h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences

– Faculty Mathematics and Natural Science –

How can state of the art generating algorithms be used to create an interactive Artificial Intelligence that generates electronic music

Master of Science (M.Sc.)

Author:

Dennis Netzer

student number: 764885

30.11.2021

First Supervisor : Prof. Dr. Elke Hergenröther
Second Supervisor : Prof. Dr. Antje Jahn

DECLARATION

I, Dennis Netzer, declare that this thesis titled, "How can state of the art generating algorithms be used to create an interactive Artificial Intelligence that generates electronic music " and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Darmstadt, 30.11.2021

Dennis Netzer

ABSTRACT

This thesis deals with the use of state-of-the-art deep learning algorithms in the creative field of generating new content. The aim is to develop a deep learning model that can generate a random song in the electronic music genre without vocals. This is motivated by the objective to expand the existing customer base of the partner company of this thesis by presenting AI expertise on a fair or conference. The resulting model can serve as an exhibit.

There are three mayor ways to present audio data to a computer that are used for generative tasks. The physical analog waveform can be digitized by measuring the change in air pressure over time for a certain location, and then by taking a certain number of samples from these values each second. This resulting vector of values is the raw audio data. Every waveform is a superstition of several waveforms with a consistent frequency. These base frequencies can be extracted by the Fourier Transform and presented over time in a spectrogram. The last format called Midi incorporates musical knowledge into events. An event indicates which musical note from which instrument should be played in what way. Different authors tackled the task of generating music in various combinations of audio formats and deep learning architectures. The most intriguing is the WaveGAN [12] that uses a Generative Adversarial Network (GAN) to model raw audio data. It can only generate audio of one second. In 2018 an architecture called progressive growing of GANs [31] showed that GANs can be used to generate over one million pixels in a coherent image when trained iteratively. This magnitude would result in audio of over a minute. This thesis implements this algorithm for raw audio.

Raw audio is the most challenging format due to its high dimensional properties. To reduce computational cost, the generated song is kept at eleven seconds, resulting in 262,144 samples for the final sampling rate. The generator uses a CNN architecture to upsample and analyze a latent vector to a song of eleven seconds. The discriminator is a mirrored version of the generator and downsamples a song to a single value, which indicates whether the discriminator thinks the song is real or fake. Both networks grow iteratively, starting with a low sampling rate and therefore fewer samples for the eleven seconds of audio. The sampling rate along with the output size of the generator and the input size of the discriminator is doubled every iteration. In that way the model first learns lower frequencies and global structure and later local details.

The results of the conducted model development and training showed that working with raw audio data has very high computational requirements. The model can successfully generate songs that can be steered by the user. The audio lacks global coherence and still contains noise. Increasing the kernel size of the convolutional networks could improve global structure but

would increase the model size considerably. The occurring noise is mainly due to the small amount of training data and the shorter training time compared to networks of similar size. Some further development and training resources are needed to use the model as an exhibit for a fair or conference.

ZUSAMMENFASSUNG

Diese Arbeit befasst sich mit dem Einsatz modernster Deep-Learning-Algorithmen im kreativen Feld der Generierung neuer Inhalte. Ziel ist es, ein Deep-Learning-Modell zu entwickeln, welches einen zufälligen Song im elektronischen Musik Genre ohne Gesang generieren kann. Dies ist motiviert durch die Zielsetzung, den bestehenden Kundenstamm des Partnerunternehmens dieser Arbeit zu erweitern, indem KI-Expertise auf einer Messe oder Konferenz präsentiert wird. Das resultierende Modell kann als Exponat dienen.

Es gibt drei Möglichkeiten Audiodaten, die für generative Aufgaben verwendet werden, einem Computer zu präsentieren. Die physikalische analoge Welle kann digitalisiert werden, indem die Veränderung des Luftdrucks über die Zeit an einem bestimmten Ort gemessen wird und aus den Messungen jede Sekunde eine durch die Samplingrate definierte Anzahl an Messwerten gespeichert werden. Dieser resultierende Vektor von Werten sind die rohen Audiodaten. Jede Audiowelle ist eine Überlagerung von mehreren Wellen mit einer einheitlichen Frequenz. Diese Grundfrequenzen können durch die Fourier-Transformation extrahiert und in einem Spektrogramm mit zeitlicher Abhängigkeit dargestellt werden. Das letzte Format, Midi genannt, enthält musikalisches Wissen in Form von Ereignissen. Ein Ereignis gibt an, welche Musiknote von welchem Instrument auf welche Weise gespielt werden soll. Verschiedene Autoren haben sich mit der Aufgabe befasst, Musik in verschiedenen Kombinationen von Audioformaten und Deep-Learning-Architekturen zu erzeugen. Am faszinierendsten ist das WaveGAN [12], das ein Generatives Adversariales Netzwerk (GAN) zur Modellierung von Audio-Rohdaten verwendet. Es kann nur Audio mit einer Länge von einer Sekunde erzeugen. Im Jahr 2018 zeigte eine Architektur namens Progressive Growing of GANs [31], dass GANs verwendet werden können, um über eine Million Pixel in einem zusammenhängenden Bild zu erzeugen, wenn sie iterativ trainiert werden. Diese Größenordnung würde zu einem Audiosignal von über einer Minute führen. In dieser Arbeit wird dieser Algorithmus für rohe Audiodaten implementiert.

Rohes Audiomaterial ist aufgrund seiner hohen Dimensionalität das schwierigste Format. Um die Rechenzeit zu reduzieren, wird der generierte Song auf elf Sekunden beschränkt, was zu 262.144 Samples für die endgültige Samplingrate führt. Der Generator verwendet eine CNN-Architektur, um einen latenten Vektor auf einen Song von elf Sekunden hochzuskalieren und zu analysieren. Der Diskriminator ist eine gespiegelte Version des Generators und skaliert einen Song auf einen einzigen Wert herunter, der angibt, ob der Diskriminator den Song für echt oder unecht hält. Beide Netzwerke wachsen iterativ, wobei sie mit einer niedrigen Samplingrate und daher mit weniger Samples für die elf Sekunden Audio beginnen. Die Samplingrate

wird zusammen mit der Ausgangsgröße des Generators und der Eingangsgröße des Diskriminators bei jeder Iteration verdoppelt. Auf diese Weise lernt das Modell zunächst die niedrigeren Frequenzen und die globale Struktur und später die lokalen Details.

Die Ergebnisse der durchgeführten Modellentwicklung und des -trainings zeigten, dass die Arbeit mit rohen Audiodaten sehr hohe Rechenanforderungen hat. Das Modell kann erfolgreich Songs generieren, die vom Benutzer gesteuert werden können. Den Audiodaten fehlt es allerdings an globaler Kohärenz und sie enthalten immer noch Rauschen. Eine Vergrößerung der Kernelgröße der CNNs könnte die globale Struktur verbessern, würde aber die Modellgröße erheblich erhöhen. Das auftretende Rauschen ist hauptsächlich auf die geringe Menge an Trainingsdaten und die kürzere Trainingszeit im Vergleich zu Netzen ähnlicher Größe zurückzuführen. Es sind noch einige Entwicklungs- und Trainingsressourcen erforderlich, um das Modell als Exponat für eine Messe oder Konferenz zu verwenden.

ACKNOWLEDGMENTS

Zu aller erst möchte ich meinem Mitstreiter Sebbo danken der die Bearbeitungszeit um einiges angenehmer gemacht hat, ganz nach dem Motto geteiltes Leid ist halbes Leid. Danke für das Anhören der Ideen, auch wenn du sie im Detail nicht verstanden hast. Das hilft sehr beim Ordnen der Gedanken. Das gemeinsame Trainieren hat auch geholfen einen klaren Kopf zu bewahren. Des Weiteren ein riesengroßes Dankeschön an Mo, der uns den Schlüssel für den Aufenthaltsraum im Wohnheim klargemacht hat, damit wir dort in Ruhe, auch außerhalb der offiziellen Öffnungszeiten, unsere Arbeit schreiben konnten. Ohne einen solchen dedizierten Ort wäre es unmöglich gewesen, die nötige Konzentration über mehrere Stunden aufrechtzuerhalten. Die HDA hat es ja leider für nötig gehalten uns derartige Räume abzusperren. Auch ein Danke an alle, die vor allem in den letzten drei Monaten Verständnis gezeigt haben, dass die Masterarbeit Vorrang zu anderen freizeitlichen Aktivitäten hatte.

Vielen Dank auch an mein Partnerunternehmen und an alle Betreuer. Danke, dass ich die nötige Zeit, die nötigen Ressourcen und das Vetrauen bekommen habe, diese Arbeit erfolgreich abzuschließen.

Big up also to Conor and Harry for double checking the grammar of this thesis. Maybe you could at least read some good craic and learn one or two things. I hope Corona lets us see each other in summer next year.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

Acronym	Meaning
MIDI	Musical Instrument Digital Interface
CSV	comma separated values
TOH	Threshold of hearing
ADC	analog-to-digital-conversion
CD	Compact Disc
FNN	Feed Forward Network
DNN	Deep Neural Network
w.r.t.	with respect to
CNN	Convolutional Neuronal Network
RNN	Recurrent Neuronal Network
LSTM	Long-short-term-memory
VAE	Variational Autoencoder
Graphics Processing Unit	GPU
ReLU	Rectifire Linear Unit
RAM	random-access memory
BatchNorm	Batch Normalization
VAE	Variatioal Autoencoder
GAN	Generative Adversarial Network
D	Discriminator
G	Generator
JS	Jensen-Shannon
WGAN	Wasserstein Generative Adversarial Network
DCGAN	Deep Convolutional Generative Adversarial Network
VQ-VAE	Vector Quantized Variatioal Autoencoder
LSTM	Long-Short-Term-Memory
GRU	Gated Recurrent Unit
WGAN-GP	Wasserstein Generative Adversarial Network with gradient penalty

Part I

THESIS

1 INTRODUCTION

1.1 MOTIVATION

Through the ever-growing amount of data and the ever-growing computational capabilities, also German companies have picked up on the trend of using the already existing data to generate new insights and intelligent machines. This new field of Data Science and Artificial Intelligence (AI) is becoming more and more popular, not only in the research community but also in the business sector. This thesis is written in cooperation with ORDIX AG, a German consultant company based in Paderborn. The main expertise and what the company is well-known for is everything related to databases. In the recent years the goal has been to acquire knowledge in newer technologies in the areas of Big Data, Cloud and Data Science and to communicate this new expertise to already existing but also new customers. The Data Science department is the most recent one. Besides building the actual expertise, the marketing for this expertise is of great importance.

To get the attention of possible customers and to demonstrate its expertise, ORDIX AG attends fairs and conferences every year. The aim of this thesis is to gain expertise in the area of AI and to develop some sort of exhibit that can be presented on fairs and conferences to show this expertise. Music is a great way to call for attention because everyone is in some way attached to it. Furthermore, it intrigues people what a computer is already able to do. Therefore, the AI should be able to generate a random song and the user should be able to influence it to get some interactive experience.

1.2 SCOPE

The field of generative algorithm is large and has a long history. In this thesis, only deep learning techniques will be examined. These techniques only became popular within the last decade through the already mentioned vast amount of data and computational advancements. Concretely, the deep learning architectures Generative Adversarial Networks and Autoencoders with their various implementations will be considered and compared. Therefore, papers that already make use of deep learning techniques for audio generation will be reviewed. The aim is to use new knowledge to improve upon these already existing architectures. Transformer networks and the attention mechanism are out of scope for this thesis. Covering these topics in detail would go exceed the usual volume, and another student is simultaneously writing his thesis about these type of networks for the ORDIX AG and is already gathering the wanted expertise.

Audio data can be presented to a computer in several formats. Different approaches use these different types of format to already generate audio data to some success. The aim is to gather knowledge about the different representations, to compare them and to select a suitable one for this thesis.

The resulting program should allow for an interactive music generation process. Therefore, some sort of user interface is needed. The scope of this thesis does not include designing a full executable product, but rather to focus on the algorithm and to allow for some kind of user interaction. The AI will focus on learning music from the electronic music genre without vocals to make the task manageable.

In this thesis instead of AI the more academically term model or network is used.

1.3 STRUCTURE

The second chapter "Audio Signal Processing" explains what sound is in the physical world and how it can be captured and digitized such that it can be processed by a computer. The digitized signal can be analyzed in its raw format or converted into some higher level representation. The raw format along with three other audio data representations are explained at the end of the chapter. Chapter three "Deep Learning Algorithms" deals with the algorithm used to process the data. The three typical types of neural networks, feed forward, convolutional and recurrent, are explained. Every network has certain hyperparameters that have to be set prior to the training. These hyperparameters and their impact on the resulting model are presented. In the second part of the chapter, the generative architectures Autoencoders and Generative Adversarial Networks are explained along with advanced implementations. The next chapter analyzes current architectures used for audio generation. It shows what the algorithms are already capable and lays the foundation for chapter five. In this chapter, the different audio representations and possible model architectures are discussed and the most suitable ones for this thesis selected. Furthermore, requirements for the outcome of the developed model are defined. The chapter "Concept" explains the idea that is used to improve one of the architectures of chapter four and how this strategy can help to achieve the aim of the thesis and the defined requirements. Chapter seven then elaborate the developed concept and shows how this can be implemented using Tensorflow and Keras in the Python programming language. This part is accompanied by a Git repository that contains the code and the data. Chapter eight discusses the results that the model is able to produce. The last chapter summarizes the thesis and gives an outlook of what still has to be done and how the developed model can be further improved.

2 AUDIO SIGNAL PROCESSING

To approach the task of generating music, the first step is to understand music itself and how it can be presented to a machine. As will be shown, there are several ways to encode music in different levels of abstraction.

It is not the aim of the author to explain in-depth musical concepts, rather to give an overview and to explain the aspects necessary to understand the approaches that have been taken by the author and by the authors of the analyzed papers to generate new music.

2.1 AUDIO REPRESENTATION

In its most basic form, sound can be seen as waves travelling through the air as alterations in air pressure. The vibration of an object causes the air molecules to oscillate and transmit energy. This deviation in air pressure from its usual value can be measured and encoded into an audio signal. This audio signal contains all information needed to reproduce a piece of music.

2.1.1 *Sound and Waveforms*

In physical terms, waves can be divided into longitudinal and transverse waves. The difference is the direction of the particles' movement respective to the direction of the energy transport. When an object starts vibrating, the surrounding air molecules start to move back and forth, energizing the adjacent molecules, triggering a chain reaction. This motion results in regions of compression and regions of rarefaction. Due to this back and forth movement, the energy is transmitted in the direction of particle movement, resulting in a longitudinal wave. The upper part of Figure 2.1 depicts this longitudinal character of the wave. If now the shift in air pressure is measured, with compression resulting in higher pressure and rarefaction resulting in lower pressure, at every point between the sound source and the receiver, the sound wave can be described with a sine wave, as shown in the lower part of Figure 2.1

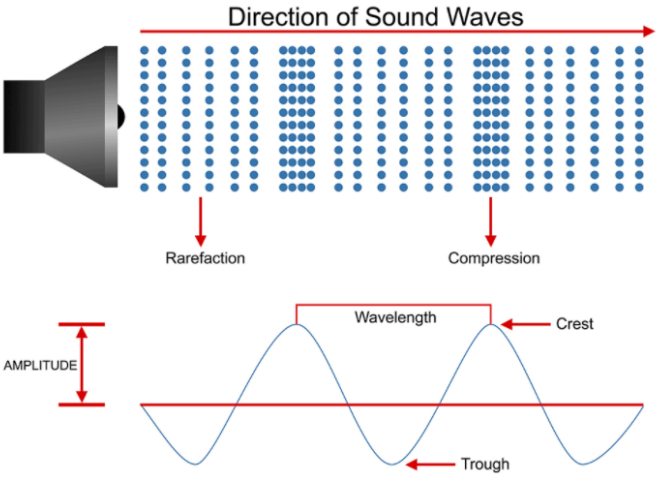

Figure 2.1: Soundwave transmitted through air [44]

The amplitude encodes the intensity of the displacement of air molecules, reflecting the loudness of the sound. The frequency of the soundwave (how many crests or troughs occur within a second) reflects the pitch of the sound, meaning how the sound is perceived in terms of higher and lower sounds.

2.1.2 *Properties of Sound*

As was shown, sound can be described with sinusoids. [9] To discuss this concept further, a classical sine function is shown in equation 2.1 and discussed in this section.

$$x(t) = A\cos(2\pi(\omega * t - \phi)) \tag{2.1}$$

2.1.3 *Frequency*

Frequency (ω) is a physical term to describe how many oscillations happen per second, measured in Hertz (Hz) or $1/s$. The higher the frequency of the sinusoid, the higher the perceived sound. Humans can generally hear sounds in between 20 Hz and $20,000$ Hz. If the oscillations occur in a consistent pattern, the sound wave is called periodic and the produced sound is called a tone. The wave form in Figure 2.1 is periodic and can be defined by the equation 2.1. Music theory coined the term pitch to refer to the perception of higher and lower sounds. A common pitch is the musical note A4, which is represented by a sinusoid with a frequency of 440 Hz.[37] Humans perceive two pure tones to be similar, if they differ by a power of two.[56] Hence, a tone with frequency 880 Hz sound similar to the pitch A4 with 440 Hz. To account for this relationship, music theory introduced the concept of an octave, where a pure tone with 880 Hz is called A5, being one octave or eight notes above A4. Since humans perceive sound logarithmically, the

perceived distance between the pitches A4 and A5 is the same as between A5 and A6. [38]

In reality, when you play a certain note on an instrument, there is not just one single harmonic sinusoid with one frequency produced. The resulting sound is a superposition of several harmonic sinusoids. If the note A4 is played, for example, the resulting sound also consists of the frequencies of the pitches A5 and A6. All occurring sinusoids are called partial, and the sinusoid with the lowest frequency is called the fundamental frequency. If the frequency of a sinusoid is an integer multiple of the fundamental frequency, it is called a harmonic partial. Instruments are generally tuned so that all occurring partials are harmonic. [16]

2.1.3.1 *Amplitude*

The amplitude (A) of the sinusoid determines the amount of energy that is transmitted by a source of sound per second. This amount can be calculated with the formula $A^2/2$ and is also called the power of sound, measured in watt (W). [9] A more known term to describe the energy of sound is the intensity level expressed by decibel (dB). The sound intensity describes the sound power per unit area (W/m^2), with dB expressing the ratio between two intensity values. Decibel is a logarithmic scale where an increase of 3dB corresponds with approximately a doubling of intensity. An overview of common intensity values is presented in figure 2.2. The threshold of hearing (TOH) is defined as the sound intensity of a pure tone that a human is able to hear. [38]

Source	Intensity	Intensity level	× TOH
Threshold of hearing (TOH)	10^{-12}	0 dB	1
Whisper	10^{-10}	20 dB	10^2
Pianissimo	10^{-8}	40 dB	10^4
Normal conversation	10^{-6}	60 dB	10^6
Fortissimo	10^{-2}	100 dB	10^{10}
Threshold of pain	10	130 dB	10^{13}
Jet take-off	10^2	140 dB	10^{14}
Instant perforation of eardrum	10^4	160 dB	10^{16}

Figure 2.2: Typical intesity values given in W/m^2, decibles and by a factor compred with the TOH [38]

The term loudness refers to a perceptual property of sound. Sounds with the same intensity may be perceived differently by an individual depending on the age of the individual, the frequency of the sound and the duration of the sound. [17]

2.1.3.2 *Phase*

The phase (ϕ) does not have a perceptual correspondence in the physical world. It solely indicates where the sinusoid is in its cycle at time zero. So, it can be thought of as a time-shift of the sinusoid. [9]

2.1.3.3 *Timbre*

Timbre is a concept that helps musicians to distinguish between a musical tone that is played with the same pitch and the same loudness by different instruments. To make timbre more tangible, the energy behavior over time and the energy distribution across the occurring partials is analyzed.

Figure 2.3 shows that at the beginning of a sound there is a spike of energy called the attack phase where the sound builds up. In that phase, there are many non-harmonic and non-periodic partials present due to noise that comes with the physical behavior of an instrument. After this spike in energy, the noise components fade away in the decay phase and the sound stabilizes into the sustain phase. In the final release phase, the musical tone fades away. The duration, the amplitude, and the shape of these phases vary a lot depending on the instrument, thus determine how the sound is perceived. [56]

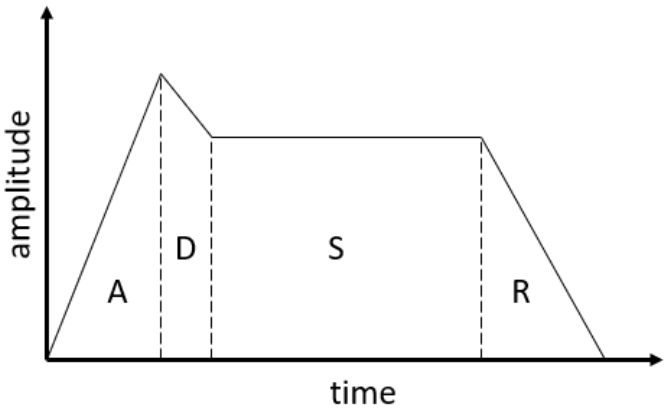

Figure 2.3: Attack (A), Decay (D), Sustain (S), Release (R) phases of a muiscal tone

To analyze which partials make up a certain sound, the intensity of the occurring frequencies can be visualized. The resulting plot is called a spectrogram. A more detailed explanation is given in section 2.3.

2.1.3.4 *Duration*

As seen in the preceding subsection, the duration of a sound influences its timbre and its loudness, and therefore how the sound is perceived. It refers to how long a certain note is played.

2.2 DIGITAL SIGNALS

So far, audio was presented as air pressure oscillations in the physical world, defined by certain sinusoids. These type of signals that occur in the real world are called analog signals. Analog signals are continuous and have therefore an infinite number of values. An example of an analog signal is the sine function in equation 2.1. To be able to process these continuous signals, it is necessary to convert it into some discrete representation that a

computer is able to save and manipulate. This conversion is called analog-to-digital-conversion (ADC) or digitization. [45] In the field of audio, the two techniques sampling and quantization are used.

2.2.1 *Sampling*

Sampling is a technique used to convert the continuous time values of an analog signal to a finite number that can be stored on a computer. Concretely, the applied procedure is the equidistant sampling where a positive real number T of samples with equal distance to each other is taken from the analog signal. This leads to the equation 2.2, where $x(n)$ is the n_{th} sample of the analog signal x_a, taken at time $n * T$. T is the sampling period and defines the distance in seconds between two samples. The inverse of T is called the sampling rate or the sampling frequency F_s, and indicates the number of samples taken per second, measured in Hertz (Hz).

$$x(n) := x_a(n * T) \tag{2.2}$$

To not lose spectral information, that means, the spectrum of the analog signal can be fully recovered from the spectrum of the discrete signal, the sample rate has to be selected large enough. C. E. Shannon and H. Nyquist laid the foundation for the Shannon Nyquist sampling theorem [40][57], which states that, if there is a sufficient condition for a sample rate established, a finite signal can capture all the spectral information of an analog signal. Specifically, the sampling rate must be double the size of the highest frequency that the discrete signal should capture. This sampling rate presented in equation 2.3 is known as the Nyquist rate, with the highest frequency F_{max} captured by the discrete signal called the Nyquist frequency. That is the reason songs on a Compact Disc (CD) are sampled with a sampling rate of $44,100\ Hz$. Humans can hear sounds up to about $20,000\ Hz$ so with this sampling rate, all frequencies that humans care about are captured by the discrete signal on the CD.

$$F_s = F_{max} * 2 \tag{2.3}$$

To better understand what happens when the sampling rate is below the Nyquist rate, an analog signal in the form of a sine wave with frequency of $1\ Hz$ is plotted in Figure 2.4 in blue. This analog signal is then sampled with the Nyquist rate of $2\ Hz$ and with a sampling rate of $1\ Hz$ and plotted with the red and yellow lines respectively. The sampled points have been interpolated with straight lines. The higher frequencies of the analog signal, that lay above the Nyquist frequency, occur in the discrete signal as lower frequencies. The analog signal in our example, sampled with a sample rate of $1\ Hz$, results in the yellow signal in a frequency of $\frac{1}{10} Hz$. Each frequency above the Nyquist frequency in the analog signal has a corresponding lower

frequency (an alias) in the discrete signal. These artifacts are thus called aliasing.

For a mathematical proof and more details, please refer to the original papers of H. Nyquist [40] and C. E. Shannon [57] or to [45].

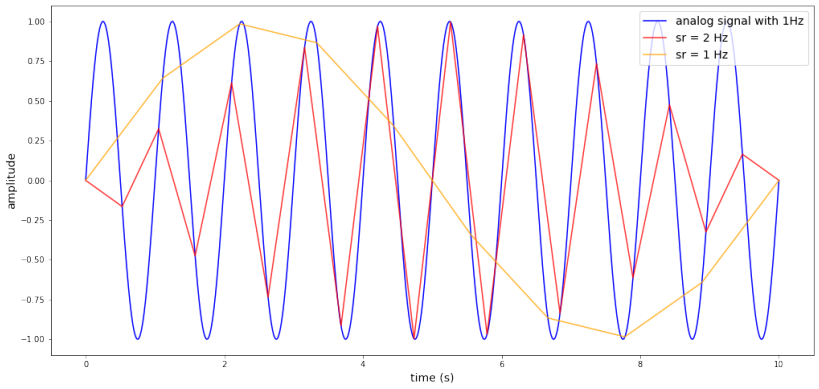

Figure 2.4: Sine wave with 1 Hz sampled with sample rates of 2 Hz and 1 Hz

2.2.2 *Quantization*

So far, it was shown how the continuous time component of an analog signal can be converted to a discrete representation. However, the amplitude of an audio signal in its physical form is also continuous and has to be discretized. Therefore, each amplitude value of the analog signal gets represented by a value from a finite set of values. The summed difference between the original and the quantized values is called the quantization error.

Figure 2.5 depicts the quantization procedure with the analog signal 0.9^t as an example. Every sample $x_q(n)$ is mapped to its closest value of the quantizer (here values between 0 and 1 with step size 0.1). The more levels of quantization are used, the lower the quantization error. In practice, this is often achieved by simply truncating the value to a certain precision.

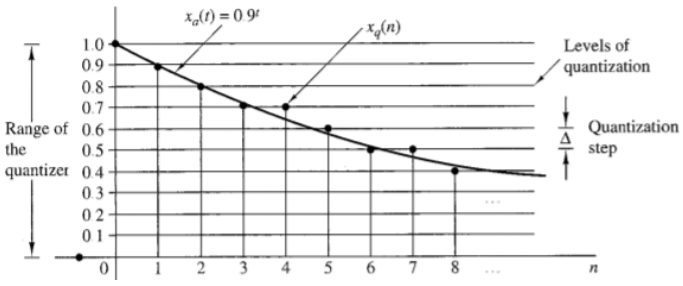

Figure 2.5: Quantization of an analog sinal 0.9^t [45]

For CD quality, 16bit quantization is used. That means, the amplitude is represented by 65,536 possible values. This value is again chosen in a way that the information loss due to the discretization is not noticeable for humans.

2.3 FOURIER ANALYSIS

This section will give a high-level overview about the Fourier Analysis named after the French mathematician and physicist Jean Baptiste Joseph Fourier and its application in digital signal processing, as it is needed to understand further concepts explained in this thesis. For technical and mathematical details and proofs, please refer to the literature linked in this section.

Fourier analysis describes the study of using trigonometric functions to define or approximate other, more complex functions. For audio processing, it can be used to figure out of which underlying frequencies a signal is composed.

2.3.1 *Fourier Transform*

Fourier transform is the specific process of transforming a signal that depends on time into a signal that depends on frequency, and is part of the Fourier analysis. This is also called a transformation from the time domain to the frequency domain. For audio signal processing, the idea is to decompose a given signal into its overlaying sinusoids. The signal is compared with various sinusoids of different frequencies. For each considered frequency, there is a magnitude coefficient and a phase coefficient determined. The magnitude coefficient is large for sinusoids that are similar to the signal. That means that the signal contains a periodic component at that sinusoid frequency. The phase coefficient indicates, for which phase of the sinusoid, the comparison yields the highest magnitude coefficient. Since sinusoids are periodic, the phase has to be considered only in a certain interval and the magnitude coefficients repeat themselves.

The original signal can be reconstructed with the magnitude and the phase coefficients of all the periodic components of the signal. Therefore, all sinusoids of all the frequencies are superimposed and weighted by their magnitude coefficient and shifted by their phase coefficient. This reconstruction is called the Fourier representation of the original signal. In Figure 2.6 a signal (blue line) is compared with different sinusoids (red line) with the frequencies 3 Hz, 10 Hz and 3.5 Hz in the plots (a), (b) and (c) respectively. In plot (a) and plot (b) there is substantial overlap between the red and the blue signal, and therefore the respective magnitude coefficient is large, meaning the blue signal probably contains frequencies of 3 and 10 Hz. The red line in plot (c) appears to have less overlap, and therefore the magnitude component of 3.5 Hz is lower. The phase coefficients were selected in a way that leads to the maximum overlap.

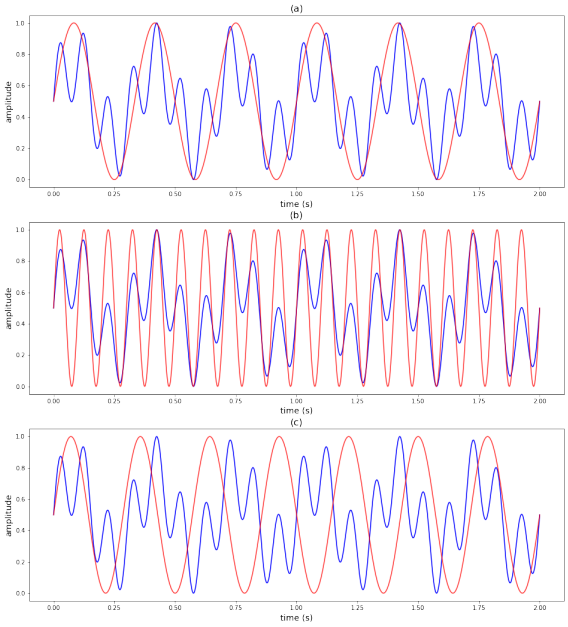

Figure 2.6: Comparison of ground signal with signal of various frequencies

Mathematically, to determine how similar two functions are, the integral of the inner product of these functions is taken. If the signals are similar, most of the coefficients of the values for some time t are both positive or both negative, resulting in a positive value. If the signals are dissimilar and their coefficients for some time t are different, the resulting values are negative, resulting in a lower integration value and therefore a lower magnitude coefficient. The pair of magnitude and phase coefficient is represented as a complex number in the complex space. The Fourier transform $\hat{f}(\omega)$ can then be calculated by Formula 2.4. Hence, the real part of the complex number is obtained by comparing the analog signal $f(t)$ to a cosine function and the imaginary part by comparing the analog signal to a sine function. For more details, please refer to [38] [41] [55].

$$\hat{f}(\omega) = \int_{t\in\mathbb{R}} f(t)\cos(-2\pi\omega t)dt + i\int_{t\in\mathbb{R}} f(t)\sin(-2\pi\omega t)dt \tag{2.4}$$

The result of the Fourier transform is perfectly displayed by Müller [38] with Figure 2.7 showing the resulting magnitude coefficients for the waveform of the note C4 played by different instruments. It shows that depending on the instrument, the magnitude of the present frequencies in C4 changes.

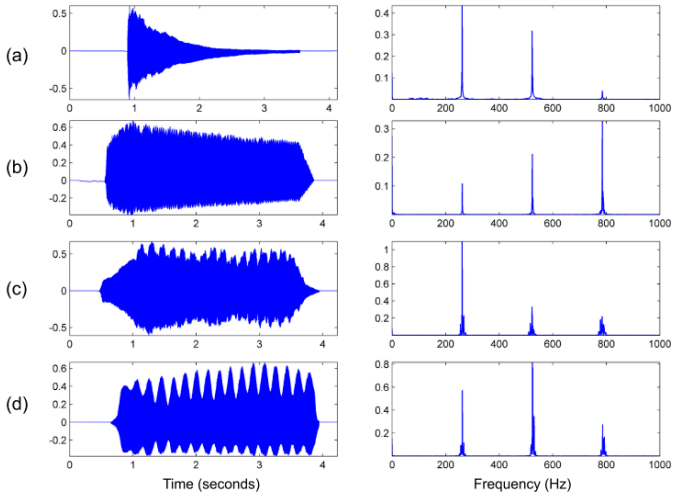

Figure 2.7: Waveform and magnitude coefficients of the note *C*4 of different intruments (a) Piano, (b) Trumpet, (c) Violin, (d) Flute

For digital signals instead of the integral, the sum of the pointwise multiplication of the sampled values is taken, where the regarded trigonometric function is sampled in the same way as the analog signal. To make it computational feasible, only a finite number of samples and a finite number of frequencies is considered. This is called the discrete Fourier transform. If certain mathematical conditions are enforced, this algorithm can be made much more efficient by exploiting redundancies of sinusoids with different frequencies, increasing its importance. The foundations for this faster algorithm, called the fast Fourier transform, were laid by Joseph Fourier and Carl Friedrich Gauss [26].

2.3.2 *Spectrograms*

So far, the Fourier transform transmuted the signal from the time domain to the frequency domain. The information about the time was lost in this process. If an algorithm is to generate a new sound, it has to be provided with information about the distribution of the frequencies over time. Dennis Gabor laid with his in 1946 published paper [19] the foundation for an algorithm now called short-time Fourier transform. He applied the Fourier transform to time windows. Therefore, a window function, that is zero outside a certain interval, is multiplied with the signal and shifted along the time axis by a certain hop length, producing the respective signal windows.

In the discrete case, a certain number of samples lie in each window. For each window, a finite number of frequencies is considered when computing the Fourier transform. This results in bins with complex numbers for all considered frequencies for each window, as explained in 2.3.1. The squared magnitude of these complex numbers can now be represented in color in a 2-d heatmap, with the time on the horizontal axis and the frequencies on the vertical axis. This representation is called a spectrogram and offers an overview of the frequency distribution of a signal over time. To make also

frequencies with little energy visible, the magnitude can be transformed into a dB representation.

A spectrogram for a C4 major scale played by a piano is plotted in Figure 2.8 with a logarithmic scale for the frequency. The MP3 file for the audio to create the spectrogram was downloaded from [18]. The fundamental frequency of the note C4 is 262 *Hz*, which is shown by the lowest most left horizontal line. The horizontal lines above are the harmonic partials of the note C4. The scale is played upwards from C4 to the C5 and reversed.

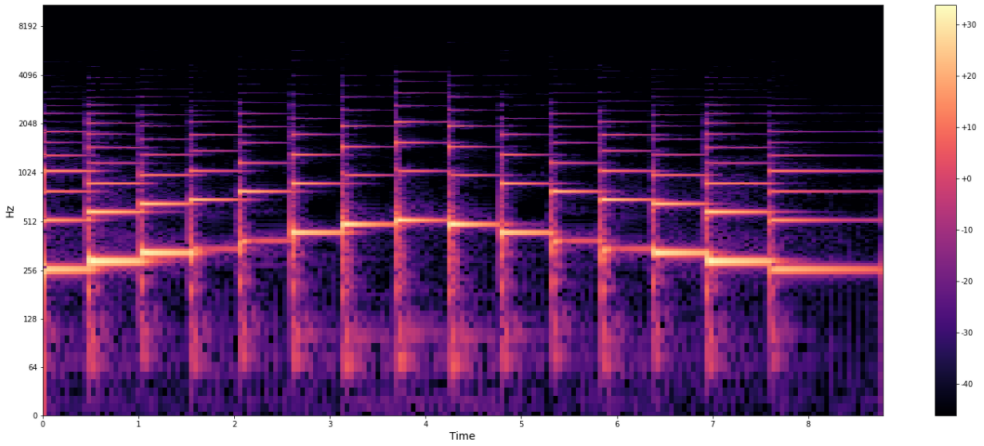

Figure 2.8: Spectogram for Major C4 scale

2.4 ENCODING OF AUDIO

So far, the reader was introduced to the physical representation of audio, how it can be saved on a digital device and to some first processing steps that lead to a visual representation of frequencies over time. To teach a computer to produce new content, it has to be presented with information that can be used by the algorithms presented in chapter 3. Four types of encoding will be presented in this section.

2.4.1 *Raw Audio*

Raw audio is the most basic format. The amplitude values of the sound wave are saved as floating-point numbers in an array-like format. If the sample rate and the bit resolution is high enough, there is nearly no information lost in the preprocessing step. The downside is the enormous amount of data that is used to save audio files. If the 16 bit encoding with a sample rate of $44,000$ *Hz* of a CD is considered, one minute of audio takes up $16bits * 44000(1/s) * 60s = 42,240,000bits \approx 5.28MB$ memory.

2.4.2 *Sheet Music Representation*

Sheet music is arguably the most known format of encoded music. In simple words, it is a visual presentation of musical notes. In an orchestra, the

different musicians with their instruments follow the instructions on which notes when and how to play printed on paper to create the musical play. Therefore, to understand sheet music, profound knowledge of music theory is needed. The instructions on the sheet music are not distinct. A musician can vary the tempo, dynamics, and articulation of the printed notes, thus creating their own style. [38] Due to its non-uniqueness in the interpretation and its graphical presentation, this type of encoding usually is not used for generative tasks.

2.4.3 *Symbolic Representation*

Symbolic representation is the most used format for generative tasks. Several distinct entities are defined that tell the computer what specific sound, in which loudness and for how long it has to produce. There is a finite number of entities, so that each piece of music is fully defined by its composition of entities. The two most common formats are the piano-roll representation and the Musical Instrument Digital Interface (MIDI) representation. A piano-roll representation is a two-dimensional sheet where the abscissa encodes the time and the ordinate encodes the pitch to play. With this encoding, the beginning, the duration, and the pitch of the sound is uniquely defined. As the name indicates, this format is mainly used to encode a play performed on a piano. It lacks information on how fast the keys were pressed, therefore missing the loudness of the played note.

MIDI files contain information on how a specific instrument was played or how a computer should play an instrument. The most important features for this task are MIDI note number, note-on, note-off and velocity. In MIDI files for a piano, the note-on and note-off events tell the computer when to press and release a key. Each note-on or note-off event is equipped with its corresponding MIDI note number and velocity. The velocity feature defines the velocity with which the key is pressed or released encoded with an integer between 0 and 127 and thus controlling the intensity of the produced sound or its attenuation. This velocity value corresponds with the concepts of timbre and loudness that were introduced earlier in this chapter. The MIDI note number encodes the pitch of the sound with an integer between 0 and 127. For the piano example, it indicates which key is to be pressed. With the equation 2.5 the corresponding frequency F to the MIDI number p can be inferred. The note A4 with the MIDI note number 69 hence leads to the frequency 440 Hz.

$$F_{pitch}(p) = 2^{(p-69)/12} * 440 \tag{2.5}$$

To allow multiple instruments playing concurrently, a MIDI event can provide a channel to which an instrument can be assigned. MIDI files measure the time in ticks. At the beginning of the file, the number of ticks per quarter note is defined. The time feature of a MIDI event consists of the number of ticks that determines after how many ticks after the last MIDI event the corresponding MIDI event is executed. Therefore, the duration of a note is

specified by how many ticks after the note-on event, the note-off event is executed.

Midi files do not contain the audio signal itself, and thus their size is small compared to other formats such as MP3 or WAV. Thanks to their small size, their standardized format and their easy manipulation, MIDI files are widely used to save, distribute and generate music. Since 1985 there is even an organization that helps to promote and further develop this technology. [60] The MIDI format itself is not human-readable, but can be converted to a comma separated value (CSV) format. Since September 2019, there is a respective python package called midicsv. [46]

2.4.4 *Image Representation*

As seen in subsection 2.3.2 music can be represented as an image. Therefore, image processing and image generation techniques can be applied to the spectrogram representation. The pixel values of the spectrogram are saves as floating-point number in an 2-d array.

3

DEEP LEARNING ALGORITHMS

In this chapter, the deep learning algorithms and overall architectures that are used in generative tasks are discussed. The aim is to give an overview over the possibilities of choices in the field of music generation and provide information about the parameters that have to be considered when creating a deep learning system.

3.1 ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks are widely known in their first developed and most basic form, the Feed Forward Network (FFN). For spreadsheet like data, this type of network performs good. It struggles with spatial data or data with a time dependency, as it is the case for audio data. Thus, other types of network have been developed.

For an extensive overview about the historical developments and contributions of different researchers that lead to the presented algorithms, please refer to the thorough technical report of Jürgen Schmidhuber [54], an internationally acknowledged researcher in the field. More recent advancements and claims to specific algorithms will be appropriately cited.

3.1.1 *Feed Forward Networks*

To provide an overview about the general functionality of Artificial Neural Networks and to provide a better entry point for readers with less background in this field, the principles of basic FFN will be elucidated in this first subsection. Most of these principles are also used in the other types of neural networks.

3.1.1.1 *General structure of a FFN*

A FFN consists of an input layer, one or more hidden layer and one output layer. If it has more than one hidden layer, it is also called a deep neural network. The input layer is a representation of some numerical input data that is sent through the network. One neuron represents one feature. The neurons of the hidden layers conduct the calculations of the network. Every neuron receives the output from the preceding layer, runs its calculations and sends its output to the succeeding one. In the output layer, the information is consolidated and transformed to a suitable interval of values. The number of output values depends on the number of neurons in this layer. Because every neuron is connected with all the neurons of the preceding and the

succeeding layer, this type of layers is also called a fully connected layer. In Figure 3.1 this structure is shown.

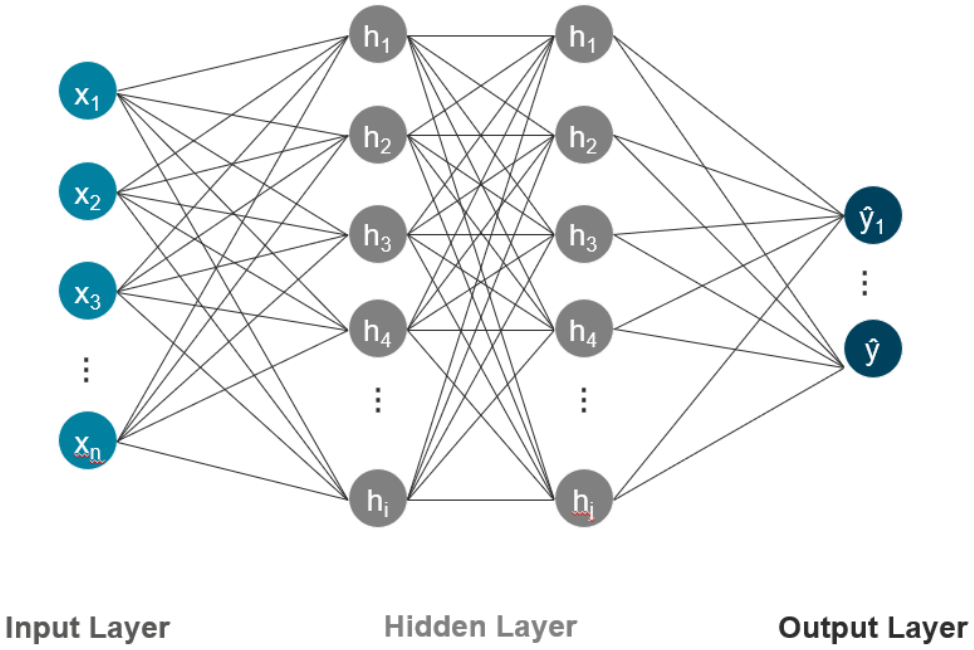

Figure 3.1: Structure of a feed forward network

3.1.1.2 *Calculations of a single neuron*

To better understand the calculation, a single neuron and its dependencies is depicted in Figure 3.2. This schema is the same for neurons of the hidden and the output layer. The neurons of the input layer are merely a representation of the input values without any calculation happening. Every connection between two neurons is assigned a weight w_i. The output of a neuron is represented with an a_i.

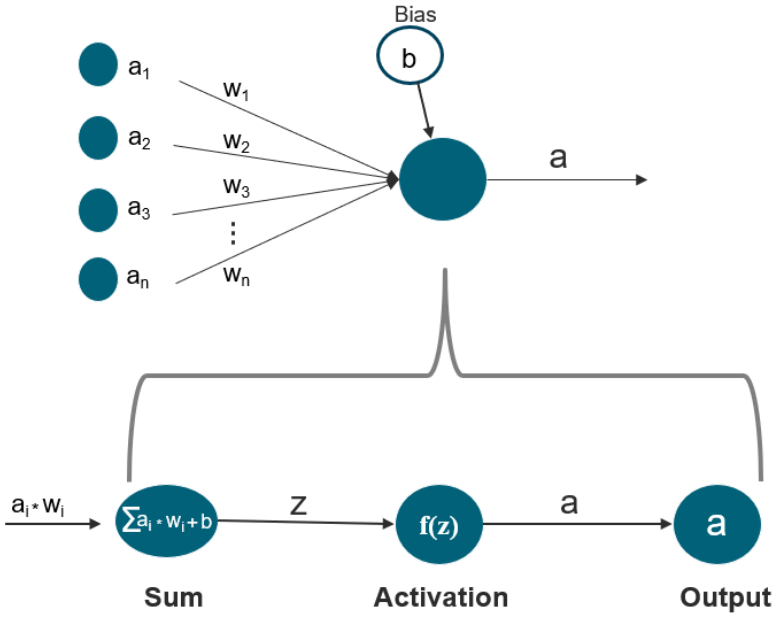

Figure 3.2: Schema of the calculation of a single neuron

The considered neuron takes the products of the outputs of the preceding neurons with their respective weights $a_i * w_i$ as an input. The weights therefore determine in what way the output is considered in the next layer. These products are summed up, and a bias b is added. The bias is a floating-point number that shifts the input into a positive or negative direction. The bias and the weights are adjusted during the training of the network.

In the next step, this sum is fed into an activation function f, which maps each input value to an output value. The simplest activation function is a linear function $f(x) = x$ where each output value equals its input value. For better training stability and to get the desired output for neurons in the output layer, activation functions usually map the input values to a distinct interval of values. Activation functions are separately discussed in 3.1.4.2. The output of the activation function gets then sent to the next layer, or is returned as the prediction, in the case of the output layer.

3.1.1.3 *Training process*

FFN are trained in a supervised manner. That means the output of the network is compared to what was expected to be the network's output. Therefore, appropriate training data is needed. To explain the training concept, a FFN for a binary classification problem is considered. The network has a single output neuron with its output mapped to a value between zero and one. The output value hence signify how certain the network is that the input data is part of category one.

To assess the network's quality, a loss function has to be designed that gives feedback to the network on how well it is performing. For this example, the simplest loss function is the squared difference between the prediction and the true output value. The loss gets smaller when the network's predic-

tion is closer to the true value, and bigger when the network performs badly. That is a desired property of a loss function. Loss functions are discussed in more depth in section 3.1.4.4.

During the training process, the network adjusts its weights and biases to improve its performance assessed by the loss function. Mathematically, it tries to minimize the loss function by altering its trainable parameters. There is no analytical feasible solution to directly calculate the optimal values for all parameters of the network, in a way that minimizes the loss function. Therefore, a numerical approach called gradient descent is used. It calculates the gradient of the loss function with respect to (w.r.t.) the weights and biases. Then a step in the direction of the negative gradient is taken by updating the weights and biases accordingly. The aim is to update the parameters step by step until a local minimum is reached. Figure 3.3 illustrates this procedure graphically in a simplified way. Every red arrow represents one gradient descent step. The step size can be adjusted by a learning rate. A higher learning rate results in larger steps. Selecting a good learning rate is part of the optimization for gradient descent. A wide variety of optimization algorithms have been developed. Section 3.1.4.3 explains the most important optimizers for this thesis.

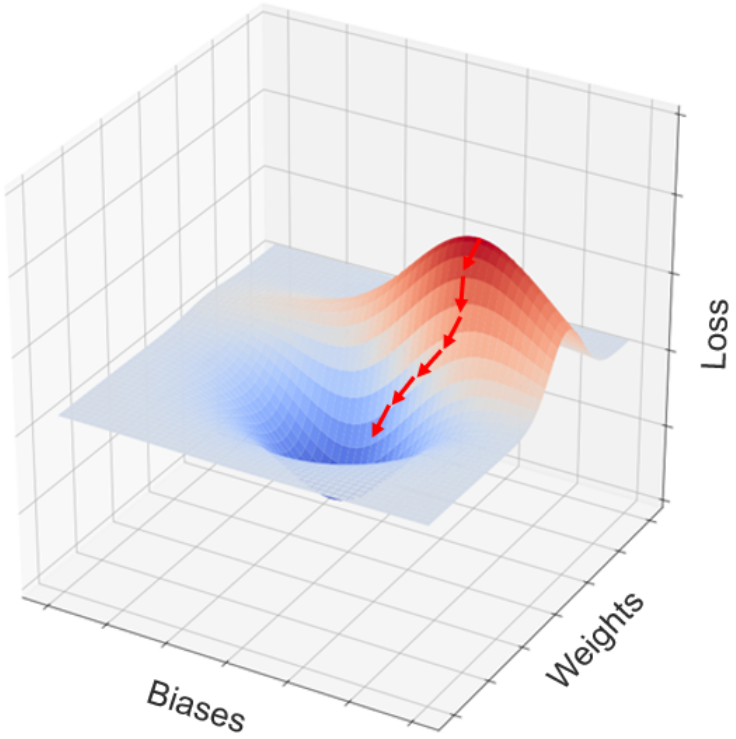

Figure 3.3: Gradient descent steps for a loss function represented by red arrows

The loss is calculated only with the output value of the network. Hence, the influence of the parameters of the output layer is directly related to the loss. The hidden layers do not have this direct feedback. Therefore, the one loss function has to be used to also update the weights and biases of the

hidden layers. Mathematically, the output of the network depends on all weights of the network. This relationship can be presented with a chain of products. For the simple network shown in Figure 3.4 and a linear activation function $f(x) = x$, this chain is given by equation 3.1.

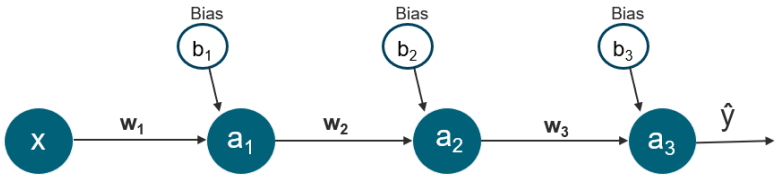

Figure 3.4: Simple FFN with two hidden layers and one neuron per layer

$$\begin{aligned}\hat{y} = a_3 &= a_2 * w_3 + b_3 = (a_1 * w_2 + b_2) * w_3 + b_3 \\ &= ((x * w_1 + b_1) * w_2 + b_2) * w_3 + b_3\end{aligned} \tag{3.1}$$

To update the parameters of the earlier layers, the differential of the loss function w.r.t to their parameters can be calculated using the chain rule. This backward flow of the error is called backpropagation. For the example network, the gradient of the loss function $Loss(weights, biases, y, X) = (y - \hat{y})^2$ w.r.t. the weight w_2 is calculated by equation 3.2 and w.r.t the weight w_1 by equation 3.2. It is calculated by plugging equation 3.1 into the loss function and then by differentiating this formula w.r.t. to the weight. The formula to update the weights is given by equation 3.4 with alpha being the learning rate.

$$\frac{\Delta Loss(weights, biases, y, X)}{\Delta w_2} = 2 * (y - \hat{y}) * a_1 * w_3 \tag{3.2}$$

$$\frac{\Delta Loss(weights, biases, y, X)}{\Delta w_1} = 2 * (y - \hat{y}) * x * w_2 * w_3 \tag{3.3}$$

$$w_{i+1} = w_i - \alpha \frac{\Delta Loss(weights, biases, y, X)}{\Delta w_i} \tag{3.4}$$

Note that the biases are updated the same way. For bigger networks, the formulas get exponentially more complex, but the underlying principles stay the same. For a more complex example of the presented principles, please refer to [4]. Further information can additionally be found in [15] and [24].

3.1.2 *Convolutional Neural Networks*

Convolutional neural networks (CNNs) are mostly known for their application in the field of Computer Vision for image processing, object detection

and classification. As the name indicates, CNNs have at least one layer that performs a mathematical operation called convolution. For this, the data has to exist in a grid-like topology. That means adjacent data points are related to each other. The pixels of images are numerical values in a 2-D grid, where a group of pixels is needed to define an object. But also audio data can be processed by convolutions. The audio data is merely a 1-D grid representation, with coherent data depending on time. Hence, there are 1-D and 2-D convolutional operations. Following, the concept of 1-D convolutions will be explained. But the same concept and wording applies to 2-D convolutions as well.

3.1.2.1 *Convolutions*

A convolution is basically the operation of applying a kernel function to some data with the explained format. Figure 3.5 shows a simple example of a kernel function that calculates the average value for a sample x_j and its two neighbors to the left and to the right.

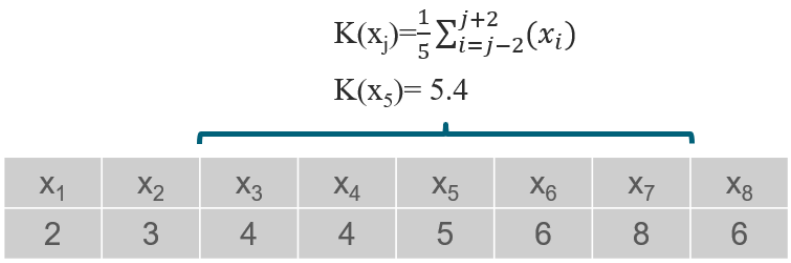

Figure 3.5: Simple convolution operation on a vector

The x values or the function that produces these values is called the input, the kernel function K is called the kernel and the output is referred to as the feature map of the convolution. In this example, the kernel is of size five. The feature map is calculated by sliding the kernel with a certain step size over the input data and calculating the value for every input sample. For a 2-D operation the kernel slides from left to right and after reaching the end it slides downwards to the next row of pixels. If the step size is one, the feature map has the same dimensions as the input. To calculate the feature map value for x_1 the input data has to be extended, since x_1 does not have neighbors to its left side. The easiest way is to pad the needed number of zeros to the input, in this case two zeros to the left side. This strategy is therefore called Zero Padding. Thus, the feature map value of x_1 is calculated by: $K(x_1) = \frac{0+0+2+3+4}{5} = 1.8$. Following this procedure, the full feature map for a step size of one is given by the vector $[1.8, 2.6, 3.6, 4.4, 5.4, 5.8, 5.0, 4.0]$ Other padding strategies will be explained when they are needed.

In CNNs for the convolution operation, the kernel function is a matrix of weights that is multiplied with the input values and updated during the training. The network learns which input values are more important and should therefore be assigned a higher weight. In CNNs, there is usually more than one kernel applied to the input, resulting in several feature maps.

In that way, the network has the possibility to portray different information by different feature maps. To each feature map, a bias is added.

3.1.2.2 *Pooling*

Pooling describes the application of a kernel that gives a summary statistic. That could be the maximal value of the input for the max pooling operation or the average input value for the average pooling operation. So, the example of the last section was basically the application of average pooling. The difference to the convolution operation is that the weights of the kernel are determined by the network itself for the convolutions. Furthermore, pooling operation usually reduce the size of the input. A common approach is to set the step size to two, which leads to a reduction by factor two for 1-D representation and to a reduction of factor four for 2-D representations.

Pooling helps the network to make their representation of certain features invariant to translation of the input. That means, if a convolution is to detect a face, pooling helps to detect the face in different locations of the picture. Because pooling summarizes the information over a certain region, the information about the feature are kept. In that way, by stacking several convolution layers and pooling layers, the information in the input can be concentrated by continuously reducing the size of the feature maps. [8]

3.1.2.3 *Transposed Convolutions*

Increasing the step size when applying a kernel to some input leads to downsampling. This can be done either by using a convolution operation such that the network learns its own parameters when executing the downsapmling or with a pooling operation with a priorly defined procedure.

For some use cases, upsampling is required as well. The simplest way to do this is by replicating the occurring values. This operation is often simply called upsampling. If the network should decide by itself how the upsampling is done, transposed convolutions are used. [14] It is done by creating a new, longer vector by inserting zeros between the input values. Then each input value is multiplied with each parameter of the kernel and the results are copied to the new vector. For example, with a kernel size of three, each input values results in three new values. By writing the resulting values into the new vector, some results of different input values might overlap. It is possible to take the average or the sum of these values and write it to the vector.

3.1.3 *Recurrent Neural Networks*

3.1.3.1 *Original RNN*

Recurrent Neural Networks (RNNs) are used for sequence data, where the length of the data can be variable. Examples for sequence data are text data where the words or the characters represent the sequence, or audio data

where the samples of the amplitude represent the sequence. In essence, a simple RNN can be seen as a FFN with one input neuron, one hidden neuron and one output neuron that processes one sample at a time. The output of the first network is passed as an input to the next network that processes the next sample in the sequence. The information that is passed from one neuron to the next is also called the internal state of the network. Through this internal state, information from earlier inputs can be used to influence predictions at the current time step. The amount of time that information from a certain time step is kept in the hidden state is not fixed. As will be shown, it depends on the weights and on the input data.

In Figure 3.6 the schema of a RNN is pictured. One neuron at time step t takes the hidden state h_{t-1} of the previous neuron and an input x_t as an input, processes it, and then passes its hidden state h_t to the next neuron. At every time step t, an output y_t is returned. This does not have to be the case. For classification problems, the networks only return a single output at the end of the sequence.

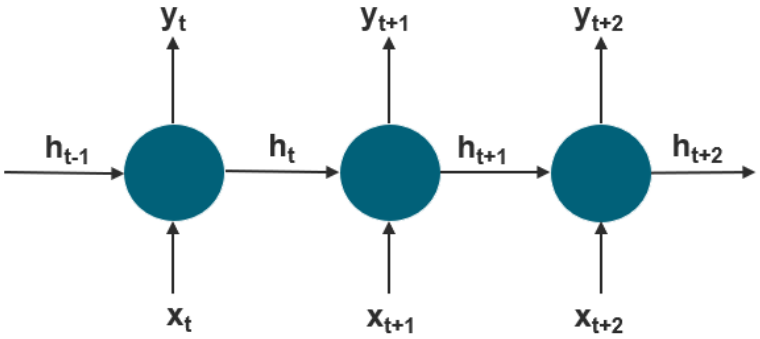

Figure 3.6: A simple visualization of the principles of a RNN

To calculate the hidden state h_t and the output y_t, equations 3.5 and 3.6 are used respectively. f_1 and f_2 are activation functions. W_{hh}, W_{hx} and W_{yh} are the weights matrices used by the network and b_h, b_y the bias vectors.

$$h_t = f_1(W_{hh} * h_{t-1} + W_{hx} * x_t + b_h) \tag{3.5}$$

$$y_t = f_2(W_{yh} * h_t + b_y) \tag{3.6}$$

These formulas indicate that the weight matrix for every time step t is reused. That means the weights are shared across time. To train the network, a loss can be computed at every time step by comparing the output value y_t with the expected value from the training data. The sum of these individual losses is the total loss of the network. To now adjust the weights, an algorithm called Backpropagation Through Time defined by Werbos [64] is used. To adjust the weights matrices, the errors for all past time steps are considered. Additionally, the error of the current time step flows backwards

through the network to account for the effect that a previous hidden state has on the current error. This same principle was explained in section 3.1.1.3 where the error of the FFN was propagated backwards to determine the influence of an early neuron on the error. For long sequences, this involves a repeated multiplication of the weight matrices. As pointed out by Hochreiter [27] [28], this results in vanishing or exploding gradients. If the weights lie in an interval (-1,1), the change in the loss function w.r.t to the weights of an early neuron gets infinitesimal small, resulting in a vanishing gradient. If the weights are big, a repeated multiplication results in extremely high gradients, making the training unstable due to large changes of the weights for every time step.

This vanishing or exploding gradient problem is not specific to RNNs. Deep FFNs and CNNs with many layers face this problem as well. Possible solutions will be discussed in 3.1.4. An extensive explanation of RNNs can be found in [1].

3.1.3.2 *Long-Short-Term-Memory Networks*

RNNs have an internal state that keeps information about past events. As was explained, the time span over which information is kept depends solely on the weights and the inputs. [6] This time span in often not larger than five to ten discrete time steps, making it useless for many tasks. A solution is offered by Long-Short-Term-Memory (LSTM) Networks developed by Sepp Hochreiter and Jürgen Schmidhuber in 1997. [29] LSTM networks enforce constant error flow back into earlier time steps through some special units called memory cells with gate units. The memory cell consists of several mathematical operations. A memory cell is basically a more complex RNN neuron. Each memory cell has a cell state that contains information from previous memory cells. The memory cell can now add or remove information from the cell state. This is done by the gates, which consist of neural network layers with specific activation functions that only let certain information from the input and the previous output through. Each memory cell then decides which information is returned as the output. This output is also given to the next memory cell. Through this complex architecture, LSTM network can save information over a longer period of time.

3.1.4 *Hyperparameters*

Hyperparameters describe the parameters that the model engineer can set prior to the training. They define how the training process is executed and in what way the model will adjust its trainable parameters during training. Besides selecting the right type of model for the considered use case, the hyperparameter selection is the most important aspect for a successful machine learning model with a stable training process.

3.1.4.1 *Layer and network size*

The easiest to understand hyperparameter is the size of the model. For FFNs, that relates to the number of hidden layers and the number of neurons per layer. The more layers and neurons in the FFN, the better the network is able to represent the relationship between the input and the output data. However, bigger networks are harder to train. The number of parameters that have to be adjusted during training grows rapidly with additional neurons and layers. That is because every neuron is connected with all neurons of the preceding and succeeding layer. Furthermore, with bigger networks the model engineer has to account for the vanishing gradient problem increasing the complexity of the network and the demand of respective knowledge. With increased complexity, the training time and the memory cost of the network increases as well. Another important concept to have in mind is the bias-variance trade-off. A complex model might explain the training data very well w.r.t to the already known output, but fails to generalize for new data, resulting in low performance on unseen data.

For CNNs, the size of a convolutional layer is determined by the input data, the kernel size and the number of kernels applied to the input data. For a 1-D convolutional layer, the input size is fixed by the number of samples of the input. The kernel size defines the number of samples that are considered when computing the feature map value. The number of kernels gives the network more possibilities to encode the information, presented in the input data, in different ways.

For RNNs, the network size is fully determined by the length of the input or output sequence.

3.1.4.2 *Activation functions*

Activation functions are basically just normal mathematical functions that take in some input x and output the corresponding values $f(x)$. They are usually used after every layer for all considered type of networks. In the hidden layers they help to stabilize the training and to enforce non-linearity, and in the output layer to make the output interpretable.

Consider a FFN that has to predict whether an email is spam or not. To make the output interpretable, it should output a probability of how certain it is that a given email is spam. Therefore, the output of the network has to be mapped to the interval $[0, 1]$, representing the probability. Other use cases need the values to be in a different interval. Hence, several activation functions have emerged that are used in the output layer of a network.

In the hidden layer, the network should be able to represent a non-linear relation between the input and the output. The first reason is that most use cases require a non-linear function to model its dependency. For the second reason, consider a network with only linear activation functions as the one earlier in this chapter in Figure 3.4. The output value can be calculated using the input value by multiplying the weights and adding the biases, as was shown by equation 3.1. This same relationship can be accomplished by a

network using only two neurons. To show this, the brackets in equation 3.1 are multiplied and the relationship between the weights and the biases of the two-neuron network and the four-neuron network is established as in Figure 3.7 shown.

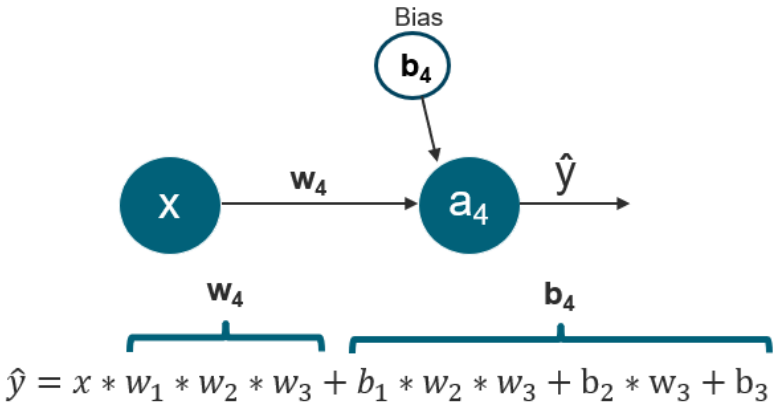

Figure 3.7: A flat 4-neuron FFN and a flat 2-neuron FNN are the same when meeting the depicted relation

Therefore, additional layers do not bring value to the network if a linear activation function is used.

After having seen why activation function are needed, the relevant ones for this thesis will be discussed in the following.

SIGMOID ACTIVATION The sigmoid activation function maps the input value to an interval between zero and one. One use case for this function was already mentioned. In the hidden layers, the sigmoid function is only used in special use cases like the attention mechanism. Since its gradient is close to zero for large positive and negative values, the vanishing gradient problem is intensified when the sigmoid function is used in the hidden layers.

$$sigmoid(x) = \frac{1}{1 + \exp(-x)} \tag{3.7}$$

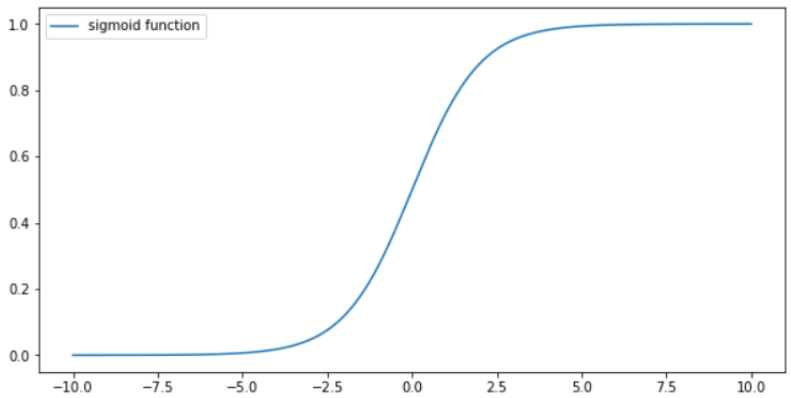

Figure 3.8: Sigmoid activation funciton

HYPERBOLIC TANGENT (TANH) The tanh activation function maps the input values to the interval $[-1, 1]$. It is used in normalized regression problems in which all values were normalized to this interval. Since sound waves are modeled with an amplitude in this interval, it can also be used in the output layer for a generative architecture. When used in the hidden layers, the vanishing gradient problem is the same as explained for the sigmoid function.

$$tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{3.8}$$

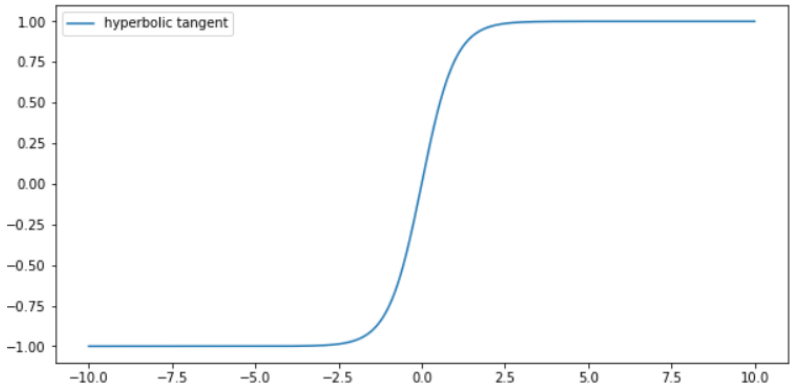

Figure 3.9: tanh activation funciton

SOFTMAX The softmax activation function is also used for output layers and represents a probability distribution like the sigmoid function. The sigmoid function is used for binary classification problems, since it outputs a value between zero and one. The softmax function outputs a probability distribution over several classes and is therefore used for multi classification problems. That means all K output values $softmax(x)_i$ for an input vector x add up to one. This is shown in equation 3.9.

$$softmax(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)} \tag{3.9}$$

RECTIFIER LINEAR UNIT (RELU) The ReLU activation function returns x if x is greater than 0 and 0 for every x less than 0. Mathematically, the ReLU function is defined by equation 3.10 plotted in Figure 3.10. Several experiments have shown that ReLU performs better than a sigmoid or a tanh activation function when used for the hidden layers. The first high-scale use in a deep learning architecture was for image classification. [33] The greatest reason for this is its computational simplicity. Since the gradient for positive values is one, the ReLU does not intensify the vanishing gradient problem. For negative values, however, the neuron outputs a zero and is therefore called "off". Since the gradient for these "off" neurons is zero, no learning is happening during the backpropagation step. If a neuron has a large negative

bias, the output of this neuron for most or all inputs is negative, hence the weights and biases of this neuron are never changed. This is known as the dead neuron problem.

$$ReLU(x) = max(0, x) \tag{3.10}$$

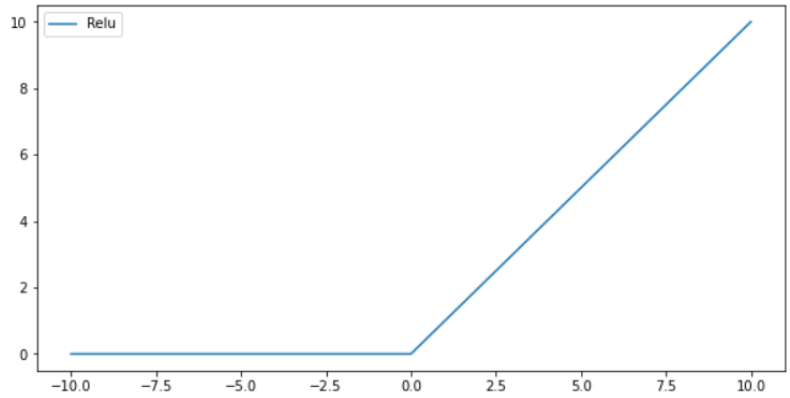

Figure 3.10: ReLU activation funciton

LEAKY RECTIFIER LINEAR UNIT As shown, for x values below zero, the ReLU function simply returns zero. That means the information about the magnitude of the negative value is lost. Every neuron that outputs a negative value is treated the same. Additionally, no learning is happening during backpropagation. Therefore, the Leaky ReLU returns the negative value scaled by a factor of 0.01 as shown in equation 3.11 and Figure 3.11.

$$LeakyReLU(x) = max(0, x) + min(0.01 * x, 0) \tag{3.11}$$

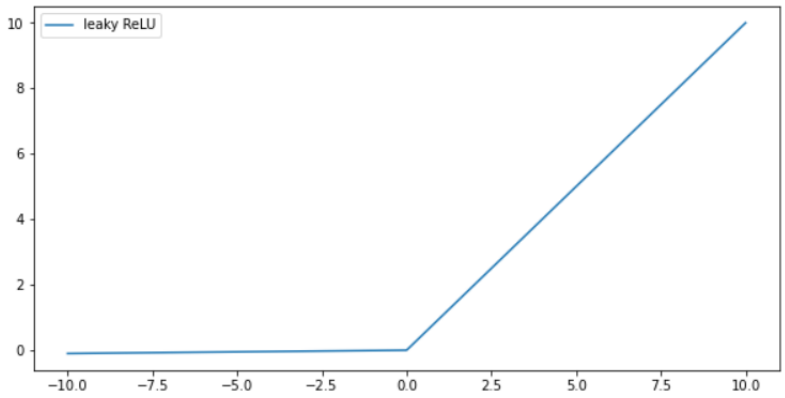

Figure 3.11: Leaky ReLU activation funciton

3.1.4.3 *Learning Rate and Optimizers*

As was seen in equation 3.4, the step size for the parameter adjustments during backpropagation is scaled by a learning rate. The learning rate, therefore,

determines if bigger or smaller steps are taken in the direction of the negative gradient of the loss function w.r.t. the adjusted parameter. The learning rate has a major influence on the success of the training. Ian Goodfellow, a leading researcher in the deep learning field and the founder of the Generative Adversarial Network (GAN) architecture, describes the learning rate as the "perhaps most important hyperparameter". [21] Figure 3.12 shows is relation to the training error. The training error gets large if the learning rate is too high or too low. A small learning rate leads to a super slow adjustment of the parameters, that might cause the network to not learn at all. A large learning rate changes the parameters in too big steps, not moving in the direction of the loss function's minimum. The optimal learning rate can only be reached by monitoring the loss of the network and tuning the learning rate accordingly.

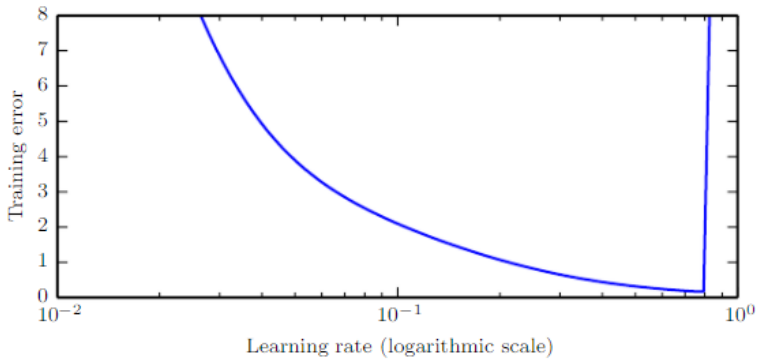

Figure 3.12: Relationship between the learning rate and the training error for an arbitrary example [21]

In addition to only the learning rate, the gradient descent algorithm can further be optimized by extending the learning rule by additional terms. The overall goal of the learning process is to find the global minimum of the loss function w.r.t to the networks' parameters. Following the gradient descent approach, the algorithm can get stuck in a local minimum or a saddle point, failing to find an optimal solution. In practice, the loss function is a multidimensional plane with many local minima and saddle points. In 1986, it was shown that adding a momentum term to the learning rule can increase the training speed and quality [49] [59]. It is an analogy to physics, where a ball gains momentum rolling down a hill and is able to surpass small hallows or flat surfaces on the way to the lowest point. In the update rule, this momentum term is a fraction of the update vector of the previous updating step that is added to the equation. The update rule 3.4 defined earlier in this chapter can therefore be extended to the equation 3.12. λ is the momentum parameter, determining how strongly the gradient of the preceding updating step should be considered at the current updating step.

$$w_{i+1} = w_i - \alpha\left(\frac{\Delta Loss(weights, biases, y, X)}{\Delta w_i} + \lambda \frac{\Delta Loss(weights, biases, y, X)}{\Delta w_{i-1}}\right)$$

(3.12)

Adding momentum was the first approach taken in trying to optimize the learning process of ANNs trained by gradient descent. Since then, many algorithms have emerged tackling this problem. The two most known and commonly used algorithms today are called RMSprop and Adam.

So far, the learning rate was a single rigid hyperparameter that was set at the beginning of the training and stayed the same during the training process. RMSprop Adam use an adaptive learning rate for every parameter. That means, the learning rate is adjusted over time and every parameter has its own learning rate. This follows the fundamental idea that parameters that are already close to the optimum need less adjustment than other parameters. The adjustment of the learning rate over time also shows to have better convergence towards the optimum. The idea of using different learning rates for every parameter and every time step was first proposed by John Duchi and his colleagues with an algorithm called Adagrad. [13] Jeffrey Dean and his colleagues have then shown that this idea of an adaptive learning rate works well for deep networks. [10] RMSprop is an extension of the Adagrad and was first introduced in a lecture from Geoffrey Hinton. [39]

RMSprop uses a moving average of the squared past gradients for each weight to scale the learning rate at every time step for every parameter.

Adaptive Moment Estimation (Adam) builds further on the idea of RMSprop and not only stores a moving average of squared past gradients but of the past gradients as well. The moving average of the squared past gradients is still used to scale the learning rate. The moving average of the past gradients is used as the step size for the parameter adjustment. Adam was introduced in 2015 by Diederik P. Kingma and Jimmy Lei Ba. [32]

3.1.4.4 *Loss functions*

The loss function is usually not considered a Hyperparameter. The loss function depends on the considered use-case and should therefore be selected prior to the tuning of the network. Sometimes there is more than one loss function possible for the desired outcome. Therefore, it is a valid view to consider the loss function a hyperparameter that has to be tuned and can be changed during the development of the network. The only important property of a loss function is that its reduction should lead to the desired behavior of the network. Since loss functions are very specific to the regarded use case, the relevant ones will be discussed in later chapters.

3.1.4.5 *Batch size*

The batch size basically determines after how many training examples the network update its weights. It plays a vital role for memory allocation because it defines for how many samples the computer has to calculate and save the gradients before executing a gradient descent step.

The vanilla gradient descent updates the weights after having seen all training examples. With the "knowledge" how every training example influences the loss, the algorithm can take a carefully thought step in the direction of the negative gradient. However, this can be very slow for deep learning architectures and intractable for data sets that do not fit into random-access memory (RAM).

The stochastic gradient descent updates the weights after every single training example. This is much faster than the vanilla approach but might lead the weights to take steps into random directions due to the variance of the training set.

Therefore, the algorithm called mini-batch gradient descent tries to combine both advantages and executes the optimization step after seeing a certain number of samples called a batch. In that way, the variance is lower without having to process every single training example before taking a step.

However, it was shown that stochastic gradient descent can be faster and perform better for datasets with low variance than mini-batch gradient descent. Furthermore, the random steps that are sometimes taken by the algorithm can lead to a better solution by forcing the algorithm to explore a wider range of possible values for the parameters. [34]

3.1.4.6 *Normalization*

Normalization refers to scaling a variable to a well-defined scale. The two most common techniques are the min-max normalization and the z-score standardization. For the min-max normalization, the maximum value is scaled to a one and the minimum value is scaled to a zero and all other values respectively. With the z-score standardization, the values are scaled with the z-score to a mean of zero and a standard deviation of one.

For FFN the input is usually scaled with the min-max normalization to speed up the training process. For data that follows a normal distribution, z-score standardization is used. As was already seen, the learning process on ANN involves a multiplication of the input data with different weights and the calculation of a loss value. Having every input data scaled to the same or similar values helps the network to faster identify patterns because it does not have to account for larger and smaller input values and every input value influences the loss in the same magnitude. [7] [34]

Since normalizing the input values worked, new approaches also tried to normalize the outputs of the activation function of every layer. That lead to two major benefits. The training speed further increased and the initialization of the weights was less important.

The gradients for the parameters of a layer are calculated under the assumption that the parameter of the other layers do not change. In practice, all parameters are updated simultaneously. Hence, the minimum that the parameters tried to reach is now in a different spot. Another way to see it is that the distribution of the output values of every layer is constantly changing. The authors of the paper that introduced Batch Normalization (BatchNorm) referred to this as "internal covariant shift". [30] BatchNorm tries to

coordinate the updates of the parameters across layers by normalizing the outputs of the activation functions with the z-score normalization per mini-batch, resulting in similar distributions after every parameter update. Since the mean and the variance depend highly on the samples in a mini-batch, the z-score scaled value is multiplied by a learnable parameter γ and shifted by a learnable parameter β. A later paper suggest that the benefit of faster convergence comes from smoothing the landscape of the loss function. [52] With this smoothed hyperplane, greater steps can be taken towards the optimum. This smoothed hyperplane also reduced the need for proper weight initialization.

After BatchNorm further normalization techniques were developed like Weight Normalization [50] or Layer Normalization [5] that try to further optimize the learning process for specific problems.

3.1.4.7 *Regularization*

The bias-variance trade-off was already mentioned in 3.1.4.1. Regularization techniques aim to improve exactly this concept. They limit the capacity of the model by enforcing some constraint. Thus, the model's ability to learn is weakened. This usually reduces the performance on the training data but might increase the performance on unseen data, improving the ability to generalize the learned relation between input and output. To achieve this, several techniques can be used.

Parameter penalty is a technique not exclusive for deep learning algorithms. It adds another term to the loss function. This term penalizes, for example, large weights, reducing the model's free parameter choice. α is a hyperparameter that controls the impact of the penalty term Ω.

$$Loss(weights, biases, y, X) = Loss(weights, biases, y, X) + \alpha\Omega(weights) \quad (3.13)$$

Other common regularization techniques include Dropout, Data Augmentation and Early Stopping.

3.2 GENERATIVE ARCHITECTURES

3.2.1 *Autoencoders*

3.2.1.1 *Original Autoencoder*

An autoencoder is an ANN architecture that aim it is to copy its inputs to its outputs. The number of neurons in the input and in the output layer are therefore the same. The idea now is to introduce a constraint in the form of a bottleneck layer, which is a hidden layer with fewer neurons than the input and output layer. In that way, the network cannot use all information given by the input data when trying to make a copy. Its aim now is to learn which information is important and which information can be discarded. The gen-

eral structure of an autoencoder is depicted in Figure 3.13. It consists of two ANNs, the encoder and the decoder. These network can be all discussed types of ANN and are usually a mirrored version of each other. They are connected by the bottleneck layer.

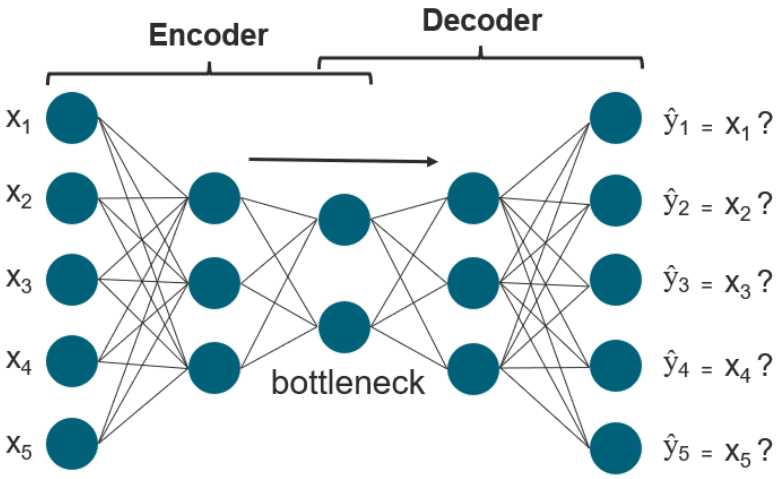

Figure 3.13: Basic Structure of an Autoencoder with FFNs

The task of the encoder is to compress the input data to the sparse representation in the bottleneck layer. This sparse representation is called the latent representation in the latent space. Next, the decoder tries to reconstruct the original data from the latent representation. The encoder and the decoder are trained together by comparing the output of the decoder to the original data and backpropagating the error backwards through the network.

By learning the latent representation of the data, an autoencoder can be used as a dimensionality reduction technique such as the principal component analysis. To be used as generative models, the latent space of the latent representation has to be modeled. Subsequently, new points in the latent space can be given to the decoder to generate a new data sample.

Autoencoders in this simple form have been used by different authors starting in the 90s. [54]

3.2.1.2 *Variational Autoencoders*

A more recent development is the Variational Autoencoders (VAE) used for generative tasks. [62] A later paper from the same authors with more in-depth information can be found with [63].

The problem of the normal autoencoder is that there is no guarantee that the data in the bottleneck layer is organized in a way that makes adequate sampling possible. The features of the latent representation could lay randomly in a multidimensional space, and sampling of these features could lead to arbitrary and unwanted results.

VAEs try to give the latent space suitable properties such that it can be more easily modeled. It does this by regularizing the encoded distribution of the latent representation. The general structure of the architecture stays

the same. Only the bottleneck layer changes. Instead of encoding the input as mere points, they are encoded as a multivariate normal distribution. Then points from this distribution are sampled and given to the decoder for decompression. The VAE is trained to learn the mean vector and the covariance matrix of the distribution, which is now the latent space. The regularization happens because, in practice, this distribution is kept close to a standard normal distribution. To enforce this constraint, a regularization term is added to the loss. Concretely, the Kulback-Leibler divergence between the computed distribution and the standard normal distribution is used. This basically leads to a distribution where points that lay close together in the latent space are decompressed to similar results, and all points sampled from this distribution lead to realistic results. For the in-depth information and the underlying mathematics, please refer to the cited papers.

3.2.2 *Generative Adversarial Networks*

3.2.2.1 *Original Generative Adversarial Network*

Generative adversarial networks (GAN) follow the idea of a two player game. Both players try to improve to be able to beat the other player. A GAN therefore consists of two parts, a generator G and a discriminator D. G tries to generate new samples that fool D and D tries to distinguish fake samples from real ones.

GANs were formally introduced by Ian Goodfellow and his team in 2014 [22], while Schmidhuber claims the credit for this idea with his earlier work. [53] However, the name GAN was coined by Ian Goodfellow and is credited by most of the community by ongoing citations. In this first version, both the G and the D consist of FFN, but all types of networks can generally be used.

The general structure of a GAN is shown in Figure 3.14.

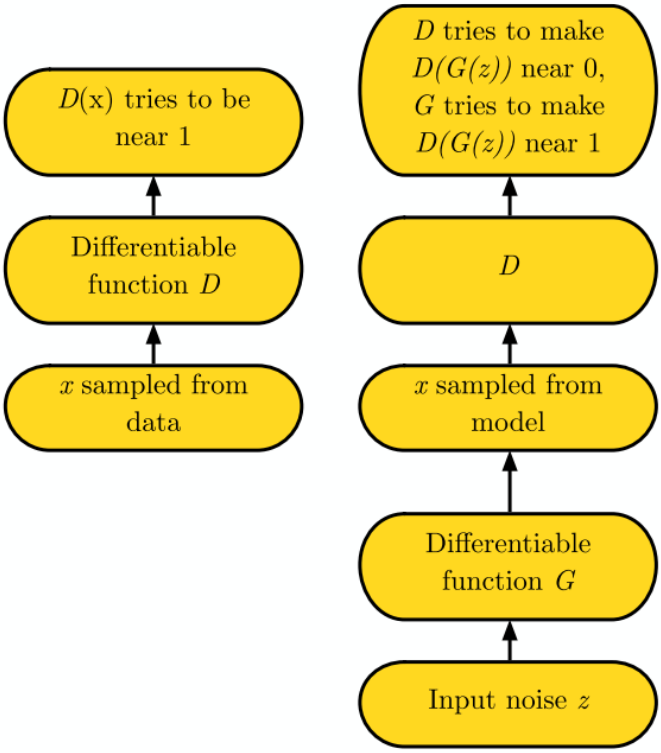

Figure 3.14: Basic idea of a GAN with the output $D(x) = 1$ for real samples and $D(x) = 0$ for fake samples [20]

On the left side, D is presented with a sample from the real training data. It aims to correctly identify it as real. The activation function of the output layer of D is the sigmoid function. Therefore, output values close to one indicate real samples and output values close to zero indicate fake samples. On the right side, D is presented with a fake sample generated by G. To generate that sample, G is fed some random input noise, for example, a vector of samples drawn from a uniform distribution. G processes this vector and sends its output to D. G tries to fool D by achieving that D outputs a one for its fake sample. D tries to correctly identify this fake sample as fake by outputting a zero. This is called a minimax game, where G tries to maximize the error of D and D tries to minimize it.

D can be trained directly by comparing its output to the expected value. The error can then be backpropagated. To train G, the output of D is needed. For every generated sample, it can be examined whether G successfully fooled D. Concretely, the loss functions in equation 3.14 and 3.15 are used.

$$Loss_D = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}} log(D(x)) - \frac{1}{2}\mathbb{E}_{z\sim p_z} log(1 - D(G(z))) \tag{3.14}$$

$$Loss_G = -\frac{1}{2}\mathbb{E}_{z\sim p_z} log(D(G(z))) \tag{3.15}$$

p_{data} refers to the probability distribution of the training data and p_z to the probability distribution of the noise variable z. G, therefore, implicitly defines a new probability distribution p_g that is obtained by the samples $G(z)$ with $z \sim p_z$. The aim of G can hence be formulated as minimizing the divergence of p_{data} and p_g. In this specific case, it can be shown that for an optimal D this minimax game resembles the minimization of the Jensen-Shannon (JS) divergence. For the mathematical proof, please refer to the linked papers of Ian Goodfellow.

Goodfellow suggests an iterative training of both models where D is trained k times for every training iteration of G. His reason is that a G that is too fast too good, can collapse too many input values to the same output values that worked well when minimizing the loss.

3.2.2.2 *Wasserstein Generative Adversarial Network*

As was mentioned in the previous subsection, the original GAN tries to minimize the JS divergence under the condition of an optimal D. If two distributions differ too much, the gradient of the JS divergence will diminish because the values approach some upper boundary. With no gradient, the generator is unable to learn and to minimize the divergence. This means if p_{data} and the initial p_g are too different, and it is too easy for D to distinguish between fake and real, no improvement is happening. For this reason, D and G have to be kept close to some equilibrium and applying this algorithm to complex data with a complex probability distribution is hard or even impossible. This was shown by Martin Arjovsky and his colleagues. [2]

To tackle this problem, he and his colleagues have suggested in a later paper from the same year the usage of a different divergence measurement called the Wasserstein or Earth-Mover distance. [3] The resulting GAN is called a Wasserstein Generative Adversarial Network (WGAN). The advantage is that the Wasserstein distance has a smoother gradient everywhere, leading to a more stable training. The generator learns, regardless of how well it or the discriminator performs. Arjovsky illustrates this with Figure 3.15. Even though p_{data} (here: "Density of real") and p_g (here: "Density of fake") have no overlap, this new approach results in a gradient on all parts of the space. The discriminator in a WGAN is sometimes called a critic because it does not output a probability through a sigmoid function, rather a critic score through a linear activation function.

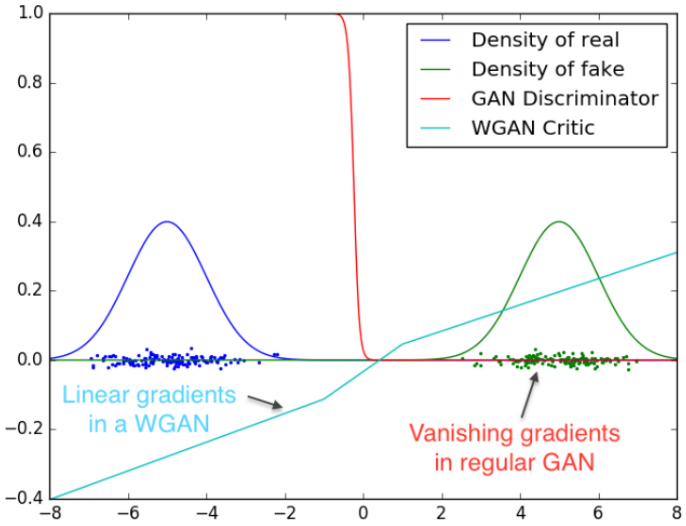

Figure 3.15: Optimal discriminator and critic when learning to differentiate two Gaussians [3]

The critic score is a feedback score for the generator that reflects how fake or real the critic thought the sample was.

The only problem with the wasserstein distance is that it is intractable. To make an approximation possible, a certain constraint has to be enforced. Concretely, the discriminator has to be a 1-Lipschtiz function. Arjovsky and his colleagues showed that it is possible to meet this constraint by using a linear activation function in the output layer and by clipping all weights such that they lay inside a certain interval $(-c, c)$ where c is a hyperparameter.

3.2.2.3 *Wasserstein Generative Adversarial Network with Gradient Penalty*

As already pointed out in their first paper, weight clipping is a "[...]clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used." [3] The hyperparameter c therefore has to be tuned carefully and acts as a regularization method reducing the capacity of the model.

A different approach to enforce the Lipschitz constraint is the gradient penalty, resulting in a network called Wasserstein Generative Adversarial Network with gradient penalty (WGAN-GP). It was developed also by Martin Arjovsky with a different team in 2017 only several months after the WGAN paper. [23]

The idea is to penalize the norm of the gradient of the critic w.r.t to its input. A function is 1-Lipschtiz if it has gradients with norm at most one everywhere. To achieve this, they add a penalty term to the critic's loss function that penalizes gradients that deviate from one. This penalty term is shown in equation 3.16. $\tilde{x}$ is a sample drawn from the distribution $P_{\tilde{x}}$. $P_{\tilde{x}}$ is

a data distribution defined by sampling uniformly along straight lines between pairs of points sampled from the data distribution P_{data} and P_g. λ is a hyperparameter.

$$gp = -\lambda * \mathbb{E}_{\tilde{x} \sim P_x}[(||\Delta_{\tilde{x}} D(\tilde{x})||_2 - 1)^2] \tag{3.16}$$

4 CURRENT ARCHITECTURES FOR ADUIO GENERATION

In the last couple of years, thanks to the ongoing developments of respective algorithms, several authors addressed the topic of music generation with deep learning architectures. Different approaches with different architectures have been taken. This chapter aims to give an overview about the work that has already been done by other authors and to point out their advantages and disadvantages that help to develop the concept in chapter 6.

4.1 WAVENET

The first paper that approach the task of modelling raw audio was the WaveNet paper from Aaron van der Oord and his colleagues in 2016. [42][1] Inspired by other generative networks for images and text, their goal was to generate raw audio waveform for human speech and musical fragments that are only a few seconds long.

Their generative process is a product of conditional probabilities of a waveform $x = x_1, ..., x_T$ as follows:

$$p(x) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1}) \tag{4.1}$$

Each sample x_t is therefore conditioned on all previous samples. This is called an autoregressive model. To model this, they use a CNN without any up or down sampling layers. So, the input and the output size of the network is the same. To make one sample depend on only previous ones, they use several so called dilated casual convolutions. As was explained in 3.1.2, normal convolutions take values to the left and to the right into consideration when calculating the feature map value. Casual convolutions have filters where only values to the left side are considered. For 2-d convolutions, this concept is called a masked convolution. To now distend the receptive field of one sample, they use stacked dilated casual convolution. as shown in Figure 4.1.

1 Samples: https://deepmind.com/blog/article/wavenet-generative-model-raw-audio

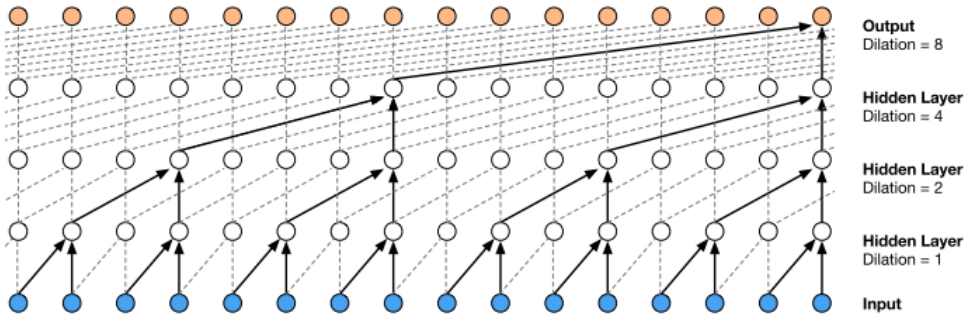

Figure 4.1: Idea of dilated casual convolution [42]

This dilated convolution skips samples along the sequence when calculating the feature map value. Hence, they do not use all previous sample to predict the next one, but rather define the receptive field based on the number of stacked convolutions and the dilation factor. The reason for this is the computational efficiency.

The raw audio that they use is encoded with 16 bit, resulting in 65,536 possible values for the amplitude for each time step. The output layer of the network uses the softmax function, predicting which amplitude value is the most probable. To make this more computational efficient, they further reduce the number of possible values by quantizing it to 256 values. The network is optimized by maximizing the log-likelihood of the data w.r.t. the parameters.

It is also possible to condition the network to some discrete feature h by extending the model as shown in equation 4.2.

$$p(x|h) = \prod_{t=1}^{T} p(x_t|x_1, ..., x_{t-1}, h) \tag{4.2}$$

The author states that training a CNN with casual convolutions is much faster than training a RNN because all output values can be predicted in parallel during training because the input data x is known for all time steps t. A RNN has to predict each sample subsequently, resulting in longer training time. Creating new content is rather slow for the CNN approach as well because to generate a new sequence, every generated sample has to be fed back into the model to produce the next sample in the sequence. Their model lacked long-term dependency, resulting in a change of style, volume, and quality every second.

4.2 MIDINET

As the name suggests, MidiNet is an architecture that uses the MIDI format to generate new sound. The architecture uses the GAN algorithm and was developed by Li-Chia Yang and his colleagues in 2017. [65][2] They use a 2-D MIDI representation of pop music songs. The MIDI files are divided

2 Samples: https://soundcloud. com/vgtsv6jf5fwq/sets

into intervals with a fixed time length called bars. Each time step can have one or more notes assigned to it. In that way, the model does not generate melodies as a continuous sequence but rather one bar after another, in a successive manner. The generator uses random noise to generate a MIDI file, and the discriminator has to predict whether a MIDI file is real or fake. Hence, they use the GAN architecture as explained in Section 3.2.2.1 with a sigmoid activation function in the discriminator. To achieve a larger receptive field, they train another CNN called the conditioner CNN to incorporate information from previous bars into the generator CNN. To account for the problem of mode collapse, they add a regularization term to the loss function but do not give mathematical explanation to why they take this approach. They can also condition their network by using the chords given by the MIDI files. Chords are basically a set of notes that are used for a certain section of a song. In the generation process, the network can be told which notes from which chord should be used for the bars. Besides note and chord information, the remaining data that MIDI files contain was not used.

4.3 SAMPLERNN

The author Soroush Mehri and his colleagues combined a FNN and several RNN to generate raw audio data. They called the resulting architecture developed in 2017 SampleRNN. [36][3] They state that other approaches that use high-level representation of audio (such as spectrograms or MIDI) often result in degraded quality and the necessity of corrective measurements that require extensive domain knowledge. To model long-range dependencies, they use a hierarchy of three RNNs that each processes audio on different clock-rates. The lowest model operates at each individual sample, and the higher models on an increasing longer timescale and a lower resolution. Each model conditions the model below it, and the lowest model outputs the samples. This general structure is shown by Figure 4.2. Each sample of the higher models is upsampled.

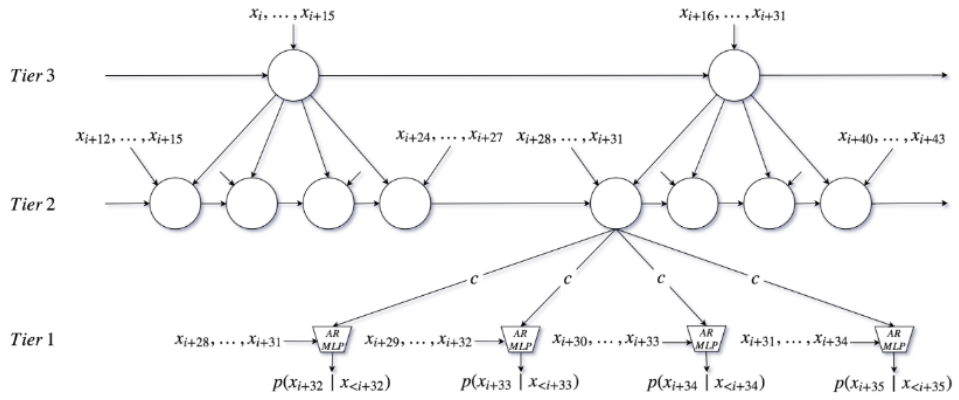

Figure 4.2: General idea of the SapleRNN with a hierarchy of three RNNs and an upsampling factor of four [36]

3 Samples: https://soundcloud.com/samplernn/sets

In the lowest level, a FFN with a softmax activation function is used to predict the sample value. They quantized the possible output values of a 16 bit encoding to 256 as in the WaveNet architecture. To predict a sample, past samples are needed and therefore in the generating processe, the model has to be run repeatedly to feed each sample back into the model to predict the next one. For their RNNs they use LSTM and Gated Recurrent Unit (GRU) units but find that GRU units work slightly better. GRU is another variant of the memory cells used in LSTM.

4.4 MELNET

MelNet is an architecture based on the autoregression idea. It uses audio in the spectrogram representation to generate new content. It was developed by Sean Vasquez and Mike Lewis in 2019. [61][4] They state that "[. . .] long-range dependencies are difficult to model directly in the time domain [and they] show that they can be more tractably modelled in two-dimensional time-frequency representation such as spectrograms". WaveNet and SampleRNN only take into consideration a fraction of a second when generating new samples, and therefore lack high-level structure. The temporal axis of a spectrogram is much more compact, allowing for longer dependencies. But the authors also recognize that through loss of information, local characteristics of the audio cannot be modelled with high-fidelity. To limit the information loss when converting raw audio into a spectrogram and during the inverse operation, they try to create the spectrograms with as much detail as possible. They achieve this by decreasing the hop length and increasing the number of frequency bins.

As other autoregressive models for image data generate an image by predicting the image pixel for pixel, their network generates a new spectrogram by iterating over the time and the frequency axis of the spectrogram. To model this, they use a complex RNN architecture with LSTM cells. They state that an RNN architecture has shown to be superior in the past to CNN architecture for modelling spectrograms. They use a stacked architecture with a time-delayed stack extracting time-depending information and a frequency-delayed stack extracting frequency-depending information. The output of layer l of the time-delayed network is fed as an input into layer l of the frequency-delayed stack.

The time-delayed stack uses three one-dimensional RNNs. One runs along the time axis, one runs along the frequency axis, and one runs backwards along the frequency axis. The output at each layer is the concatenation of the three RNN hidden states. The frequency-delayed stack uses only one one-dimensional RNN that runs along the frequency axis. They used piano music and generated samples of 10 seconds.

4 Samples: https://audio-samples.github.io

4.5 WAVEGAN

A first approach of applying a GAN architecture to raw audio was taken by Chris Donahue and his colleagues with the development of the WaveGAN in 2019. [12][5] Their network is capable of generating coherent sounds of one second, suitable for sound effect generation. The biggest advantage over the already existing WaveNet that they mentioned was the speed advantage in the generating process. GANs can generate all samples simultaneously when generating a new sound. Besides WaveGAN, they also developed a network called SpecGAN that generates sound with the image representation of the sound in form of spectrograms. The authors emphasize that "[...] spectrograms are non-invertible and cannot be listened to without lossy estimation or learned inversion models". For the SpecGAN they therefore design a spectrogram that allows for approximate inversion. For both networks, they use the Deep Convolutional Generative Adversarial Network (DCGAN) architecture by Alec Radford and his colleagues from 2016. [47] Since the DCGAN architecture was aimed at image processing tasks and uses 2-D convolutions, they flatten it to operate on one dimension for the WaveGAN.

The DCGAN architecture tries to define setting that allow stable training for deeper GANs with convolutional layers. The first adjustment that they made was the use of the all convolutional net. [58] They do not use pooling layers for the upsampling and downsampling tasks, but rather convolutions with an increased step size such that the network can learn the best parameters for these tasks itself. Next, they avoid stacking fully connected layers (layers from a FFN) on top of or before the CNN. They reshape the input layer of the generator to the shape needed for the first convolutional layer and flatten the last convolutional layer of the discriminator and feed it directly into an activation function. This proofed to stabilize the training and yielded better results for deeper networks. For the third setting, they used of BatchNorm to allow gradients to flow into deeper layers and to avoid problems of poor parameter initialization. They used BatchNorm for all layers besides the input layer of the discriminator and the generator output layer. In experiments, this also proofed to be the best setting. In the generator, they used the ReLU activation function for all hidden layers. For the discriminator, the Leaky ReLU activation function worked well.

The WaveGAN and SpecGAN architecture use both the WGAN-GP strategy for the learning process. Samples produced by the WaveGAN were better than samples from the SpecGAN based on subjective measurement of human listeners. For this reason, the architecture of WaveGAN will be explained in the following.

The generator of the WaveGAN is shown in Figure 4.3. The first column describes the type of layer that is used with the stride length for the transpose convolutions. In the second column the first number is the kernel size as it was explained in this thesis. The second number indicates the number of input feature maps, and the third number the number of output feature

5 Samples: https://chrisdonahue.com/wavegan_examples/

maps. d is a hyperparameter that adjusts the number of filters used in the network. c is set to one for this network and corresponds to a concept explained in chapter 7 in this thesis. The last column is the output shape of each layer, where n is the batch size, the middle number is the output length and the last number is the number of feature maps.

Operation	Kernel Size	Output Shape
Input $\boldsymbol{z} \sim \text{Uniform}(-1, 1)$		$(n, 100)$
Dense 1	$(100, 256d)$	$(n, 256d)$
Reshape		$(n, 16, 16d)$
ReLU		$(n, 16, 16d)$
Trans Conv1D (Stride=4)	$(25, 16d, 8d)$	$(n, 64, 8d)$
ReLU		$(n, 64, 8d)$
Trans Conv1D (Stride=4)	$(25, 8d, 4d)$	$(n, 256, 4d)$
ReLU		$(n, 256, 4d)$
Trans Conv1D (Stride=4)	$(25, 4d, 2d)$	$(n, 1024, 2d)$
ReLU		$(n, 1024, 2d)$
Trans Conv1D (Stride=4)	$(25, 2d, d)$	$(n, 4096, d)$
ReLU		$(n, 4096, d)$
Trans Conv1D (Stride=4)	$(25, d, c)$	$(n, 16384, c)$
Tanh		$(n, 16384, c)$

Figure 4.3: Structure of the generator of the WaveGAN [12]

They start with a vector of length 100 with samples drawn from a uniform distribution between -1 and 1. This vector is then reshaped and fed into a transposed convolutional layer. They use a filter size of 25 to have a large enough receptive field to model longer-range dependencies. They then upsample by a factor of four until they have a vector with $16,384$ samples. Since they use a sample rate of 16 kHz, this results in an audio file of slightly longer than a second. The discriminator is a mirrored copy of the generator, where they use convolutions with a step size of four to downsample the audio file to a single number. The kernel size of the convolutions is as well 25. They do not use BatchNorm in the generator nor in the discriminator. They do not give a reason why.

4.6 MUSICVAE

MusicVAE is a recurrent VAE with a hierarchical decoder. It was developed by Adam Roberts and his colleagues for the Google Magenta project. [48][6] They use Midi files to model sequences of musical notes. Their reason to use a VAE is that it results directly in a latent space with which targeted changes to the output can be made. The hierarchical decoder RNN helps the model to learn long-term structure in the music. The RNNs that are used in the VAE produces the output sequence autoregressively. The overall architecture is shown in Figure 4.4. For the encoder, they use a two-layer LSTM network. The last hidden state of the second layer LSTM is then fed into two fully connected layers to produce the latent distribution parameters of the bottleneck layer z. For the decoder, the latent vector is passed through a

6 Samples: https://goo.gl/magenta/musicvae-examples

fully connected layer and then fed as an input to a two-layer LSTM network called the conductor. The original input sequence is segmented into U subsequences. The conductor network is built in such a way that it outputs U numbers of outputs called the embedding vector c. Each value of c is then individually passed through a shared fully connected layer to produce the initial hidden states for the decoder RNN. The decoder RNN also consists of a two-layer LSTM network and produces a distribution over Midi events with a softmax activation function. They produced 38 seconds long drum and piano pieces.

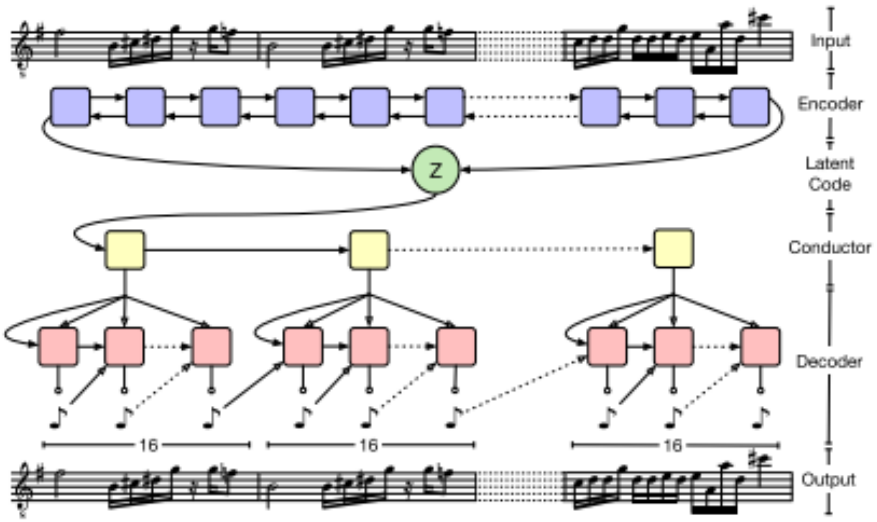

Figure 4.4: Structure of the MusicVAE [48]

4.7 JUKEBOX

Jukebox is arguably by far the best music generating architecture currently developed. It is a product by the OpenAI team and was published in 2020. [11][7] It can generate coherent songs up to multiple minutes with sung lyrics in different genres. It uses the raw audio format with a $44,100\ Hz$ sample rate and a 32 bit encoding. To make this complex task feasible, they compress the audio data with an autoencoder to a lower space. Concretely, they use a Vector Quantized Variational Autoencoder (VQ-VAE). The difference to the normal VAE is the discretized bottleneck layer. They use a codebook with discrete values and map every output value of the bottleneck layer to its nearest value in the codebook. They use three different VQ-VAE. Every VQ-VAE uses non-casual, dilated 1-D convolutions with different hop-length (dilation factor) for the dilation. To compress the data, they use downsampling convolutions in the encoder and upsampling convolutions in the decoder. They conditioned on artist and genre. To generate new audio, they use an autoregressive transformer network. It is a progression of the RNN architecture with the possibility to allow for longer dependencies, which was outside

7 Samples: https://jukebox.openai.com/

the scope of this thesis. This transformer network uses the three latent representations generated by the three VQ-VAE to upsample this compressed representation back to the original space.

5

ANALYSIS AND REQUIREMENTS

5.1 ANALYSIS OF AUDIO REPRESENTATION

The first question to ask when building a generative system for music is which representation of audio to use. In section 2.4 different types were explained. In chapter 4 different approaches used spectrograms, Midi files or raw audio representation to tackle this generative task and all approaches were able to produce audio in some form. All representations come with advantages and disadvantages that will be described and discussed in the following.

MIDI Midi files are a high-level representation of the audio. They do not contain the audio itself and therefore are smaller in memory size compared to MP3 or WAV format. This allows the algorithm to process the Midi files faster, resulting in reduced training time or more data different files can be used. The reduced complexity also makes the task of learning the structure of the data easier. This reduces the requirements of the used architecture and increases the probability of the desired result. With Midi files, it is also possible to condition the model on certain chords, velocity or other aspects of the music. Since Midi files are used by musicians and encode their way of thinking in the process of making music, knowledge in music theory is needed to fully understand the information in Midi files and how it plays together to define the music. However, Midi files cannot capture all the diversity of music. In the process of expressing the actual music with a Midi file, information is lost. They are also limited to predefined events with certain frequencies of certain instruments that they can concatenate. Therefore, it is impossible for an algorithm to produce new sounds. Midi files have to be generated when producing a song. It is not possible to convert an existing MP3 file into a Midi representation. Hence, the data that can be used for generative tasks is limited to already existing data sets.

SPECTROGRAMS Like Midi files, spectrograms compress the actual music to some high-level representation, resulting in information loss. Spectrograms are also invertible. This means after the algorithm has generated a new file in the form of a spectrogram, the conversion into an audible representation is again a lossy operation. This limits the achievable quality of the audio. Since spectrograms compress the time into certain bins, it is easier to model long-term dependencies with a spectrogram than with raw audio. An advantage over Midi files is that spectrograms can be generated out of arbitrary MP3 or WAV audio files. Therefore, an individual data set can be generated that fits the needs of the developer. Moreover, more research has

been dedicated to the generation of images so far. Already well-working architectures can hence be used for the task of audio generation with spectrograms as well.

RAW AUDIO Raw audio data represent the waveform of the audio directly. The discretized wave can be saved in a vector. Most architectures that modelled raw audio data have so far used a 16 kHz sampling rate. That means $16,000$ values are saved for each second of audio. For a song of one minute, the model has to predict $960,000$ values. This makes the task of modeling raw audio data challenging and computationally expensive. To model long-range dependency in the raw audio format, very deep architectures with wide receptive fields are needed. These deep architectures are hard to train and often have to perform many calculations that are computationally expensive and therefore need a long time to execute. Humans can perceive frequencies up to 20 kHz. As was explained in section 2.2, a sample rate below 40 kHz results in some information loss during the digitization because frequencies above half of the sampling rate are not represented in the digitized audio signal. For most music, a sample rate of 16 kHz is sufficient to capture all relevant frequencies. Modelling in the raw audio representation, allows for higher fidelity. The sounds are processed by the model as they are, and the model is not limited to some already pre-defined entities. The discretized waveform can also be extracted from every raw audio file format, such as MP3 or WAV. The developer can create their own dataset to train the model.

5.2 ANALYSIS OF ARCHITECTURES

When building a generative deep learning model, two different questions have to be answered. Which type of network should deal with the data processing, and how should the overall architecture of the model be designed to foster the generative process. For audio data, the processing, and prediction can be done by RNNs as well as CNNs. The two most common architectures for deep generative modelling are autoencoders and generative adversarial networks, but also stacked RNN architectures are possible.

There is no clear scientific belief which type of network is better suited for processing sequential data. Aaron van den Oord, says that RNNs are harder to train and parallelize than CNNs. He showed this by comparing his PixelCNN network with a PixelRNN network for the same task. [43] But as was shown in chapter 4 other authors preferred RNNs. However, because of their autoregressive property, RNNs take longer than CNNs to generate new content. For an autoencoder and a GAN architecture, the network has to predict new data, which increases the training time with RNNs.

There is also no clear scientific belief whether autoencoders or GANs are the better deep generative architecture. It is still a matter of active research. However, more authors seem to prefer GANs over autoencoder. An advan-

tage of GANs is that the generator never sees the training data. The generator is trained solely with gradients flowing backwards from the discriminator. In that way, components from the training data are not copied directly. Content from VAEs often resemble the input data. A disadvantage of GANs is that there is no explicit latent representation of data that it generates. The generator is fed a random vector that it maps to the representation of the generated content. VAEs learn a latent space based on real data, therefore the interpolation in this latent space works better.

5.3 REQUIREMENTS AND DESIGN CHOICES

When selecting the audio representation, there is a trade-off between the higher complexity and the audio quality and diversity that the model can ideally produce. For this thesis, the author has decided to tackle the task of raw audio generation. To fully take advantage of the information encoded in a Midi file, musical knowledge is needed that the author does not thoroughly possess. By presenting the model raw audio data from solely one genre, the model can learn the important aspects of the music by itself. Another reason is that in the future, with increasing computational power, models for raw audio generation will likely surpass all other music generation models because the model is not limited in any way. Raw audio also increases the importance of a deep model with the right settings, which is the essential data science related objective of this thesis.

After the decision which audio representation to use, the next decision has to be how the generative deep learning model should be designed. Due to the shorter training time of CNN compared to RNN, the author will use a CNN-based architecture for this thesis. Another important aim of this thesis is to create a model where the user can influence the generation process. Autoencoder as well as GAN-based architecture allow for interpolation of the latent space and have the possibility to condition on additional features. The author decided to implement a GAN-based architecture. So far, not too much research has been done on applying GAN architectures to raw audio data. The WaveGAN architecture is able to only produce audio of one second. The aim of the thesis is to improve upon this architecture to allow for longer sequences. To keep it computationally feasible, the aim is to generate musical pieces of around ten seconds. The genre of electronic music without vocals will be used. The model should be able to model long-term dependencies. Therefore, the underlying beat of the musical piece must not alternate too much over the period of ten seconds. The requirements for the deep generative model of this thesis can be summarized by Table 5.1.

Table 5.1: Requirements of the deep generative model that will be developed in this thesis

Requirement 1	The model has to produce a song of at least 10 seconds
Requirement 2	The model has to model long-term dependencies. The underlying beat must not alternate too much, resulting in a coherent song
Requirement 3	The user should be able to influence the produced song in some way
Requirement 4	The song has to be free of noise, and it should be pleasant to listen to it

6 CONCEPT

6.1 GROWING GAN GENERAL IDEA

The WaveGAN is based on the DCGAN architecture that stabilizes the training for deeper GAN networks. However, The WaveGAN can only output 16384 samples. At a sampling rate of 16 kHz at least $160,000$ samples are needed to produce a song of ten seconds or longer. Increasing the size of this architecture by simply adding new layers results in an unsatisfying outcome, as was tested by the author https://github.com/dennis31197/Master_Thesis/tree/master/WGAN-GP. Inspired by the paper "Progressive Growing of GANSs For Improved Quality, Stability, and Variation" [31] of Tero Karras and his colleagues from the year 2018, the aim of this thesis is to achieve the bigger network size by progressively growing the discriminator and the generator of the GAN. The paper showed that it is possible to apply this strategy for image generation to generate coherent images of 1024×1024 pixels with good quality. They started with downscaled images of 4×4 pixels and trained the networks for this resolution. Then in each iteration they doubled the resolution by adding new layers to the networks while keeping the parameters of the already trained layers. The idea is that it is easier to learn the mapping from a random initial latent vector to a complex high-resolution image in steps rather than directly. First, the GAN learns global structure of the images through the lower resolution and in later iterations more and more details. It is yet to be shown that the same principle can be used for 1-D audio data as well. With their chosen end resolution, their network can generate $1,048,576$ samples, which corresponds to 65.5 seconds of audio with a sample rate of 16 kHz. The architecture that they used for the fully grown network is shown in Figure 6.1. In the first column of both the generator and the discriminator, the type of layer and the kernel size for the convolutional layers is indicated. In the second column, the activation function is named. The first number of the output shape is the number of feature maps (also called channels) and the last two numbers indicate the current resolution of the image. In the last column, the number of parameters that each layer uses is shown. The generator is fed with a latent vector with points sampled from a standard normal distribution. This latent vector is upsampled by transposed convolutions to the first 4×4 image. For every growing iteration, one convolutional block consisting of one upsampling layer (through replication) and two convolutional layers with a 3×3 kernel is added. The generator outputs a 3×1024×1024 image, where the three channels are the three different colors of an RGB image. Likewise, the discriminator takes a 3×1024×1024 image as an input and downscales the image with average pooling until it outputs a single number. The wasserstein loss is used, there-

fore, the last layer of the discriminator uses a linear activation function. The discriminator and the generator are simultaneously scaled such that the output size of the generator matches the input size of the discriminator for every iteration.

Generator	Act.	Output shape	Params
Latent vector	–	512 × 1 × 1	–
Conv 4 × 4	LReLU	512 × 4 × 4	4.2M
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Upsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Upsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Upsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Upsample	–	512 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	1.2M
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Upsample	–	256 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	295k
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Upsample	–	128 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	74k
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Upsample	–	64 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	18k
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Upsample	–	32 × 1024 × 1024	–
Conv 3 × 3	LReLU	16 × 1024 × 1024	4.6k
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 1 × 1	linear	3 × 1024 × 1024	51
Total trainable parameters			**23.1M**

Discriminator	Act.	Output shape	Params
Input image	–	3 × 1024 × 1024	–
Conv 1 × 1	LReLU	16 × 1024 × 1024	64
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 3 × 3	LReLU	32 × 1024 × 1024	4.6k
Downsample	–	32 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Conv 3 × 3	LReLU	64 × 512 × 512	18k
Downsample	–	64 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Conv 3 × 3	LReLU	128 × 256 × 256	74k
Downsample	–	128 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Conv 3 × 3	LReLU	256 × 128 × 128	295k
Downsample	–	256 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Conv 3 × 3	LReLU	512 × 64 × 64	1.2M
Downsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Downsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Downsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Downsample	–	512 × 4 × 4	–
Minibatch stddev	–	513 × 4 × 4	–
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
Fully-connected	linear	1 × 1 × 1	513
Total trainable parameters			**23.1M**

Figure 6.1: GAN Architecutre to generate 1024×1024 images [31]

Another strategy that they implemented is that they do not introduce the higher resolution suddenly, but rather fading in the new convolutional blocks smoothly. Figure 6.2 shows this strategy.

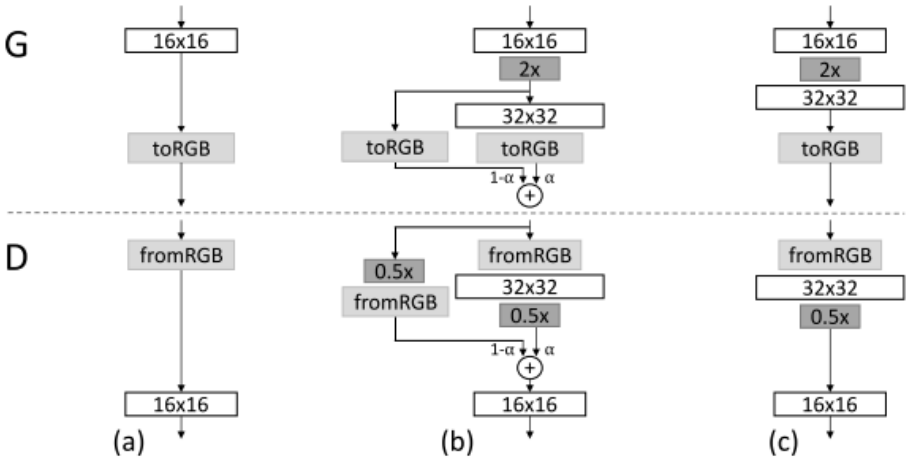

Figure 6.2: Strategy of fading in new convolutional blocks in the growing GAN architecture [31]

The fields "toRGB" and "fromRGB" stand for the operation of transforming data to an RGB image or the inverse. In (a) the generator outputs an 16×16 image that is sent to the discriminator. (b) represents the fade in phase where a new higher resolution is introduced. The 16×16 image is upsampled by a

factor of two and sent through two different routes. In one path, the upsampled version is directly turned into an RGB image and in the other path it is first passed through another convolutional block represented by the 32×32 field. The circle with the plus represents a weighted sum layer where the two inputs are scaled with α and $1 - \alpha$ respectively and then added up. α is slowly increased from zero to one, increasing the weight of the new convolutional block until the state (c) is reached. The reason for the smooth fade in is that the network should not unlearn the already learned mapping by changing the parameters too much when presented with an entirely new representation of the data. By smoothing in the higher resolution, the network has time to slowly adapt.

6.2 GROWING GAN FOR AUDIO DATA

To use the growing GAN algorithm, as it was implemented by Karras, for audio data, some adjustments have to be made. Details of the implementation and the specific architecture choices will be discussed in the next chapter. In this section, it will be shown how the growing GAN algorithm can help to meet the in the previous chapter defined requirements for this thesis.

REQUIREMENT 1 The length of the song depends on two factors. The first factor is the number of samples generated by the generator network in its output layer. The second factor is the sample rate that is used. The higher the sample rate, the more samples are needed to produce one second of audio, increasing the quality of the audio. Requirement 1 can therefore be met by having a generator network with the required output size defined by the sample rate. In the simplest form, the generator could map the input latent vector directly to the output size for a low sample rate. The growing GAN algorithm is hence not necessarily needed to meet requirement 1. However, training a GAN architecture with a high enough sample rate and the required output size right away results in insufficient quality. The output size and the sample rate have to be selected deliberately.

REQUIREMENT 2 To model long-term structure, every generated sample has to be conditioned on enough surrounding samples. In the WaveGAN paper, the authors explain that for the musical note A4 with a frequency of 440 Hz and a sample rate of 16 kHz, 36 samples are needed to complete a whole cycle of the soundwave. [12] To model the lowest frequency that humans can hear (20 Hz) at the same sampling rate, $\frac{16000Hz}{20Hz} = 800$ samples are needed. To model coherent change in the frequency over time, an even wider receptive field is needed. When using convolutional layers, the receptive field of every output sample is determined by the kernel size of the convolutional filters and the number of stacked layers. This concept was well shown by Oord in the WaveNet paper explained in section with Figure 4.1. The WaveNet is an autoregressive model, so every sample is conditioned only on previous ones with casual convolutions. The GAN architecture can predict all samples at

once, and can therefore also use future samples for the prediction. This concept is depicted in Figure 6.3 with a simplified case of a kernel size of five. Each sample of the upper layer takes nine samples into consideration if two convolutional layers are stacked.

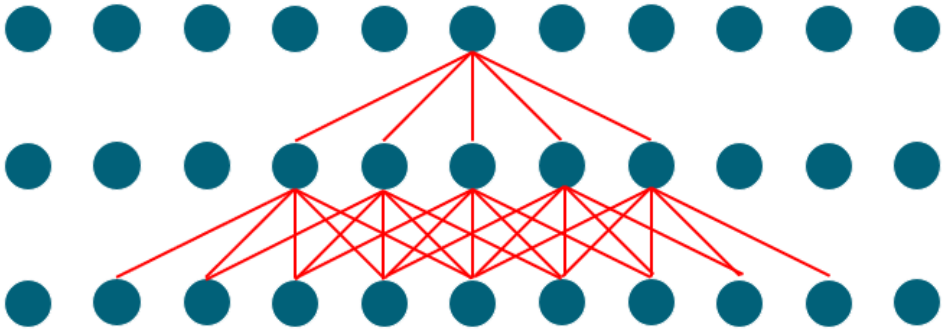

Figure 6.3: Receptive field for normal convolutions with a kernel size of five

For a deeper network like the WaveGAN or the network shown in Figure 6.1, the generator starts with a shorter latent vector that is upsampled several times. For a deep enough network and a large filter size, every value of the latent vector influences every output sample if the vanishing gradient problem is handled well. Therefore, these type of networks have the ability to model long-term structure. As was already mentioned in the last section, learning this mapping directly is a complex task. The growing GAN algorithm decomposes this complex task into smaller, more feasible ones.

It starts with learning a lower representation of the data. For audio data, that corresponds to a lower sampling rate. If, for example, a sampling rate of 100 *Hz* is used, the network has to predict only 100 samples for one second of audio. This way, it is easier to learn the global structure of the song. According to the Shannon Nyquist sampling theorem, all frequencies above half the sampling rate are lost. This lower representation therefore contains only the low frequencies of the signal that are first learned by the model. The Shannon Nyquist sampling theorem additionally says that the higher frequencies are mapped to lower alias frequencies that are contained as well in the signal as artifacts. In music processing, anti-aliasing filters can be used to reduce this effect. In this thesis, a standard python library for music processing will be used that does not contain such filters.

For every growing iteration of the network, the sampling rate is increased simultaneously with the network size, such that the length of the song stays the same for every iteration. The model is thus introduced to higher frequencies every iteration. As Karras pointed out, following this strategy, in each iteration the model is asked a much simpler question compared to discovering a mapping from a latent vector to some complex data right away. The model can concentrate on lower frequencies and global structure at the beginning, and then on the high-fidelity features in later iterations.

REQUIREMENT 3 To manipulate a song, a lower representation of the song is needed, or the model has to be conditioned on some predefined features. The GAN architecture takes a latent vector as input that it maps to a song. Even if this latent vector is just sampled from a probability distribution, the network learns to map these number to the complex data structure. When giving the network the same input vector twice, the output is both times the same. The input vector therefore determines the song and can be manipulated. When the network is trained well, latent vectors that are close in the hyperspace should result in similar songs. Another way to manipulate the song is by giving the network an additional feature as an input. The generator is then presented by a random latent vector plus, for example, the beats per minute of the song or the genre. When the network is trained, these features can be used to steer the outcome of the generator. Both ways are possible for the growing GAN architecture.

REQUIREMENT 4 The main aspect of good sound quality is the sampling rate used to process the audio signal. If it is too low, higher frequencies cannot be captured. The growing GAN architecture helps to further improve sound quality for longer sequences, with the same principle explained for requirement 2. The network does not have to learn too many relations at once, which improves the quality of the outcome. Additionally, the smooth fade in facilitates to stabilize the growing iteration. If the outcome is still noisy, an anti-aliasing filter can further be used to avoid artifacts when using a low sampling rate and therefore to improve the sound quality.

6.3 DATA COLLECTION

The model should be able to model arbitrary music. Hence, different songs from the electronic music genre without vocals were collected from the internet. The songs had to be in the MP3 or WAV format and should be legally free to use to avoid any legal complications. The website where the music was collected from is "Free Music Archive". [1] The author listened to songs from the subgenres "Techno", "Minimal Electronic" and "Drum & Bass" to make sure they do not contain vocals and are of decent quality. That way, 252 songs in the MP3 format were collected. The data preprocessing will be explained in the following chapter.

1 https://freemusicarchive.org/

7 IMPLEMENTATION WITH TENSORFLOW AND KERAS

7.1 PRELIMINARY CONSIDERATIONS

Before preprocessing the data and designing the specific model, some overall design choices have to be made. Both the generator and the discriminator will be progressively grown. Therefore, the number of growing iterations along with the initial and the final sampling rate have to be defined. The sampling will be done by the python library Librosa. [35] It helps programmers to extract information out of audio data. In this thesis, it is used to extract the waveform data from the MP3 files and to visualize it. When loading the waveform data with Librosa, the sampling rate can be specified. The final sampling rate should be high enough to provide a sufficient audio quality. CD quality uses a sampling rate of 44100 Hz, as it is also used in the Jukebox model. To lower the computational cost and especially the memory consumption of the model, the final sampling rate will be half that size for this thesis. The author listened to the songs downsampled to 22050 Hz and the audio quality was for this use case only negligibly compromised. The growing strategy is based on the original paper of Karras, where the resolution is doubled for every iteration with eight iterations. The starting sampling rate is therefore $\frac{22050}{2^8} = 86.1328125$, which is also the lowest possible sampling rate for Librosa. The original growing GAN architecture of Karras was able to produce $1,048,576$ samples in the fully grown size, which would be $\frac{1,048,576}{22050} = 47.55$ seconds of audio. To again lower the computational cost and the memory consumption of the network, the final output size will be a quarter of this, resulting in $262,144$ samples and 11.89 seconds of audio, still satisfying the requirements. The output size of the generator and the input size of the discriminator also grows by a factor of two each iteration, starting by $\frac{262,144}{2^8} = 1024$ samples in the first level, such that the song duration always stays the same. This is because for a lower sampling rate, fewer samples are needed for the same duration. Thus, increasing both by the same factor keeps the duration the same.

7.2 DATA PREPROCESSING

The discriminator of the GAN has to be provided with real training data. Therefore, the MP3 files have to be converted into vectors of float numbers, representing the amplitude values of the wave. The *load* method of the Librosa library is used.

```
librosa.load(path, sr=22050, mono=True,
             offset=0.0, duration=None,
             dtype=<class 'numpy.float32'>,
```

```
res_type='kaiser_best')
```

It takes seven arguments. The *path* argument is the path to the input file. *sr* stands for the sampling rate. *mono* indicates whether the signal should be converted to type mono. Stereo signals save two or more waveforms in parallel. This way, 3-D audio effects can be produced. Mono signals only store one waveform. The *duration* is the time in seconds that is loaded, and *offset* defines after how many seconds it starts to load the signal. With *dtype* the python data type can be defined to store the data. *res_type* is the resample type that defines how a signal is interpolated when the sampling rate used to load the signal differs from the sampling rate that the signal is currently stored with.

For the growing GAN strategy, the data has to be loaded and stored with nine different sampling rates, as defined in the previous section. The *mono* argument is kept true, so the network only has to deal with one waveform. The network cannot process varying input lengths. Therefore, every MP3 file has to be divided into sections of equal length for every iteration. The first level discriminator takes 1024 samples as an input. The MP3 files are loaded entirely (this corresponds to keeping the *duration* argument at None) and then sliced into vectors with 1024 values each. The *offset* argument is set to five to avoid modelling the initial build up of a song. The python data type is kept at default, as is the *res_type* argument. The default encoding is 16bit and cannot be directly changed with Librosa *load* method. With the same procedure, the training data for all nine levels is generated. Every vector is saved as an NumPy array in a dedicated folder for every level.

7.3 STRUCTURE OF THE MODEL

In Appendix in Table A.1 and Table A.2 the structure of the generator and the discriminator respectively for the final level is shown. The following subsections explain the concepts and components of the networks.

7.3.1 *Input and Output Blocks*

The input of the generator is a random latent vector sampled from a probability distribution. The WaveGAN architecture uses a uniform distribution between negative one and one. The same sampling for the latent vector will be used in this thesis, since this distribution already simulates a waveform. There is no general rule to select the right length of the latent vector. The more values the generator gets as an input, the more freedom it has to generate the output and the easier it gets to map the input to the output. At the same time, it is preferable to have a small input vector such that the generator learns a compact latent space that can easily be explored and interpolated. The WaveGAN architecture uses a latent vector of size 100 to generate 16,384 samples and the original growing GAN architecture a latent vector of size 512 to generate 1,048,576 samples. Following this order of magnitude, the latent vector for this thesis will be of size 256. As in the

WaveGAN architecture, the latent vector will then be passed through a fully connected layer and reshaped into the shape 1024×4 representing the output of a 1-D convolution with four feature maps. This representation will then be passed as input into the further network. The output size of the generator depends on the iteration level and is doubled every iteration starting by 1024. The output layer will be a convolutional layer with only one feature map. The tanh activation function is used, which maps the feature map values to values between negative one and one representing the waveform.

The discriminator takes the output of the generator and the prepared training data as an input. Therefore, the input size is already defined and changes for every iteration. This input is then processed by a convolutional layer with kernel size one with a leaky ReLu activation function. This was done by Karras in the original progressive GAN architecture. The kernel of size one allows the discriminator to scale and shift the values if necessary to facilitate the discrimination. This representation is then passed as input into the further network. The output of the last convolutional layer of the discriminator will be flattened and connected to a single neuron with a linear activation function. Using several fully connected layers on top of a CNN architecture was proven disadvantageous by the DCGAN paper. [47]

7.3.2 *Convolutional Blocks*

For the generator, every convolutional block that is added for a new iteration consists of an upsampling layer and two convolutional layers. For the upsampling the Keras UpSampling1D layer with the standard parameters is used which replicates every value by a factor of two. This type of upsampling keeps the already learned structure of the soundwave. The convolutional layers have a kernel size of 25 and the number of kernels used for the layers decreases for later levels. It is better to use a larger kernel size, but this increases the computational cost and memory consumption significantly. The number of kernels at the initial blocks is greater, so the model has more possibilities to save different aspects of the song at the beginning but has to consolidate this information into a single soundwave over time. The number of kernels is mainly copied from the original growing GAN paper from Karras, since it already worked well for learning global structure in images. Every convolutional layer has a Leaky ReLU activation function with a leakiness of 0.2 as suggested by Karras. The normalization mechanisms are explained in subsection 7.3.4.

The discriminator is a mirrored implementation of the generator. For every block, the input is processed by two convolutional layers and then downsampled by the *AveragePooling1D* layer from Keras with a pool size and stride length of two. The padding strategy is set to "same". It pads the first few and last few values of the vector to the beginning and the end, respectively, such that the pooling operation can be performed for every position of the vector. The convolutional layers also have a kernel size of 25 and the number of kernels is increasing for later blocks. The leaky ReLu activation function

with a leakiness of 0.2 is used as well. In the last convolutional block before the output block, another layer constructed by Karras in the original growing GAN paper is added. This layer is called Minibatch Standard Deviation and calculates statistics over a minibatch and adds it as a feature map. This idea originally stems from Salimans and his team and was simplified by Karras. [51] For every position of the vector, the standard deviation over the minibatch for every feature map is calculated. The average of all values for every position and every feature map is then calculated, such that only one number remains. This number is replicated to the original size of a feature map. The resulting constant feature map is finally added to the data and sent to the next layer. The generator in a GAN struggles with modelling the variation found in the training data, as pointed out by Salimans. Adding this additional statistic helps the network to increase its variation.

7.3.3 *Loss Function for the Generator and Discriminator*

The loss function of the network will be the wasserstein loss, enforced with the gradient penalty. The discriminator therefore outputs a critic score through a linear activation function. To calculate the loss used to train the discriminator, the discriminator is shown one batch of real songs and one batch of fake songs created by the generator. The discriminator calculates the critic score for every sample in the real and the fake data. It then averages the score for the real and the fake data, respectively. The loss can now be calculated by subtracting the average score of the real data from the average score of the fake data. Hence, the goal of the discriminator is to output large numbers for real songs and small numbers for fake songs. It does not matter if the numbers are positive or negative, as long as the distance between the scores for fake and real data is large enough. Additionally, the gradient penalty is added to this loss and scaled by a weight set to ten. Another term that is added by Karras to this loss is a drift score, that calculates how far the discriminator's output has drifted from zero. Large numbers for the loss can impede the training, indicating exploding gradients. Therefore, a large negative or positive output is penalized by this term. This was observed to be only needed for the last two levels of the growing strategy. The final loss equation can now be expressed by Equation 7.1.

$$\begin{aligned} d\_loss = & fake\_critic\_score\_mean - real\_critic\_score\_mean + \\ & gp * gp\_weight + drift * drift\_weight \end{aligned} \tag{7.1}$$

The loss function of the generator is simply the negative average score for the fake songs from the discriminator. The discriminator tries to output small numbers for fake samples and large numbers for reals samples. Therefore, if the generator successfully generates songs that the discriminator assesses as real, the discriminator will output a large number, which leads to a large negative loss. If the discriminator successfully recognizes the generated songs

as fake, it will output a small positive or large negative number, resulting in a large loss for the generator. The generator loss is shown in Equation 7.2.

$$g\_loss = -fake\_critic\_score\_mean \tag{7.2}$$

7.3.4 *Normalization in the Generator and Discriminator*

Karras introduced pixelwise feature vector normalization for the generator. The same normalization technique will be used in this thesis, since it already showed good results for a very deep architecture. The WaveGAN architecture does not use any normalization technique, neither in the generator nor in the discriminator. Another advantage is, that it does not have any learnable parameters compared to BatchNorm. It scales the feature vector for every position to unit length. The feature vector is the vector of feature map values for the same position (for images it is pixels). This is done by Equation 7.3 where b_x is the scaled value at position x and a_x the original.

$$b_x = \frac{a_x}{\sqrt{\frac{1}{N}\sum_{j=0}^{N-1}(a_x^j)^2 + \epsilon}} \tag{7.3}$$

As in the WaveGAN and the original growing GAN architecture, no normalization technique will be used for the discriminator.

7.3.5 *Fade in of a new Convolutional Block*

After the generator and discriminator are trained for one level, the new level is faded in smoothly. In Figure 7.1 the output of the Keras method *plot_model* for the fade in model of the second level is shown. The first level outputs 1024 samples. This is upscaled to 2048 for level two. The fade in generator takes the output of the last convolutional block of the previous level and splits it into two paths. Therefore, all layers from the previous level with the according weights are reused. At the top of Figure 7.1 the last convolutional layer with its normalization and activation of level one forwards its output to two upsampling layers. The right path sends the upsampled signal directly into an output layer. In the left path, the upsampled signal is first processed by another convolutional block of the level two generator before being passed into an output layer. The "weighted_sum_layer" takes the outputs from both paths and multiplies them by a weight before summing them up. The left output is multiplied by α and the right output by $1 - \alpha$. The weight alpha is incrementally increased from zero to one after every epoch. Thus, at the beginning of the training, the new convolutional block of level two does not influence the outcome. When alpha reaches one, the transformation from level one to level two is completed and the right path can be dropped.

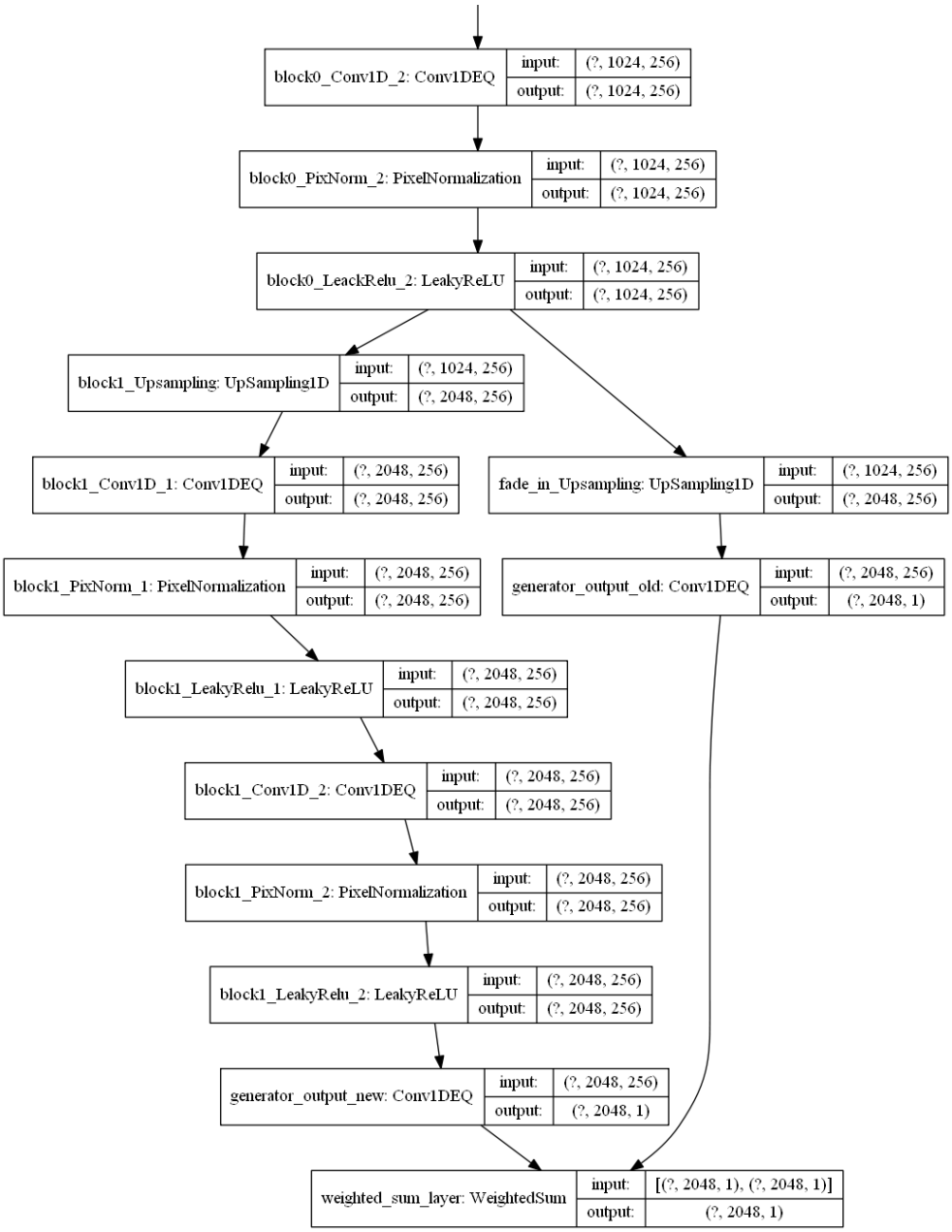

Figure 7.1: Fade in process of smoothing in level two for the generator

The new level of the discriminator is faded in the same way. The difference is that the discriminator is extended at the beginning of the network to be able to receive the larger input size. Both networks are trained together and the weight increase for alpha happens at the same time with the same factor.

7.3.6 *Other Hyperparameters and Settings of the Model*

The main idea of selecting the hyperparameters is to make the model able to learn efficiently, even for deep architectures. Since, the settings for the growing GAN architectures of Karras worked well for a similar architecture, the same optimizer with its parameters will be used. The generator and the

discriminator will therefore be trained with the Adam optimizer with the parameters $\alpha = 0.001$, $\beta_1 = 0$ and $\beta_2 = 0.99$. Karras also observed that using Adam alone still leads to suboptimal training due to exploding or vanishing gradients. This lead to introducing another restriction to the model called equalized learning rate. The weights are scaled at runtime for every layer with a per-layer normalization constant c. Equation 7.4 shows the formula.

$$\hat{w}_i = \frac{w_i}{c} \tag{7.4}$$

The per-layer normalization constant c was defined by He in 2015 and is given by equation 7.5. [25] It calculates the standard deviation for the weights of each layer, with k being the kernel size and d the number of filters. The weights are hence scaled by dividing by their standard deviation. This prevents them from becoming too small or too large

$$c = \sqrt{\frac{2}{k^2 * d}} \tag{7.5}$$

Another hyperparameter to consider is the discriminator learning steps. The mathematical assumption for the wasserstein loss holds true for an optimal discriminator. Therefore, usually the discriminator is trained several steps for every training step of the generator. Karras deviates from this strategy and alternates directly between optimizing the generator and the discriminator. For the WaveGAN architecture, the discriminator was trained with five optimization steps before optimizing the generator once. In this thesis, a factor of two will be used to still have some benefits of a better discriminator without increasing the computational requirements too much.

7.4 STRUCTURE OF THE ACCOMPANYING GIT REPOSITORY

The code for the whole thesis can be found in the following Git repository: https://github.com/dennis31197/Master_Thesis. The code was written in Jupyter Notebooks. The author decided to keep the code in Jupyter Notebooks to be able to add markdown fields to provide explanations additional to the comments in the code. In the folder "Growing_GAN" is all the code for the growing GAN algorithm. The notebook "preprocessing.ipynb", converts the MP3 files into the needed format as described in section 7.2. The notebook "model.ipynb" contains the code to build and train the whole model, consisting of the generator and the discriminator for the different levels. The notebook "generating_music.ipynb" loads an already trained generator and uses it to generate new music. The folder "Monitoring" contains the final discriminator and generator models.

7.5 TRAINING TIME AND HARDWARE

The first four levels of the network were trained on a Nvidia GeForce GTX 1650 Max-Q GPU and on a computer with 16 GB RAM for a week. The full

networks were trained for 14 epochs and the fade in networks for 7 epochs with a batch size of 16 for the first two levels, 8 for the third level and 4 for the fourth level. Due to the increasing network sizes, the 4 GB of vRAM and 16 GB of RAM were not sufficient for later levels. The training was further executed on a Nvidia Tesla T4 GPU with 16 GB vRAM on a computer with 26 GB RAM in the Google Cloud. Level five was trained with a batch size of 8 for 6 epochs for the fade in networks and 12 epochs for the full networks. Level six was trained with a batch size of 8 for 5 and 10 epochs, respectively. For level seven, a batch size of 4 was used, and the networks were trained for 3 and 4 epochs, respectively. The last level of the model could not be trained because of a timeout error that occurred and that will be discussed in the next chapter. The whole training of levels five to seven took eight days.

7.6 CODE DEBUGGING AND OPTIMIZATION STRATEGIES

For Keras models, graph execution is automatically enabled. That means before the training begins, the whole network is converted into an executable graph that is then internally optimized. This lengthens the time that the program needs to start the training process but reduces the training time per batch significantly. In order that the graph can successfully build and executed, all data types have to be TensorFlow own datatypes. All functions that are involved in the training process can only take tensors as input and only return tensors or None. There are other prerequisites that have to be considered with graph execution that are linked in the Git repository. Besides these prerequisites, another downside of graph execution is the enormous memory consumption or deep models. Models that can be trained with eager execution result in an out of memory error for graph execution. This is due to the graph being built in RAM and has to be loaded into vRAM for training.

Eager execution can be explicitly set prior to the training. This helps with debugging the code because it is run line by line and possible errors are shown at the exact spot. Graph execution can result in a crash of the machine if it takes too long to build the graph, if certain prerequisites are not met or if an error occurs during the execution of the graph. The program does not know where the error occurs during the execution and therefore, no explicit error message is shown.

To make sure the graph can be successfully built, the code was tested for lower levels with a smaller network size to reduce the building time. The used functions for the training process are the same for all levels, so if the graph is built for the lower level, it can technically be built for the final level as well.

The whole training with all growing iterations was tested with only two batches and one epoch for every level to make sure it can run entirely without an error and that the monitoring works as intended.

To lower the strain on the RAM, batch generation was used to provide the training data to the model without having to load the whole dataset. In the

fit method of the Keras model, the number of batches that should be loaded into memory can be set. This helps to optimize the training time. The batch loading is done by the CPU and the batch processing is done by the GPU. With this strategy, the GPU never has to wait for a batch and can be used efficiently without having all data in memory. For this thesis, 8 batches were loaded into RAM. After every epoch of training, the python garbage collect and the TensorFlow clear session is called. Both commands delete objects from memory that are no longer needed. This also helps reducing the load on the RAM.

8

RESULTS

Figure 8.2 shows the loss history of level seven for the generator and the discriminator in epoch four. The generator loss in orange has a high volatility. It is defined by the critic score of the discriminator for the generated samples of the generator. Fluctuating values indicate that the discriminator has not found a good rule to differentiate between real and fake samples. The discriminator loss stays close to zero, indicating that the critic scores for fake and real samples for a training batch of the discriminator are of similar magnitude with a different sign. This behavior still needs further investigation.

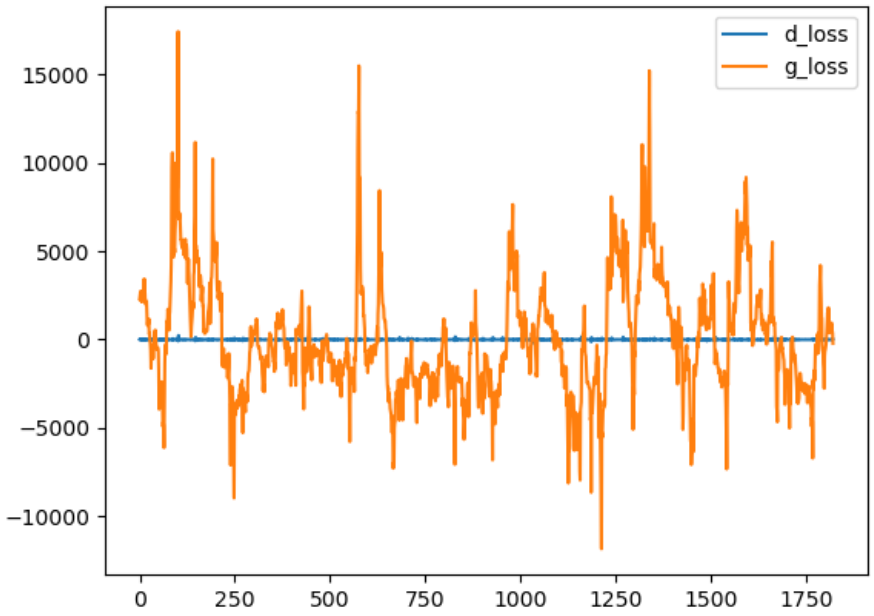

Figure 8.1: Loss history for epoch four for the discriminator (d_loss) and generator (g_loss) of level seven; The y-axis is the loss value and the x-axis is the batch number

Figure 8.2 shows the loss history of the generator and discriminator of level three for epoch 14. The generator loss has again a high volatility, but the values are closer together in terms of magnitude. This shows that the discriminator does a better job identifying real and fake samples than it did in Figure 8.1. There is also a downwards trend recognizable, indicating that the generator improves during this epoch. The discriminator loss fluctuates as well around zero.

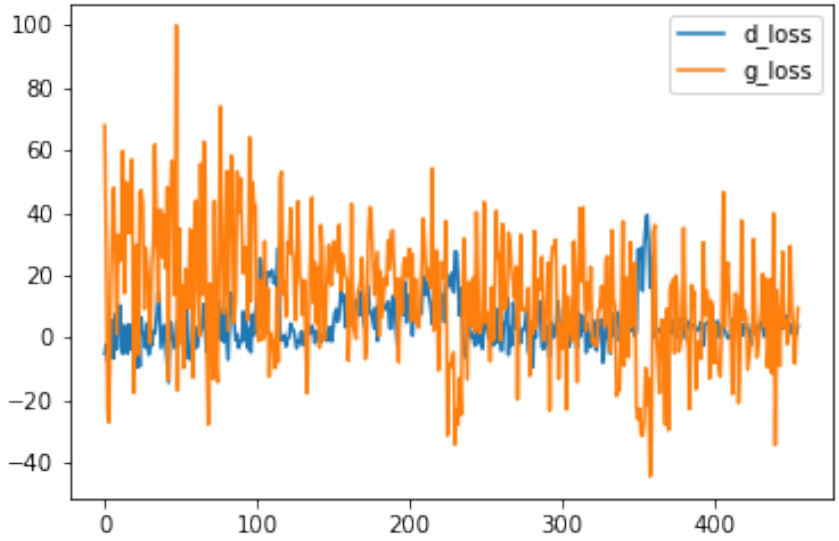

Figure 8.2: Loss history for epoch 14 for the discriminator (d_loss) and generator (g_loss) of the third level; The y-axis is the loss value and the x-axis is the batch number

A possibility to visualize the song is to plot of its waveform. In Figure 8.3 the waveform of an eleven second segment of a song from the raining data is shown. In Figure 8.4 the waveform of a fake song generated by the generator. The real song tends to have more of a clear beat that stays along the whole song. Whereas, the waveform of the generated song seems to have a rather random pattern.

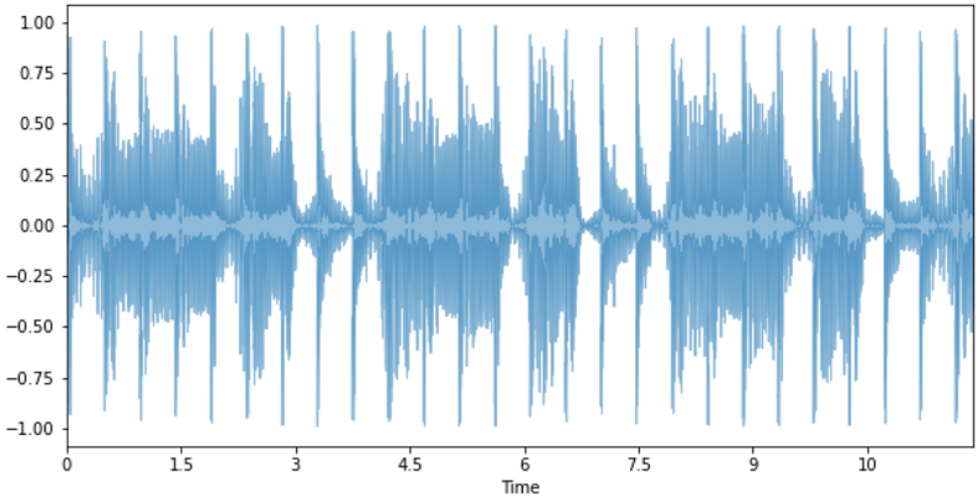

Figure 8.3: Waveform of a eleven second segment of a real song from the training data

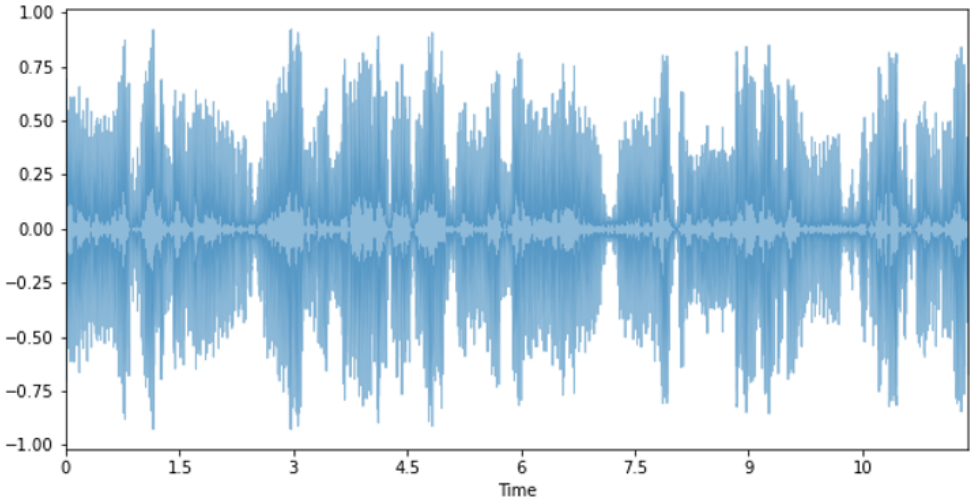

Figure 8.4: Waveform of a eleven second segment of a fake song created by the generator

The loss values and the plot of the waveform only give a notion of how well the model is actually doing. To assess the quality of the GAN properly, the generated songs have to be listened to. Therefore, the Jupyter Notebook "generating_music.ipynb" from the Git repository can be used. The written code loads the latest generator and lets the user generate a random latent vector that is used by the generator to generate a song. Another option is to manipulate the single values of the latent vector with sliders to steer the outcome of the song.

The model successfully outputs a song of eleven seconds and fulfills requirement 1. To generate a random song, the model needs less than one second. The user has also the possibility to alter the soundwave of the song. The latent space has to be further explored to recognize possible dependencies between the values of the latent vector and the resulting song. Hence, requirement 3 is fulfilled. The songs still contain a lot of noise, and the beat changes over the course of the song. Nevertheless, some structure can be identified, and the outcome is a step into the right direction. The model has to be further improved, and the last level has to be trained to also fulfill requirements 2 and 4. The author refrained from sending the results to people to listen to and to judge the quality of the song, because the audio quality is still no good enough to compare it to the training data.

The idea to use the growing architecture to reduce the computational cost by giving the network simpler tasks in every step did not succeed fully. Especially in the earlier levels, a decreasing generator loss could be observed, as shown in Figure 8.2. A longer training time therefore would further improve the generator. The resources for this thesis were limited and hence restricted the author in the testing and training of the model. One epoch for final seven took ten hours of training time on a Nvidia Tesla T4 GPU in the Google Cloud. To not produce too high cost, the number of epochs was lowered for the later levels. As seen in Figure 8.1 after only four epochs the discriminator of level seven was still very unsure about whether the song was real or fake.

Google also deals with a shortage of GPUs at the moment and sometimes could not provide a respective machine. To train the last level of the model, it was planned to use the better Nvidia Tesla V100 GPU, which was not available for the required period. Training the final level with the available machine resulted in a timeout error before the actual training started. This timeout only occurred for graph execution. This means the creation of the network graph cannot be finished. For graph execution, Tensorflow does not output error messages, hence, the problem of insufficient memory can only be assumed. The network starts training when eager execution is enabled, but one epoch would need 40 hours to complete. When at the beginning, the whole algorithm with all levels was trained with only two batches and one epoch, no timeout occurred.

The training data with 252 songs which were divided into 7288 segments was also small. In the original growing GAN paper, they used 30,000 images and trained the model with eight Nvidia Tesla V100 GPUs with a vRAM of 32GB each for four days. If a linear relation is assumed, that would result in 32 training days on that respective GPU. Currently, the best architecture for audio generation using raw audio is Jukebox. They used 256 Nvidia Tesla V100 GPUs for several days.

To address this problem, the whole architecture of the model has to be modified. In the current architecture, two convolutional layers for every convolutional block are used. Using only one layer with more filters would reduce the memory capacity of the network and would make it easier to train. Tensorflow also uses a 32bit float format to save all values of the model. The raw audio data was saved in a 16bit format. By creating the model with also a 16bit encoding, the memory capacity could have been reduced without losing valuable information. In chapter 4 dilated convolutions were introduced. By using this type of convolutions, the receptive field of the network could be increased without changing the needed memory. This would help the network with learning long-term dependencies. Using a high-level representation of audio like Midi files or Spectrograms would highly reduce the requirements of the network and the training. Until more computational power is easier and cheaper accessible, the use of these high-level formats should be favored over the use of raw audio data for all use cases besides research.

9 SUMMARY AND OUTLOOK

This thesis laid a good foundation for further research in the audio generation area. The different data representations that can be used were explained and benefits as well as disadvantages presented. Audio data depends on time and can be processed by different types of networks. So far, there has not emerged a single best approach, and algorithms change and improve constantly. This thesis used CNN networks paired with a GAN architecture to model raw audio data. Raw audio data is very complex and high dimensional. Therefore, networks struggle to detect and model long-term dependencies. Until the computational power of computers suffice to easily train deep enough models, it is a good idea to use a lower dimensional representation of audio. This means to divert to Midi files or to use Autoencoders to get the lower representation of the raw audio data to train a generative model. Midi files also give already pre-defined events that make sure that the generated sound is melodious. The generator of the GAN in this thesis was fed with a latent vector of values from a uniform distribution between negative one and one. Its task was to make sense of this randomness and find structure related to the structure of a song. Instead of giving the generator a random vector, it could be given a latent representation of the songs calculated by an Autoencoder. The downside is that it would make the generator more likely to generate songs similar to the training data. Another way that might help the generator to model long-term dependency is to use dilated convolutions to increase the receptive field of the convolutional layers without increasing the computational cost.

The final generator of the trained model can generate audio segments of eleven seconds. There is no long-term structure, but some kind of beat can be identified. To use this model at a fair, the audio quality has to be increased. This can be done by collecting more training data and by training the model for the final level and for a loner time with a better GPU with more vRAM. The program has an interactive interface for the generation process. To present this to possible customers, the user interface should be developed with an appropriate framework. This thesis could serve as the foundation for later student project of the ORDIX AG. The model could be further improved or, with the acquired knowledge, it is possible to develop a more target-oriented model.

Part II

APPENDIX

APPENDIX A

Table A.1: Structure of the generator in the final level

Generator	**Act.**	**Norm.**	**Output shape**	**Params**
Latent vector	-	-	256×1	-
Dense	-	-	4096×1	$1,052,672$
Reshape	-	-	1024×4	-
Conv1D (25)	LeakyReLu	PixNorm	1024×256	$25,856$
Conv1D (25)	LeakyReLu	PixNorm	1024×256	$1,638,656$
Upsampling	-	-	2048×256	-
Conv1D (25)	LeakyReLu	PixNorm	2048×256	$1,638,656$
Conv1D (25)	LeakyReLu	PixNorm	2048×256	$1,638,656$
Upsampling	-	-	4096×256	-
Conv1D (25)	LeakyReLu	PixNorm	4096×128	$819,328$
Conv1D (25)	LeakyReLu	PixNorm	4096×128	$409,728$
Upsampling	-	-	8192×128	-
Conv1D(25)	LeakyReLu	PixNorm	8192×64	$204,864$
Conv1D (25)	LeakyReLu	PixNorm	8192×64	$102,464$
Upsampling	-	-	16384×64	-
Conv1D (25)	LeakyReLu	PixNorm	16384×64	$102,464$
Conv1D (25)	LeakyReLu	PixNorm	16384×64	$102,464$
Upsampling	-	-	32768×64	-
Conv1D (25)	LeakyReLu	PixNorm	32768×32	$51,232$
Conv1D (25)	LeakyReLu	PixNorm	32768×32	$25,632$
Upsampling	-	-	65536×32	-
Conv1D (25)	LeakyReLu	PixNorm	65536×32	$25,632$
Conv1D (25)	LeakyReLu	PixNorm	65536×32	$25,632$
Upsampling	-	-	131072×32	-
Conv1D (25)	LeakyReLu	PixNorm	131072×16	$12,816$
Conv1D (25)	LeakyReLu	PixNorm	131072×16	$6,416$
Upsampling	-	-	262144×16	-
Conv1D (25)	LeakyReLu	PixNorm	262144×16	$12,816$
Conv1D (25)	LeakyReLu	PixNorm	262144×16	$6,416$
Conv1D (1)	tanh	-	262144×1	17
Total trainable parameters				**$7,896,017$**

Table A.2: Structure of the discriminator in the final level

Discriminator	**Act.**	**Norm.**	**Output shape**	**Params**
Input Song	-	-	262144×1	-
Conv1D (1)	LeakyReLu	-	262144×16	32
Conv1D (25)	LeakyReLu	-	262144×16	6,416
Conv1D (25)	LeakyReLu	-	262144×16	6,416
AveragePooling	-	-	131072×16	-
Conv1D (25)	LeakyReLu	-	131072×16	6,416
Conv1D (25)	LeakyReLu	-	131072×32	12,832
AveragePooling	-	-	65536×32	-
Conv1D (25)	LeakyReLu	-	65536×32	25,632
Conv1D (25)	LeakyReLu	-	65536×32	25,632
AveragePooling	-	-	32768×32	-
Conv1D (25)	LeakyReLu	-	32768×32	25,632
Conv1D (25)	LeakyReLu	-	32768×64	51,264
AveragePooling	-	-	16384×64	-
Conv1D (25)	LeakyReLu	-	16384×64	102,464
Conv1D (25)	LeakyReLu	-	16384×64	102,464
AveragePooling	-	-	8192×64	-
Conv1D (25)	LeakyReLu	-	8192×64	102,464
Conv1D (25)	LeakyReLu	-	8192×128	204,928
AveragePooling	-	-	4096×128	-
Conv1D (25)	LeakyReLu	-	4096×128	409,728
Conv1D (25)	LeakyReLu	-	4096×256	819,456
AveragePooling	-	-	2048×256	-
Conv1D (25)	LeakyReLu	-	2048×256	1,638,656
Conv1D (25)	LeakyReLu	-	2048×256	1,638,656
AveragePooling	-	-	1024×256	-
MiniBatchStDev	-	-	1024×257	-
Conv1D (25)	LeakyReLu	-	1024×256	1,645,056
Conv1D (25)	LeakyReLu	-	1024×256	1,638,656
Flatten	-	-	262144×1	-
Dense	linear	-	1×1	262,145
Total trainable parameters				**8,724,945**

BIBLIOGRAPHY

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning*. 2018. ISBN: 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0.

[2] Martin Arjovsky and Léon Bottou. "TOWARDS PRINCIPLED METHODS FOR TRAINING GENERATIVE ADVERSARIAL NETWORKS." In: (2017), pp. 1–17. arXiv: arXiv:1701.04862v1.

[3] Martin Arjovsky, Léon Bottou, and Soumith Chintala. "Wasserstein GAN." In: (). arXiv: arXiv:1701.07875v3.

[4] Mariette Awad and Rahul Khanna. *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*. 2015. ISBN: 978-1-4302-5989-3. DOI: https://doi.org/10.1007/978-1-4302-5990-9.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer Normalization." In: (2015). arXiv: arXiv:1607.06450v1.

[6] Yoshua Benigo, P. Simard, and P. Frasconi. "Learning Long-Term Dependencies with Gradient Descent is Difficult." In: *IEEE Transactions on Neural Networks* (1994). URL: https://ieeexplore.ieee.org/document/279181.

[7] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. http://people.sabanciuniv.edu/berrin/cs512/lectures/Book-Bishop-Neural%20Networks%20for%20Pattern%20Recognition.pdf. 1995.

[8] Y-Lan Boureau, Jean Ponce, and LeCun Yann. "A Theoretical Analysis of Feature Pooling in Visual Recognition." In: (2009). URL: https://www.di.ens.fr/willow/pdfs/icml2010b.pdf.

[9] Mads G. Christensen. *Introduction to Audio*. 2019. ISBN: 978-3-030-11781-8. DOI: 10.1007/978-3-030-11781-8.

[10] Jeffrey Dean et al. "Large Scale Distributed Deep Networks." In: *Advances in Neural Information Processing Systems 25 (NIPS 2012)* (2012), pp. 1–11.

[11] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. "Jukebox: A generative model for music." In: *arXiv* (2020). ISSN: 23318422. arXiv: 2005.00341.

[12] Chris Donahue, Julian McAuley, and Miller Puckette. "Adversarial audio synthesis." In: (2019), pp. 1–16. arXiv: 1802.04208.

[13] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: 12 (2011), pp. 2121–2159.

[14] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning." In: (2018), pp. 1–31. arXiv: arXiv:1603.07285v2.

[15] Laurene Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall international editions. Prentice-Hall, 1994. ISBN: 9780133341867. URL: https://dl.matlabyar.com/siavash/Neural%20Network/Book/Fausett%20L.-Fundamentals%20of%20Neural%20Networks_%20Architectures,%20Algorithms,%20and%20Applications%20(1994).pdf.

[16] Neveille H. Fletcher and Thomas D. Rossing. *The Physics of Musical Instruments*. 1998. ISBN: 978-1-4419-3120-7. DOI: https://doi.org/10.1007/978-0-387-21603-4. URL: https://www.researchgate.net/publication/215646583.

[17] Harvey Fletscher and W. A. Munson. "Loudness, Its Definition, Measurement and Calculation." In: *The Bell System Technical Journal* (1933).

[18] *Free Scales*. https://www.youraccompanist.com/free-scales-and-warm-ups/reference-scales. [Online; accessed 04-October-2021].

[19] Dennis Gabor. "Theory of Communication." In: (1946). URL: http://wearcam.org/gabor1946.pdf.

[20] Ian Goodfellow. "Generative Modeling." In: (2016). arXiv: arXiv:1701.00160v4.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[22] Ian Goodfellow, Jean Pouget-abadie, Mehdi Mirza, Bing Xu, and David Warde-farley. "Generative Adversarial Nets." In: (2014). arXiv: arXiv:1406.2661v1.

[23] Ishaan Gulrajani, Faruk Ahmed, Martin Arovsky, Vincent Dumoulin, and Aaron Courville. "Improved Training of Wasserstein GANs." In: (2017). arXiv: arXiv:1704.00028v3.

[24] Simon Haykin. *Neural Networks: A Comprehensive Foundation Second Edition*. Pearson Education, 1999. ISBN: 81-7808-300-0. URL: https://cdn.preterhuman.net/texts/science_and_technology/artificial_intelligence/Neural%20Networks%20-%20A%20Comprehensive%20Foundation%20-%20Simon%20Haykin.pdf.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers : Surpassing Human-Level Performance on ImageNet Classification." In: (). arXiv: arXiv:1502.01852v1.

[26] Michael T. Heidemann, Don H. Johnson, and Charles Sidney Burrus. "Gauss and the history of the fast Fourier transform." In: (1985). DOI: https://doi.org/10.1007/BF00348431.

[27] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen." In: (1991).

[28] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions." In: (1998). DOI: 10.1142/S0218488598000094.

[29] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory." In: (1997). DOI: 10.1162/neco.1997.9.8.1735.

[30] Sergey Ioffe and Christian Szegedy. "Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: (2015). arXiv: arXiv:1502.03167v3.

[31] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. "PROGRESSIVE GROWING OF GANs FOR IMPROVED QUALITY, STABILITY, AND VARIATION." In: (2018), pp. 1–26. arXiv: arXiv:1710.10196v3.

[32] Diederik P Kingma and Jimmy Lei Ba. "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION." In: (2015), pp. 1–15. arXiv: arXiv:1412.6980v9.

[33] Alex Krizhevsky and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: (2012), pp. 1–9. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[34] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. In: (). URL: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

[35] Brian Mcfee, Colin Raffel, Dawen Liang, Daniel P W Ellis, Matt Mcvicar, Eric Battenberg, and Oriol Nieto. "librosa : Audio and Music Signal Analysis in Python." In: Scipy (2015), pp. 18–24.

[36] 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." In: *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings* (2017), pp. 1–11. arXiv: 1612.07837.

[37] Arthur Mendel. *Pitch in Western Music since 1500. A Re-Examination*. Vol. 50. 1978, pp. 1–93+328. DOI: https://doi.org/10.2307/932288. URL: https://www.jstor.org/stable/932288.

[38] Meinard Müller. *Fundamentals of Music Processing*. 2021. ISBN: 9783030698072. DOI: 10.1007/978-3-319-21945-5.

[39] *Neural Networks for Machine Learning Lecture 6e*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. [Online; accessed 22-October-2021].

[40] H. Nyquist. "Certain Topics in Telegraph Transmission Theory." In: (1928).

[41] Tim Olson. *Applied Fourier Analysis*. 2017. ISBN: 978-1-4939-7393-4. DOI: https://doi.org/10.1007/978-1-4939-7393-4.

[42] 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." In: (2016), pp. 1–15. arXiv: 1609.03499. URL: http://arxiv.org/abs/1609.03499.

[43] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. "Conditional image generation with PixelCNN decoders." In: *Advances in Neural Information Processing Systems* (2016), pp. 4797–4805. ISSN: 10495258. eprint: 1606.05328.

[44] *Pasco Sound Waves*. https://www.pasco.com/products/guides/sound-waves. [Online; accessed 19-July-2021].

[45] John G. Proakis and Dimitris K. Manolakis. *Digital Signal Processing, 4th Edition*. 2007. ISBN: 9780132341998. URL: http://www.elcom-hu.com/Electrical/Digital\%20Singnal\%20Proccessing/4th/4th\%20Digital\%20Signal\%20Processing\%20-\%20Proakis\%20and\%20Manolakis.pdf.

[46] *Python package: midicsv*. https://pypi.org/project/py-midicsv/#data. [Online; accessed 15-July-2021].

[47] Alec Radford, Luke Metz, and Soumith Chintala. "UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS." In: (2016), pp. 1–16. arXiv: arXiv:1511.06434v2.

[48] Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. "A hierarchical latent vector model for learning long-term structure in music." In: *35th International Conference on Machine Learning, ICML 2018* 10 (2018), pp. 6939–6954. arXiv: 1803.05428.

[49] David E. Rumelhart, McClelland James L., and PDP Research Group. *Parallel Distributed Processing, Volume 1*. https://mitpress.mit.edu/books/parallel-distributed-processing-volume-1. MIT Press, 1986.

[50] Tim Salimans. "Weight Normalization : A Simple Reparameterization to Accelerate Training of Deep Neural Networks." In: (2016). arXiv: arXiv:1602.07868v3.

[51] Tim Salimans, Ian Goodfellow, Vicki Cheung, Alec Radford, and Xi Chen. "Improved Techniques for Training GANs." In: (), pp. 1–10. arXiv: arXiv:1606.03498v1.

[52] Shibani Santurkar, Dimitris Tsipras, and Andrew Ilyas. "How Does Batch Normalization Help Optimization ?" In: (2018). arXiv: arXiv:1805.11604v5.

[53] Jürgen Schmidhuber. "Generative Adversarial Networks are Special Cases of Artificial Curiosity (1990) and also Closely Related to Predictability Minimization (1991)." In: (2020). arXiv: arXiv:1906.04493v3.

[54] Jürgen Schmidhuber. "Deep Learning in Neural Networks : An Overview." In: (2014), pp. 1–88. arXiv: arXiv:1404.7828.

[55] Valery Serov. *Fourier Series, Fourier Transform and Their Applications to Mathematical Physics*. 2017. ISBN: 978-3-319-65262-7. DOI: https://doi.org/10.1007/978-3-319-65262-7.

[56] William A. Sethares. *Tuning, Timbre, Spectrum, Scale*. 2005. URL: https://www.researchgate.net/publication/215646583.

[57] Claude E. Shannon. "Communication in the Presence of Noise." In: (1949).

[58] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. "STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET." In: (2015), pp. 1–14. arXiv: arXiv:1412.6806v3.

[59] Richard S. Sutton. In: (). URL: http://incompleteideas.net/papers/sutton-86.pdf.

[60] *The MIDI Association*. https://www.midi.org/about. [Online; accessed 14-July-2021].

[61] Sean Vasquez and Mike Lewis. "MelNet: A Generative Model for Audio in the Frequency Domain." In: Figure 1 (2019). arXiv: 1906.01083. URL: http://arxiv.org/abs/1906.01083.

[62] Max Welling and Diederik P. Kingma. "Auto-Encoding Variational Bayes." In: (2014). arXiv: arXiv:1312.6114v10.

[63] Max Welling and Diederik P. Kingma. "An Introduction to Variational Autoencoders." In: (2019). arXiv: arXiv:1906.02691v3.

[64] Paul Werbos and National Science Foundation. "Backpropagation through time : what it does and how to do it." In: November 1990 (1990). DOI: 10.1109/5.58337.

[65] Li Chia Yang, Szu Yu Chou, and Yi Hsuan Yang. "Midinet: A convolutional generative adversarial network for symbolic-domain music generation." In: *Proceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017* (2017), pp. 324–331. arXiv: 1703.10847.