Bachelor thesis
Computing Science

Radboud University

Generating Kick Drum Samples with Neural Networks

Author:
Wouter Damen
s1028002
Wouter.Damen@student.ru.nl

First supervisor/assessor:
David van Leeuwen
d.vanleeuwen@science.ru.nl

Second assessor:
Gijs van Tulder
g.vantulder@cs.ru.nl

June 6, 2021

Abstract

The kick drum sound is a central piece in many styles of modern electronic music. In this research we explore current methods in neural audio synthesis and inspect their performance on the specific task of unconditional kick drum generation. To this end, we define a measure to evaluate the quality of a single kick drum sample. We then compare models from related work and propose a progressive growing scheme for the WaveGAN model. We find in our experiments that this Progressive WaveGAN outperforms the other models on our learning task and validate this using a formal listening experiment. The code for our model and the quality measure is available on GitHub[1].

[1] https://github.com/apeklets/kicks-thesis

Contents

Chapter 1

Introduction

In recent years there has been rapid improvement in the quality of AI image synthesis. Neural networks can now generate images or even video footage that humans can not tell are fake [1]. However, audio synthesis seems to significantly lag behind. A lot of work has been done in the area of speech synthesis, which has produced realistic results, but musical AI has a long way to go.

This research focuses on the generation of kick drum samples. These play an important role in any modern electronic music composer's toolkit. In many genres such as House[1] and Techno, the kick drum is often the loudest element in a mix, and is the main driver of the rhythm. So, artists spend a lot of time finding or designing the right kick drum sample that fits the context.

Furthermore, it is a simple yet challenging problem for neural synthesis. A kick drum is a short signal that does not repeat and where every frequency matters, from 20 Hz to 20000 Hz. The network needs to simultaneously learn the timbre and texture, a moving pitch and a volume envelope.

The main question is whether we can use current methods in neural audio synthesis to generate adequate kick drums using a relatively small dataset. This already raises a new question, namely what it means for a kick drum to be "adequate". One requirement for this is a sampling rate of at least 44.1kHz, just like any kick drum sample you would find in a sample pack or on the internet. That is however not a complete answer to the question.

So, we start by defining a quality measure for kick drums specifically. This could be used even outside of this learning task to evaluate samples. We then explore how current work in audio synthesis performs on this task and apply some methodology from image synthesis to improve the results. Finally, we perform a listening test to investigate the validity of both the quality measure and the produced samples.

[1] We have compiled a little playlist of reference tracks, https://spoti.fi/3bNM8ot

Chapter 2

Preliminaries

In this chapter we give an overview of some fundamental concepts that later chapters build upon. We further lay out some notation and terminology regarding audio and neural networks. In the last section of this chapter we dive into some more advanced mathematical theory behind a specific class of neural networks called GANs.

2.1 Digital Audio Terminology

Digital audio is represented as a series of samples, each sample consisting of a fixed number of bits. A single sample represents the amplitude of the signal at that time. The amount of samples taken in a second is called the *sampling rate* or *sampling frequency*, whereas the number of bits per sample is called the bit depth. For high resolution audio, common values are 44.1 kHz, 48 kHz or 92 kHz and 16-bit, 24-bit or 32-bit. In this report, we will denote the sampling rate by f_s.

The Shannon-Nyquist sampling theorem [2] proves that a sampling rate of f_s can uniquely represent all frequencies below $f_s/2$. In short, this is because you need at least two points to define a sine curve. So, if $f_s = 44.1$ kHz, we can record and reproduce all frequencies between 0 Hz and 22050 Hz. The human audible range falls comfortably between these two frequencies, which is why 44.1 kHz is a common choice for f_s. The frequency $f_s/2$ is often referred to as the *Nyquist frequency*.[1]

Note that the term 'sample' can have multiple meanings in this report. We call a single point in a digital signal a 'sample', but we sometimes also call a full signal a 'sample'. For instance, a 1-second long 'kick drum sample' is a sequence of 44100 numbers, and each of those numbers is also a 'sample'. It should usually be clear from the context which is meant.

[1] For an educational deep dive into sampling rates, see https://www.youtube.com/watch?v=-jCwIsT0X8M

2.2 Neural Network Basics

The simplest but most abstract way to view a neural network is as a function parametrized by a set of parameters θ, which takes an input x and produces an output $M(x;\theta)$. One then has to define what the training goal should be. This goal is to minimize the loss function, $\mathcal{L}$, which represents some distance to the optimum. It is usually calculated based on the input from the dataset and the desired output. For instance, if the goal is to classify samples, one has a dataset of sample and label pairs $\{(x_i, y_i)\}$, and computes $\hat{y}_i = M(x_i)$. After which the loss is computed as some distance measure between $\{y_i\}$ and $\{\hat{y}_i\}$.

The goal of training a neural network is to find the values for the parameters θ that optimize the loss value. This is achieved by *backpropagation*, where the loss is viewed as a function of the parameters, $\mathcal{L}(\theta)$ and differentiated. In the simplest case, one updates the parameters by

$$\theta_i := \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i}(\theta)$$

Where the value η is called the *learning rate*. Although many loss functions are defined in terms of the entire dataset, in practice processing every sample in the data for every update step would be infeasible. So, one takes only a single *batch* of say 32 or 64 samples at each step to estimate the loss. This approach is called *stochastic gradient descent*. More complicated methods of updating the parameters also take into account past update steps to achieve more long term stability. One example of which is the Adam method [3].

2.3 Convolutional Networks

A fully connected layer in a neural network, where each of the input nodes is connected with each of the output nodes, can be represented as a single matrix multiplication. The elements of the matrix are the parameters we want to learn, here called 'weights'.

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Fully connected networks are great for classification tasks on low-dimensional datasets and a small number of labels. However, as the input dimensionality and complexity of the task grows, the amount of parameters needed to fully connect all of the layers grows very quickly. In the case of image or audio processing tasks, the input dimensionality can quickly grow to millions of values. If the input is 1 second of 44.1kHz

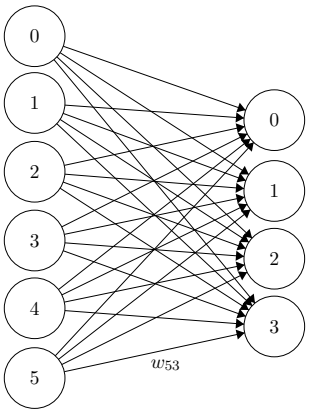


Figure 2.1: A small fully connected network. Each connection is indexed as w_{ij}, one connection is labeled as example.

audio, a single fully connected layer will already contain $1.94 \cdot 10^9$ parameters. If each of those is a 32-bit float, that's almost 8GB worth of parameters!

This is where convolutional networks come in. The idea is that a network may only need to consider a small part of the input at once. Whereas a fully connected layer considers all of the input for every output node it has. The classic example is that of an image net classifying faces; it can consider the eyes, nose and mouth seperately for different output nodes.

The discrete convolution product between two 1-dimensional sequences x and y is defined as:

$$(x \star y)[n] = \sum_{i=-\infty}^{\infty} x[n+i] \cdot y[i]$$

In the case of convolutional networks, x would be the input of the layer, and y is a short sequence of parameters called the *convolution kernel*. Both sequences are of course finite, and so the operation is better written as:

$$(x \star y)[n] = \sum_{i=0}^{k} x[n+i] \cdot y[i]$$

Where k is the length of y, called the *kernel size*. An example of a convolution product is the moving average, where the kernel is just $y = (\frac{1}{k}, ..., \frac{1}{k})$. In general, a convolution product is more like a moving weighted average.

The optimizer will then adjust the values inside the kernels in each of its layers until it optimizes the loss. The benefit of these convolutional layers is that they share parameters across the output nodes, greatly reducing the amount of parameters that are being learned.

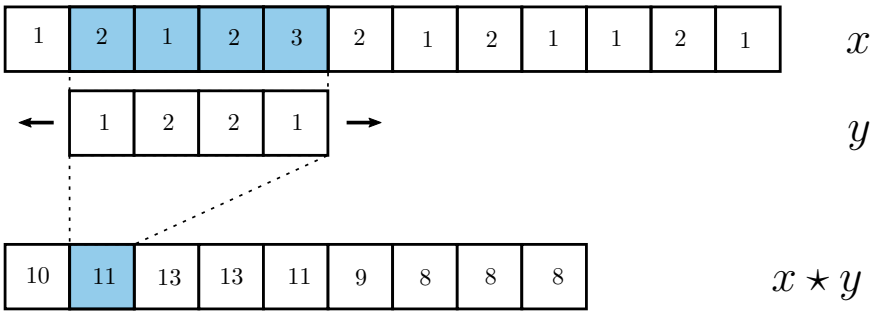


Figure 2.2: Example 1D convolution product.

For convolutional networks such as those in [4], there are multiple kernels being learned at each layer. If you see a description as "1D convolution layer with kernel size $(k, N_{\text{in}}, N_{\text{out}})$", this means that layer takes N_{in} input channels, and produces N_{out} different output channels, where for each output channel there is a k-length convolution kernel being learned. For each output channel out_i, the convolution product is taken over all input channels:

$$\text{out}_i[n] = b_i + \sum_{j=1}^{N_{\text{in}}} (\text{in}_j \star y_i)[n]$$

Where y_i is the convolution kernel for that output channel, and b_i is the layer *bias*.

2.4 Three Generative Models

Our next discussion will concern three different approaches to creating neural networks that generate new data instead of classifying or otherwise processing already existing data. The goal is to give a high-level overview of these methods, leaving the details to be read in the various research papers that apply them.

2.4.1 Generative Adversarial Networks (GANs)

In a GAN, two neural networks play an adversarial game in which they try to maximize or minimize the same loss value. The generator network G takes random noise as input, and outputs fake datapoints. The discriminator network D is then tasked with classifying whether its input is a real datapoint from the dataset or a datapoint generated by G. The generator tries to minimize the classification accuracy, while the discriminator is trying to maximize it [5].

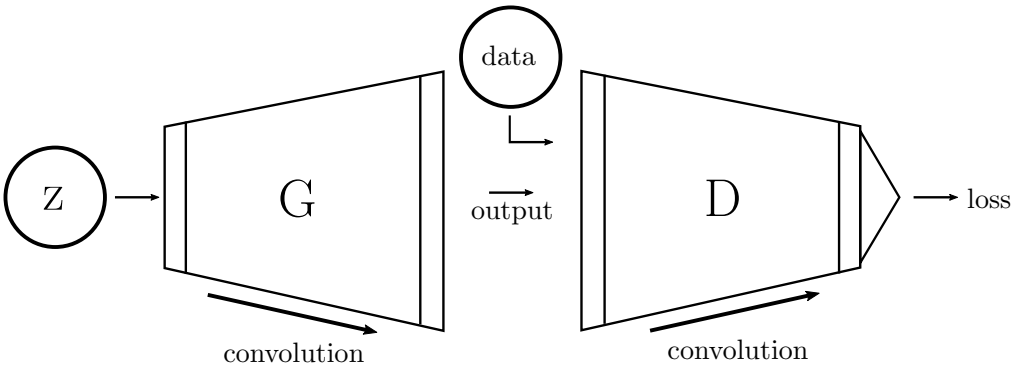


Figure 2.3: A schematic GAN architecture.

The loss function used by the GANs we evaluate in this report is called the WGAN-GP loss, short for "Wasserstein GAN with gradient penalty" [6]. We discuss this particular loss function in section 2.5. For the training method, it is important that G and D train such that one model does not get a large advantage over the other. Most approaches let D make some N training steps for each training step of G, where N is in the order of 5 to 10.

2.4.2 Variational Autoencoders (VAEs)

Variational Autoencoders look similar to GANs, but with the direction of upsampling and downsampling switched. The two networks are now called the encoder E and the decoder D. The samples from the dataset are encoded to a low-dimensional representation in the *latent space* Z, and are then decoded to reconstruct the original sample. The networks work together to minimize the distance between the original and reconstructed data. Generation of new samples is then done by taking a random element $z \in Z$ and computing $D(z)$.

The easiest choice for the distance measure between input and output, d, would be the Euclidean distance, but as we will also discuss in section 3.3, this does not effectively model the perceptive distance between the input and output. Instead, the loss is computed as a distance measure between probability distributions. Read more in section 2.5 and in [7].

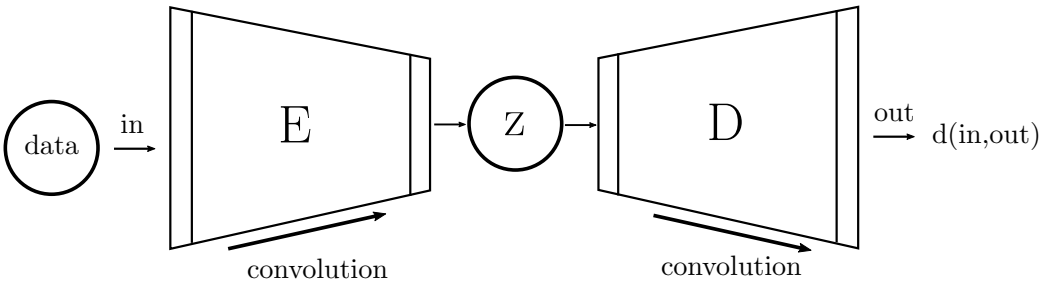


Figure 2.4: A schematic VAE architecture.

2.4.3 Recurrent Neural Networks (RNNs)

A recurrent neural network is any neural network that uses previous output(s) from its last layer as part of the input of its first layer. In a generative setting, this often means that we run the network multiple times to produce multiple outputs, and then string them together. For example, if the model works on windows of 512 samples, and we want 4096 samples, we compute

$$x_i = M(x_{i-1}; \theta)$$

And the final output will obtained by combining x_1 through x_8, where x_0 is some sort of initial vector. The theory is that the RNN learns to encode some kind of internal state that helps it keep track of where it is in the generation process, and that it will use this to generate its next output. The network is then trained with example sequences (in our case kick drums) with the goal being to minimize the reconstruction loss.

2.5 Probability Theory

In the previous section, we glossed over the details of the loss functions for the various architectures. To illustrate the kind of mathematics that is behind these methods, we examine in this section the loss functions for GANs.

To reason about datasets and samples in mathematical formulas, the dataset is modeled as a probability distribution, where each sample has the same chance of getting picked. This is called the *data distribution*, denoted p_{data}. The generator further takes as input samples from the *noise distribution*, denoted p_z. The range of output samples that the generator can produce from this is called the *generator distribution*, denoted p_g. So, sampling a value $x \sim p_g$ means taking a random sample $z \sim p_z$, and computing $x = G(z)$.

Using this notation, the GAN loss function was originally defined in [5] as:

$$\mathcal{L}(G, D) = \mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{x \sim p_g}[\log(1 - D(x))]$$

The notation $\mathbb{E}_{x\sim p}[f(x)]$ denotes the *expected value* for the function f when computed over the distribution p. In practice, this is implemented by simply taking a batch of N samples, and computing the mean $\frac{1}{N}\sum_{i=1}^{N} f(x_i)$.

For what is inside the expected values, note that the discriminator is trying to maximize this loss, while the generator is trying to minimize it. So, the discriminator aims to return 1 for a data sample, and 0 for a fake sample. The generator however tries to fool the discriminator, and wants the exact opposite to happen.

A later paper, [8], proposes the use of the Wasserstein distance for the GAN loss function. This is defined[2] as a distance measure between probability distributions:

$$W(p_1, p_2) = \sup\{\mathbb{E}_{x\sim p_1}[f(x)] - \mathbb{E}_{x\sim p_2}[f(x)] \mid f \text{ is 1-Lipschitz}\}$$

A function f is K-Lipschitz for some constant $K > 0$ if for all inputs x and y we have

$$||f(x) - f(y)|| \leq K \cdot ||x - y||$$

This f in the case of GANs becomes our discriminator function. While training D, we get closer and closer to this supremum. We now want D to be 1-Lipschitz, whereas previously any function from the input space to $[0, 1]$ was permitted. Since D is also assumed to be differentiable, this means we want to have $||\nabla_x D(x)||_2 \leq 1$ for all x in the input space.

In [6], this constraint is enforced by adding *gradient penalty* which is meant to make $||\nabla D||$ approach 1. This is called the WGAN-GP loss, and it is the loss function used by the GANs we evaluate in this report:

$$\mathcal{L}_{GP}(G, D) = \mathbb{E}_{x\sim p_{\text{data}}}[D(x)] - \mathbb{E}_{x\sim p_g}[D(x)] + \lambda\mathbb{E}_{\hat{x}\sim p_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2]$$

Where $p_{\hat{x}}$ is obtained by sampling on straight lines between points in p_x and p_g. So sampling a $\hat{x} \sim p_{\hat{x}}$ means sampling $x_0 \sim p_x$, $x_1 \sim p_g$, $\lambda \sim Un(0, 1)$ and computing $\hat{x} = \lambda x_0 + (1 - \lambda)x_1$.

If we keep G constant, and train the discriminator D until it maximizes this loss, we have found an approximation of the supremum we were looking for, and the loss value will be almost equal to the value of $W(p_{\text{data}}, p_g)$. In this context, the optimal generator G returns the exact data distribution that we started with. Then $W(p_{\text{data}}, p_g) = 0$. Note that $W(p_1, p_2)$ can never be smaller than 0, since we can always take f to be a constant function, which is certainly 1-Lipschitz, so the supremum is at least 0.

In practice, one never trains D to convergence for each training step of G. This

[2] [8] defines this in somewhat more complicated terms. To have the notation in line with the other formulas, we use the definition from [9].

is often not feasible computationally. Moreover, stochastic gradient descent is not guaranteed to find the absolute maximum and so the situation described above is unlikely to occur.

Chapter 3

Methodology

3.1 Quality Measure

The goal of this part of the research is to define an objective quality measure for kick drum samples, that can be used to formally evaluate a model. This should give more direction to the model evaluation metrics that a generic distance measure can not provide.

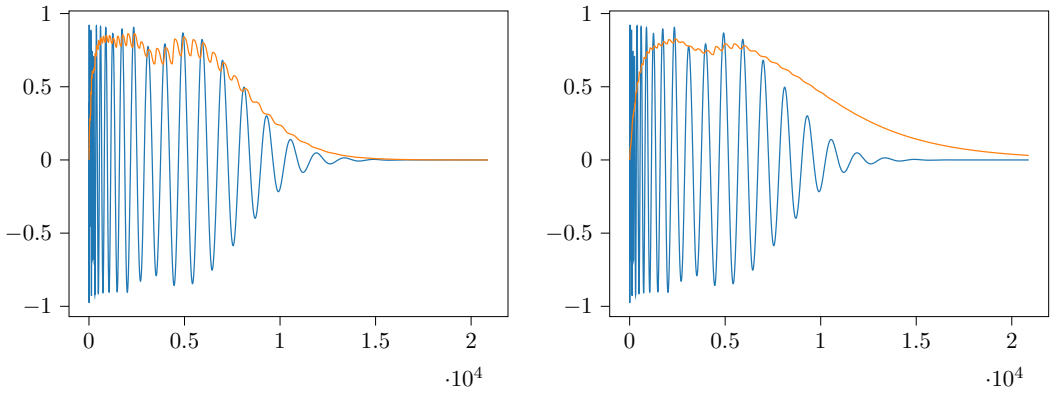
3.1.1 Decay Envelope Modelling

An important part of how a kick drum is perceived is the way the volume decreases over time. This is referred to as the *decay envelope* of the kick drum, whereas the very start of the kick drum is the *transient*. The transient is where most of the high frequency content is concentrated, and it the part that pokes most through a mix. In this paragraph, we evaluate the smoothness of a decay envelope by looking at the peaks of the waveform.

Two common methods to model envelopes in signal processing are to employ a windowed RMS value, or an envelope follower. The first is used to estimate perceived loudness on a short timespan, while the second responds better to peak loudness. Both are often used to make other processes in a chain respond to the input's envelope, e.g., in dynamic range compressors or limiters [10, 11].

In the case of kick drums, an envelope follower's output will often differ significantly from the visual shape of the waveform (see figure 3.1). This is mainly because the timescales of the lowest frequencies in a kick drum cause these methods to overfit to the actual oscillations instead of the overall shape of the volume, and increasing their scope makes them overshoot instead.

This gives the following insight: the shape of the envelope is found in the peaks, and not in the troughs. The weakness of the above methods is that they also process the troughs. The following method uses this insight and fits a global shape onto the kick



(a) Overfit, attack=5ms, release=100ms (b) Overshoot, attack=20ms, release=400ms

Figure 3.1: Examples of an envelope follower overfitting and overshooting the waveform shape.

drum that can then be used to detect local deviations from that shape.

First, detect all peaks in the waveform, we do this using Scipy's `find_peaks` algorithm, with a width of 64 samples and a minimum height of 10^{-2}. Adding this width constraint instead of just detecting each point at which the slope switches direction ensures that there are at most two peaks detected for each oscillation of the fundamental frequency.

Since the peaks are both negative and positive, we take the absolute value and use the method of least squares [12] to fit the following sigmoid function to the detected peaks:

$$\sigma(t, L, t_0, k) = \frac{L}{1 + e^{-k(t-t_0)}}$$

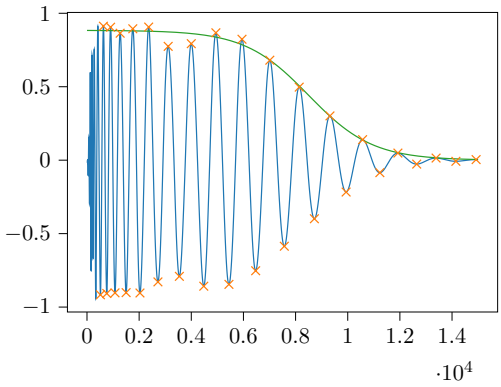
In short, the aim of the least squares method is to find values for the parameters L, x_0 and k such that if (t_i, x_i) are the peaks, we minimize the following sum:

$$S = \sum_{i=1}^{N_{\text{pks}}} (|x_i| - \sigma(t_i, L, t_0, k))^2$$

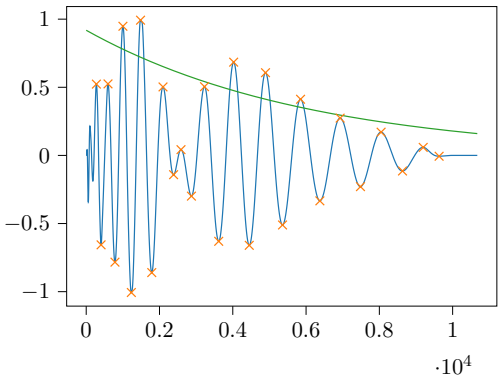
To get faster convergence in the least squares method we first apply a low-pass filter at 1 kHz to the kick drum, such that the often noisy high frequency contents of the transient do not generate too much randomness in the peaks that skews the sum in the wrong direction. After all, we are currently interested in the decay, not the transient. We then initialize the least squares method with the following parameter values:

$$L = \max_i x_i, \quad t_0 = \text{median}_i(t_i), \quad k = -1 \cdot 10^{-4}$$

If the least squares method converges, the quality of the kick drum decay Q_1 is returned as the mean squared error (MSE) of the fitted curve to the peaks, which



(a) MSE $\approx 9.88 \cdot 10^{-5}$
Smooth envelope which shows the sigmoid shape.



(b) MSE $\approx 4.60 \cdot 10^{-2}$
Bad envelope which shows an interference pattern.

Figure 3.2: Examples of the least squares method fitting the sigmoid to a good (a) and bad (b) sample.

is just $\frac{S}{N_{\text{pks}}}$. However, there are many cases where the method takes very long to converge, and as such we set a limit to the amount of iterations. If this limit is exceeded, we just return the MSE of the curve at the last iteration.

3.1.2 Flatness of the Frequency Response

It has long been known in the study of human perception that the ear does not perceive every frequency in the same way. In particular, lower frequencies are perceived to be less loud than higher frequencies played at the same amplitude. Since the early 1930s, researchers have produced various sets of curves that intend to map the frequencies of the human audible spectrum to a relative level such that all frequencies are perceived by the ear as being equally loud.

A *weighting filter* is an audio filter that is used to weight each frequency before analyzing the loudness of a signal. There are many different weighting filters, each with their own applications. Of interest in this case is the A-weighting filter, which is based on the Fletcher-Munson [13] curves and is intended to correlate well with human perception, as well as the K-weighting filter, which was intended for loudness normalization in TV broadcasts (and more recently is also used on streaming services like Spotify).

The A-weighting filter is less applicable here, because of the aggressive rolloff of high frequencies. In practice one never sees a musical signal that actually matches the inverse of this A-weighting curve. Furthermore, the A-weighting filter has been critisized for only being accurate for quiet pure tone signals, as it was based on the 40-phon Fletcher-Munson curve.

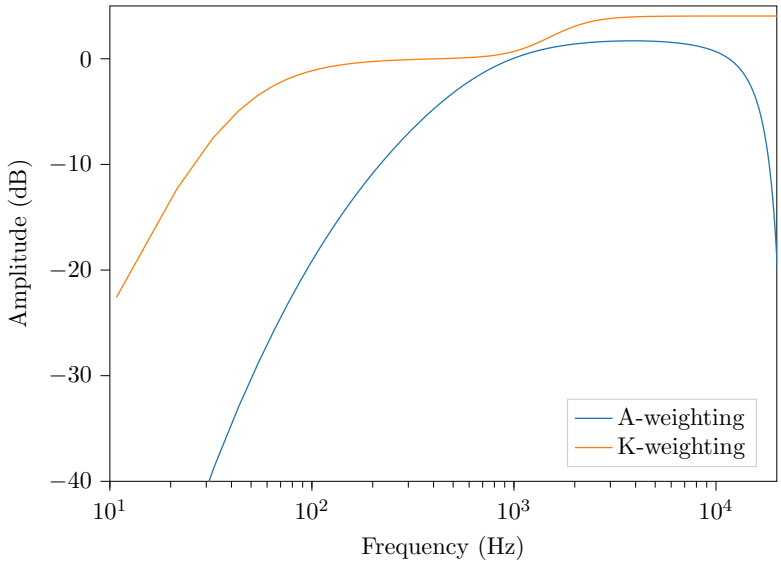


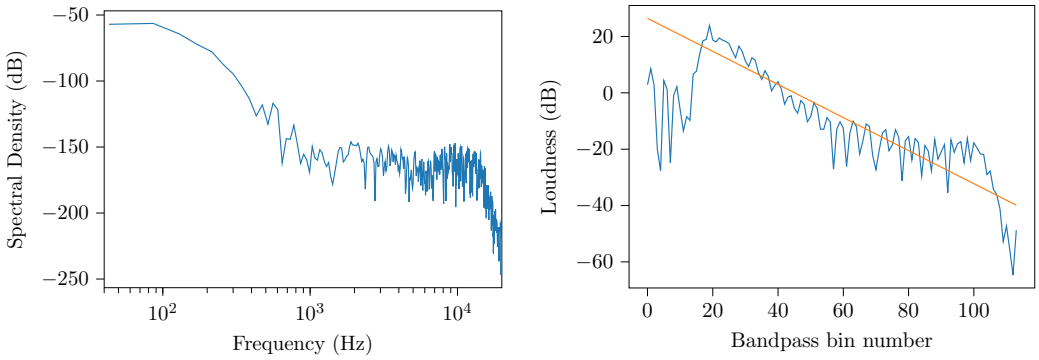
Figure 3.3: Frequency response of A-weighting and K-weighting filters.

From our experience, the K-weighting filter more closely matches the perception of louder wideband signals like noise, a full mix or in this case a kick drum. An easy way to try this[1] is to apply the inverse[2] of the K-weighting and A-weighting filters to a white noise sample, and playing them back to hear which of the two sounds more flat.

We apply the K-weighting filter to the kick drum signal before evaluating the flatness of the frequency spectrum. With *flatness* it is meant here that the signal is perceived to have the same amount of energy in every part of the audible spectrum. We determine this using the Constant-Q Transform (CQT), which divides the frequency spectrum using bandpass filters, and calculates the loudness for each band. The constant value Q is the ratio of the center frequency to the bandwith of each band. An example value for this is $Q = \frac{1}{\sqrt{2}}$, which yields a bandwith of precisely an octave [14].

Having obtained the CQT spectrum (in dB), we procede, similarly to the previous section, by fitting a linear curve through the spectrum. The slope of this curve, and the deviation of the spectrum from this curve, will define the quality. This fitting is again done using the least-squares method, which returns the slope a, the offset b

[1] Assuming you have access to a Digital Audio Workstation (DAW). There are many free options.
[2] Or dial in an EQ curve that comes close enough.



(a) Original power spectrum.

(b) CQT after K-weighting.

Figure 3.4: Example PSD and CQT of a kick drum sample. The linear fit has slope $a \approx -0.587$.

and the residue r. The quality of the frequency response is then defined as:

$$Q_2 = \begin{cases} -(a+\mu) + r/\sigma & \text{if } (a+\mu) \leq 0 \\ 2(a+\mu) + r/\sigma & \text{if } (a+\mu) \geq 0 \end{cases}$$

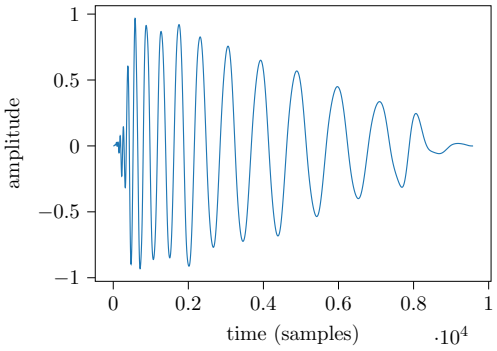
Where $\mu = 0.4$ and $\sigma = 250$ are used to weight the different slope and residue values. A μ of 0.4 means that a CQT spectrum with a downward slope of $a = -0.4$ dB/bin will get the best quality. We further punish upward slopes by doubling the value in those cases. The values for μ and σ were chosen by listening to samples from the dataset and contrasting them against their slope and residue values.

3.1.3 Finding the Fundamental Tone

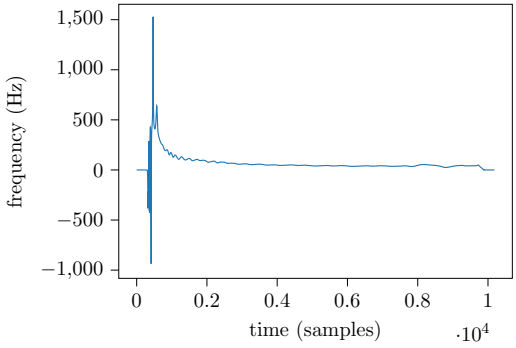
A kick drum's decay is meant to occupy the lowest frequencies in a piece of music. In this section we define a quality feature by looking for the fundamental frequency. The first thought when tasked with finding the frequency of a particular signal, is to use the Fourier transform and then find the first peak. However, it is difficult to get a precise measurement because the FFT subdivides the frequency spectrum into bins, and the amount of bins in the first 100 Hz is always small compared to the total number of bins.

Instead, the Hilbert transform can be used to approximate the phase at each point in the signal, and then we can determine the *Instantaneous Frequency (IF)* by taking the derivative [15]. Let $x(n)$ denote a discrete signal, and H the Hilbert transform, then this calculation can be written as follows:

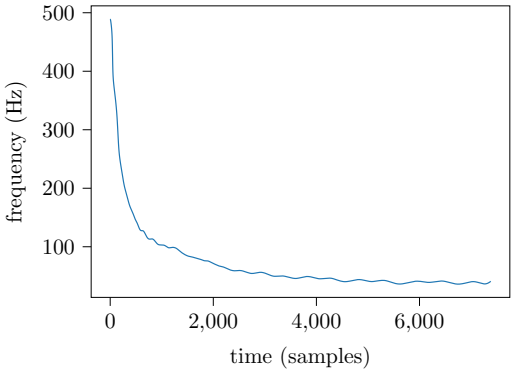
$$y(n) = H[x](n) := A(n)e^{\phi(n)i}$$



(a) Lowpassed kick drum signal $x(n)$



(b) Instantaneous Frequency $f_i(n)$



(c) Approximated pitch envelope $f_p(n)$

Figure 3.5: Example of finding the pitch envelope of a kick drum. This evaluates to $f_e \approx 38.87$, which is an E♭. Notice the IF is very inaccurate, even negative, at the very start.

The Hilbert Transform returns a complex-valued signal, and we want to know the phase component ϕ, so we take the complex argument:

$$\phi(n) = \arg y(n)$$

Now we take the discrete derivative to obtain the Instantaneous Frequency f_i:

$$f_i(n) := -\frac{f_s}{2\pi}(\phi(n) - \phi(n-1))$$

The Hilbert transform is perfect in theory, but in practice we can only get finite precision, and as such we still need to account for a noisy signal. Therefore, we apply a low-pass to the kick drum signal before applying the transforms, and we also take a moving average of the instantaneous frequency f_i to produce the approximated pitch envelope f_p.

The pitch envelope of a kick drum will be mostly decreasing, and so we estimate the fundamental tone by looking at the lowest point in the pitch envelope. The estimated pitch f_e is then returned as the average pitch value in a window of 1000 samples around the lowest point.

$$f_e = \frac{1}{1000} \sum_{k=-500}^{499} f_p(k + \text{argmin} f_p)$$

We then define the pitch quality Q_3 by linearly interpolating between the frequencies 20 Hz, 25 Hz, 65 Hz and 100 Hz, such that an f_e between 25 and 65 gets the best quality, and the quality decreases outside of this range.

$$Q_3 = \begin{cases} 1 & \text{if } f_e \leq 20 \\ 1 - \frac{1}{5}(f_e - 20) & \text{if } 20 \leq f_e \leq 25 \\ 0 & \text{if } 25 \leq f_e \leq 65 \\ 1 - \frac{1}{35}(100 - f_e) & \text{if } 65 \leq f_e \leq 100 \\ \frac{1}{100} f_e & \text{if } f_e \geq 100 \end{cases}$$

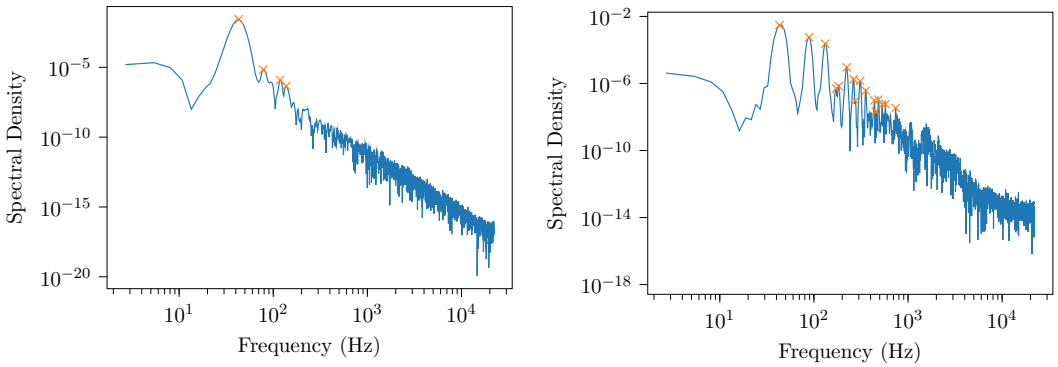
3.1.4 Measuring Distortion

The last feature we want to consider for our quality measure is *distortion*. Before we define this term, let us first consider the concept of *overtones*. Musically, a (harmonic) overtone of a frequency is any integer multiple of that frequency. The particular pattern of overtones is what gives an instrument its character. In a tonal signal, the fundamental tone is the first harmonic, and the n-th harmonic is at n times the fundamental's frequency.

In signal processing, *distortion* is often measured by comparing a signal before and after applying a certain processing chain. For example, one might be interested in measuring how transparent an analog amplifier is, and so researchers will measure its effect on a pure sine wave signal. Any overtones present in the output are considered to be distortion. If V_i is the amplitude of the i-th harmonic, then the *Total Harmonic Distortion (THD)* [16] is defined as:

$$\text{THD} = \frac{\sqrt{V_2^2 + V_3^2 + ...}}{V_1}$$

However, the THD is always computed based on a reference signal. In the case of kick drum signals, we do not have such a reference. Furthermore, we want to measure how distorted the kick drum sounds, which isn't always the same as measuring the exact volume of each of the overtones. Consider for example a white noise signal; for any reference signal, the THD would be very large, but the noise does not "sound" distorted because all of the frequencies are perfectly masked.



(a) Clean fundamental tone.
$Q_4 \approx 2.35 \cdot 10^{-4}$

(b) Fundamental tone not clean.
$Q_4 \approx 1.93 \cdot 10^{-1}$

Figure 3.6: Example of two spectra with peaks highlighted.

Moreover, the intuitive way to compute the THD is to use the Fourier transform. This is more than adequate for a signal with static pitch, but for kick drums, where the pitch changes over time, we run into some issues. Because both the fundamental frequency and its overtones move with the pitch, we see that the DFT averages some overtones to one frequency bin above or below an integer multiple of the fundamental.

So, all of these observations in mind, we forgo the notion of overtones and just look for any strong peaks in the frequency spectrum. Using the same peak finding algorithm [12] as mentioned in section 3.1.1, with prominence and height adjusted according to the range of peaks found in the dataset, we obtain a list of peaks $P_1, ..., P_k$ and define the "Inharmonic Distortion" quality measure Q_4 by

$$Q_4 = \frac{\sqrt{P_2^2 + P_3^2 + ... + P_k^2}}{P_1}$$

The frequency spectrum that is analyzed here is obtained by applying the Welch method [12, 17] on the sample, but starting at a fifth of the duration. We start this late to avoid the biggest part of the pitch envelope, and so obtain more accurate peaks.

3.1.5 Putting it all together

So far we have defined four seperate quality features, Q_1 through Q_4. Not all of them occupy the same range of values when applied on the dataset, and they are not all equally important. So, we have to decide on a set of weights to apply to the features.

Before we do that, we define one final quality feature, a really simple one, based

on the RMS value of the kick drum. This is just intended to measure the overall loudness. Note that each sample will be peak normalized before calculating the features.

$$Q_5 = 1 - \text{RMS}(sample)$$

We now give the weights for the final quality measure:

$$Q = 10Q_1 + 0.5Q_2 + Q_3 + 2Q_4 + 0.5Q_5$$

These weights were decided on by inspecting the performance on the dataset, as well as some examples of bad kick drums. The aim was to get the measure to evaluate good kick drums with a quality somewhere between 0.0 and 1.0, so that any value higher than 1.0 would indicate a flawed sample.

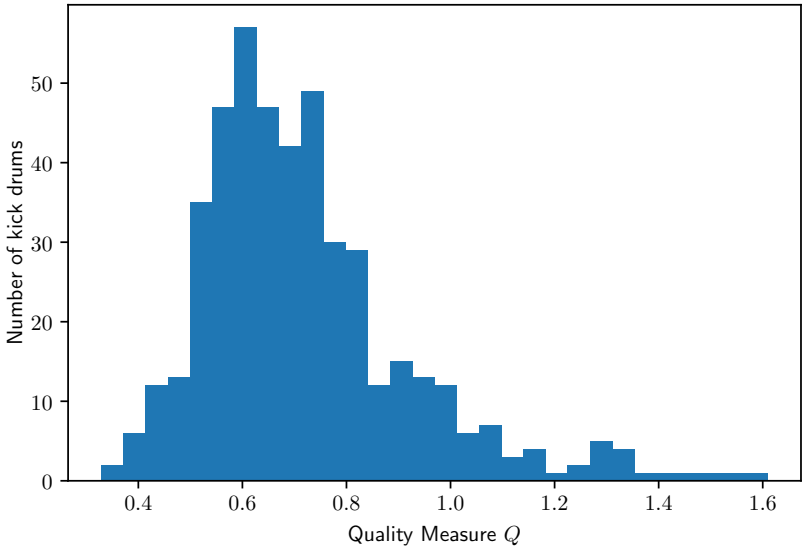


Figure 3.7: Histogram of quality measure on the dataset.

In Figure 3.7 we see that most of the kick drums from the dataset have a Q between 0 and 1, with some outliers. It is hard to disagree with the quality measure in labelling these outliers as flawed. However, when inspecting these samples, the biggest contributor to their high value is the fundamental frequency being below 25 Hz. We keep these samples in the dataset because they can be useful as top layer in our data augmentation. (See section 3.2)

Seeing how the quality measure performs on the dataset is one thing, but it is also important that the quality measure assigns a bad score to recordings that are *not*

Quality Q	Sample 1	Sample 2	Sample 3
Snare drum	3.145	3.611	2.848
Conga drum	2.188	2.893	.
Piano note	1.866	1.926	3.424
Pink noise	1.915	2.079	1.898
Bird sounds	2.426	2.080	.

Table 3.1: Quality values of some samples that are not kick drums. Where necessary, sounds were shortened using sigmoid-like envelope shapes.

kick drums. We tried some examples and list their quality values in table 3.1.

We note that a snare drum, which is very similar to a kick drum but with a higher pitch, gets a worse score than bird sounds, which are completely unrelated. In the scope of kick drums, it makes sense to give a snare and conga drum these quality scores, but the scores for the other samples do not really fit into the scale.

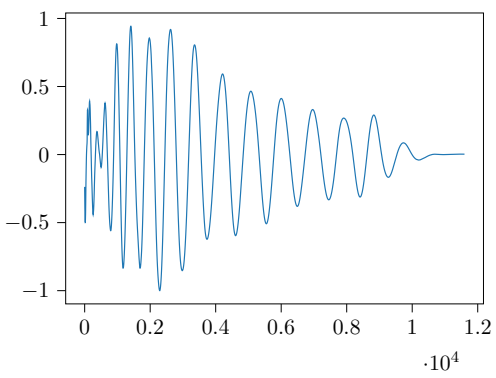
In particular, the pink noise seems to fool the algorithms for $Q1$ and $Q3$, and so we should be somewhat careful when computing the quality measures over a neural network's output early in the training, as these are often very much like noise.

3.2 Data Augmentation

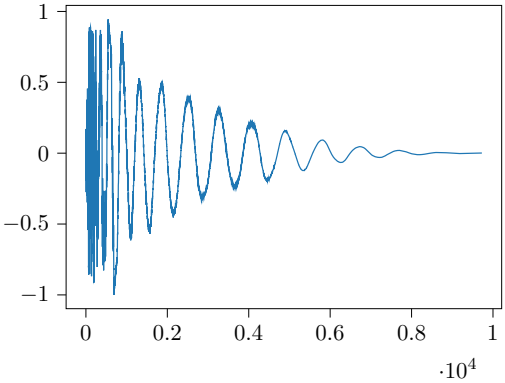
A technique ubiquitous in music, from classical to modern pop, is that of *layering*. This is when multiple similar sounds are combined, such that the result fits the piece better than the originals did. In an orchestra, the cellos and violins can become layers of the same sound by playing the same melody lines. In electronic music, the same principle is applied to drum samples.

However, one needs to take more care in this latter case. The cello and violin are designed to occupy different parts of the spectrum. Whereas if you carelessly layer two kick drums, clashes are bound to occur. The biggest issue you will run into is phase cancellation in the lowest frequencies. To avoid these clashes, practicioners will often apply a highpass filter to one layer and a lowpass filter to the other. Keeping the low end character of one kick drum, and adding the high end texture of another.

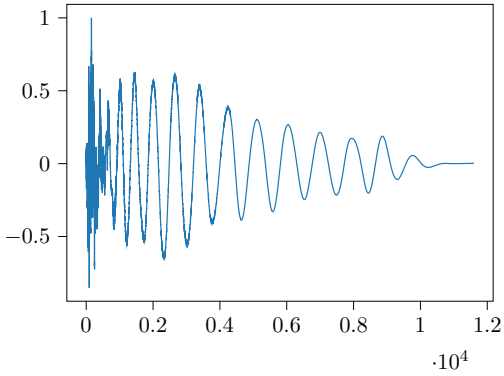
We take inspiration from this technique to produce our method of data augmentation. We produce two versions of the dataset, one sorted by the quality combination $5Q_1 + Q_3 + Q_5$ and another by the combination $Q_2 + Q_4$. The first mostly describes the low end, and the second mostly describes the high end. We take the first half of both sorted versions to obtain two mostly non-overlapping sets. We can then take



(a) Sample 1, $Q \approx 1.003$



(b) Sample 2, $Q \approx 0.753$



(c) Sample 1 and 2 layered, $Q \approx 0.665$

Figure 3.8: Example of layering where the combination is "better" than the seperate parts.

the cartesian product of these halves and combine each tuple by lowpassing the first element, and highpassing the second element.

Lastly, we prune any samples with a bad enough quality. In this way, we take the ~ 450 sample dataset and obtain an augmented dataset of over 50 thousand samples. We can do this efficiently by applying the augmentation during training, storing only as many augmented samples in memory as are needed for a single batch.

This method is also not limited to a single cutoff frequency. Here we chose 400 Hz, but any value between 300 Hz and 600 Hz could work as well. So, if needed one could also allow multiple cutoff frequencies in the augmentation and thus obtain an even larger, or more varied, augmented dataset. The samples generated will be similar, but never exactly the same.

Listing 3.1: Pseudocode for layering augmentation

```
function layer(sample1, sample2, freq):
    # Butterworth filters
    lp = butter_lp(sample1, freq, order=3)
    hp = butter_hp(sample2, freq, order=4)
    return normalize(lp + hp)

low = sort(dataset, key=Q1)
high = sort(dataset, key=Q2)

N = size(dataset)
augmented = []
for i = 0 to N/2:
    for j = 0 to N/2:
        sample = layer(low[i], high[j])
        if quality(sample) < 1.5:
            append(augmented, sample)
```

3.3 Other Measures

In section 3.1 we have defined a quality measure that is computed over a single instance of a kick drum. When it comes to evaluating models, it is also important to compare the perceptual similarity of the output to the original dataset. Otherwise, one could construct a "model" that simply returns the sample from the dataset with the best quality score on each run, and so achieve the best possible model evaluation.

It is not sufficient here to take a simple distance measure between your input and output, as this does not match perception. To understand this, consider the following example: We generate two 1-second long white noise signals at 44.1 kHz, and compute the Euclidean distance in $\mathbb{R}^{44100}$ between these signals. Each sample in the signals is a random number between -1 and 1, and so the Euclidean distance will just be the sum of 44100 random numbers between 0 and 4. Yielding a distance of around 170 to 172 on most runs. Whereas a human listener would be unable to tell the two apart!

A solution to this, used by many papers in image or audio synthesis, is to apply a strong external audio classification network to the problem. This classifier has not been trained on any of the to be evaluated data. Then one can compute distances based on the internal represesentation that model uses to classify. The intuition is that if the model is able to accurately classify audio, its internal representation must hold a lot of compressed information about the input.

The Fréchet Audio Distance (FAD) [18] computes the similarity between the inputs and outputs of an audio processing system by comparing their 128-dimensional

feature embeddings computed by the VGGish model[3]. The similarity is essentially computed as the Wasserstein-2 (see also section 2.5) distance between the embedded distributions. These are however assumed to be multivariate normal distributions $p_1 = \mathcal{N}(\mu_1, \Sigma_1)$, $p_2 = \mathcal{N}(\mu_2, \Sigma_2)$ and so the formula can be simplified to:

$$F(p_1, p_2) = ||\mu_1 - \mu_2||^2 + \text{trace}(\Sigma_1 + \Sigma_2 - 2\sqrt{\Sigma_1 \Sigma_2})$$

We can now conduct the experiment with random white noise signals, but using the FAD. When we use the FAD to compare 1500 seconds of white noise, the resulting distance is 0.004884. Which is indeed a very small distance compared to the values seen in [18].

[3]https://github.com/tensorflow/models/tree/master/research/audioset/vggish

Chapter 4

Experiments

We now move on from the preporatory methodology to the actual task at hand: training and evaluating models. We compare models that have their open-source code available in a state that is (close to) runnable. Small changes to code to deal with compatilibity issues are fine, but rewriting large parts of code just to do a single experiment was outside of the scope of this research. We compare the results of this chapter to other related work we did not attempt to run in chapter 5.

All of the models in this chapter were trained on a single GTX 2080 Ti with 11GB of VRAM. This limits the size of the models, but eliminates the need to rewrite models to train on multiple GPUs, and ensures that all models got the same resources which enables a fair comparison. We focus mostly on PyTorch [19] implementations, as this reduces compatibility issues compared to Tensorflow[1] [20]. Links to the respective GitHub repositories and our final source code are documented in the appendix, which also contains more configuration details such as the hyperparameters we used in each of the experiments.

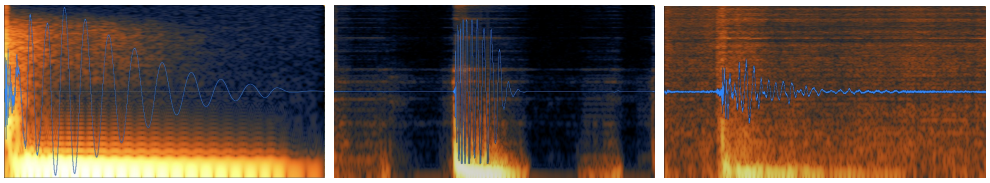
A couple of audio examples from each section are hosted on an accompanying web page[2]. We recommend listening to them with headphones or good speakers, as differences in the low frequencies are otherwise hard to pick out.

4.1 Model Descriptions

4.1.1 WaveGAN

WaveGAN [9] approaches the problem of audio synthesis by processing the waveforms through a deep convolutional GAN (DCGAN) [4], where the input is reshaped to a 2D array instead of a 1D array, and then 1D convolution is applied to each channel. At each layer in the generator, the amount of channels is halved, and the

[1] The release of Tensorflow 2.0 means any code from before 2019 needs to be updated.
[2] https://dev.wdamen.com/kicks/



(a) (b) (c)

Figure 4.1: Combined spectograms and waveforms of a sample from (a) the dataset, (b) WaveGAN with phase shuffling and (c) WaveGAN without phase shuffling. Both after around 8 hours of training.

length of each channel is quadrupled. Until at the last layer there is only 1 channel (or 2, in case of stereo signals) and each channel contains the waveform outputs. The discriminator does the same, but in the opposite direction.

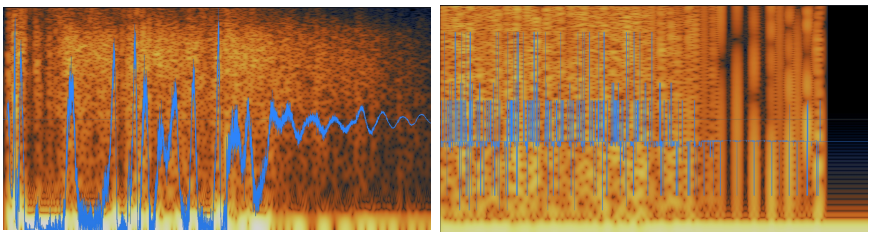
What is unique about WaveGAN compared to other audio GANs, is the addition of "phase shuffling" layers in the discriminator network. These apply a randomized shift to each channel at that stage in the network. Figure 4.1 shows the results with and without these layers enabled, and it is clear that phase shuffling greatly reduces the noisyness.

According to the authors, this phase shuffling is to stop the discriminator from detecting artifacts at a specific phase. If we disable the phase shuffling, the generator is forced to add a lot of noise to its output to mask those artifacts.

We make the following two changes and consider the resulting model to be our baseline in the rest of the experiments:

Padding Policy We also see in figure 4.1 that the samples from WaveGAN are centered in the frame instead of starting at the start. This is because all input samples from the dataset are padded to fit the input size. We changed the padding method so that the padding is only added at the end, as we want all kick drums to start at the same time.

Increased Sample rate As with most other research in this area, the WaveGAN model was initially meant for a sample rate of 16 kHz. To increase this to 44.1kHz, we follow the updated Tensorflow version on GitHub and add a sixth layer to the PyTorch version to increase the output size to 32768 samples, and interpret this as $\sim$0.74 seconds of audio sampled at 44.1 kHz. The resulting architectures are documented in the appendix.



(a) (b)

Figure 4.2: Combined spectograms and waveforms of a sample from (a) WaveRNN and (b) WaveNet

4.1.2 WaveRNN and WaveNet

We will now have a look at two recurrent models, WaveRNN [21] and WaveNet [22]. These both were originally intended for speech synthesis, but WaveNet has also been tested on musical applications [23, 24]. We tried running WaveNet and WaveRNN on both our non-augmented and augmented datasets, but did not get anything that resembles a kick drum as output.

We found that it is hard to get the receptive field large enough within the memory limitations of our single-GPU training environment. The lowest frequencies of our data would have periods close to or larger than the size of the receptive field, and we believe this caused the poor results. Perhaps with more training data, more training time and a larger model size these models would stand a chance to compete with the GAN models, but within the scope of this research we chose to discontinue experimenting on these recurrent models.

4.1.3 Progressive Growing for WaveGAN

With the limited success of the recurrent models, we return to the WaveGAN model and try to improve its results. In this section we propose a progressive growing scheme for WaveGAN. A Progressively Growing GAN (PGAN) learns one layer at a time, starting at low resolution and increasing the resolution over time. It is responsible for some of the largest advances in image synthesis [25, 26].

The way that one reduces the resolution of digital audio is through resampling. The Shannon-Nyquist theorem tells us that if we reduce the sample rate, we lose part of the frequency spectrum. This is why progressive growing intuitively makes perfect sense for this application; we first train the model to generate an accurate low end, and then gradually add higher frequencies.

The intermediate layers of WaveGAN have 2-dimensional shapes, and only in the last layer does the output become 1-dimensional. This choice of architecture forces the layers to be added at the start of the generator and at the end of the discrimi-

nator, which is the opposite of what happens in a regular PGAN. The implications of this on the intermediate model's architectures is documented in the appendix.

# of layers	Output length	Sampling rate (Hz)	Nyquist frequency (Hz)
1	32	43.07	21.53
2	128	172.26	86.13
3	512	689.06	344.53
4	2048	2756.25	1378.13
5	8192	11025	5512.5
6	32768	44100	22050

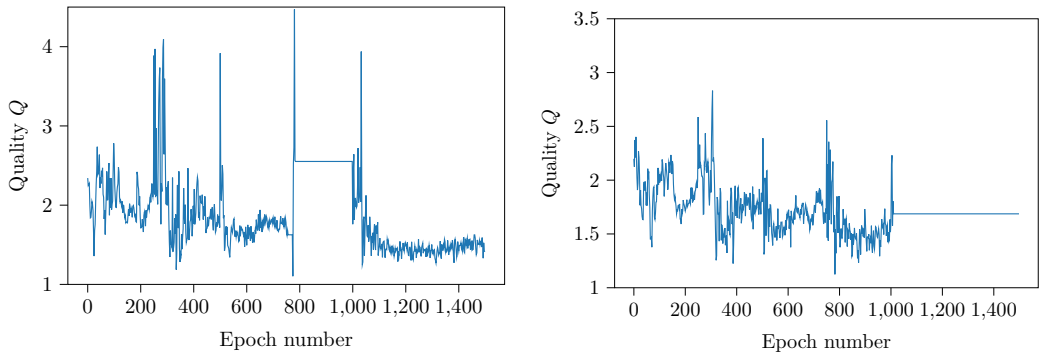
Table 4.1: Output sizes and sample rates for each progressive layer step.

Using just the first layer would have the model operating on mostly inaudible frequencies, so we initialize at 2 layers. Then, every N epochs another layer is added, until we reach 6 layers, at which point the rest of the epochs are spent training the full model.

The first run of this progressive WaveGAN version immediately produced better samples than before. However, on subsequent runs we noticed some numerical instabilities at the start of each layer. Sometimes, the newest layer gets stuck at a converged state right after that layer is added. Only on the next layer will the model continue learning. If this happens in the final layer, most of the training is completely wasted. We consider two different ways to solve this problem:

Layer Fading The way that these instabilities are resolved in [25], is by linearly fading in the newest layer. The output from the previous layer is upsampled using interpolation and combined with the output of the new layer. However, the fact that the newest layer is added at the start in our case makes implementing this fix more complicated. We essentially need to apply the entire model twice, once with the newest layer, once without, and then perform upsampling and weighting. This considerably slows down training. We refer to this as α-fading, because the constant used to weight the old and new layers is denoted by α in [25].

Learning Rate Schedule We try an alternative way to fix the instabilities. Each time a new layer is added, we decrease the Adam learning rate to a tenth of the original value, and linearly increase it to return to the original value halfway through the training of the layer. This also seems to prevent the model from getting stuck at the new layer, and avoids the speed penalty of the layer fading method. We call this method η-fading, as the learning rate is often denoted by η. We are not aware of any progressive GANs that use this method, but the idea of using a learning rate schedule is certainly not new [27].



(a) Stuck at layer 5

(b) Stuck at layer 6

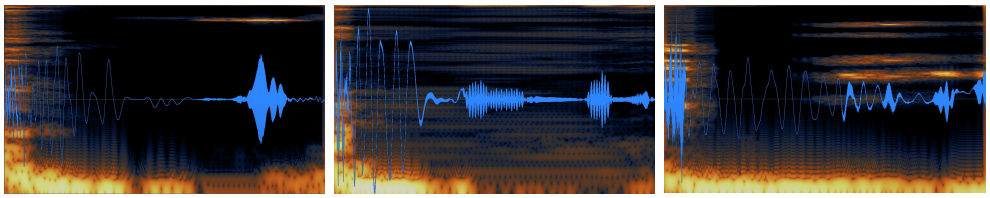
Figure 4.3: Numerical instabilities: Average quality measure evaluation of 20 samples for each point. Measured every other epoch.

4.1.4 Spectral Representations

The authors of WaveGAN also included another model they called SpecGAN, which computes a spectrogram representation of the input data, and trains like an image net to generate new spectrogram images. These are then converted back to audio using the Griffin-Lim [28] algorithm.

A concern we had with the idea of using spectrogram representations for generating kick drums is that the Fourier Transform tends to be innacurate around the edges of a signal. Most STFT (Short-time Fourier Transform) implementations will use a windowing function, fading out the edges of each processing block. Therefore, the very start of the kick drum will be faded in before computing the spectrogram representation. This part of the signal, the transient, is very important here, moreso than in other situations like speech synthesis or melodic instruments.

The authors already found in their paper that WaveGAN outperforms SpecGAN, and as such it is not surprising that our Progressive WaveGAN would also perform better. The samples SpecGAN produced were very distorted in comparison to WaveGAN, as if the image net is generating many overtones that are not present in the original data. Furthermore, it did not seem to improve much after the first few hours of training (see figure 4.4).



(a) (b) (c)

Figure 4.5: Combined spectograms and waveforms of a sample from (a) complex after 27.6h of training, (b) mag-if after 10.5h of training and (c) a 24h+ mag-if run that we ended up discarding.

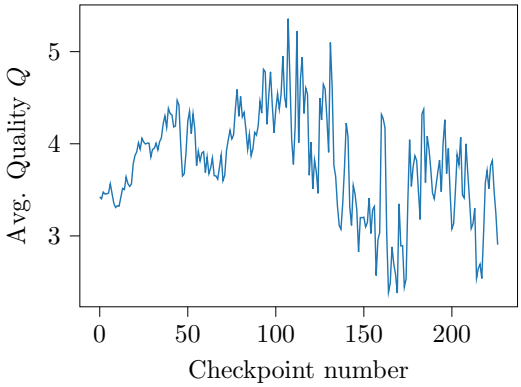


Figure 4.4: Average quality measure evaluation for SpecGAN, 20 samples for each point. Checkpoint every ∼5 minutes.

A more recent paper, [29], applies a standard PGAN architecture to the task of generating these spectrograms, and also compares numerous different ways to represent the audio in such a spectrogram. We refer to these models as 'Nistal PGAN'. They conclude their best performing representations are a complex-valued STFT (complex), and an STFT with a magnitude and instantaneous frequency component (mag-if). We compare both of these representations when trained on our augmented dataset.

For the two spectral representations, we notice artifacts that are more pronounced than in WaveGAN (see figure 4.5). The random generation is also less consistent, the same trained model sometimes generating samples that are not like kick drums at all.

We also tried the 'waveform' representation, but we did not manage to get anything better than a single click. This again shows how the WaveGAN architecture is uniquely suited for the task of modeling raw audio. Neither the recurrent networks nor this more standard PGAN can compete, at least within our constraints.

4.2 Quality Measure Comparison

After training each of the models, we compare the quality measure evaluation of 100 randomly generated samples. We use the WaveGAN without progressive growing trained on the original dataset as our baseline, and also include the statistics for the dataset and a subset (size ∼6.5k) of the augmented dataset. Besides this baseline, all other models were trained on the augmented dataset. The FAD is computed between the embeddings of the model samples and the original dataset.

Model Name	Training Time	Avg. Q	Min. Q	Max. Q	FAD
Dataset	-	0.720	0.329	1.610	-
Augmented	-	0.650	0.428	1.420	1.648
WaveGAN (Baseline)	24.5h	0.911	0.656	3.369	5.302
WaveGAN (augment)	23.0h	0.918	0.581	2.003	3.591
WaveGAN (α-fading)	65.7h	0.689	0.493	1.567	5.617
WaveGAN (η-fading)	36.8h	0.702	0.471	2.602	3.891
Nistal PGAN (mag-if)	10.5h	1.122	0.684	2.919	7.449
Nistal PGAN (complex)	27.6h	1.795	0.723	4.125	8.842

Table 4.2: Quality measure statistics for the different models.

From the values in the table we see that the spectral representations are outmatched by the WaveGAN versions in every column, except for the training time. Curiously, we also did a run for the PGAN mag-if that took over 24 hours, but the results we got were actually far worse than the 10.5 hour run. We also see in the FAD column that the spectral artifacts give a larger distance to the original data than the noisy artifacts in waveform processing do.

What is also surprising, is that the progressive variants of WaveGAN actually produce a lower average Q than is found in the original dataset. The caveat here is that the quality measure is not very good at picking out the artifacts in the generated samples. This is also reflected in the FAD column, which shows that all models have at least a distance of 3.5, which is a significant distance on the scale in [18].

4.3 Listening Experiment

Our final experiment is a small listening experiment. We do this to investigate to what extent listener's perceptions match the quality measure. We also include samples generated by WaveGAN and Nistal's PGAN in this experiment, and thus obtain some information about which models the listeners prefer.

The listener is presented with pairs of samples, drawn from two datasets, and is asked to choose the one they prefer. The specific question we ask is "Which of these samples sounds more like it came from a paid sample pack?". We compare the following conditions in this experiment:

1. Original dataset
2. Augmented dataset
3. A set of 'bad' samples.
4. WaveGAN (Baseline)
5. WaveGAN (η-fading)
6. PGAN mag-if

The pairs are generated out of two pools, the data pool $(1, 2, 3)$ and two times the model pool $(4, 5, 6)$, in which we take the cartesian products. For the model pools, we leave out the (i, i) pairs. We further add three pairs $(1, 4), (1, 5), (1, 6)$ to compare the models to the original data. For each pair, a sample is selected randomly from each condition. After generating the pairs, we randomly choose to repeat 20% of the pairs, and then shuffle the order. Giving a total of 28 pairs, 4 of which are repetitions.

This seperation of pools was chosen so that we can focus more on how the models compare to each other, by only comparing each model to the dataset once. Since we expect the data to win over the models anyway, it would be less productive to also compare the augmented dataset to the model samples.

4.3.1 Quantitative Results

The experiment ran for a couple of days, and we managed to get 29 fully filled out responses. Leaving out repetitions, this means $29 \cdot 24 = 696$ pairs were judged. We additionally had 6 participants who did not finish the experiment, with a total of 34 judged pairs.

We are interested in both the level of agreement of the listeners with the quality measure, and also the level of preference for each of the models relative to each other. Whenever the listener picks the sample in a pair which has the lower Q value, we say that the listener 'agrees' with the measure for that pair.

Leaving out repetitions, we find agreement in 60.7% of cases. This is significantly better than random guessing, but it is not an amazing score. However, this is without taking into account that some pairs are just hard to decide. For instance, if we only look at pairs where the distance in quality ΔQ is bigger than 1, we actually find 73.3% agreement. This gives rise to the graph in figure 4.6.

Win %	Data	Augm	Bad	Baseline	η-fading	mag-if
Data	-	56.67	77.05	84.38	67.74	86.67
Augm	43.33	-	74.19	-	-	-
Bad	22.95	25.81	-	-	-	-
Baseline	15.62	-	-	-	31.45	74.79
η-fading	32.26	-	-	68.55	-	77.5
mag-if	13.33	-	-	25.21	22.5	-

Table 4.3: Head-to-head matrix. Percentage of times the condition in that row got chosen over the condition in that column.

We included repetitions to verify that listeners were consistent in their choices. Most listeners (24 out of 29) made the same choice in all four repetitions. While the other 5 changed their mind twice.

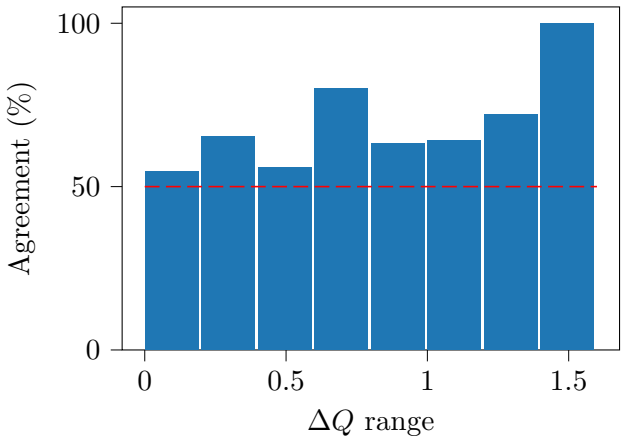


Figure 4.6: Level of agreement in different ranges for ΔQ. Each range is 0.2 wide.

From table 4.3, we see that the three models follow the same ordering WaveGAN-η > WaveGAN baseline > PGAN mag-if as in table 4.2. This is a nice result which suggests that even though the quality measure is a bit inconsistent, it gives accurate results when averaged over multiple samples.

4.3.2 Qualitative Results

At the end of the experiment we asked the listener to describe what aspects they focused on. Unsurprisingly, most of them mentioned the presence of artifacts in the AI generated sounds. Across the responses there were also mentions of the aspects

we described in chapter 3: the transient, the decay or "tail", level of distortion, frequency balance and pitch.

Beyond this, we get into more subjective descriptive terms. We find listeners describing a kick drum as "tight", "punchy", "thumpy" or even "flabby". This is also where we find some disagreement between different listeners. Some listeners say they prefer short and punchy kick drums, whereas other listeners prefer kick drums with more body and bass.

The word "punchy" or "punch" definitely stood out the most here. This mostly has to do with the pitch envelope of the sample. Perhaps the method in section 3.1.3 could be enriched by fitting some exponential decay to the pitch envelope. A kick drum sample with a more defined pitch envelope will be more "punchy".

We also asked the listeners whether they noticed that the experiment contained repetitions. The majority of listeners said they did notice. A couple of them indicated having changed their mind even though they were aware of the repetitions.

Chapter 5

Related Work

In this chapter we discuss some other related work in this area that we did not apply to our learning task. This was either due to time constraints, availability constraints or it simply being outside of the scope for this project. Still, we can compare and contrast our results with the results from this related work, to get a better idea of the current state-of-the-art.

One thing to note right away is that we did not come across any papers in the conducting of this research that used sampling rates of 44.1kHz or above. The highest we found was 22.05kHz, in [30]. So perhaps our goal of generating full-spectrum resolution audio was a tad ambitious compared to this.

5.1 Conditional Synthesis

In this research we almost exclusively focus on unconditional generation of kick drums. Each time we want to generate a new sample, we pick a random latent variable and pass it through the generator network. However, there is an entire class of models that learn to generate multiple different classes of data.

For instance, if you want to model a musical instrument, you want to have control over which note the network generates. This is what is done in [23], [31] and also [29]. All three focus on the NSynth dataset, which was introduced by [23] and contains lots of samples from various musical instruments.

More closely related to our learning task are [32] and [33]. These papers also work on generating drum sounds, but by conditioning on various perceptual features. Their datasets both come from freesound.org, and they both use the same feature set[1]. Though freesound is an easy way to obtain drum samples, they are not always of the highest quality. This is what we sought to avoid with our data selection, as the

[1] https://github.com/AudioCommons/ac-audio-extractor

generated data will at best be slightly worse than the input data. Still, the results[2] of [32] are impressive given what went in.

5.2 More On Spectrograms

In chapter 4 we briefly touched on why we think spectral representations are less likely to work for the specific task of generating drum sounds. We also mentioned SpecGAN and its use of the Griffin-Lim algorithm to convert spectrograms back into waveforms. This is needed because SpecGAN discards the phase component of the STFT calculation, and just uses the magnitude. When converting back to audio, the phase component has to be reobtained by an estimation.

However, [34] advises the use of a different algorithm for phase reconstruction, namely 'Phase-gradient heap integration (PGHI)'. In their paper, the authors note they specifically designed their 'TiFGAN' architecture to be similar to that of SpecGAN, but show that they achieve better performance with the use of PGHI.

In [29], the authors do not discard the phase information for their 'complex' and 'mag-if' presentations. However, for other representations in [29], namely those based on the Constant-Q Transform (CQT), this problem of inversion is even bigger. We further note that [29] also uses Griffin-Lim for their Mel-scale based representations.

5.3 Other Generative Methods

The methods we discussed in section 2.4 and applied in chapter 4 by no means constitute an exhaustive list of generative methods in machine learning. Two methods we have not discussed so far are *normalizing flows* and *diffusion models*. There is some recent work using these methods for audio synthesis, namely [35] and [30].

Diffusion models work by adding noise to the dataset in multiple steps and then teaching a network to reverse each step. Then, one can start with a noise sample and work backwards to generate new data. OpenAI recently published a paper, [36], claiming their diffusion model beats GANs on image synthesis. In [30], this method is applied to speech synthesis, and the model outperforms WaveGAN in unconditional speech generation according to various metrics.

Normalizing flows are in a way a subset of VAEs (see section 2.4.2) in that they learn to encode a latent representation that can later be decoded. However, normalizing flows require each layer to be an invertible mapping. This way, the latent representation can be decoded by the same network that encoded it. Furthermore, it has better probability distribution properties that allow for an improved loss func-

[2] https://sites.google.com/view/drumgan

tion and fast training. This is applied in [35] to model the functionality of a digital synthesizer's parameters.

Chapter 6

Conclusions

From our experiments we conclude that out of the models we tried, our new Progressive WaveGAN model with either η-fading or α-fading performs best on this learning task. We also conclude that spectral representations are not well suited for the task of generating kick drum sounds. That is not to say that spectral representations are useless in audio generation; related work shows that it is certainly useful in melodic and speech synthesis.

As for our quality measure, we conclude that it can aid in evaluating sample quality. However, we have seen two important caveats. One, the quality measure does not do enough to punish the presence of artifacts in signals. Two, there is a lot of subjectivity involved in evaluating kick drum samples, so that objective judgements are hard to obtain.

We have also seen that the quality measure can be applied to data augmentation through sample layering. The produced augmented samples were classified by human listeners as being only slightly worse in quality. This augmentation strategy may also be applicable to other learning tasks in audio.

Overall, we conclude there is more work needed before high fidelity neural audio synthesis at 44.1kHz can become the norm. Perhaps our Progressive WaveGAN is a step in the right direction.

6.1 Future Work

We list some possible future directions that build upon this research.

Other datasets We can not conclude from our experiments that our Progressive WaveGAN truly is a 'better' model than other models in related work. In line with the scope of our research, we only evaluated the models on a small dataset and relatively short training times. The next step would be to try Progressive WaveGAN

on an open dataset like NSynth to enable a more direct comparison.

Quality as loss It could be beneficial to use a quality measure similar to ours as part of the loss function for the GAN generator. Would the network learn to produce higher quality samples? Or would it learn to fool the quality measure just like it tries to fool the discriminator? The best way to achieve this is to define the measure in terms of PyTorch/Tensorflow modules. This would result in a different quality measure than the one we defined for this research.

More varied data augmentation The augmentation method we propose in section 3.2 could be adapted to produce a more varied augmented dataset. We already noted that the cutoff frequency does not need to be constant. One could also experiment with applying pitch shifting and all-pass filters to the lower layer, or adjusting the volume and decay of the upper layer.

Optimized code The PyTorch code we wrote to adapt WaveGAN into a progressive growing scheme is a bit messy and can probably benefit from optimization. Especially the α-fading version is very slow, reversing the speed gains the progressive growing scheme initially provided before we added fading.

Spectral Discriminators We believe the main problem in spectral representations is inversion. However, spectral representations make for great audio classifiers. Perhaps there could be some benefit in making a GAN where the generator works on raw audio, but the discriminator uses spectral representations.

Oversampling In all of our experiments, we targeted 44.1 kHz sampling rates by directly processing at that resolution. The next step could be to process at a higher rate, and then downsample the result. We believe this could be effective, because we noticed in our experiments that the worst artifacts had very high pitches. If we increase the sampling rate, perhaps those artifacts' pitches will move with it, and so move into the inaudible range before being low-passed out.

6.2 Discussion

We already noted in section 6.1 that this research used a small dataset and short training times. Thus potentially leaving a lot of performance on the table, especially for the recurrent models. One criticism of our research process could be that we spent more time training and tuning the hyperparameters of the WaveGAN variants than we did for the other models. Most of the hyperparameters for each model were left unchanged from when we cloned the respective repositories, but it could certainly be the case that the hyperparameters for the WaveGAN model happened to be more suited for this task and that a more dedicated search in the other models would have yielded better results.

Another valid criticism of our experiments is that we did not include the WaveGAN-α variant in our listening experiment, even though it got the best quality evaluation. Unfortunately, this is due to the timeline of this project, GPU cluster availability and the α-fading variant having the longest training time. Still, the fact that the listening experiment found the exact same ordering on models as the quality measure gives us confidence that the α-fading variant would have performed similarly to the η-fading variant.

6.3 Acknowledgements

We would like to end this report by thanking David van Leeuwen for acting as supervisor and Nik Vaessen for his help with PyTorch, getting set up on the GPU cluster and feedback on this report. We would also like to thank the participants in the listening experiments for their contribution and their patience with the slow loading times that came as a result of running the Raspberry Pi 4 as a web server.

Bibliography

[1] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of StyleGAN," 2020.

[2] C. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[4] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[5] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[6] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein GANs," 2017.

[7] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2014.

[8] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," 2017.

[9] C. Donahue, J. McAuley, and M. Puckette, "Adversarial audio synthesis," in *International Conference on Learning Representations*, 2019.

[10] J. M. Kates, "Principles of digital dynamic-range compression," *Trends in Amplification*, vol. 9, no. 2, pp. 45–76, 2005.

[11] J. Schmidt and J. Rutledge, "Multichannel dynamic range compression for music signals," in *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, vol. 2, pp. 1013–1016 vol. 2, 1996.

[12] P. Virtanen, R. Gommers, T. E. Oliphant, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[13] H. Fletcher and W. A. Munson, "Loudness, its definition, measurement and calculation," *The Bell System Technical Journal*, vol. 12, no. 4, pp. 377–430, 1933.

[14] J. C. Brown and M. S. Puckette, "An efficient algorithm for the calculation of a constant Q transform," *The Journal of the Acoustical Society of America*, vol. 92, no. 5, pp. 2698–2701, 1992.

[15] B. Boashash, "Estimating and interpreting the instantaneous frequency of a signal. ii. algorithms and applications," *Proceedings of the IEEE*, vol. 80, no. 4, pp. 540–568, 1992.

[16] D. Shmilovitz, "On the definition of total harmonic distortion and its effect on measurement interpretation," *IEEE Transactions on Power Delivery*, vol. 20, no. 1, pp. 526–528, 2005.

[17] P. Welch, "The use of fast fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms," *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967.

[18] K. Kilgour, M. Zuluaga, D. Roblek, and M. Sharifi, "Fréchet audio distance: A metric for evaluating music enhancement algorithms," 2019.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alch'e-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[20] M. Abadi, A. Agarwal, P. Barham, and Tensorflow Contributors, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[21] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. van den Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," *CoRR*, vol. abs/1802.08435, 2018.

[22] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A generative model for raw audio," 2016.

[23] J. H. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, and M. Norouzi, "Neural audio synthesis of musical notes with wavenet autoencoders," *CoRR*, vol. abs/1704.01279, 2017.

[24] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "SampleRNN: An unconditional end-to-end neural audio generation model," 2017.

[25] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of GANs for improved quality, stability, and variation," in *International Conference on Learning Representations*, 2018.

[26] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *CoRR*, vol. abs/1812.04948, 2018.

[27] L. N. Smith, "Cyclical learning rates for training neural networks," 2017.

[28] D. Griffin and J. Lim, "Signal estimation from modified short-time fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 2, pp. 236–243, 1984.

[29] J. Nistal, S. Lattner, and G. Richard, "Comparing representations for audio synthesis using generative adversarial networks," in *2020 28th European Signal Processing Conference (EUSIPCO)*, pp. 161–165, 2021.

[30] Z. Kong, W. Ping, J. Huang, K. Zhao, and B. Catanzaro, "DiffWave: A versatile diffusion model for audio synthesis," 2021.

[31] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "GANSynth: Adversarial neural audio synthesis," in *International Conference on Learning Representations*, 2019.

[32] G. R. Javier Nistal, Stefan Lattner, "DrumGAN: Synthesis of drum sounds with timbral feature conditioning using generative adversarial networks," in *2020 21st International Society for Music Information Retrieval Conference*, 2020.

[33] A. Ramires, P. Chandna, X. Favory, E. Gómez, and X. Serra, "Neural percussive synthesis parameterised by high-level timbral features," pp. 786–790, 2020.

[34] A. Marafioti, N. Perraudin, N. Holighaus, and P. Majdak, "Adversarial generation of time-frequency features with application in audio synthesis," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 4352–4362, PMLR, 09–15 Jun 2019.

[35] P. Esling, N. Masuda, A. Bardet, R. Despres, and A. Chemla-Romeu-Santos, "Universal audio synthesizer control with normalizing flows," 2019.

[36] P. Dhariwal and A. Nichol, "Diffusion models beat GANs on image synthesis," 2021.

Appendix A

Appendix

A.1 Code Sources

We forked our models from the following GitHub repositories, and adapted the code to fit the goals of the research (think of 44.1kHz sampling rates, quality measure evaluations, memory limitations and checkpoint generation)

- WaveGAN & SpecGAN in Tensorflow: https://github.com/chrisdonahue/wavegan
- WaveGAN in PyTorch: https://github.com/auroracramer/wavegan
- WaveRNN: https://github.com/fatchord/WaveRNN
- WaveNet: https://github.com/golbin/WaveNet
- Nistal PGAN: https://github.com/SonyCSLParis/Comparing-Representations-for-Audio-Synthesis-using-GANs

Our final source code is slightly messy but can be found at https://github.com/apeklets/kicks-thesis, which also includes the source code for the quality measure and a number of notebooks documenting various informal experiments conducted during the design of the quality measure.

A.2 Configuration Details

A.2.1 WaveGAN 44.1kHz

As described in chapter 4, we adapted the PyTorch WaveGAN implementation to generate 44.1kHz samples according to the following architectures. Here d is the model dimension, we used 48 in our results, and c is the amount of channels, which in our case is 1. We use an 8-dimensional latent space.

Operation	Kernel Size	Out Shape	Activation
Input $z \sim Un(-1, 1)$		(8)	
Dense Layer	$(8, 512d)$	$(512d)$	
Reshape		$(16, 32d)$	ReLU
TransConv (s=4)	$(25, 32d, 16d)$	$(64, 16d)$	ReLU
TransConv (s=4)	$(25, 16d, 8d)$	$(256, 8d)$	ReLU
TransConv (s=4)	$(25, 8d, 4d)$	$(1024, 4d)$	ReLU
TransConv (s=4)	$(25, 4d, 2d)$	$(4096, 2d)$	ReLU
TransConv (s=4)	$(25, 2d, d)$	$(16384, d)$	ReLU
TransConv (s=2)	$(25, d, c)$	$(32768, c)$	Tanh

Table A.1: Adapted WaveGAN generator for 44.1kHz

Operation	Kernel Size	Out Shape	Activation
Input		$(32768, c)$	
Conv (s=2)	$(25, c, d)$	$(16384, d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, d, 2d)$	$(4096, 2d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, 2d, 4d)$	$(1024, 4d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, 4d, 8d)$	$(256, 8d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, 8d, 16d)$	$(64, 16d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, 16d, 32d)$	$(16, 32d)$	LReLU ($\alpha = 0.2$)
Reshape		$(512d)$	
Dense	$(512d, 1)$	(1)	Sigmoid

Table A.2: Adapted WaveGAN discriminator for 44.1kHz

We further used a base learning rate of $\eta = 1e - 4$, Adam optim parameters $\beta_1 = 0.5$, $\beta_2 = 0.9$, and a gradient penalty constant of $\lambda = 10$. Batch shuffling was disabled for our experiments, and we used a batch size of 32 to reduce memory usage. Our baseline model ran for 750 epochs, with 20 batches per epoch and 5 discriminator updates per epoch.

A.2.2 WaveGAN layers

To achieve a model that works at any number of layers, we had to adapt the input and output stages of the WaveGAN model. Depending on the number of layers, we discard part of the output from the dense layer in the generator such that the convolutions match perfectly and produce a 1D output of the correct length at the end. In the discriminator, we then have to fill the output back up with zeros such

that the final dense layer gets the right input shape. Below is an example architecture description when the number of layers is 3:

Operation	Kernel Size	Out Shape	Activation
Input $z \sim Un(-1, 1)$		(8)	
Dense Layer	$(8, 512d)$	$(512d)$	
Drop samples		$(64d)$	
Reshape		$(16, 4d)$	ReLU
TransConv (s=4)	$(25, 4d, 2d)$	$(64, 2d)$	ReLU
TransConv (s=4)	$(25, 2d, d)$	$(256, d)$	ReLU
TransConv (s=2)	$(25, d, c)$	$(512, c)$	Tanh

Table A.3: Progressive WaveGAN generator at layer 3

Operation	Kernel Size	Out Shape	Activation
Input		$(512, c)$	
Conv (s=2)	$(25, c, d)$	$(256, d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, d, 2d)$	$(64, 2d)$	LReLU ($\alpha = 0.2$)
Conv (s=4)	$(25, 2d, 4d)$	$(16, 4d)$	LReLU ($\alpha = 0.2$)
Reshape		$(64d)$	
Fill zeros		$(512d)$	
Dense	$(512d, 1)$	(1)	Sigmoid

Table A.4: Progressive WaveGAN discriminator at layer 3

All other hyperparameters stayed the same. For our final η-fading run, we trained for 1500 epochs, adding a new layer every 250 epochs. The α-fading run lasted 1000 epochs, with a new layer every 150 epochs.

A.2.3 PGAN configs

For our longest runs of this model, we started with the configuration as found on the GitHub repository for the Nistal PGAN (see A.1), and configured the model to run for the same amount of iterations as outline in their paper. That is 200k iterations for the first 4 scales, and 300k iterations for the last scale. We changed the base learning rate to $2 \cdot 10^{-4}$, and set the latent space dimensionality to 8. Of course, we also set it to run at 44.1 kHz sampling rates. The output length was 26000 samples long. This is the configuration that was used in our results for the 'complex' representation.

However, as we mentioned in section 4.2, we managed much better results for the

'mag-if' representation if we trained for less iterations. This run only ran for 150k iterations on the first 3 scales, 160k on the 4th scale and 300k on the last scale. This run further used a higher base learning rate of $6 \cdot 10^{-4}$.