez and F. Vico. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research*, 48:513–582, November 2013.

[30] 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.

[31] Seth* Forsgren and Hayk* Martiros. Riffusion - Stable diffusion for real-time music generation, 2022.

[32] Alexander L. Fradkov. Early History of Machine Learning. *IFAC-PapersOnLine*, 53(2):1385–1390, January 2020.

[33] 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.

[34] 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.

[35] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, May 2015. Number: 7553 Publisher: Nature Publishing Group.

[36] 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].

[37] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. Deep AutoRegressive Networks, May 2014. arXiv:1310.8499 [cs, stat].

[38] Hongyu Guo, Yongyi Mao, and Richong Zhang. Augmenting Data with Mixup for Sentence Classification: An Empirical Study, May 2019. arXiv:1905.08941 [cs].

[39] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949. Published: Hardcover.

[40] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models, December 2020. arXiv:2006.11239 [cs, stat].

[41] Jonathan Ho and Tim Salimans. Classifier-Free Diffusion Guidance, July 2022. arXiv:2207.12598 [cs].

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[43] 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.

[44] 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.

[45] 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].

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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].

[51] Stefano Kalonaris and Anna Jordanous. Computational Music Aesthetics: a survey and some thoughts. Dublin, Ireland, August 2018. Accepted: 2018-07-09.

[52] Chris Dongjoo Kim, Byeongchang Kim, Hyunmin Lee, and Gunhee Kim. AudioCaps: Generating Captions for Audios in The Wild. In *NAACL-HLT*, 2019.

[53] 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.

[54] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].

[55] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, December 2022. arXiv:1312.6114 [cs, stat].

[56] 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.

[57] 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].

[58] 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.

[59] Gianluca Maguolo, Michelangelo Paci, Loris Nanni, and Ludovico Bonan. Audiogmenter: a MATLAB Toolbox for Audio Data Augmentation, January 2022. arXiv:1912.05472 [cs, eess].

[60] 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.

[61] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. 1969.

[62] 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].

[63] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, 2013.

[64] Tom M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. McGraw-Hill, New York, 1997.

[65] 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.

[66] 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.

[67] 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].

[68] 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].

[69] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural Discrete Representation Learning, May 2018. arXiv:1711.00937 [cs].

[70] 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].

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Hieu Pham, Xinyi Wang, Yiming Yang, and Graham Neubig. Meta Back-Translation. In *International Conference on Learning Representations*, 2021.

[76] Domagoj Pluščec and Jan Šnajder. Data Augmentation for Neural NLP, February 2023. arXiv:2302.11412 [cs].

[77] Yanmin Qian, Hu Hu, and Tian Tan. Data augmentation using generative adversarial networks for robust speech recognition. *Speech Communication*, 114:1–9, November 2019.

[78] 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.

[79] 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].

[80] 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].

[81] 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.

[82] Edison Research. The Infinite Dial 2021, March 2021. Section: Featured.

[83] Danilo Jimenez Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows, June 2016. arXiv:1505.05770 [cs, stat].

[84] 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.

[85] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015. arXiv:1505.04597 [cs].

[86] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[87] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, 1986.

[88] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[89] 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.

[90] 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.

[91] 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.

[92] R. Murray Schafer. *The Tuning of the World*. Knopf, 1977. Google-Books-ID: SIufAAAAMAAJ.

[93] Connor Shorten, Taghi M. Khoshgoftaar, and Borko Furht. Text Data Augmentation for Deep Learning. *Journal of Big Data*, 8(1):101, July 2021.

[94] 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].

[95] David Sonnenschein. *Sound Design: The Expressive Power of Music, Voice, and Sound Effects in Cinema*. Michael Wiese Productions, 2001.

[96] 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.

[97] 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.

[98] 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.

[99] 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].

[100] University of York. What is Computer Science?

[101] 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.

[102] 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].

[103] Ashvala Vinay and Alexander Lerch. Evaluating generative audio systems and their metrics, August 2022. arXiv:2209.00130 [cs, eess].

[104] 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].

[105] 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.

[106] Lilian Weng. Flow-based Deep Generative Models. *lilianweng.github.io*, 2018.

[107] 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.

[108] 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].

[109] 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].