Kiel University of Applied Sciences

Bachelor-Thesis

A Data-Driven Approach to Wavetable-Synthesis

Niklas Wantrupp

Matr.-No.: 928817
Course of Studies: Information Technology

supervised by
Prof. Dr. Robert Manzke and Prof. Dr.-Ing. Gunnar Eisenberg

August 20, 2019

Abstract

The generation of raw audio, using generative deep learning models, is a field which recently received a lot of attention with the proposal of WaveNET or WaveGAN. Both models use different deep learning architectures and are thus capable of generating novel sounds from a learned prior distribution. The systems are mainly adopted in text-to-speech environments.

This thesis, however, concentrates on the generation of raw audio for creative domains like sound design. Therefore TableGAN is presented, a generative model which is capable of synthesizing single-cycle waveforms from a learned prior distribution with the ability to interpolate smoothly between different points within the latent space. Furthermore, a wavetable oscillator framework is proposed which is capable of playing back the created wavetables.

Contents

1 Introduction **3**

2 Technical Background **5**
2.1 Wavetable Synthesis 5
2.1.1 Related Synthesis Forms 6
2.1.2 Wavetable Creation 8
2.1.3 Wavetable Oscillators 17
2.1.4 The Reference Model Wavetable Synthesis Process . . 22
2.2 Data Driven Sound Generation 22
2.2.1 Deep Learning Background 24
2.2.2 Convolutional Neural Networks (CNNs) 27
2.2.3 Generative Adversarial Networks (GANs) 29
2.2.4 WaveGAN Architecture 33

3 The Data-Driven Wavetable Oscillator **39**
3.1 Audio File Preparation 40
3.2 The proposed TableGAN Model 41
3.3 Sampling the TableGAN Latent Space 45
3.3.1 Wavetable Post Processing 46
3.4 Proposed Oscillator 47

4 Evaluation **48**
4.1 TableGAN Training Process 49
4.2 Inception Score (IS) Measures 50
4.2.1 Analyzation Process 51
4.3 Comparison between Test-Data Inner-Class Similarities 54
4.4 Comparison between Model Class Distributions 56
4.5 Nearest Neighbor Comparison 58
4.6 Evaluation of Space Continuity 59

5 Results, Discussion and Limitations **62**

6 Conclusion and Future Directions **64**

List of Acronyms **66**

List of Figures **67**

List of Tables **69**

Bibliography **70**

Appendices **75**

A **76**

B **78**

C **80**

D **83**

E **86**

1. Introduction

Since 2016 intense research on applications for artificial intelligence has been done in Google's Magenta Project[1], analysing the creative processes of music creation. In 2017 Google's first approach of generating raw audio with the help of artificial intelligence, the Neural-Synthesizer (NSynth)[2], was introduced. A Synthesizer based on deep neural networks, which creates sounds with a data-driven approach in mind, based on the WaveNet-Algorithm (Oord et al. 2016).

The NSynth-Algorithm works by using an autoencoder network to find a representation of the sound in the domain of the network. A decoder network is used to back convert the encoded sound, aiming to maintain the original characteristics of the sound. Sound fusion is carried out in the transformed domain, resulting in a hybrid, sounding different than a classical mix (time-domain addition) of simultaneously played sounds (Oord et al. 2016).

Similar to WaveNET, WaveGAN was introduced by Donahue, McAuley, and Puckette (2018). It uses a GAN to learn a latent representation of the provided test data and is able to synthesize audio samples of a length of one second at a sampling rate of 16 kHz.

[1]Further information can be found at: Magenta
[2]Further information can be found at: NSynth

This thesis will examine the application and modification of the WaveGAN-Algorithm to approach sound generation in a classical manner. Classical wavetable oscillators are holding an unspecified amount of periodic waveform cycles which can be swept through to modify the harmonic content of the created signal. When users want to create their own wavetables from audio samples, synthesisers like *Xfer-Records Serum*[3] allow users to do so by choosing between different algorithms to produce their wavetables. These results however need some careful modification to ensure that unwanted artifacts are avoided when modulating the position of the wavetable.

By building a deep neural network based on the WaveGAN-Algorithm a wavetable synthesis framework is created, which produces single-cycle waveforms based on the given audio material. This audio material can be created from a quantity of audio samples to examine the impact on musical timbre, when mixing takes place in the compressed domain of the network. Furthermore, the interpolation process between different waveform-cycles in the wavetable should be tackled by the algorithm as well, producing smooth results when changing the wavetable position. In addition, the feasability of this approach should be examined with realtime modulation of the wavetable position in mind.

Therefore TableGAN (presented in section 3.2) is proposed, a model for synthesizing single-cycle wavetables, accompanied by a classical FFT-based synthesis process from audio files (presented in section 2.1.4) and a wavetable oscillator framework to playback created wavetables (presented in section 3.4). All implementations can be found on the accompanying USB-Stick [4].

[3]Further information can be found at: Xfer-Records Serum
[4]The Contents of the USB-Stick are shown in Appendix E

2. Technical Background

2.1 Wavetable Synthesis

Wavetable synthesis arose from the need for a synthesis method that can be employed in low power hardware to synthesize sounds of arbitrary pitch and harmonical content. Due to the unavailability of hardware floating-point units (FPU's), calculations of trigonometric functions were implemented using software approximations. Furthermore wavetable oscillators (also called table lookup oscillators) were used, because the calculation of those higher-level math functions was too expensive for realtime synthesis (Boulanger and Lazzarini 2011).

Precalculating a cycle of a waveform and storing it into a lookup table of fixed size made it possible to cycle through that table at different speeds to generate a steady signal of a particular pitch. Wavetable synthesis, in general, should not be confused with PCM sample buffer playback, where an entire PCM file, saved in memory, can be played back at different rates (Bristow-Johnson 1996).

Even though FPUs implemented in hardware and computer systems are much faster nowadays, wavetable synthesis still is intensively used in modern synthesizers. A reason for that is the significant advantage over other sound

generation methods. I.e., the method does not only offer the possibility to mimic classic waveform shapes like saw, square, triangle, or sine, but it is also allows to produce more complex, harmonically rich waveforms (Boulanger and Lazzarini 2011).

Modern synthesizers allow users to load their sounds into the oscillator. A pitch detection algorithm tries to extract single-cycle waveforms into different frames, which reside in memory. Those frames can be exchanged dynamically in realtime, producing a change in harmonical content as the sound evolves (Bristow-Johnson 1996).

Other synthesis techniques like additive synthesis are also able to produce harmonically rich sounds that can change their harmonic content over time, but the process of stacking sine waves at different frequencies in realtime is far more computationally intensive than playback and looping of a wavetable (Franck and Välimäki 2012).

There are two main approaches to implement wavetable synthesis. The first approach, called wavetable cross-fading, which this thesis will focus on, aims at playing back two different wavetables at one time and to linearly interpolate their values. In contrast the second approach, called multiple wavetable synthesis, focuses on mixing different wavetables, each one having a temporal envelope function (Franck and Välimäki 2012).

2.1.1 Related Synthesis Forms

Before describing the technical details of wavetable synthesis, closely related methods are being discussed to demarcate those from one another.

Sample and Synthesis (S&S)

S&S describes synthesis forms that use a sample playback oscillator as a raw sound source. In contrast to wavetable synthesis, S&S loads a whole PCM file into the memory. Although typically implemented in digital form, an S&S synthesizer also can be implemented using analog hardware, e.g. the mellotron, which used magnetic tape for sample storage (Russ 2009).

For playback, samples usually are replayed once, instead of looping a single cycle of the waveform, although this is also possible with modern instruments (Russ 2009). The variation of pitch can be achieved with up- and down-sampling methods, playing back the sound at different sampling rates. For this process, a steep reconstruction filter is needed to eliminate alias frequencies at half of the sampling rate (Russ 2009). Another method which is also incorporated for pitching the S&S oscillator is interpolation. This is presented in section 2.1.3.

Granular Synthesis

Granular synthesis is another form that can work with audio files. Contrary to wavetable synthesis, in granular synthesis small sections (usually 10-100 milliseconds) get extracted from a source file. These so called "grains" are mixed with other small grains, each with an envelope function, to make sure each grain starts and stops at zero to avoid discontinuities. At playback, these grains get reorganized, with control over the number of grains, their length, and their repetition rate over time. *Martin Russ* states, that: "In some ways, granular synthesis can be considered as the limiting case of wavetable synthesis, where the table of waveforms is swept very rapidly to give a constantly changing waveshape" (Russ 2009).

2.1.2 Wavetable Creation

The creation of wavetables from a given audio file is a nontrivial task. At first, a pitch detection algorithm has to be employed to find the right sample positions within the given buffer to cut out frames. Those frames need to be stretched out or compressed regarding a specific table size to create the wavetables.

Because the table can contain harmonics that are above Nyquist (Shannon 1949), it should be bandlimited, or algorithms for suppressing aliasing artifacts need to be applied. When band-limiting a table to the maximum number of possible harmonics it should be noted that scanning through the table at a frequency higher than the root frequency of the table creates alias frequencies (Boulanger and Lazzarini 2011). Alias frequencies can be avoided through the creation of bandlimited tables for specified frequency ranges. All steps in creating wavetables from arbitrary audio files will be introduced in the following sections.

Pitch Detection

The task of pitch detection is the first important step in creating wavetables, each containing a single cycle of an input audio file. For pitch detection, time and frequency domain approaches can be used to detect the pitch at a given time t_0. In Zölzer (2011), a good selection of available algorithms are shown. Here, the average magnitude difference function (AMDF) and the YIN-algorithm are presented, which both represent time-domain approaches.

AMDF The average magnitude difference function, first presented by Ross et al. (1974) is a correlation function recommended by Bristow-Johnson (1996) for the task of pitch detection for wavetable creation. The AMDF

is defined by

$$D[\tau] = \frac{1}{N} \sum_{n=0}^{N-1} |x[n] - x[n+\tau]|, \tag{2.1}$$

as introduced by Ross et al. (1974). x[n] is a sample sequence which is multiplied with a rectangular window. τ defines the lag number, which is between 0 and $N-1$.

Equation (2.1) should show a minimum at the lag number of the period T of a given periodic signal and further minimum peaks with less amplitude at period multiples (Muhammad 2011). The frame size N of the rectangular window must be chosen carefully, as no pitches can be detected which period would exceed the frame length (Bristow-Johnson 1996). The pitch period can be estimated by

$$T = \text{argmin}_{\tau=\tau_{min}}^{\tau_{max}} D[\tau], \tag{2.2}$$

as proposed in Ross et al. (1974).

Because the AMDF starts with value zero at lag zero and is often nonzero at the period due to imperfect periodicity, a lower limit has to be set on the search range, as the algorithm would choose lag zero as the period (Cheveigné and Kawahara 2002). When the input signal is noisy, AMDF has problems in estimating the right pitch. In those cases, $2T$ or $\frac{T}{2}$ are often predicted as pitch, which is known as "double-pitch error" or "half-pitch error" respectively(Muhammad 2011).

In Figure 2.1 (a) the performance of the AMDF algorithm under noisy conditions can be observed. Because the input signal has a significant proportion of noise, the AMDF has difficulties in estimating the right pitch. Caused by the falling trend of the AMDF in the latter half, a completely wrong period was estimated, as figure 2.1 (a) shows. The sampling rate of the input is $f_s = 44100Hz$. The estimated period length in samples T_s by

AMDF is 1023, so the estimated frequency f_0 is $f_0 = \frac{f_s}{T_s} = \frac{44100Hz}{1023} \approx 43.1Hz$.

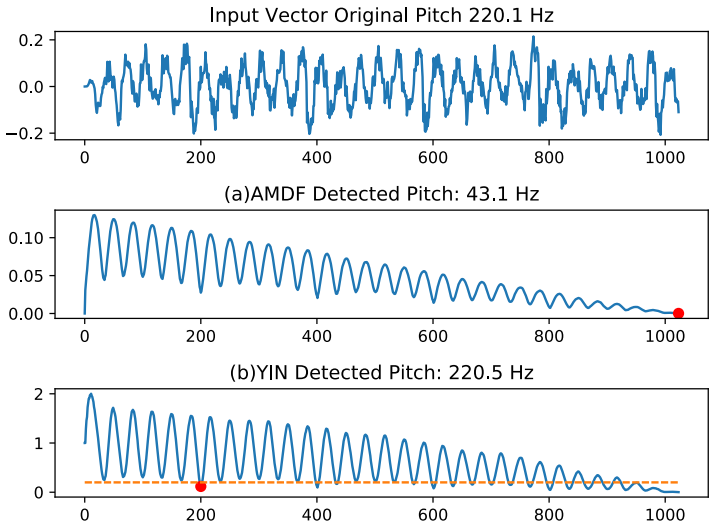

Figure 2.1: Comparison of (a) AMDF and (b) YIN Pitch Detection Performance

YIN To overcome those shortcomings Cheveigné and Kawahara (2002) propose the YIN pitch detection. YIN is calculated in 5 steps. The first step is to calculate the lag values:

$$D_e[\tau] = \sum_{n=0}^{N-1} (x[n] - x[n+\tau])^2, \tag{2.3}$$

as proposed by Cheveigné and Kawahara (2002). Equation (2.3) squares the difference term, instead of taking the absolute value as equation (2.1) does.

To avoid setting a lower limit for the search range, and to be able to define a threshold where a local minimum has to fall below, in order considering it as a valid pitch candidate, the YIN-algorithm is normalized as proposed by Cheveigné and Kawahara (2002). The resulting "cumulative mean normalized

difference function" is defined as:

$$D'_e[\tau] = \begin{cases} 1 & \tau = 0 \\ \frac{D_e[\tau]}{\frac{1}{\tau}\sum_{n=1}^{\tau} D_e[n]}, & else, \end{cases} \tag{2.4}$$

presented by Cheveigné and Kawahara (2002). The "cumulative mean normalized difference function" starts at 1 and drops just below 1, where the lag value $D_e(\tau)$ falls below the average (Cheveigné and Kawahara 2002).

To minimize the "octave error" phenomenon, an absolute threshold is defined, where a lag value has to fall below to be considered a valid pitch candidate. When none is found, the global minimum is chosen instead. To also consider periods which are non-integer multiples of the sampling rate, the local minima of $D'_e[\tau]$ are interpolated with their direct neighbor values parabolically to select the period estimate (Cheveigné and Kawahara 2002). Figure 2.1 (b) shows the pitch detection using the YIN-algorithm. The dashed orange line displays the threshold a pitch candidate has to fall below. YIN detects the right pitch with a deviation of $\sim 0.4Hz$ in this specific case.

The applied pitch detection used in the reference model is YIN, where the input file is being analyzed in frames of $L = 2048$ samples at a sampling rate of $f_s = 44100Hz$. This allows to detect frequencies down to $f_{min} = \frac{f_s}{L} \approx 21.5Hz$. Because the bottom of the hearing range of a human is at approximately $20Hz$, the frame length of $L = 2048$ will be sufficient. The YIN-algorithm will analyze one frame at a time and will proceed at a hop size which is dependent on the detected period length, so adjacent wavetables include the corresponding cycles of the input file.

Prevention of Aliasing Components

Because the cycle found by the pitch detection algorithm will be played back at different speeds resulting in pitches that could be above Nyquist, care must be taken to avoid the emergence of alias frequencies.

One possible method presented in Franck and Välimäki (2012) is to integrate the wavetable multiple times before storing it into memory, and when playing it back, the signal is differentiated as many times as it was integrated. This technique leads to a significant reduction in amplitude of the alias frequencies (Franck and Välimäki 2012).

The technique which is incorporated in the presented reference model is another bandlimited approach. Within this approach a stack of tables is created for every wavetable[5]. Every table within this stack is a representation of the input wavetable with a different amount of harmonics which can be played back within a specific frequency range.

Determine Maximum Number of Harmonics Foremost the maximum number of harmonics have to be specified and with this the spacing of adjacent wavetables. The spacing is important as it determines the amount of aliasing which could occur at playback. The maximum number of harmonics H_{max} from a specified frequency f_0 up to Nyquist can be calculated by:

$$H_{max} = \left\lfloor \frac{\frac{f_s}{2}}{f_0} \right\rfloor, \tag{2.5}$$

as shown by Redmon (2012). Given a sampling rate of $f_s = 44100Hz$ and a frequency of $f_0 = 20Hz$ the resulting maximum number of Harmonics would be: $H_{max} = \left\lfloor \frac{\frac{44100}{2}}{20} \right\rfloor = 1102$. When using a table spacing of an octave, which

[5]referred to as wavetable stacks

results in a doubling in frequency per table, the first table would span from $20Hz - 40Hz$. This would result in creating a fair amount of harmonics above Nyquist, as $40Hz \cdot 1102 = 44080Hz$ would be the highest harmonic created.

The first step to meet the Nyquist criteria would be to consider choosing a different table spacing. The reference model uses a table spacing of a minor third. Because an octave is a doubling in frequency and consists of 12 semitones, the ratio r_1 of one semitone can be described as:

$$r_1 = \sqrt[12]{2} = 2^{\frac{1}{12}} \tag{2.6}$$

With equation (2.6) the table spacing of a minor third would result in: $r_3 = 2^{\frac{3}{12}} = 2^{\frac{1}{4}} \approx 1.1892$. When equation (2.5) takes that spacing into account it can be rewritten as follows:

$$H'_{max} = \left\lfloor \frac{\frac{f_s}{2}}{f_0 \cdot r_s} \right\rfloor \tag{2.7}$$

The subscript s determines the number of semitones and r_s the table spacing ratio which can be calculated with $r_s = 2^{\frac{s}{12}}$.

Using equation (2.7) with a minor third would result in $H'_{max} = 927$ harmonics for the first table. The first table would span form $f_0 = 20Hz$ to $f_1 = 20Hz \cdot r_3 \approx 23.78Hz$ with the highest harmonic of f_0 at $h_{0,max} = 18540Hz$ and f_1 at $h_{1,max} = 22048Hz$. With this approach no aliasing is accepted with the trade-off, that for f_0 the highest harmonic is at $18540Hz$ which is below the threshold of $20000Hz$.

Bandlimiting Process With the base wavetable constructed the bandlimiting process can be applied. For that process the given input vector $x_{wt}[k]$

has to be transformed into the frequency domain using the DFT resulting in $X_{wt}[n] = DFT\{x_{wt}[k]\}$. Using H'_{max} all harmonics $h_n > H'_{max}$ are set to zero, to bandlimit the given wavetable. Because the complex DFT is used, the corresponding negative frequencies also have to be set to zero (Redmon 2012). To get rid of possible DC offsets the first frequency bin should be set to zero resulting in $X_{wt}[0] = 0$ (Redmon 2012). Next, the $IDFT$ of that signal is taken to transform it back into the time domain. Because the input signal is real-valued the real part of the $IDFT$ is taken to get the resulting wavetable $x'_{wt}[k] = Re\{IDFT\{X_{wt}[n]\}\}$, which then is normalized to $[-1, 1]$ (Redmon 2012).

This process needs to be repeated until $H'_{max} = 1$, decreasing the number of harmonics by $H'_{max} = H'_{max} \cdot \frac{1}{r_s}$ for every following table in the stack. The reference model creates objects for every table within that stack, holding the table itself, the table length and the highest frequency at which the table can be played back to avoid aliasing as proposed by Redmon (2012).

Figure 2.2 shows the first eight tables of a wavetable stack. It can be observed that every successive wavetable decreases in the number of harmonics, enabling it to be played back at a higher pitch without creating alias frequencies.

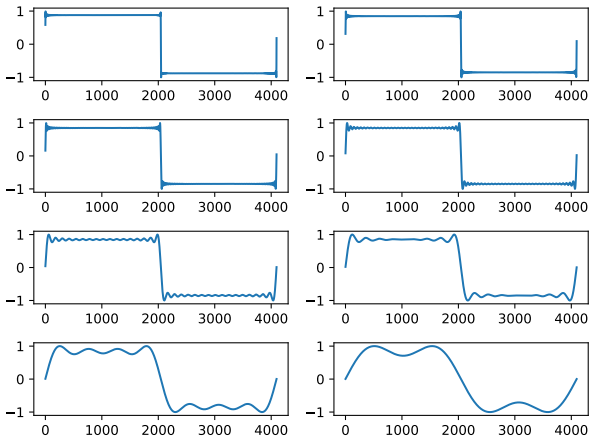

Figure 2.2: Representation of the 8 first wavetables of a square wave within a wavetable stack, with $r_s = 2$

Phase Alignment The subsequent wavetables which are created are not necessarily phase-aligned to each other. When automating the wavetable position and interpolating between two adjacent wavetables, null values could be created when not properly phase-aligned to each other (Bristow-Johnson 1996). To ensure a proper phase alignment between two adjacent tables, the cross-correlation is calculated:

$$\begin{aligned} r_{xy}[\tau] &= \frac{1}{N}\sum_{k=0}^{N-1}(\overline{x[k]} \cdot y[k+\tau]) \\ &= \frac{1}{N} IDFT_\tau\{\overline{X} \cdot Y\}, \quad \tau = 0, 1, 2...L-1, \end{aligned} \tag{2.8}$$

as shown by Smith (2010).

L depicts the length of the analysis frame and x and y are the input vectors in the time domain of two adjacent wavetables. The cross-correlation function is a measure of similarity vs. offset, where the maximum of the function represents the offset between the wavetables (Smith 2010). Following this, $d = argmax\, r_{xy}$ will give the delay of y in respect to x, where d is

the delay in samples. To align both tables y has to be delayed by d samples into positive or negative direction depending whether $d > \frac{L}{2}$.

Figure 2.3 shows the described process. In 2.3 (a) and (b) the signal vectors x and y are being displayed. 2.3 (c) shows the calculated cross-correlation vector with $d = 2048$ and 2.3 (d) shows the corrected y vector which was shifted by $d = 2048$ samples.

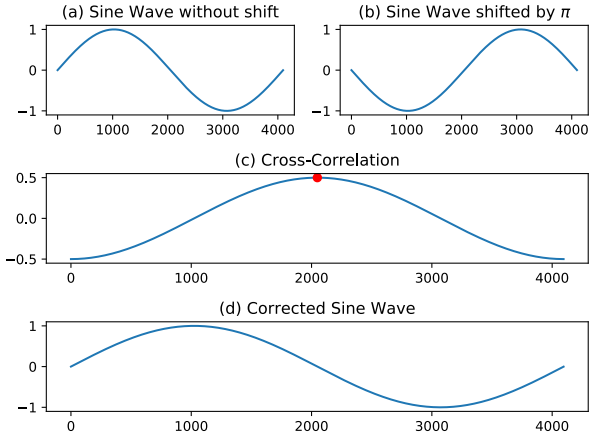

Figure 2.3: Cross-Correlation Process (a) Sine Wave without shift (b) Sine Wave shifted by π (c) Cross-Correlation (d) Corrected Sine Wave

Avoiding Discontinuities To avoid discontinuities at wavetable edges, Bristow-Johnson (1996) recommends multiplying the analyzed wavetable by a Hanning window. However, as this multiplication leads to a drastic alteration of the original waveform shape, another approach is taken. The proposed approach multiplies the wavetable with a modified *tanh*-window $s[k]$ of length $\frac{N}{2}$ which is calculated by:

$$s[k] = 0.5 + 0.5 \cdot tanh(\frac{x[k] - \frac{N}{2}}{b}) \tag{2.9}$$

Where s is the tanh-window, x the input x coordinate vector, $\frac{N}{2}$ the turning point of the *tanh*-function and b depicts the gradient.

To smooth the edges of the wavetable x_{wt}, the first $\frac{N}{2}$ samples are multiplied with s and the last $\frac{N}{2}$ samples with $1 - s$. Figure 2.4 (b) shows the resulting smoothed waveform when applied to a sawtooth wave. In contrast to 2.4 (a), where the sawtooth is heavily alterated by the hanning window, only the edges of the wavetable are modified.

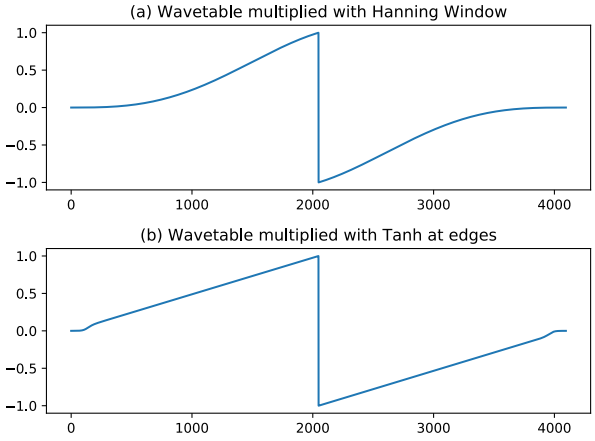

Figure 2.4: Comparison Hanning-Window Smoothing and tanh-smoothing (a) Sawtooth smoothed with Hanning (b) Sawtooth smoothed with *tanh*

2.1.3 Wavetable Oscillators

The Oscillator is responsible for playing back the wavetables residing in memory. Pitch shifting is done with sampling rate conversion techniques. With downsampling, the pitch is raised and with upsampling the pitch is lowered (Franck and Välimäki 2012). The challenges arising from those mechanisms are to avoid distortion, which occurs when the wavetable is played back at an increment unequal to one. To reduce this distortion, which is called truncation noise, interpolation algorithms are applied (Boulanger and Lazzarini

2011). The following sections introduce a trivial oscillator approach, which then is being modified to meet the requirements of modern music production systems.

The Trivial Wavetable Oscillator Approach

Appendix A shows a slightly modified Python implementation of the proposed wavetable oscillator by Boulanger and Lazzarini (2011). This trivial wavetable oscillator is not expecting a wavetable stack so it would produce aliasing when playing back wavetables at pitches other than the table pitch. The algorithm calculates the sample increment based on the normalized frequency and the length of the wavetable. This value is added to the current phase after every iteration to progress through the table.

Algorithm 1 Trivial Wavetable Oscillator

```
1:  curphase = 0
2:  incr = (f0/fs) * L
3:  while not at end of processing block do
4:      index = int(curphase)
5:      out[i] = table[index]
6:      curphase+ = incr
7:      while curphase ≥ L do
8:          curphase− = L
9:      end while
10:     while curphase < 0 do
11:         curphase+ = L
12:     end while
13: end while
```

Algorithm 1 shows the pseudocode for the implementation. Line 7 - 12 implement the wrap-around so that the phase wraps around when reaching the end of the table. A while loop is used to allow increments bigger than the table length (Boulanger and Lazzarini 2011). Line 5 describes the critical part of the trivial approach. The index which is read from the table is obtained

by truncating the fractional part of the phase position. Truncation leads to distortion and the problem that it is not possible to playback the wavetable at the exact desired frequency (Boulanger and Lazzarini 2011; Pirkle 2015). Boulanger and Lazzarini (2011) state to gain acceptable quality using the truncation algorithm a minimum table size of 8192 samples is needed.

Interpolation Methods for Wavetable Playback

To overcome the arising issues caused by truncating the fractional part of the phase position, different interpolation methods are available. Following Pirkle (2015), linear and third-order Lagrange interpolation methods are commonly applied to minimize the effect of truncation noise.

Linear Interpolation Linear interpolation uses the fractional part of the phase position to interpolate between the current integer position and the next. The current output value of y can be calculated with:

$$y = y_1 + (y_2 - y_1) \cdot (p - x_1), \tag{2.10}$$

as proposed by Boulanger and Lazzarini (2011). Where x_1 is the integer part and p the fractional part of the phase position, y_2 and y_1 denote the table values at the current and the next phase position. Because y_2 can be greater than the table length, a modulo operation has to be implemented to wrap around the value to the start of the table. Another method is to extend the table length by one and copy the first value to the last table position (Boulanger and Lazzarini 2011). Appendix B shows the implementation of the wavetable oscillator, using linear interpolation as suggested by Boulanger and Lazzarini (2011).

Third Order Lagrange Interpolation The third order lagrange interpolation $N = 3$ uses $N + 1 = 4$ points around the target value for interpolation (Pirkle 2015). Zölzer (2008) and Smith (2010) propose to calculate the output sample with:

$$y[x] = \sum_{n=0}^{N} l_n[x]y[k], \tag{2.11}$$

where x depicts the phase parameter and $l_n(x)$ is calculated with:

$$l_n[x] = \prod_{j=0, j\neq n}^{N} \frac{x - x_j}{x_n - x_j} \tag{2.12}$$

Appendix C shows a Python implementation of the third order lagrange interpolation method, inspired by the implementation in Pirkle (2015). When examining the source code, it should be noticed that the complexity of the lagrange algorithm is $O(n^2)$ on the contrary to the complexity of $O(n)$ of the linear interpolation algorithm.

Comparison of Interpolation Algotihms Figure 2.5 shows a comparison between the spectra of the different methods (a) truncation, (b) linear interpolation and (c) Lagrange interpolation for playback of a wavetable. The wavetable length is $L = 1024$ and the frequency is $f_0 = 441Hz$ at a sampling rate of $f_s = 44100Hz$. It can be observed that the truncation algorithm produces the most distortion, as the peaks in the frequency bins go up to $\sim -65dB_{FS}$. The highest peaks with linear interpolation are at $\sim -125dB_{Fs}$ and for the Lagrange interpolation there are barely any distortions above $-250dB_{Fs}$.

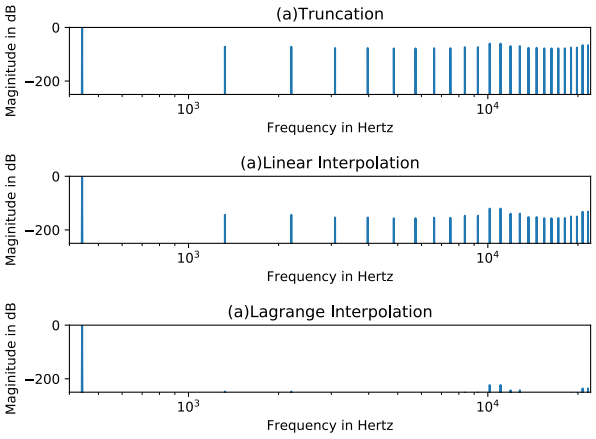

Figure 2.5: Comparison of Interpolation Algorithms with table length of $L = 1024$ (a) Truncation, (b) Linear Interpolation, (c) Lagrange Interpolation

Figure 2.6 shows the same comparison but with a table length of $L = 4096$. One can clearly see that the truncation algorithm has a high level of distortion frequencies, with peaks at about $\sim -77.5dB_{FS}$. The linear interpolation produces the highest peaks at $\sim -150dB_{Fs}$, whereas the Lagrange interpolation produces no distortion frequencies above $-250dB_{Fs}$. Following that observation, it can be said that, with the table containing more samples, the distortion is less obvious to the hearer. This meets with the statement made by Boulanger and Lazzarini: "To obtain acceptable quality, a table size of 8,192 points or greater is usually required, depending on the sampling rate" (Boulanger and Lazzarini 2011).

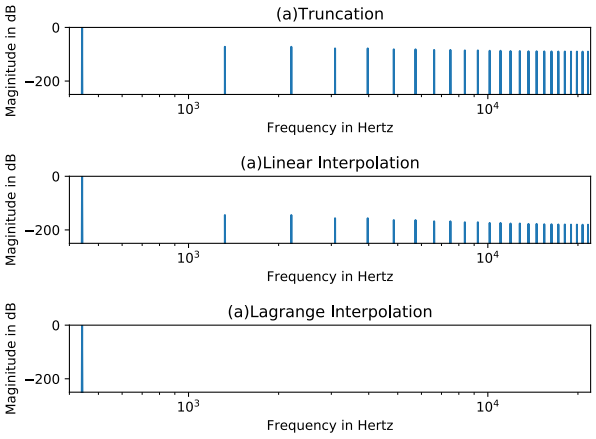

Figure 2.6: Comparison of Interpolation Algorithms with table length of $L = 4096$ (a) Truncation, (b) Linear Interpolation, (c) Lagrange Interpolation

2.1.4 The Reference Model Wavetable Synthesis Process

For the synthesis process, a reference model has been built to create wavetables from given audio files. Those wavetables are created utilizing the methods presented above. For playing back the created wavetables, an oscillator has been implemented, which is presented in chapter 3.4. Information on how to use the wavetable framework appendix D should be inspected.

2.2 Data Driven Sound Generation

Deep learning is a fast-evolving domain which is mostly used in the field of image and voice recognition or more recently translation (Briot, Hadjeres, and Pachet 2017).

It describes a subset of tools in the field of machine learning (ML), based on artificial neural networks, where deep means multiple layers which can

decompose a complex structure into simpler representations. General applications of deep learning are classification problems and predictions, but more recently, also the generation of content (Briot, Hadjeres, and Pachet 2017). As Briot, Hadjeres, and Pachet (2017) state, the success of this technology is based on three points:

- Availability of mass data,
- availability of efficient and affordable computing power,
- recent technical advances in the field of machine learning.

Given the recent success of generative models in image generation, researchers started to adopt similar techniques in the field of music generation. Those methods have been widely adopted with the fundamental difference that audio data is represented as a one-dimensional time series of audio samples, instead of a two-dimensional instantaneous image (Purwins et al. 2019). Applications of generative models in the audio domain could be amongst others:

- Generation of semantic information,
- extraction of information,
- classification of audio data,
- generation of raw audio.

This thesis focuses on the generation of raw audio, using a deep learning model. Current state-of-the-art techniques for generation of raw audio include SampleRNN, a model based on recurrent neural networks (RNN) (Mehri et al. 2017), NSynth, which is based on WaveNET and uses an autoregressive approach (Engel, Resnick, et al. 2017; Oord et al. 2016) and GANSynth and WaveGAN, both using a generative adversarial network (GAN) (Engel, Agrawal, et al. 2019; Donahue, McAuley, and Puckette 2018).

It is important to state that deep learning is not the only method for generative music creation. Methods like gaussian mixture models, hidden markov models, or non-negative matrix factorization are also used but often outperformed by deep learning models, when sufficient data is available as Briot, Hadjeres, and Pachet (2017) claim.

Modelling the complex structure of raw audio waveforms with generative models is a challenging task because as well as the local structure producing high fidelity audio, the longterm structure needs to be captured to generate globally consistent audio data (Vasquez and Lewis 2019).

2.2.1 Deep Learning Background

The core concept of deep learning are artificial neural networks (ANNs) which are ideal for solving complex ML tasks (Géron 2017). In the following section the fundamental concepts of deep learning are layed out. A more thorough introduction can be found in (I. Goodfellow, Bengio, and Courville 2016; Géron 2017).

The evolution of neural networks started in 1957 with the invention of the Perceptron by *Frank Rosenblatt*, which consists of one layer of threshold logic units (TLUs) (Géron 2017). Figure 2.7 shows a TLU. It consists of a neuron or unit which takes a weighted sum and a bias term that is always 1. After the calculation a step function is applied to the result. In most cases the heavyside function $H(x) = \begin{cases} 0 & x < 0 \\ 1, & x >= 0. \end{cases}$ is used (Géron 2017).

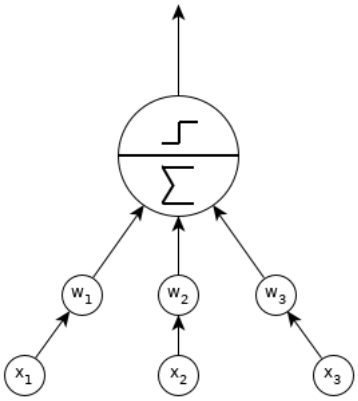

Figure 2.7: Threshold Logic Unit (Géron 2017)

A Perceptron is composed of a single layer containing 1 to n TLUs with each neuron connected to all inputs. Because the Perceptron fails to solve simple problems like calculating the logical XOR-function, Multi-Layer-Perceptrons (MLPs) were introduced.

MLPs overcome the shortcomings of Perceptrons, by stacking up more than one layer of Perceptrons on top of each other. MLPs have an input layer, which just passes the input through to the network. Then 1 to n hidden layers are added before one final layer called the output layer outputs the computed values. When $n >= 2$ the ANN is called a deep neural network (DNN) (Géron 2017).

The MLP approach also replaces the step function by the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ which is called the activation function of the neuron. It is trained using the backpropagation algorithm introduced by Rumelhart, Hinton, and Williams (1986). The algorithm makes a prediction for one training step which is called the forward pass. It measures the error, using a loss function, and then propagates the error back layer per layer through the network with every connection, measuring how much it contributes to the error and

finally tweaking the connection weights, using the gradient descent optimization algorithm (Géron 2017). Modern architectures of the MLP often use other optimization, activation, and loss functions, which are all optimized towards different use cases.

Gradient Descent The gradient descent algorithm is an optimization algorithm to optimize the weights of neuron connections. It does so by measuring the local gradient of the loss function with regards to a parameter vector θ and optimizes the parameters into the direction of the descending gradient with a defined step size called the learning rate. The gradient descent is finished when it converged to a minimum (Géron 2017).

Batch Normalization Batch normalization is a technique applied to reduce vanishing/exploding gradients while training the network. The vanishing gradient problem happens, when gradients are getting smaller with every lower layer of the network. This results in virtually unchanged weights, which ensure that the training process never converges to an optimal solution (Géron 2017).

The exploding gradient problem is the exact opposite of the vanishing process. The gradients are growing over time, resulting in large weight updates which lead to divergence of the algorithm (Géron 2017). To overcome the vanishing/exploding gradient problem, alternative activation functions can be used to reduce those problems at the beginning of the training significantly. However, those problems can come back during training.

The objective of the batch normalization algorithm is to prevent this by adding a batch normalization layer in front of each activation function to zero-center and normalize the inputs. This is achieved by using the mean

and standard deviation over the mini-batch[6] to zero-center and normalize the input. It then scales the input with a factor γ and shifts the output by the parameter β. These parameters are learnable for each batch normalization layer (Géron 2017).

2.2.2 Convolutional Neural Networks (CNNs)

CNN architectures are composites of convolutional layers, pooling layers, and a fully connected layer. The input of the network gets analyzed with a fixed number of filters with a specified kernel size, which describes the length of a filter. Those filters are convoluted over the input sequence with a given step size, the stride. When reaching the edge of the input, the zero-padding value determines how the filter behaves at edges (Briot, Hadjeres, and Pachet 2017).

Depending on the chosen architecture, it is a common practice to have more than one convolutional layer. After the convolution step, the results are passed to a pooling layer, which passes the strongest signal in the frame on to the next step. Max pooling, for example, passes the highest value on to the next layer (Briot, Hadjeres, and Pachet 2017). Pooling serves for abstracting the input and just passing the most relevant signals on to the next layer, which is the fully connected layer.

The fully connected layer represents a standard feedforward network, in which normally a softmax activation function for classification problems triggers the output of the network (Briot, Hadjeres, and Pachet 2017).

[6] mini-batch or batch: training data is split into mini-batches of a given batch size

Receptive Fields of Convolutional Neural Networks

Unlike fully connected layers, convolutional units depend on a region of the input feature map. Just this specified region will trigger the weights of the corresponding unit (Luo et al. 2016). This region is called the receptive field of the convolutional unit.

Central values in the input feature map have more impact on the output, as they have multiple ways of propagating through the network in contrast to outer values. To increase the size of the receptive field, convolutional layers can be stacked, which grow the receptive field linearly by the kernel length. Another approach is to use subsampling. This increases size multiplicatively (Luo et al. 2016). To calculate the size of the receptive field of each layer the following equations are used:

$$\begin{gathered} n_{out} = \lfloor \frac{n_{in} + 2p - k}{s} \rfloor + 1 \\ n_{in} : number\ of\ input\ samples \\ n_{out} : number\ of\ output\ samples \\ k : convolution\ kernel\ size \\ p : convolution\ padding\ size \\ s : convolution\ stride\ size \end{gathered} \tag{2.13}$$

$$\begin{gathered} j_{out} = j_{in} \cdot s \\ r_{out} = r_{in} + (k - 1) \\ j : distance\ between\ two\ adjacent\ features \\ r : receptive\ field\ size, \end{gathered} \tag{2.14}$$

as proposed by Hien (2017).

Calculating and planning the receptive field size while designing the

model, has a significant impact on the results. Especially in audio aplications a sizeable receptive field is needed to catch longterm structures and dependencies in audio data (Oord et al. 2016; Donahue, McAuley, and Puckette 2018).

2.2.3 Generative Adversarial Networks (GANs)

Generative adversarial networks were introduced by (I. J. Goodfellow et al. 2014). They consist of a discriminator (D) and a generator (G), where D is fed by real data or a generated sample from G in an alternating manner. Gs objective is to generate a synthetic sample from an input noise vector (I. J. Goodfellow et al. 2014). D classifies the input to the probability that the data is either real or synthetic. This adversarial process can be described as a minimax two-player game with:

$$\min_{\mathrm{G}}\max_{\mathrm{D}} V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))], \quad (2.15)$$

as introduced by I. J. Goodfellow et al. (2014).

In equation 2.15 p_{data} describes the distribution of the training data and p_z describes the noise distribution. The objective of the adversarial net is to learn Gs distribution p_g over the data x (I. J. Goodfellow et al. 2014). During training D is trained to maximize the probability of assigning the right labels to the given input samples. G is trained to minimize the term $\log(1 - D(G(z)))$ (I. J. Goodfellow et al. 2014). In practice equation 2.15 might not produce sufficient gradients for G to learn well because D can reject samples from G with high conviction in the beginning of the training phase, as the samples from G are clearly different from the training data (I. J. Goodfellow et al. 2014). To overcome this, the objective of G is to

maximize $\log(D(G(z)))$ instead of $\log(1-(D(G(z))))$. This provides G with much stronger gradients in the beginning of the training, resulting in:

$$\min_{\mathrm{G}} \max_{\mathrm{D}} V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[-\log(D(G(z)))], \quad (2.16)$$

as introduced by I. J. Goodfellow et al. (2014). To make sure D is maintained near the optimum, it is trained k times per training iteration of G. This is because training D and G iteratively in the inner training loop leads to overfitting of D and is inefficient (I. J. Goodfellow et al. 2014).

The training of the adversarial network is finished when D is not able to tell whether a sample either comes from the training set or is generated by G. G is then able to produce synthetic samples, which model the training data (Briot, Hadjeres, and Pachet 2017).

GAN Optimization

GAN architectures based solely on the proposed model by I. J. Goodfellow et al. (2014) are highly unstable in training. The unstability occurs because the architecture suffers from the following problems:

- **mode collapse:** G produces samples with a small diversity
- **diminishing gradients:** D is getting too good at the classification of real/fake data, resulting in diminishing gradients of G and no or slow learning of G taking place
- **oscillating loss:** model parameters destabilize and oscillate

, as laid out by Hui (2018a) and Foster (2019).

Mode collapse is a problem, where generated samples converge to the same sample. This is known as full mode collapse. Full mode collapse is

not a common problem. Instead, it often happens that the mode partially collapses, meaning that a range of similar samples are produced (Hui 2018a).

The GAN architecture relies on gradient descent as an optimization algorithm. With two networks competing against each other, the learning process can be very slow in the beginning. Since D can tell the difference between real and fake easily, the gradients for G are diminishing, which slows down the learning process (Hui 2018a).

A GAN converges when it finds the Nash equilibrium of equation 2.15 (I. J. Goodfellow et al. 2014). Convergence is achieved when one side does not want to change its action regardless of what the opponent might do. Hui (2018c) shows that using gradient descent cost functions may never converge in a minimax-game.

Those instabilities of the original architecture proposed by I. J. Goodfellow et al. (2014) have been theoretically analyzed by Arjovsky and Bottou (2017). Arjovsky and Bottou (2017) showed that the original loss function, shown in equation 2.15, suffers from diminishing gradients and also the alternative cost function in equation 2.16 has problems with fluctuating gradients, resulting in poor quality samples. This led researchers to trying to find new loss functions that can solve those problems. The loss function which is presented here is the Wasserstein GAN gradient penalty (WGAN-GP) algorithm proposed by Gulrajani et al. (2017), which is used both in WaveGAN and in the proposed TableGAN architectures.

WGAN-GP uses the Wasserstein loss function instead of binary cross-entropy. This requires the training data to be labelled $y = 1$ for real and $y = -1$ for fake instead of $y = 1$ and $y = 0$. Additionally, removing Ds last layers activation function leads to output predictions in the range of $[-\inf, \inf]$. Because of that, D is designated as critic in the context of WGAN-GP. Ds

loss is calculated via comparing the predictions for a real image $p_r = D(x)$ to the response $y = 1$ and predictions for the generated images $p_g = D(G(z))$ to the response $y = -1$ (Foster 2019). This results in the following loss function for D:

$$\min_{\mathrm{D}} - (\mathbb{E}_{x \sim p_x}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))]), \tag{2.17}$$

as presented by Foster (2019). For G the loss is calculated by comparing the predicitons for generated images $p_g = D(G(z))$ to the response $y = 1$. Therefore, the WGAN-GP loss function for G results in:

$$\min_{\mathrm{G}} - (\mathbb{E}_{z \sim p_z}[D(G(z))]), \tag{2.18}$$

as shown by Foster (2019).

Because the WGAN-GP algorithm allows the output predictions to grow very large, which usually should be avoided in neural networks, an additional constraint has to be placed on D. The additional requirement is that the critic is a *1-Lipschitz continuous function*. D satisfies this requirement when it follows the equation with x_1 and x_2 being two subsequent samples:

$$\frac{|D(x_1) - D(x_2)|}{|x_1 - x_2|} \leq 1, \tag{2.19}$$

as shown by Foster (2019).

This equation requires a limit on the rate at which the predictions of two subsequent samples can change. This requirement is enforced by the gradient penalty algorithm, which includes a term in Ds loss function which penalizes the model if the gradient norm of the critic derivates from 1 (Foster 2019).

Figure 2.8 shows the algorithm to calculate the WGAN-GP algorithm, which uses Adam with shown parameters as the optimization algorithm. The

first step is to iterate k times through Ds training process, where in line 7, the gradient penalty is added to the original D loss per batch sample. In line 8 the weights of D are updated with the Adam optimizer and the calculated losses of a given batch. After k iterations Gs weights are updated in line 12 of the algorithm.

The advantage of WGAN-GP over other loss functions is the ability to converge. This allows the training process to become more stable and allows for a higher generator model complexity (Hui 2018b).

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
1: **while** θ has not converged **do**
2: **for** $t = 1, ..., n_{\text{critic}}$ **do**
3: **for** $i = 1, ..., m$ **do**
4: Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
5: $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
6: $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
7: $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
8: **end for**
9: $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
10: **end for**
11: Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
12: $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
13: **end while**

Figure 2.8: WGAN Gradient Penalty Algorithm taken from (Gulrajani et al. 2017)

2.2.4 WaveGAN Architecture

The WaveGAN architecture is capable of modeling raw audio waveforms and producing up to one second of audio at a sampling rate of $f_s = 16000Hz$ (Donahue, McAuley, and Puckette 2018). It is based on the DCGAN-architecture presented in Radford, Metz, and Chintala (2015), which uses convolutional networks in both the G and D.

DCGAN can generate images from a learned latent space; in contrary

WaveGAN's objective is to produce raw audio data. Because audio data is more likely to show periodicity than image data, the filters used in the convolutional network need a higher receptive field than those used for image generation. To achieve this, WaveGANs G uses a transposed convolution operation, with a filter length of 25 and an upsampling factor of 4. D also uses a filter length of 25 and a stride of 4 to mirror the generator (Donahue, McAuley, and Puckette 2018).

Figure 2.9 shows the growth of the receptive field size within the WaveGAN architecture. Following results can be seen: The deeper the layer is located in the network, the larger its receptive field grows. Hence, it could be said that the deeper layers capture longterm structure in audio data, while the first layers capture short term structure.

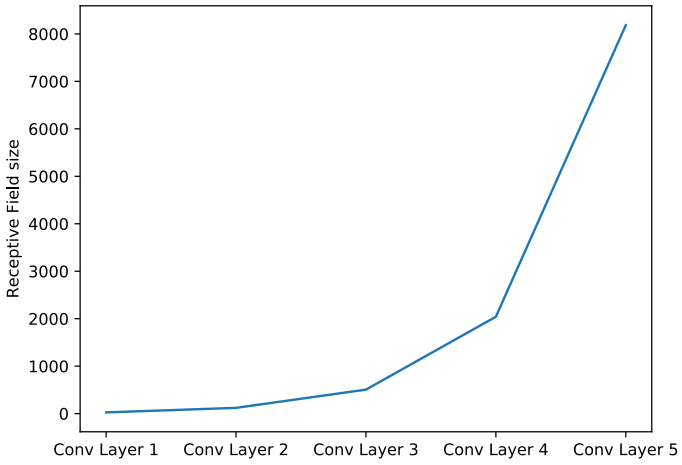

Figure 2.9: Growth of receptive field size in WaveGAN

Data encoding is 32-Bit float in G as well as in D. This leads to WaveGAN having a higher headroom than WaveNET, which is using 8-Bit μ-law encoding (Oord et al. 2016). Using generative adversarial networks also has

the benefit, that sound generation can be parallelized, which allows the generation of up to 1 hour of audio in less than 2 seconds (Donahue, McAuley, and Puckette 2018). The audio generation in WaveNET has to feedback output audio data back into the input, which is much slower than WaveGANs approach (Donahue, McAuley, and Puckette 2018; Oord et al. 2016).

Phase Shuffling

Generative models, which upsample by transposed convolution, are known to produce checkerboard artifacts, as shown in figure 2.10.

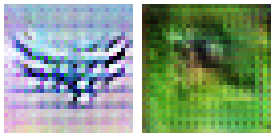

Figure 2.10: Checkerboard Artifacts in generated images taken from (Odena, Dumoulin, and Olah 2016)

Such periodic patterns are less common in images, so that the discriminator learns to reject them. In the field of audio signal processing, analogous artifacts are perceived as pitched noise (Donahue, McAuley, and Puckette 2018). Those artifacts can overlap with frequencies from real data occurring at specific phase positions, allowing D to reject those patterns, which can restrain the optimization problem (Donahue, McAuley, and Puckette 2018).

To prevent D from learning those patterns, phase shuffling with hyperparameter n is applied to the feature maps, which modifies the phase from $-n$ to n before inputting it to the next layer of D. As G is already fed with a uniformly distributed noise vector, this process just needs to be applied in D (Donahue, McAuley, and Puckette 2018). This makes Ds job to distinguish between real and fake harder.

Figure 2.11 shows the process of phase shuffling with $n = 1$. This process is applied in every layer of the discriminator to perturb the phase of the feature maps. n is sampled from a uniform distribution, and the missing samples after shifting are filled via reflection.

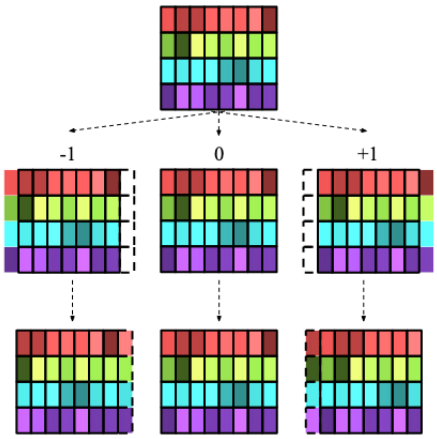

Figure 2.11: Phase Shuffling Process taken from (Donahue, McAuley, and Puckette 2018)

Model Architecture

Table 2.1 and 2.2 show the architectures of WaveGANs G and D respectively, where b denotes the batch size, and c denotes the number of audio channels. For the training of WaveGAN b was set to 64 and c to 1 (Donahue, McAuley, and Puckette 2018). In table 2.1 it is shown how an output tensor with shape $(n, 16384, c)$ is crafted out of the input noise vector with a latent size of 100. The upsampling takes place in the transposed convolution layers, where stride corresponds to the upsampling factor.

Operation	Input Shape	Output Shape
Input $z \sim Uniform(-1, 1)$	(b, 100)	(b, 100)
Dense	(b, 100)	(b, 16384)
Reshape	(b, 16384)	(b, 16, 1024)
ReLU	(b, 16, 1024)	(b, 16, 1024)
TransConv1D(Stride=4)	(b, 16, 1024)	(b, 64, 512)
ReLU	(b, 64, 512)	(b, 64, 512)
TransConv1D(Stride=4)	(b, 16, 1024)	(b, 256, 256)
ReLU	(b, 256, 256)	(b, 256, 256)
TransConv1D(Stride=4)	(b, 256, 256)	(b, 1024, 128)
ReLU	(b, 1024, 128)	(b, 1024, 128)
TransConv1D(Stride=4)	(b, 1024, 128)	(b, 4096, 64)
ReLU	(b, 4096, 64)	(b, 4096, 64)
TransConv1D(Stride=4)	(b, 4096, 64)	(b, 16384, c)
Tanh	(b, 16384, c)	(b, 16384, c)

Table 2.1: WaveGAN Generator Architecture

Table 2.2 shows WaveGANs D architecture. It can be observed that the input and output shapes of the transposed convolution of G and convolution operations of D are reversely mirrored. With this approach, D can classify the given input samples real or fake.

The optimizer used for WaveGAN is Adam with $\alpha = 0.0001$, which describes the learning rate and $\beta_1 = 0.5$ and $\beta_2 = 0.9$. The used loss algorithm is the presented WGAN-GP, while the number k denotes the discriminator updates per generator update, which is $k = 5$. The hyperparameter n of the phase shuffling algorithm is set to $n = 2$ (Donahue, McAuley, and Puckette 2018).

Operation	Input Shape	Output Shape
Input x or G(z)	(b, 16384, c)	(b, 16384, c)
Conv1D(Stride=4)	(b, 16384, c)	(b, 4096, 64)
LReLU($\alpha = 0.2$)	(b, 4096, 64)	(b, 4096, 64))
PhaseShuffle($n = 2$)	(b, 4096, 64)	(b, 4096, 64))
Conv1D(Stride=4)	(b, 4096, 64)	(b, 1024, 128)
LReLU($\alpha = 0.22$)	(b, 1024, 128)	(b, 1024, 128)
PhaseShuffle($n = 2$)	(b, 1024, 128)	(b, 1024, 128)
Conv1D(Stride=4)	(b, 1024, 128)	(b, 256, 256)
LReLU($\alpha = 0.2$)	(b, 256, 256)	(b, 256, 256)
PhaseShuffle($n = 2$)	(b, 256, 256)	(b, 256, 256)
Conv1D(Stride=4)	(b, 256, 256)	(b, 64, 512)
LReLU($\alpha = 0.2$)	(b, 64, 512)	(b, 64, 512)
PhaseShuffle($n = 2$)	(b, 64, 512)	(b, 64, 512)
Conv1D(Stride=4)	(b, 64, 512)	(b, 16, 1024)
LReLU($\alpha = 0.2$)	(b, 16, 1024)	(b, 16, 1024)
Reshape	(b, 16, 1024)	(b, 16384)
Dense	(b, 16384)	(b, 1)

Table 2.2: WaveGAN Discriminator Architecture

3. The Data-Driven Wavetable Oscillator

The goal of this thesis is to present a deep learning model that can produce single-cycle waveforms which can be interpolated by the model. The generated wavetables are then prepared for playback through a wavetable oscillator which can automate the wavetable position of a given wavetable set produced by the model. Hereby, one aim is to produce smooth interpolations between wavetables with the help of the generative model. Furthermore, another aim is to enable playback automations of the wavetable position without audible artifacts.

The deep learning model is based on WaveGAN (Donahue, McAuley, and Puckette 2018) and DCGAN (Radford, Metz, and Chintala 2015) because GAN architectures are able to generate content in parallel, in contrast to autoregressive models like WaveNET (Oord et al. 2016) or recurrent neural networks like SampleRNN (Mehri et al. 2017). Other GAN architectures like SpecGAN, which are based on waveform generation through spectrogram images, are not promising for the use case of wavetable generation because of their lossy conversion strategy back to raw audio using griffin-lim (Donahue, McAuley, and Puckette 2018).

The proposed TableGAN model can generate n interpolations between two chosen wavetables. The generation process is part of the following presented wavetable oscillator ecosystem consisting of five different steps:

- Preparation of audio files,
- training of proposed TableGAN model for wavetable generation,
- sampling and interpolating in the TableGAN latent space,
- post-processing resulting wavetables,
- and playing back wavetables via the proposed oscillator.

All implementations are using *Python 3* as language and *keras* with *tensorflow*-backend as deep learning framework. The single programs which are providing the functionality and a trained model on an example dataset are provided via the accompanying USB-Stick and their usage is explained in appendix D.

3.1 Audio File Preparation

The preparation of audio files is needed to train the model with single-cycle waveforms. For the preparation a selection of files gets analyzed with the proposed YIN-algorithm (section 2.1.2). Then the files are sliced, given the periods estimated by YIN. As a post-processing step the resulting wavetables get smoothed at the edges to avoid discontinuities using the proposed tanh-smoothing (section 2.1.2).

When the slicing and post-processing steps for every file are finished, it is necessary to ensure that all files hold the same number of created wavetables. If the files contain a different amount of tables, the file with the least number

of tables defines the maximum amount of tables per file. All remaining tables within the files will be discarded. This is necessary because every file represents a class in the context of a deep learning problem. When creating batches for the training process, this approach leads to a uniformly distributed set of training samples.

3.2 The proposed TableGAN Model

TableGAN is a deep learning model based on both WaveGAN and DCGAN. The objective of the trained G is to be able to generate sounds from its learned distribution p_g, which should mimic the real data distribution p_r. The learning process takes place in an adversarial minimax-game, where D and G are trying to maximize their opponents loss function while minimizing their own.

Because TableGAN is supposed to produce single-cycle waveforms of length $L_T = 4096$ instead of $L_W = 16384$, which is the length of WaveGANs generated samples, it can leave out one layer of transposed convolution in G and accordingly one layer of convolution in D.

This is also noticeable in figure 3.1, where the growth of the receptive field is shown. It can be observed, that the last layer of convolution has a receptive field of $r_{4,TG} = 2041$ which is approximately half of the table length $\frac{L_T}{2} = 2048$. This length is sufficient to observe longterm structure in the audio data, like in WaveGAN, which has a receptive field size of $r_{5,WG} = 8185$ at a length of $L_{WG} = 16384$. The exponential growth of receptive field size is achieved through both: stacking four convolutional layers and through using subsampling with a stride $s = 4$ like in WaveGAN (Donahue, McAuley, and Puckette 2018). It can be observed that the deeper the location of a

convolutional layer in D is, the more it is capable of recognizing long-term structures. For G the exact opposite is true because the layer structure is inverted.

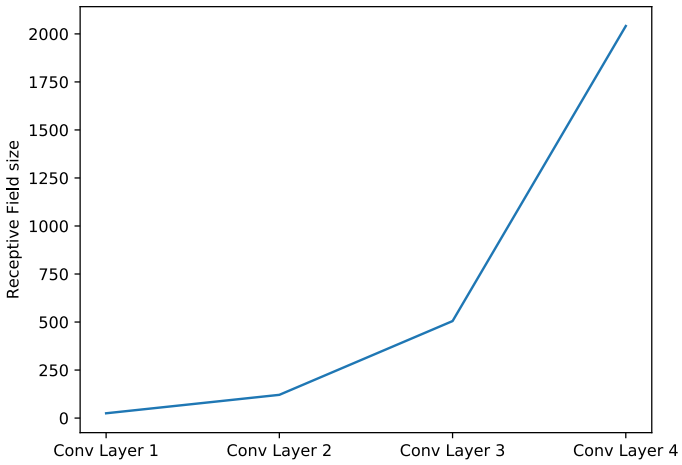

Figure 3.1: Growth of receptive field size in TableGAN

The aim to synthesize single-cycle waveforms is an advantage compared to WaveGAN because this results in the table length being the sampling rate of the signal rather than the sampling rate of $f_s = 16000Hz$ which WaveGAN uses. This is due to the fact that a single cycle of a waveform is sampled at the length of the table, resulting in the wavetable having no specific sampling rate. The table length of $L_T = 4096$ results in a similar architecture like DCGAN. While DCGANs G has an output dimension of $(b, 64, 64, c)$, TableGAN has an output dimension of $(b, 4096, c)$. So TableGAN can be considered a flattened modified 1D version of DCGAN (Radford, Metz, and Chintala 2015). The main difference is that TableGAN is not using batch normalization and has a different loss function and optimizer, using WGAN-GP and ADAM, respectively like WaveGAN with the same hyperparameters.

Table 3.1 and table 3.2 show the layer architectures of TableGANs G and D. The basic model uses no batch normalization and no phase shuffling. The two alternative models marked with $*$ and $\star$ use either batch normalization or phase shuffling with hyperparameter $n = 2$ like in WaveGAN.

The implementation of the model uses the *keras* framework with *tensorflow* backend. For G the layer class *Conv2DTranspose* is used because *keras* does not provide an equivalent 1D version. To fit the input data to the dimensionality of the transposed convolution operator, a dimension is added with an extra layer preceding the transposed convolution and a layer to extract the dimension again thereafter. All other implemented layers are using the shapes defined in table 3.1 and 3.2. The implementation of the WGAN-GP algorithm is taken from Lai (2017). The definitions of the basic and the two alternative models can be found on the USB-Stick via *Repository/src/model_creation*.

Operation	Input Shape	Output Shape
Input $z \sim Uniform(-1, 1)$	(b, 100)	(b, 100)
Dense	(b, 100)	(b, 8192)
Reshape	(b, 8192)	(b, 16, 512)
ReLU	(b, 16, 512)	(b, 16, 512)
$*$ BatchNorm	(b, 16, 512)	(b, 16, 512)
TransConv1D(Stride=4)	(b, 16, 512)	(b, 64, 256)
ReLU	(b, 64, 256)	(b, 64, 256)
$*$ BatchNorm	(b, 64, 256)	(b, 64, 256)
TransConv1D(Stride=4)	(b, 64, 256)	(b, 256, 128)
ReLU	(b, 256, 128)	(b, 256, 128)
$*$ BatchNorm	(b, 256, 128)	(b, 256, 128)
TransConv1D(Stride=4)	(b, 256, 128)	(b, 1024, 64)
ReLU	(b, 1024, 64)	(b, 1024, 64)
$*$ BatchNorm	(b, 1024, 64)	(b, 1024, 64)
TransConv1D(Stride=4)	(b, 1024, 64)	(b, 4096, 1)
Tanh	(b, 4096, 1)	(b, 4096, 1)

Table 3.1: TableGAN Generator Architecture

Operation	Input Shape	Output Shape
Input x or G(z)	(b, 4096, 1)	(b, 4096, 1)
Conv1D(Stride=4)	(b, 4096, 1)	(b, 1024, 64)
LReLU($\alpha = 0.02$)	(b, 1024, 64)	(b, 1024, 64))
$\star$ PhaseShuffle($n = 2$)	(b, 1024, 64)	(b, 1024, 64))
$*$ BatchNorm	(b, 1024, 64)	(b, 1024, 64)
Conv1D(Stride=4)	(b, 1024, 64)	(b, 256, 128)
LReLU($\alpha = 0.022$)	(b, 256, 128)	(b, 256, 128))
$\star$ PhaseShuffle($n = 2$)	(b, 256, 128)	(b, 256, 128)
$*$ BatchNorm	(b, 256, 128)	(b, 256, 128)
Conv1D(Stride=4)	(b, 256, 128)	(b, 64, 256)
LReLU($\alpha = 0.02$)	(b, 64, 256)	(b, 64, 256)
$*$ BatchNorm	(b, 64, 256)	(b, 64, 256)
$\star$ PhaseShuffle($n = 2$)	(b, 64, 256)	(b, 64, 256)
Conv1D(Stride=4)	(b, 64, 256)	(b, 16, 512)
LReLU($\alpha = 0.02$)	(b, 16, 512)	(b, 16, 512)
$\star$ PhaseShuffle($n = 2$)	(b, 16, 512)	(b, 16, 512)
$*$ BatchNorm	(b, 16, 512)	(b, 16, 512)
Reshape	(b, 16, 512)	(b, 8192)
Dense	(b, 8192)	(b, 1)

Table 3.2: TableGAN Discriminator Architecture

3.3 Sampling the TableGAN Latent Space

To be able to create interpolations between two wavetables, the latent space needs to be sampled. This is done by sampling two noise vectors z_1 and z_2 from a uniform distribution $z \sim Uniform(-1, 1)$.

To traverse between those two locations in latent space, an interpolation algorithm is used. Interpolations are also often used in research to show that a model has not just memorized how to mimic training data, but instead successfully learned the underlying distribution (White 2017).

In this case, the interpolation is used to produce smooth transitions between two given wavetables by the network. These interpolations then are

post-processed and played back via the proposed wavetable oscillator. The different interpolations can be chosen via the wavetable position parameter. The objective hereby is to produce smooth, time-varying harmonic spectrum changes without audible artifacts with the automation of the wavetable position.

White (2017) proposes to use the spherical linear interpolation algorithm (slerp) instead of a normal linear interpolation because the latent space of generative models is highly dimensional with an underlying gaussian or uniform prior. When using linear interpolation, it traverses through places within that prior, which are very unlikely to show up (White 2017). The used slerp algorithm is defined as:

$$slerp(q_1, q_2; t) = \frac{\sin(1-t)\Omega}{\sin\Omega} q_1 + \frac{\sin(t\Omega)}{\sin\Omega} q_2, \tag{3.1}$$

where $\sin\Omega = q_1 \cdot q_2$ and $q_1 \cdot q_2$ is the dot product of unit vectors of q_1 and q_2 respectively, as proposed by Shoemake (1985). For t the inequality $0 \geq t \leq 1$ has to be suffused. The created interpolations between z_1 and z_2 are then fed into TableGAN, which makes the predictions and generates the resulting wavetables.

3.3.1 Wavetable Post Processing

The post-processing step takes the generated wavetables and transforms every wavetable into a wavetable stack, using the operations presented in chapter 2.1.2. This process is neccessary to band limit the created tables and to avoid aliasing artifacts when playing back the wavetables through the proposed wavetable oscillator. The whole process uses the same parameters as the operations in chapter 2.1.2.

For every wavetable created the wavetable stack which is composited of *WaveTable*-objects holding the table, the table length and the highest frequency at which the table can be played back is saved to a *JSON*-file. The wavetable oscillator can then process those *JSON*-files.

3.4 Proposed Oscillator

The proposed wavetable oscillator can load multiple *JSON*-files and extract the saved wavetable stacks consisting of *WaveTable*-objects. The wavetable *JSON*-files which can be read by the oscillator are generated by either the reference model synthesis engine presented in chapter 2.1.4 or by the sampling and interpolation process of TableGAN presented in chapter 3.3.

The playback process is based on the linear interpolation wavetable oscillator algorithm presented in appendix C. All given wavetable stacks are loaded into the oscillator. During playback, the table at a given wavetable and wavetable stack position is chosen. When the wavetable position is between two neighboring tables, they are linearly interpolated.

4. Evaluation

The Evaluation of model performance is a critical part in comparing the strengths and shortcomings of different model architectures. While GANs are creating appealing results, there is still no consensus on how to measure if one GAN performs better than another. This is due to the lack of robust and consistent metrics, which is an ongoing area of research. The main issue is that the probability of $p_g(x)$ cannot be computed explicitly, leading many researchers to adapt to qualitative comparisons which are subjective and can be misleading (Lucic et al. 2017).

To overcome those shortcomings two different quantitative measures have been introduced, the Inception Score (IS) (Salimans et al. 2016) and the Fréchet Inception Distance (FID) (Heusel et al. 2017). Both, IS and FID presume a pretrained image-classifier. IS, which is widely adopted for GAN measurement, is based on the idea, that generated samples should have low entropy when being evaluated by the classifier, as well as being diverse and covering all classes (Borji 2018).

FID was introduced, because IS does not compare real with fake data, leading to IS being sensitive in favoring "memory GANs". FID, however, embeds generated samples into a feature space given by a specific layer of Inception net. The embedded layer is considered a continuous multivariate

gaussian, where mean and covariance are estimated for both real and generated data. To quantify the quality of samples, the Fréchet Distance between both Gaussians is calculated (Lucic et al. 2017). For evaluating TableGAN FID is not practicable because Heusel et al. (2017) suggest using at least 20K samples for comparison between real and generated data. TableGAN, however, was only trained on 119 different samples.

To adjust for the problems of IS in favoring "memory GANs" and being sensitive to mode collapse, different evaluation methods are additionally applied to validate the different model architectures. The applied methods are:

- Inception Score for evaluation of quality and diversity of generated samples,
- evaluation of similarities within classes of training data using Pearson Correlation Coefficient (PCC),
- evaluation of mode drop and mode collapse through comparison of class distributions,
- nearest neighbor comparison between ground truth and different architectures to detect overfitting,
- evaluation of space continuity through interpolating the latent space to ensure the model can produce novel samples.

4.1 TableGAN Training Process

Because of the limited computational budget, the models were trained for a fixed number of iterations and not until convergence using the IS, like Donahue, McAuley, and Puckette (2018) propose for WaveGAN. This restriction

had to be made because the training of a TableGAN model for 20K iterations took about two hours on a *Nvidia Tesla K80*-GPU, with checkpoints every 20 iterations. The calculation of the IS also took around 2 hours for one model. By using IS as an early stopping criteria, this would have added around 2000 hours of extra time for training per model, which is not sufficient.

The approach taken here was, therefore, to train the three different model architectures for 20K iterations and then measure the IS. The model which scored the highest IS was trained again for 200K iterations, which is the number of iterations WaveGAN took to converge using IS (Donahue, McAuley, and Puckette 2018). The training of TableGAN for 200K iterations took about 20 hours to complete on the same system.

4.2 Inception Score (IS) Measures

The IS is the measure that is most widely adopted to GAN evaluation. It uses a pre-trained classifier, which in the original implementation proposed by Salimans et al. (2016), uses the Inception model. Using IS as an automatic method to evaluate samples, correlates well with human evaluation as Salimans et al. (2016) claim.

The classifier is applied to every generated image to get the conditional label distribution $p(y|x)$. Images that contain a clearly discernable object should have a conditional label distribution with low entropy, meaning, that the classifier can tell with high confidence to which class the image belongs. Because it is expected that the model does not only learn images from one class, the marginal distribution $\int p(y|x = G(z))dz$ should have high entropy (Salimans et al. 2016).

The combined metric, which is proposed, can be calculated by using the

Kullback-Leibler-Divergence (KL-Divergence), which measures the distance between two distributions. The resulting IS can be calculated via:

$$exp(\mathbb{E}_x KL(p(y|x)||p(y))), \tag{4.1}$$

as proposed by Salimans et al. (2016). With the KL-Divergence defined as:

$$KL(P||Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)}, \tag{4.2}$$

as introduced by Kullback and Leibler (1951).

Salimans et al. (2016) propose to evaluate the metric over an amount of approximately 50K samples because a part of the metric measures diversity. A model that generates diverse, high-quality images will score a high IS, whereas a model that generates low-quality samples will score a low IS. The upper bound of IS is defined by the number of class labels (Mack 2019). The drawback of the IS is, that it favors "memory GANs" and is highly sensitive to mode collapse. Models, which store all training samples and models which collapse into creating just one sample per class can score a high IS. Consequently IS does not take into account whether the real distribution was learned, because it does not consider real samples when measuring the score (Heusel et al. 2017).

4.2.1 Analyzation Process

Because this thesis treats generated audio samples instead of images, the audio data first has to be converted into a corresponding visual representation to be classified. Single generated wavetables have been converted to mel-scaled spectrograms, using the Short-Time Fourier transform (STFT) with a

hop size of 512 samples and a window-length of 2048 samples. The resulting frames which are created by the STFT are then scaled, using the mel-scale with 26 mel-filters. The mel-scale is a perceptual scale that was judged by human listeners to find equal distances in between different pitches (Stevens, Volkmann, and Newman 1937). Applying the mel-scaling process to the STFT, the resulting visual representations are close to how humans perceive audio data.

For classification of generated wavetables, a specific classifier has been trained. At first, a transfer-learning approach was taken, where the top layer of the Inception model was changed to a softmax layer with seven outputs following the total class count of the experiment. For the training process the training data of TableGAN which included 119 different wavetables of seven classes, was split into a test and a training set, where the test set included four samples of every class and the training set included 12 samples of every class.

Because this approach led to a classifier which scored just 65% on the test data and was overfitting on the training data, the approach of Donahue, McAuley, and Puckette (2018) was taken in creating a custom classifier. The architecture of the used classifier is the same as in Donahue, McAuley, and Puckette (2018) with four layers of batch-normalization, followed by a convolutional layer with stride $s = 1$, a kernel $k = (5, 5)$ and a relu activation function and max-pooling. Those four layers are followed by a flatten layer and another batch-normalization and finally a dense layer with a softmax activation function. Only the input dimensions of the network have been changed to 200 by 400 sized RGB images. This approach led to a classifier, which scored $93,75\%$ on the test data.

For the analyzation, all models generated 50K samples which were then

converted to mel-scaled spectrograms to be classified and with the predictions of the classifier the IS was calculated over all samples. The resulting scores can be observed in table 4.1.

Architecture	Inception Score
TableGAN	$\mathbf{3.34 \pm 0.02}$
TableGAN (200K)	2.26 ± 0.02
TableGAN with Batch Normalization	1.79 ± 0.01
TableGAN with Phase Shuffle	2.45 ± 0.02

Table 4.1: Model Inception Scores

It can be seen, that the basic TableGAN architecture, which was trained for 20K iterations scores the highest IS with a standard deviation of 0.02. This is the basic architecture without phase suffling and batch normalization. The batch normalization architecture scores the worst IS, which could be assumed as Gulrajani et al. (2017) propose to avoid batch normalization when using WGAN-GP as a loss function. Interestingly, the architecture using phase shuffling, scores a significantly worse IS than the basic TableGAN architecture, while Donahue, McAuley, and Puckette (2018) showed that WaveGAN scored the highest IS using phase shuffling in D. Also the basic TableGAN architecture trained for 200K iterations scores a significantly lower IS than the architecture which was trained for 20K iterations. When focusing on IS, it can be assumed that a longer training duration is not an indicator for better GAN performance.

4.3 Comparison between Test-Data Inner-Class Similarities

In order to facilitate comparison of class distributions, the similarity of class elements among themselves needs to be investigated. Taking the inner-class similarities into account, allows for a more grounded assessment of different evaluation outcomings.

For the analysis of similarities between audio-files, the Pearson Correlation Coefficient (PCC) is utilized. The PCC analyzes how two signals are correlated. When two signals are completely positively correlated, all points on a scatterplot are lying on an ascending straight line which results in the PCC being $r = 1$. When two signals are uncorrelated, meaning there are no relationships between the two signals, the PCC is $r = 0$. When two signals, however, are completely negatively correlated, all data points are lying on a straight descending line (Papula 2016). This means that the two signals have a phase offset of π . The PCC of input vectors x and y is calculated by:

$$r = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sqrt{(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2)(\sum_{i=1}^{n} y_i^2 - n\bar{y}^2)}}, \tag{4.3}$$

as presented in Papula (2016).

To analyze the inner-class similarities the PCC of every sample among themselves has been calculated and plotted in scatterplots using thin points for the scatter plots. With this approach the overall inner-class similarity trend can be visualized, where multiple correlation points are joining. Figure 4.1 shows the resulting scatter plots of all classes and table 4.2 shows the means and standard deviation of the PCC per class.

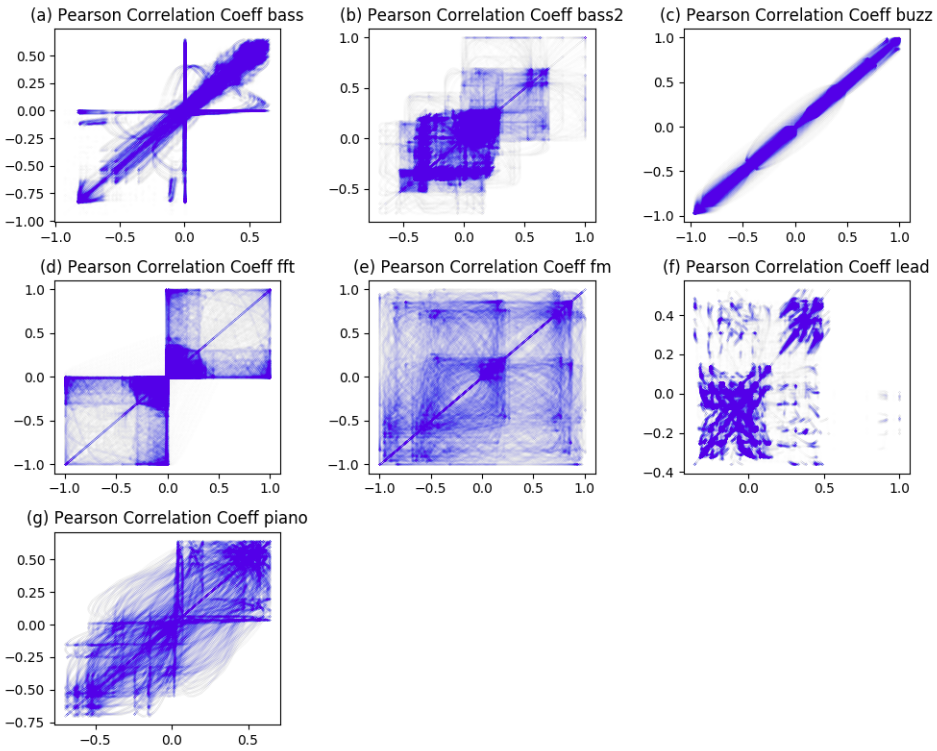

Figure 4.1: Scatter-Plots of Correlation between Inner-Class Samples (a) Bass Class (b) Bass 2 Class (c) Buzz Class (d) FFT Class (e) FM Class (f) Lead Class (g) Piano Class

Class	Mean	Standard Deviation
Bass	0.78	±0.35
Bass 2	0.62	±0.27
Buzz	0.99	±0.01
FFT	0.59	±0.28
FM	0.23	±0.45
Lead	0.59	±0.38
Piano	0.82	±0.18

Table 4.2: Mean and Standard Deviation of Inner-Class Similarities

When observing figure 4.1 along with table 4.2 it can be seen, that the classes *Bass, Buzz* and *Piano* have the strongest correlations, with *Buzz*

clearly having the most prominent correlations within the group. The classes *Bass* and *Piano*, however, have a significant higher standard deviation. This diversification can also be seen in figure 4.1 (a) and (g).

The classes *Bass 2, FFT* and *Lead* all have a mean which is roughly centered between the interval $[0, 1]$ with all having quite high standard deviations, but all classes show a linear trend. The class *FM*, however, is scattered, which can be seen in figure 4.1 (e).

When just taking these findings into account, it would be reasonable to suggest that the learned TableGAN model would rather generate a sample that could be classified as *Buzz* than *FM*. Although the underlying class distribution is uniform, the chances for the model in choosing to generate a sample from a specific class over another is significantly higher, because of the diverse inner-class similarities.

4.4 Comparison between Model Class Distributions

GANs are sensitive to failure in modeling the entire data distribution, while still generating realistically looking images. Both, mode collapse and mode drop, can occur as a result. Mode collapse happens when the generator maps several input noise vectors to the same output. This leads to low diversity in generated samples. Mode drop occurs when modes of the real data distribution are ignored by the generated distribution because they are difficult to represent.

To evaluate whether a model collapsed, Santurkar, Schmidt, and Madry (2017) propose to train the model on a well-balanced dataset. Because of this, all audio files were sliced into the same amount of wavetables in section 3.1.

To evaluate whether a GAN generated uniformly distributed data, a classifier is trained on the dataset. The classifier then evaluates the generated data of the models, and compares the results, using a bar chart (Santurkar, Schmidt, and Madry 2017).

The classifier used in this case is the one created for the IS in section 4.2. For the analysis of the class distributions, every model generated 50K samples, which were converted to mel-spectrograms using the same parameters as in section 4.2. The resulting distributions are shown in figure 4.2.

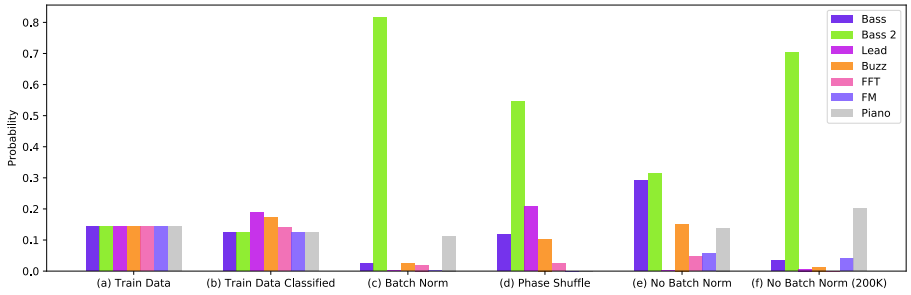

Figure 4.2: Comparison of Model Class Distributions (a) Train Data (b) Train Data Classified (c) Batch Norm (d) Phase Shuffle (e) No Batch Norm (f) No Batch Norm (200K)

Figure 4.2 (a) shows the uniformly distributed train data, while figure 4.2 (b) depicts the evaluated train data by the classifier to show if certain classes are preferred by the classifier. Figure 4.2 (c) and (d) show the class distribution of the TableGAN model with batch normalization and without batch normalization but trained for 200K iterations. Both models show severe signs of mode collapse, as the probability for generating a sample from class *Bass 2* are at 80% and 70% respectively.

Also, the model which applies phase shuffling, is not uniformly distributed. Interestingly, the *Piano* class has a very low probability of being generated by this model. All models have problems in generating samples

from the *FFT* class, and just the models without batch normalization are capable of producing samples from the *FM* class.

It can be observed that the model with no batch normalization, which was trained for 20K iterations, has a distribution that is closest to a uniform distribution when comparing all models. However, even the model in figure 4.2 (e) shows signs of mode dropping because no samples from the *Lead*-class were created.

4.5 Nearest Neighbor Comparison

The nearest neighbor comparison should detect overfitting in models. In order to achieve this, generated samples from different models are shown next to each other. For image generation models, normally the euclidean distance is taken to find nearest neighbors between model outputs and the training data.

The method to find the nearest neighbors in the audio domain is the PCC, which was presented in section 4.3. The TableGAN model creates four random wavetables. For those wavetables, the PCC gets calculated over the whole training set to find the nearest neighbors which are found computing $argmax\ r$, where r is the vector holding the PCC values per generated wavetable. To find the nearest neighbors with competing models, every model generates 1000 samples. Afterward, the PCC is computed over the generated samples, and the nearest neighbors are chosen as shown above. Figure 4.3 shows the nearest neighbors for all models and the ground truth set.

It can be observed that both the ground truth set and the generated images by the model without batch normalization (200K) are very similar

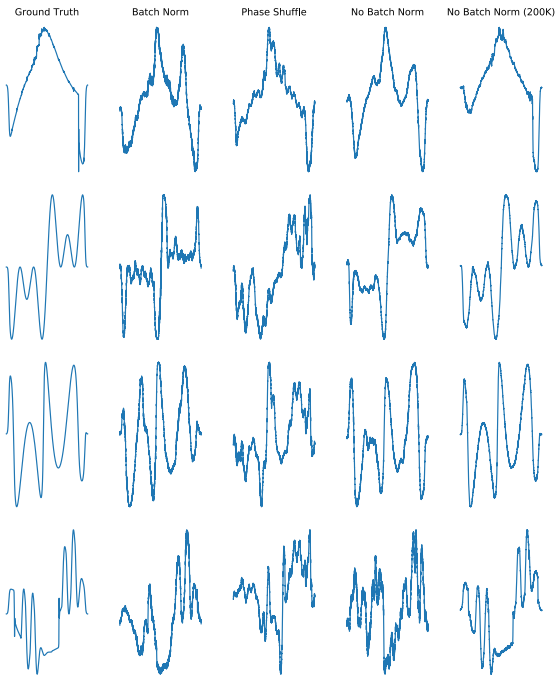

Figure 4.3: Nearest Neighbor Comparison

to each other. With the observations from section 4.3 and 4.2 the suspicion arises, that the model without batch normalization (200K) is a "memory GAN", which memorized the distribution.

The other models show no signs of overfitting as every model created samples that are similar but not the same as the ground truth data. However, the model without batch normalization (20K) produces wavetables which are closer to the ground truth data than the other competitors (except for the 200K model, which overfits).

4.6 Evaluation of Space Continuity

To evaluate what a generative model has learned, the latent space of the model can be explored. This is done by interpolating between two points

using the slerp interpolation method presented in section 3.3. Interpolating the latent space evaluates the level of detail a model was able to extract. If the generated wavetables between two points are reasonable, this method can provide another sign that a model is capable of producing new samples instead of just memorizing the test data (Borji 2018). Figure 4.4 shows the interpolations every model has created between two specific points.

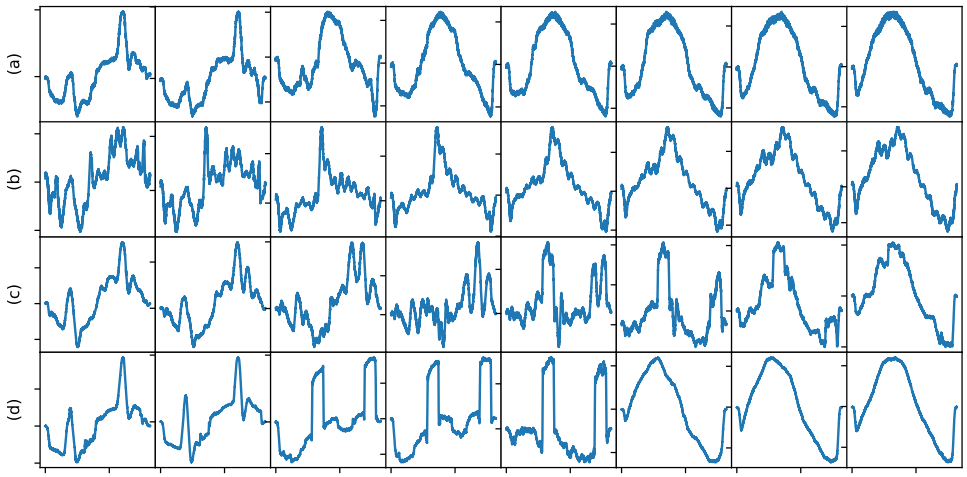

Figure 4.4: Model Interpolations between (a) Batch Norm (b) Phase Shuffle (c) No Batch Norm (d) No Batch Norm (200K)

Examining figure 4.4 shows the interpolation between two latent points of all four models. It can be seen that the model with phase shuffling (b) and the model without batch normalization (c) are producing smooth interpolations. Every successive frame is different and blends smoothly into the next. However, the model with batch normalization (a) and the model without batch normalization which was trained for 200K iterations are showing signs of mode collapse and especially (d) seems to be a "memory GAN".

This assumption is strengthened when comparing the animated *GIFs* and

audio-files which interpolate between the two points in latent space over 128 steps, which can be found at [media files for interpolation process] or on the accompanying USB-Stick via *Evaluation Files/GIFS* and *Evaluation Files/AUDIO*. [(d) GIF] and [(d) audio-file] show the interpolation process of the model without batch normalization, which was trained for 200K iterations. Those two representations strengthen the assumption further, that (d) is a "memory GAN" because of the abrupt changes in contrast to the smooth ones in the models (c) and (b).

To support this, [(c) GIF] and [(c) audio-file] are showing that the model without batch normalization is producing smooth interpolations without abrupt changes like in (d).

5. Results, Discussion and Limitations

When considering the results of the evaluation, it can be drawn to conclusion that all models are sensitive to both mode dropping, as well as mode collapse. Reviewing figure 4.2 it can be seen that the TableGAN model with batch normalization dropped the *Lead* and *FM* class and has a probability of approximately 80% in creating a sample from the *Bass 2* class. The model which incorporated phase shuffling dropped two classes, namely *FM* and, interestingly, *Piano*, the second most prominent in all other model distributions. The model which is most uniformly distributed is the model without batch normalization which was trained for 20K iterations. But even this model dropped the *Lead* class.

Observing figure 4.1 and table 4.2 it can be seen that the low degree of correlation within the *FM* class can be considered a factor in leading to the models having difficulties in learning the class representation. When looking at the nearest neighbor comparison shown in figure 4.3 and the evaluation of space continuity in figure 4.4, it can be seen that the TableGAN model without batch normalization, which was trained for 20K iterations, represents the best trade-off between wavetable similarity to the ground truth set and

the highest diversity which could be captured within the latent space. The model utilizing batch normalization and the model without batch normalization (200K) were not able to capture a diverse latent space and produce interpolations with abrupt changes. Especially the model without batch normalization (200K) has just learned the training set thus being considered a "memory GAN". It is interesting to see how the extra training time led the model to overfit on the data set.

Solely the model which utilizes phase shuffling creates smooth interpolations besides the basic TableGAN model. Those models learned the representations of the ground truth set the best by looking at the IS of the models. The model using phase shuffling and the basic TableGAN model score the highest IS with 2.45 and 3.34 respectively. Using the interpolations created by TableGAN, creates smooth fades between two random wavetables, when played back with the proposed oscillator system. Audio examples with automated wavetable position can be found in [Audio Examples] or on the USB-Stick at *Evaluation Files/Synthesized TableGAN Examples*. An area of ongoing research is trying to train both models until convergence, using a larger and more diverse data set.

6. Conclusion and Future Directions

This thesis examined the possibility of using generative deep learning models for wavetable creation and interpolations, which are ready for playback using a proposed oscillator framework.

For generating wavetables TableGAN was introduced which is based on both WaveGAN and DCGAN. TableGAN is able to be trained on a dataset of classes containing single-cycle waveforms and can afterwards synthesize wavetables using latent space representations and interpolate between two randomly selected representations. The resulting interpolations can be played back with a proposed oscillator system which has the ability to automate the wavetable position to create sounds which are evolving harmonically over time. The interesting part of the created interpolations is that the model creates those interpolations by itself through latent space representations. The resulting interpolations are very different from simple linear interpolation methods between two wavetables, providing artists and sound designers with an additional new tool for multiple sound design purposes.

All tools for creating single-cycle waveforms from audio files of arbitrary

length have been provided with the USB-Stick and can be used to create new datasets along with scripts for interpolating the latent space of the resulting model and rendering playback of interpolations to audio files using the oscillator framework.

Even though the model is capable of learning the general underlying data representations and synthesizing audio, the distribution of training data was not fully captured. Proposals for future directions in this field would include:

- Creation of a larger well-balanced dataset using single-cycle waveforms with high inner-class correlations,
- training of TableGAN models without batch normalization and with phase shuffling till convergence,
- investigation of the possibility to train a conditional TableGAN based on Lee et al. (2018).

List of Acronyms

ANN	Artificial Neural Networks
AMDF	Average Magnitude Difference Function
CNN	Convolutional Neural Network
DNN	Deep Neural Network
D	Discriminator
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
FPU	Floating-Point Unit
FID	Fréchet Inception Distance
G	Generator
GAN	Generative Adversarial Network
IS	Inception Score
IDFT	Inverse Discrete Fourier Transform
KL-Divergence	Kullback-Leibler Divergence
ML	Machine Learning
MLP	Multi Layer Perceptron
PCC	Pearson Correlation Coefficient
PCM	Pulse Code Modulation
RNN	Recurrent Neural Network
S&S	Sample & Synthesis
slerp	spherical linear interpolation
STFT	Short-Time Fourier transform
TLU	Threshold Logic Unit
WGAN-GP	Wasserstein GAN gradient penalty

List of Figures

2.1 Comparison of (a) AMDF and (b) YIN Pitch Detection Performance . . . 10
2.2 Representation of the 8 first wavetables of a square wave within a wavetable stack, with $r_s = 2$ . . . 15
2.3 Cross-Correlation Process (a) Sine Wave without shift (b) Sine Wave shifted by π (c) Cross-Correlation (d) Corrected Sine Wave 16
2.4 Comparison Hanning-Window Smoothing and tanh-smoothing (a) Sawtooth smoothed with Hanning (b) Sawtooth smoothed with *tanh* . . . 17
2.5 Comparison of Interpolation Algorithms with table length of $L = 1024$ (a) Truncation, (b) Linear Interpolation, (c) Lagrange Interpolation . . . 21
2.6 Comparison of Interpolation Algorithms with table length of $L = 4096$ (a) Truncation, (b) Linear Interpolation, (c) Lagrange Interpolation . . . 22
2.7 Threshold Logic Unit (Géron 2017) . . . 25
2.8 WGAN Gradient Penalty Algorithm taken from (Gulrajani et al. 2017) . . . 33
2.9 Growth of receptive field size in WaveGAN . . . 34
2.10 Checkerboard Artifacts in generated images taken from (Odena, Dumoulin, and Olah 2016) . . . 35
2.11 Phase Shuffling Process taken from (Donahue, McAuley, and Puckette 2018) . . . 36

3.1 Growth of receptive field size in TableGAN . . . 42

4.1 Scatter-Plots of Correlation between Inner-Class Samples (a) Bass Class (b) Bass 2 Class (c) Buzz Class (d) FFT Class (e) FM Class (f) Lead Class (g) Piano Class . . . 55

4.2 Comparison of Model Class Distributions (a) Train Data (b) Train Data Classified (c) Batch Norm (d) Phase Shuffle (e) No Batch Norm (f) No Batch Norm (200K) 57
4.3 Nearest Neighbor Comparison 59
4.4 Model Interpolations between (a) Batch Norm (b) Phase Shuffle (c) No Batch Norm (d) No Batch Norm (200K) 60

List of Tables

2.1 WaveGAN Generator Architecture 37
2.2 WaveGAN Discriminator Architecture 38

3.1 TableGAN Generator Architecture 44
3.2 TableGAN Discriminator Architecture 45

4.1 Model Inception Scores 53
4.2 Mean and Standard Deviation of Inner-Class Similarities . . . 55

Bibliography

[SVN37] Sheridan S. Stevens, John E. Volkmann, and E. B. Newman. "A Scale for the Measurement of the Psychological Magnitude Pitch". In: 1937.

[Sha49] C. E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (Jan. 1949), pp. 10–21. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1949.232969.

[KL51] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *Ann. Math. Statist.* 22.1 (Mar. 1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[Ros+74] M. Ross et al. "Average magnitude difference function pitch extractor". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 22.5 (Oct. 1974), pp. 353–362. ISSN: 0096-3518. DOI: 10.1109/TASSP.1974.1162598.

[Sho85] Ken Shoemake. "Animating rotation with quaternion curves". In: *SIGGRAPH*. 1985.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning internal representations by error propagation". In: 1986.

[Bri96] Robert Bristow-Johnson. "Wavetable Synthesis 101, A Fundamental Perspective". In: (Jan. 1996).

[CK02] Alain de Cheveigné and Hideki Kawahara. "YIN, a fundamental frequency estimator for speech and music." In: *The Journal of the Acoustical Society of America* 111 4 (2002), pp. 1917–30.

[Zöl08] Udo Zölzer, ed. *Digital Audio Signal Processing*. 2nd ed. Chichester, West Sussex: John Wiley & Sons Ltd, 2008.

[Rus09] Martin Russ, ed. *Sound Synthesis and Sampling*. 3rd ed. Burlington, MA: Focal Press, 2009.

[Smi10] Julius O. Smith. *Physical Audio Signal Processing*. Stanford CCRMA, 2010. URL: http://ccrma.stanford.edu/~jos/pasp/. (accessed: 16.06.2019).

[BL11] Richard Boulanger and Victor Lazzarini, eds. *The Audio Programming Book*. Cambridge, Massachusetts: The MIT Press, 2011.

[Muh11] Ghulam Muhammad. "Extended average magnitude difference function (EAMDF) based pitch detection." In: *Int. Arab J. Inf. Technol.* 8 (Apr. 2011), pp. 197–203.

[Zöl11] Udo Zölzer, ed. *DAFX: Digital Audio Effects*. 2nd ed. Chichester, West Sussex: John Wiley & Sons Ltd, 2011.

[FV12] Andreas Franck and Vesa Välimäki. "Higher-Order Integrated Wavetable Synthesis". In: Sept. 2012, pp. 245–252.

[Red12] Nigel Redmon. *A wavetable oscillator—Part 2*. 2012. URL: https://www.earlevel.com/main/2012/05/08/a-wavetable-oscillator%E2%80%94part-2/. (accessed: 13.06.2019).

[Goo+14] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *NIPS*. 2014.

[Pir15] Will Pirkle, ed. *DESIGNING SOFTWARE SYNTHESIZER PLUG-INS IN C++: FOR RACKAFX, VST3 AND AUDIO UNITS*. Burlington, MA: Focal Press, 2015.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *CoRR* abs/1511.06434 (2015).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Luo+16] Wenjie Luo et al. "Understanding the Effective Receptive Field in Deep Convolutional Neural Networks". In: *NIPS*. 2016.

[ODO16] Augustus Odena, Vincent Dumoulin, and Chris Olah. "Deconvolution and Checkerboard Artifacts". In: *Distill* (2016). DOI: 10.23915/distill.00003. URL: http://distill.pub/2016/deconv-checkerboard.

[Oor+16] Aäron van den Oord et al. "WaveNet: A Generative Model for Raw Audio". In: *CoRR* abs/1609.03499 (2016). arXiv: 1609.03499. URL: http://arxiv.org/abs/1609.03499.

[Pap16] Lothar Papula, ed. *Mathematik für Ingenieure und Naturwissenschaftler Band 3 Vektoranalysis, Wahrscheinlichkeitsrechnung, Mathematische Statistik, Fehler- und Ausgleichsrechnung*. 7th ed. Wiesbaden, Germany: Springer Fachmedien, 2016.

[Sal+16] Tim Salimans et al. "Improved Techniques for Training GANs". In: *ArXiv* abs/1606.03498 (2016).

[AB17] Martín Arjovsky and Léon Bottou. "Towards Principled Methods for Training Generative Adversarial Networks". In: *ArXiv* abs/1701.04862 (2017).

[BHP17] Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. "Deep Learning Techniques for Music Generation - A Survey". In: *ArXiv* abs/1709.01620 (2017).

[Eng+17] Jesse Engel, Cinjon Resnick, et al. "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders". In: *CoRR* abs/1704.01279 (2017). arXiv: 1704.01279. URL: http://arxiv.org/abs/1704.01279.

[Gér17] Aurélien Géron, ed. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. 3rd ed. Sebastopol, CA: O'Reilly Media Inc., 2017.

[Gul+17] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *NIPS*. 2017.

[Heu+17] Martin Heusel et al. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium". In: *NIPS*. 2017.

[Hie17] Dang Ha The Hien. *A guide to receptive field arithmetic for Convolutional Neural Networks*. 2017. URL: https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807. (accessed: 08.08.2019).

[Lai17] Shaofan Lai. *Implement improved WGAN with Keras-2.x*. 2017. URL: http://shaofanlai.com/post/10. (accessed: 18.08.2019).

[Luc+17] Mario Lucic et al. "Are GANs Created Equal? A Large-Scale Study". In: *NeurIPS*. 2017.

[Meh+17] Soroush Mehri et al. "SampleRNN: An Unconditional End-to-End Neural Audio Generation Model". In: *ArXiv* abs/1612.07837 (2017).

[SSM17] Shibani Santurkar, Ludwig Schmidt, and Aleksander Madry. "A Classification-Based Study of Covariate Shift in GAN Distributions". In: *ICML*. 2017.

[Whi17] Tom White. "Sampling Generative Networks". In: 2017.

[Bor18] Ali Borji. "Pros and cons of GAN evaluation measures". In: *Computer Vision and Image Understanding* 179 (2018), pp. 41–65.

[DMP18] Chris Donahue, Julian J. McAuley, and Miller Puckette. "Adversarial Audio Synthesis". In: *ICLR*. 2018.

[Hui18a] Jonathan Hui. *GAN — A comprehensive review into the gangsters of GANs (Part 1)*. 2018. URL: https://medium.com/@jonathan_hui/gan-a-comprehensive-review-into-the-gangsters-of-gans-part-1-95ff52455672. (accessed: 04.08.2019).

[Hui18b] Jonathan Hui. *GAN — Wasserstein GAN & WGAN-GP*. 2018. URL: https://medium.com/@jonathan_hui/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490. (accessed: 04.08.2019).

[Hui18c] Jonathan Hui. *GAN — Why it is so hard to train Generative Adversarial Networks!* 2018. URL: https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b. (accessed: 04.08.2019).

[Lee+18] Chae Young Lee et al. "Conditional WaveGAN". In: *ArXiv* abs/1809.10636 (2018).

[Eng+19] Jesse Engel, Kumar Krishna Agrawal, et al. "GANSynth: Adversarial Neural Audio Synthesis". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=H1xQVn09FX.

[Fos19] David Foster, ed. *Generative Deep Learning Teaching Machines to Paint, Write, Compose, and Play*. 1st ed. Sebastopol, CA: O'Reilly Media Inc., 2019.

[Mac19] David Mack. *GAN — A comprehensive review into the gangsters of GANs (Part 1)*. 2019. URL: https://medium.com/octavian-ai/a-simple-explanation-of-the-inception-score-372dff6a8c7a. (accessed: 18.08.2019).

[Pur+19] Hendrik Purwins et al. "Deep Learning for Audio Signal Processing". In: *IEEE Journal of Selected Topics in Signal Processing* 13 (2019), pp. 206–219.

[VL19] Sean Vasquez and Marshall Lewis. “MelNet: A Generative Model for Audio in the Frequency Domain”. In: *ArXiv* abs/1906.01083 (2019).

Appendices

Appendix A

```
1 import numpy as np
2
3
4 def truncating_wt_osc(out, table, table_len, block_len,
      frequency, sample_rate):
5     curphase = 0.0
6     incr = (frequency / sample_rate) * table_len
7
8     for i in range(block_len):
9         index = int(curphase)
10        out[i] = table[index]
11
12        curphase += incr
13
14        while curphase >= table_len:
15            curphase -= table_len
```

```
16
17          while curphase < 0:
18              curphase += table_len
19
20      return out
```

Appendix B

```
1 import numpy as np
2
3
4 def linear_wt_osc(out, table, table_len, block_len,
      frequency, sample_rate):
5     curphase = 0.0
6     incr = (frequency / sample_rate) * table_len
7     print(block_len)
8     for i in range(block_len):
9         index = int(curphase)
10        frac = curphase - index
11        a = table[index]
12        b = table[(index + 1) % table_len]
13
14        out[i] = a + (b - a) * frac
15
```

```
16          curphase += incr
17
18          while curphase >= table_len:
19              curphase -= table_len
20
21          while curphase < 0:
22              curphase += table_len
23
24      return out
```

Appendix C

```
1 import numpy as np
2
3 class LagrangeInterpolator:
4     def __init__(self, sample_rate, frequency,
        block_length):
5         self.phase = 0.0
6         self.__block_length = block_length
7         self.increment = frequency / sample_rate
8
9     def process(self, table):
10        phase = self.phase
11        increment = self.increment
12        table_len = len(table)
13        block_length = self.__block_length
14
15        output = np.empty(block_length)
```

```
16
17          for i in range(block_length):
18              j = int(phase * table_len)
19              x = np.empty(4, dtype=int)
20              y = np.empty(4)
21              x[0] = (j - 1) % table_len
22              x[1] = j % table_len
23              x[2] = (j + 1) % table_len
24              x[3] = (j + 2) % table_len
25
26              output[i] = self.__calculate_lagrange(x,
                    table, 4, phase * table_len)
27
28              phase += increment
29
30              if phase >= 1.0:
31                  phase -= 1.0
32
33          return output
34
35      def __calculate_lagrange(self, x, table, table_len,
            phase):
36          interpolated_value = 0.0
37
38          for v in range(table_len):
39              l = 1.0
40              for j in range(table_len):
```

```
41                  if j != v:
42                      l *= (phase - x[j]) / (x[v] - x[j])
43
44              interpolated_value += l * table[x[v]]
45
46          return interpolated_value
```

Appendix D

Wavetable Creation from Audio Files For synthesizing wavetables from audio files, a command-line program was implemented, which can be found on the accompanying USB-Stick via *Repository/src/wavetable_creation/create_reference_tables.py*. The program takes the argument *-i* specifying the input path to audio files and *-o* specifying the output directory, where wavetables should be saved.

The argument *-s* tells the algorithm whether to apply the in chapter 2.1.2 proposed tanh-smoothing or not. The argument *-m* specifies the maximum number of tables to create from one file and the parameter *-f* enables *fastwav* or not, depending on whether the input audio is encoded in 16-Bit PCM or 32-Bit PCM float ($f = true$), or $f = false$ when another PCM format is used to enable faster encoding via the *scipy* library when $f = true$ (Donahue, McAuley, and Puckette 2018).

The audio files get analyzed file by file, and at first, all cycles are extracted using the methods from chapter 2.1.2. With those wavetables, created, they get properly phase aligned using the cross-correlation function shown in chapter 2.1.2. Then wavetable stacks are built, using the operations proposed in

chapter 2.1.2, before saving every wavetable stack as a *JSON*-file to disk which can be loaded and played back via the proposed wavetable oscillator.

Audio File Preperation To extract single-cycle waveforms out of a file the provided command-line app, which can be found on the USB-Stick at *Repository/src/wavetable_creation/create_model_training_tables.py*, needs to be invoked with the argument *-i* and a string which points to an audio file directory. The argument *-o* specifies where the output directory containing the generated wavetables is created. With the argument *-m* the maximum number of wavetables can be adjusted, the default value is set to $m = 64$. The boolean argument *-s* sets whether tanh-smoothing should be applied to avoid discontinuities, and *-f* specifies whether the input audio is encoded in 16-Bit PCM or 32-Bit PCM float ($f = true$), or $f = false$ when another PCM format is used to enable faster encoding via the *scipy* library when $f = true$ (Donahue, McAuley, and Puckette 2018).

The file preparation is being executed for every file in the provided input directory. In the first place, the period of the file is estimated using YIN, which is described in chapter 2.1.2. When the first period is estimated, the first slice of the file is extracted. This slice is then stretched to the wavetable length of $L = 4096$ samples, using linear interpolation (chapter 2.1.3), and the DC component is filtered. This process is repeated until either the end of the audio file or the maximum number of tables are reached.

Subsequently, the phases of the analyzed files are aligned using the cross-correlation algorithm (chapter 2.1.2). If the argument $s = true$, the edges of the wavetables are smoothed using the previously described *tanh*-smoothing (chapter 2.1.2).

Sampling the TableGAN Latent Space The console application for generating the interpolated wavetables can be found on the USB-Stick via *Repository/src/wavetable_ creation/create_ model/interpolated_ tables.py*. When starting the application, the command line argument *-i* determines the input model with choices for *batch_ norm*, *n_ batch_ norm*, *phase_ shuffle* and *n_ batch_ norm200K*, representing the four different created models.

The command line argument *-o* determines the output directory within which the wavetables are saved and the parameter *-w* defines the wavetable prefix names, resulting in the namingconvention *<wavetablename>_ <number>.json*. The last parameter *-n* defines the total number of interpolations created by TableGAN.

Wavetable Oscillator The command line app of the wavetable oscillator can be found on the USB-Stick via *Repository/src/wavetable_ oscillation/wavetable_ oscillator.py*. It can be started with the input argument *-i* and a given input folder, where the *JSON*-files are residing. The input argument *-o* sets the output path, where the synthesized *wave*-file should be saved. The input arguments *-p* and *-w* are boolean arguments that determine whether the pitch and wavetable position should be automated, respectively. The argument *-f* sets the playback frequency in Hz and the parameter *-l* sets the playback duration in seconds. With those defined settings, the oscillator synthesizes a sound with the given parameters offline and saves it into the specified output directory.

Appendix E

Contents of the accompanying USB-Stick

1. Digital copy of the Thesis.
2. Repository with accompanying code.
3. Dataset of 119 single-cycle waveforms.
4. Evaluation files of TableGAN interpolations.

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

Kiel, 20.08.2019 ..... K. Wantia .....

city, date signature