TRANSFER LEARNING FOR AUTOENCODERS ON AUDIO DATA

Master Thesis

By

Niels Warncke

Supervised by

Thomas Goerttler

TU Berlin
Institute of Software Engineering and Theoretical Computer Science

September 2021

DECLARATION OF INDEPENDENCE

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, 29. September 2021

.....

ACKNOWLEDGMENTS

First I would like to thank Thomas Haferlach and Thomas Goerttler. I would also like to thank Daniel Heemann and Cemre Yüksel for proof reading, and I would like to thank the following AIs that assisted me with my writing: github copilot, grammarly and deepl.

SUMMARY

A recent approach towards modelling audio in a deep learning framework is Differential Digital Signal Processing (DDSP). Building blocks such as differential synthesizers are used to induce an audio specific bias into a model. As a result, Engel, Hantrakul, Gu, and Roberts [1] built an extremely data- and compute- efficient autoencoder for audio re-synthesis. Such a model can be used for various tasks including timbre transfer: here, the model receives an audio signal as input and outputs a new audio signal with the same melody but a modified timbre. Using the mentioned autoencoder, timbre transfer can be performed to the single instrument the model has been trained on.

This thesis extends and improves the DDSP autoencoder by a series of ablations. The final model can infer the timbre of a short audio signal and synthesize new melodies in the same timbre, without the need for training a completely new model for every instrument.

The major changes to the DDSP autoencoder are a stronger restriction on the latent space that enforces the model to separate melody and timbre, and the use of transfer learning to quickly adapt to new instruments. Additionally, the algorithm to compute loudness and the loss function are improved, and a new metric called cycle reconstruction loss is proposed to measure the model's performance for timbre transfer.

Finally, applications of the autoencoder for music generation and audio source separation are discussed.

ZUSAMMENFASSUNG

Ein neuerer Ansatz zur Modellierung von Audio in einem Deep-Learning-Rahmen ist Differenzierbare Digitale Signalverarbeitung (DDSP). Dabei werden Bausteine wie differentielle Synthesizer verwendet, um einen audiospezifischen Bias in ein Modell einzubringen. Als Ergebnis haben Engel, Hantrakul, Gu, and Roberts [1] einen extrem daten- und recheneffizienten Autoencoder für Audioresynthese entwickelt. Ein solches Modell kann für verschiedene Aufgaben verwendet werden, unter anderem für die Übertragung von Klangfarben: Das Modell empfängt ein Audiosignal als Eingabe und gibt ein neues Audiosignal mit derselben Melodie, aber einer veränderten Klangfarbe aus. Mit diesem Autoencoder kann die Klangfarbe auf das eine Instrument übertragen werden, für das das Modell trainiert wurde.

Diese Arbeit erweitert und verbessert den DDSP-Autoencoder durch eine Reihe von Ablationen. Das resultierende Modell kann die Klangfarbe eines kurzen Audiosignals erkennen und neue Melodien in der gleichen Klangfarbe synthetisieren, ohne dass für jedes Instrument ein komplett neues Modell trainiert werden muss.

Die wichtigsten Änderungen am DDSP-Auto-Encoder sind eine stärkere Regularisierung des latenten Raums, die das Modell dazu zwingt, Melodie und Klangfarbe zu trennen, und die Verwendung von Transfer-Lernen zur schnellen Anpassung an neue Instrumente. Außerdem wurden der Algorithmus zur Berechnung der Lautheit und die Fehlerfunktion verbessert, und es wird eine neue Metrik namens Zyklusrekonstruktionsverlust vorgeschlagen, um die Leistung des Modells bei der Übertragung von Klangfarben zu messen.

Schließlich werden Anwendungen des Autoencoders für die Musikerzeugung und die Trennung von Audioquellen diskutiert.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Autoencoders are self-supervised machine learning models trained to reconstruct inputs from a compressed representation of that input, also called the latent representation. They have been applied to various domains such as images, text, and audio. Use cases for autoencoders include among others anomaly detection, using them as a pretraining task for other downstream applications, for compression, or as a low-level module for a generative task.

This thesis builds on the work by Engel, Hantrakul, Gu, and Roberts [1], who built an autoencoder for monophonic recordings of harmonic instruments using differential digital signal processing (DDSP) techniques. Digital signal processing (DSP) is a field of signal processing that deals with the representation of signals as sequences of numbers and the generation and manipulation of those. By implementing DSP techniques in an auto-differential framework such as tensorflow, it is possible to train neural networks that make use of our prior knowledge about audio, making them extremely efficient.

In concrete terms, this autoencoder uses a pretrained model called CREPE to extract the *fundamental frequency*, a rule-based algorithm to estimate the loudness, and an additional encoder to compute the latent variable z that holds the remaining information necessary to reconstruct the signal. The decoder then controls a synthesizer that outputs the waveform. A more detailed description of this model follows in Chapter 2.

A direct application of this model is timbre transfer: a model trained on trumpet sounds will recreate any melody with the timbre of a trumpet. The purpose of this thesis is to improve this model, which from now on is called baseline model, such that it can later be used as a low-level module for the task of generating new music, similar to the VQ-VAE in Jukebox by Dhariwal, Jun, Payne, Kim, Radford, and Sutskever [2]. Another potential application that could be developed on top of such an autoencoder is audio source separation i.e. extracting the separate instruments of a polyphonic recording. A sketch of this idea is explained in Section 4.1.

In order to improve the baseline model, a number of changes are proposed, including a novel

way of using transfer learning. Transfer learning is the research area of using knowledge gained by training a model on one task in order to improve the performance on another related task. For example, if one needs to classify images into a number of custom categories, it can help to use one of the many available models trained on ImageNet. There are multiple approaches to perform transfer learning, including using the weights of the pretrained model to initialize a new model or using the first layers of the pretrained model as a feature extractor for a simpler model.
In this thesis, it is proposed to fine-tune the model on every inference sample for a short time, basically including fine-tuning in the inference pipeline, with the aim of creating an autoencoder that can handle a large variety of instruments that are not necessarily included in any training set.

The contributions of this thesis include multiple general improvements to the baseline model as well as the evaluation of the use of transfer learning. The general improvements are:

- the discovery of a bug in the algorithm for computing loudness which causes the baseline model to produce unpleasant squeaky noises in silent parts
- constraining the latent variable z to be constant over time, which allows using a single model to perform timbre transfer on multiple instruments that are included in the train set. The baseline model can only perform timbre transfer to the single instrument it has been trained on.
- a minor change to the loss function that allows better visualization and - under my subjective impression - is better aligned with the perceptual sound quality leading to more authentic reconstructions
- the proposal to use a cycle consistency loss inspired metric to measure the performance in a timbre transfer task

The rest of this thesis is organized as follows: Chapter 2 gives an overview over related work with a detailed description of the baseline model. Chapter 3 describes the proposed changes to the baseline, starting with the changes necessary to get a single model to perform timbre transfer onto different target instruments, and an evaluation of how to best fine-tune a multi-instrument

model onto new instruments. A discussion of the findings for potential use-cases and future work is given in Chapter 4. Further details on the training setup and hyperparameters can be found in Chapter C

A dashboard for exploring the different models that were trained for this thesis is available on nielsrolf.github.io?thesis - including sound samples, see Chapter B. The code written for this thesis is available on github.com/nielsrolf/ddsp.

CHAPTER 2

RELATED WORK

This thesis focuses on extending the DDSP-based autoencoder by [1] aiming to create a single model that can quickly learn to mimic diverse sounds from little data. The first part of this section contains a detailed description of this baseline model. In the second part, other machine learning models that can learn to synthesize natural data are presented.

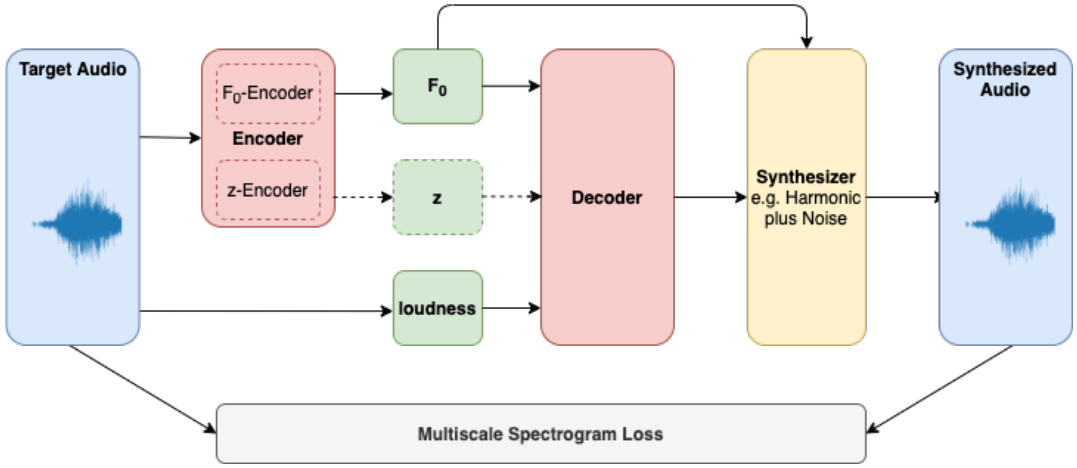
2.1 Differential Digital Signal Processing (DDSP)

An audio signal is not an arbitrary time series of air pressure values: we know that sound is often produced by vibrating objects, or gets perceptually relevant features from filters being applied to a base signal. If a note is played on an instrument such as a violin, the strings vibrate with a fundamental frequency f_0 and integer multiples of that frequency: their harmonics. A recording of sound in its raw form is called the waveform, and spectral information can be made available using a Short-Time Fourier Transform (STFT). Our perception relies mostly on spectral features of sound and much less on absolute air pressure values or the phase of a sound. For this reason, spectrograms are such a popular preprocessing step for tasks that involve audio feature extraction.

However, since a real-valued spectrogram contains no information about the phase of the sound, it is not perfectly suited for audio generation. For waveform generation, many different types of synthesizers exist that can create a wide variety of sounds.

Motivated by these observations, Engel, Hantrakul, Gu, and Roberts [1] proposed DDSP as a way of incorporating this knowledge about audio into neural network architectures. Specifically, they built an autoencoder for monophonic recordings of musical instruments. Its encoder extracts the fundamental frequency, the loudness, and optionally an additional latent variable z from the waveform. The decoder reconstructs the signal by controlling a harmonic plus filtered noise synthesizer. The architecture is shown in Figure 2.1.

Figure 2.1: Baseline DDSP architecture



The encoder computes f_0 using a pretrained model called CREPE and an additional latent variable z. From that, the decoder reconstructs the original audio by controlling a harmonic plus filtered noise synthesizer.

A harmonic plus noise synthesizer consists of two separate modules: a harmonic synthesizer and a filtered noise synthesizer. Harmonic audio occurs when an object can vibrate freely: a guitar string oscillates with a fundamental frequency and multiples of that. However natural signals often contain at least a certain amount of noise and other non harmonic elements, which is why a filtered noise synthesizer is added.

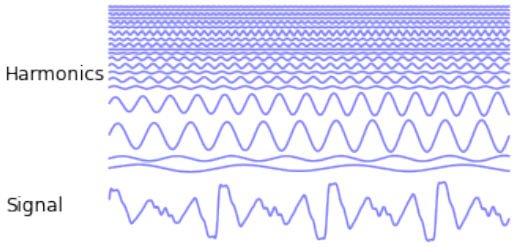
The harmonic synthesizer mixes sine waves with instantenous frequency $f_k = k * f_0$ according to a harmonic distribution $h \in \mathbb{R}^H$, where H is the number of harmonics and h is an output of the decoder. The signal is then scaled by the total amplitude a, such that the waveform at time t is computed as:

$$s_{harmonic}(t; f_0, h, a) = a_t \sum_i h_i \sin(2\pi i F_{0,t}) \tag{2.1}$$

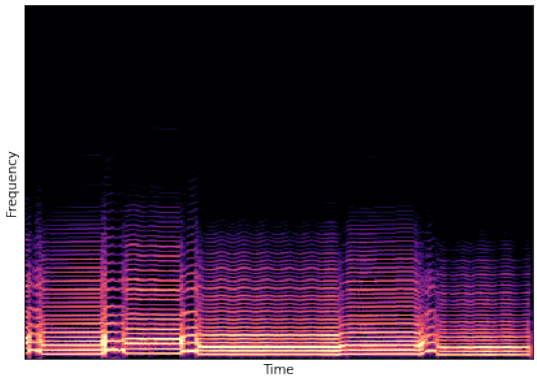
where $F_{0,t}$ is the unrolled phase

$$F_{0,t} = \frac{1}{sr} \sum_{i=0}^{t} f_{0,i} \tag{2.2}$$

An illustration of this synthesizer is shown in Figure 2.2.

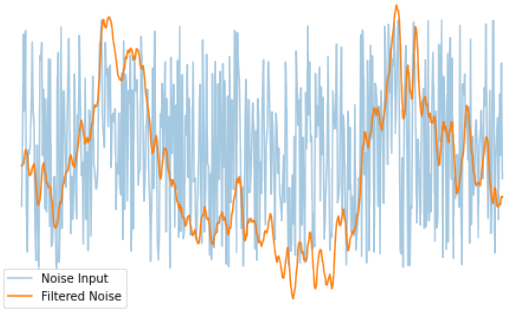


A harmonic synthesizer synthesizes a signal that is made up of scaled sine waves with frequencies that are integer multiples of f_0. This waveform corresponds to 31 milliseconds of audio.

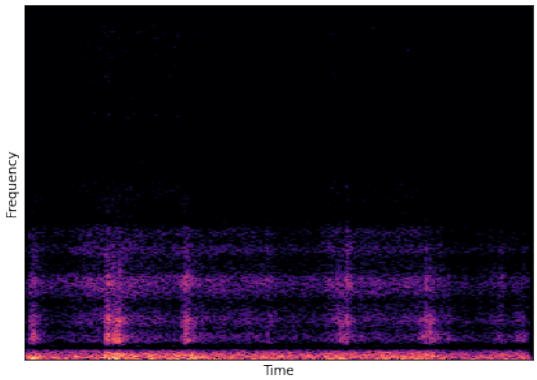


The spectrogram of 4 seconds of a harmonic waveform.

Figure 2.2: Harmonic Synthesizer



A filtered noise synthesizer receives random white noise as an input and applies frequency-band pass filters to it to generate a waveform. This waveform again corresponds to 31 milliseconds of audio.



The spectrogram of 4 seconds of audio output of a filtered noise synthesizer.

Figure 2.3: Filtered Noise Synthesizer

The filtered noise synthesizer is a band-pass filter applied to white noise - here, the neural network predicts time distributed noise magnitudes for a number of frequency bands. A waveform and a spectrogram of filtered noise are shown in Figure 2.3.

The organization of the latent space into these perceptually relevant variables and the use of synthesizers induce a structural bias on the model that is well suited for modeling monophonic audio. Furthermore, the prior that our perception depends mostly on spectral features is reflected by the use of a multi-scale spectrogram loss function [3].

Using f_0, loudness and z as the structure for the latent space was first proposed by Hantrakul, Engel, Roberts, and Gu [4] together with a WaveRNN architecture, motivated by the success

of TTS systems with informative local conditioning such as mel spectrograms. Hierarchical Timbre Painting (HPT) [5] use the same latent representation together with a ParallelWaveGAN [6] architecture in the decoder. While their sound samples are impressive and arguably a slight improvement over the DDSP baseline, I found exchanging the DDSP decoder by a Parallel WaveNet architecture to hurt performance in my initial experiments, if additional tweaks that require huge implementation efforts were not used: HPT is trained to upsample a low-resolution signal multiple times, requiring additional discriminators for each upsampling step. Training a baseline DDSP architecture is much more convenient as it trains only a single model, without the need for advanced training schedules to train different upsampling models and their respective discriminators.

Follow up work on DDSP by Hayes, Saitis, and Fazekas [7] add learned neural waveshaping units (NEWTs) to the synthesizer. These units get harmonic oscillators corresponding to f_0 as input, which are fed to a multi-layer perceptron (MLP) that uses sine waves as nonlinear activations. The MLP additionally gets time distributed audio features z as input and resembles a SIREN with a hypernetwork [8]. Using this approach allows very fast and high-quality audio re-synthesis. However, just like the original DDSP autoencoder, one model has to be trained for each target timbre. The suggested improvements of this work can potentially be combined with the methods evaluated in this thesis.

Related work by Liu, Chen, and Yu [9] similarly use DSP-inspired techniques for neural vocoding: their model is conditioned on f_0 and additional log mel-spectrogram features z, and learns to control linear time-varying filters (LTVs) in order to generate speech. The network predicts LTVs which are then applied to a periodic impulse train with frequency f_0, and to stochastic noise. LTVs applied to an impulse train yield harmonic signals which appear in voiced speech, and LTVs applied to noise add non-harmonic parts - just like the filtered noise synthesizer used in the baseline DDSP architecture. Besides applying it to a different problem, namely speech synthesis, the main difference is therefore the decoder architecture. The techniques explored in this thesis all use the same decoder as the baseline DDSP model. However, they don't necessarily depend on it and can therefore be combined with alternative

decoder/synthesizer architectures.

Another work that uses DDSP for vocoding is McCarthy and Ahmed [10]: here, a WaveNet is used on top of the individual harmonics and noise outputs of a DDSP autoencoder.

2.2 Other Approaches to Audio Modelling

WaveNet A major breakthrough for audio generation were autoregressive models like WaveNet [11] and WaveRNN [12], which are able to learn conditional and unconditional audio synthesis and the task of transforming text to speech (TTS). However, due to their autoregressive sampling, training and inference are extremely slow and due to the lack of any audio-specific bias, everything that is typical for sound has to be learned from scratch. These models are trained to maximize the log-likelihood of the audio data by factorizing

$$p(x) = p(x_1) \cdot p(x_2|x_1) \cdot \cdots \cdot p(x_n|x_1, \ldots, x_{n-1}) \tag{2.3}$$

where $x = (x_1, \ldots, x_n)$ is the audio data as a waveform. In order to sample from the model, one forward pass is necessary to obtain x_1, which is send fed back into the network to obtain x_2 and so on. Audio is typically sampled with at least 16000 Hz, which corresponds to the number of forward passes necessary to generate a single second of audio.

Non-autoregressive variants of it such as Parallel WaveNet [13] improve the inference speed, but still require huge amounts of training data and GPU time.

Other models like VQ-VAE and VQ-VAE-2 [14] [15] use a WaveNet as a building block, and additional modules to learn long-range dependencies.

Diffusion based models Diffusion based models recently became very popular for image generation [16]. For audio generation, they also show promising results: Kong, Ping, Huang, Zhao, and Catanzaro [17] introduced DiffWave, a vocoder that generates audio of spoken words from a mel-spectrogram. Diffusion models are trained to revert a process called diffusion, where noise is repeatedly added to a signal $x_{(0)}$. After a fixed number of steps T, the signal $x_{(T)}$ is approximately gaussian distributed, which means that using T denoising steps, a random vector $\hat{x}_{(T)} \sim \mathcal{N}(\mu, \sigma^2)$ can be transformed into a an audio signal $\hat{x}_{(0)}$. More precisely, the model

approximates the distribution of the signal at the previous time step, given $x_{(t)}$ as:

$$\begin{aligned} p(x_{(t-1)}|x_{(t)}, t, C) &\sim \mathcal{N}(\mu_{(t)},\, \sigma^2_{(t)}) \\ \mu_{(t)},\, \sigma_{(t)} &= f(x_{(t)}, t, C) \end{aligned} \tag{2.4}$$

where C is a control signal, e.g. features of a mel-spectrogram, and f is the forward pass of the neural network.

GAN-based Approaches WaveNet variants and diffusion-based models optimize the likelihood of the waveform. As an alternative to this, also GAN-based approaches can be used to model audio. Parallel WaveGAN [6] and HPT [5] use generators that work in the waveform domain together with dilated convolutional layers, similar to the ones used in WaveNet. GANSynth [18] uses an invertible spectral representation of the audio.
In early experiments, I tried training a DDSP-autoencoder with a GAN objective but found training difficult. The discriminator struggles to differentiate between the real and generated audio even if it is trained for many more steps than the generator.

Jukebox The current state-of-the-art in music generation is Jukebox [2]. It uses stacked transformers to learn the structure of music on multiple time-scales, and a VQ-VAE to generate music from latents. For such an approach, it is nice to have a low-level model with an encoder: in that case, the latent representation of natural music can easily be computed and used as the ground truth for another model that learns the next layer of abstraction.
The line of work on DDSP based architectures provides a potential alternative to the VQ-VAE within Jukebox. A limitation is, however, that the currently available DDSP-based architectures only work for monophonic audio. An encoder that controls multiple synthesizers to model polyphonic audio would be necessary to lift DDSP towards a general-purpose audio autoencoder. Creating a model that can capture all different kinds of monophonic audio is a first necessary step towards building a general-purpose polyphonic model, and this challenge is addressed in this thesis.

CHAPTER 3

METHODS

Consider the case that we have a short recording of an unknown instrument playing, and we want a model that allows us to mimic this instrument with new melodies - that is, we are interested in few shot timbre transfer. Using the baseline architecture, a model would need to be trained on the recordings of the new instrument, and then used to generate new melodies. An analysis of the capabilities and limitations of the baseline model is given in Section 3.1. In this thesis, the limits of fine-tuning a model to any new instruments are explored. However, before diving into transfer learning in Section 3.3, some general improvements to the model are explored that yield an improved baseline.

Using the first simple modification, the model can be forced to separate timbre from melody and perform one-shot timbre transfer for instruments it has seen during training. This improvement relies on making z constant over time, such that timbre transfer can be done by simply extracting z of a recording of a target timbre, and feeding it to the decoder together with the f_0 and loudness curve of the target melody. A more detailed analysis of this method follows in Section 3.2.1.

The total amount of information available to the decoder is strongly limited through this modification, which hurts reconstruction accuracy but improves the disentanglement of melody and timbre. Other tweaks of the model, namely a bug fix of the algorithm that computes loudness and a modification of the multiscale-spectrogram loss make up for the drop in reconstruction accuracy and yield a model that can mimic diverse instruments such as flute, trumpet, violin, bass, guitar, sounds of the iconic Roland sh101 synthesizer and to some extend even various drums.

Details on the datasets used are given in Section 3.1.2.

3.1 Baseline architecture

In order to encode the audio signal, loudness is computed using a rule-based algorithm with no trainable parameters as described by Hantrakul, Engel, Roberts, and Gu [4]. The f_0 encoder is a pretrained convolutional neural network called CREPE that operates directly on the waveform, and the z-encoder consists of fully connected and GRU-layers [19] working on mel-frequency cepstral coefficients (MFCCs) computed from the waveform. If the training data is collected from a single instrument, reconstruction is possible from only the pitch and loudness curves, without the need for a z-encoder. However, all models trained for this thesis use a z-encoder. All latent variables are time-distributed with a frame rate of 250 Hz, much below the sample rate of the waveform, which is 16000 Hz.

The decoder controls two synthesizers: an additive harmonic synthesizer and a filtered noise synthesizer. The harmonic synthesizer gets f_0 directly as input and requires a time-varying amplitude distribution over 100 harmonics which is predicted by the decoder.

The filtered noise synthesizer applies band-pass filters to white noise using 65 noise magnitudes at every time frame, which are also an output of the decoder. Fully connected layers and one GRU layer are used in the decoder.

If the model is trained on audios of a single instrument, e.g. a violin, the decoder interprets every melody as played by a violin and can therefore be used to perform timbre transfer. However, if the model is trained on multiple instruments at once, it infers the timbre from the latent variable z.

3.1.1 Measuring Timbre Transfer Performance

Evaluation of a model's capabilities to perform timbre transfer is often done by subjective listening tests. In order to automatically evaluate a large number of models in a deterministic setup, I used a cycle-consistency inspired loss function. To compute the loss, two timbre transfers are performed: a sample a, say a violin recording, is reconstructed by transferring its timbre onto an intermediate melody of a sample b, e.g. a melody extracted of someone singing. The resulting audio is then used as a target timbre for the second timbre transfer, where the pitch and loudness of a are used as the target melody to obtain $\hat{a}$. Ideally, both synthesized audios

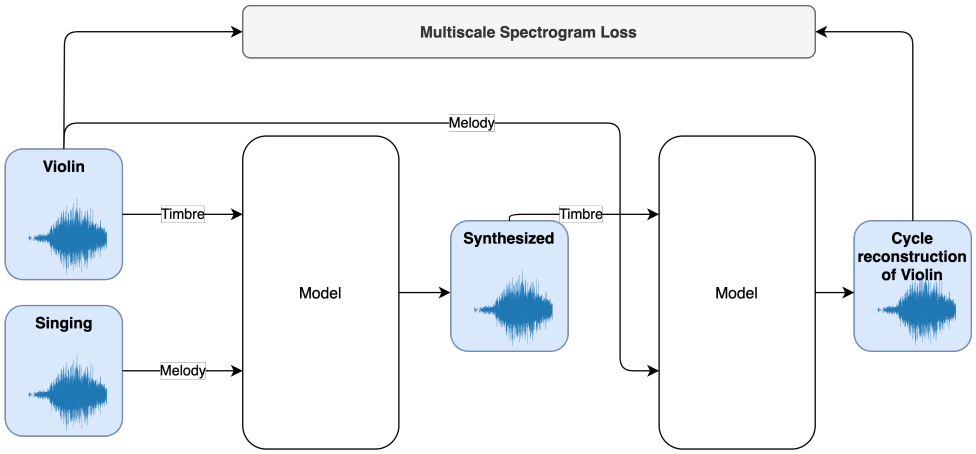


Figure 3.1: The cycle-reconstruction loss

should have different melodies but the timbre of the violin in a, and therefore $\hat{a}$ approximates a and is called the cycle reconstruction of a. A schematic representation of the loss function is shown in Figure 3.1.

A baseline model trained on monophonic recordings of the URMP dataset can reconstruct recordings of a held out test set consisting of the same instruments. However, if we use the model to perform a timbre transfer, the audio sounds noisy and the timbre is barely recognizable, which is reflected by a poor cycle reconstruction loss. Error heatmaps of respective spectrograms are shown in Figure 3.3.

3.1.2 Datasets

Multiple datasets with different instruments and varying total amounts of audio data serve as training data for the models, an overview is given in Table 3.1. The bass, guitar and drum datasets are published by the Frauenhofer IDMT institue [20] [21] [22]. The URMP dataset has been released by Li, Liu, Dinesh, Duan, and Sharma [23] and contains a large number of different instruments. Some experiments for this thesis are performed only on the URMP dataset, while others are made on all listed datasets. Last but not least, the sh101 dataset was kindly provided by Thomas Haferlach and is currently not published.

Of each dataset, only the raw audio data of monophonic recordings is used for the experiments - none of the additional meta-data. In particular have the models not been trained to explicitly classify the instrument of the audio, instead the timbre is modeled as a continuous latent vec-

Dataset	Audio Data	Instruments
bass_train bass_test	00:03:30 00:02:20	Bass
guitar_train guitar_test	00:38:26 00:22:48	Guitar
idmt_drum_train idmt_drum_test	00:34:10 00:23:43	Drum loops of acoustic drums, drum sample libraries and drum synthesizers
sh101_train sh101_test	00:48:43 00:05:26	Samples of a Roland sh101 synthesizer with different settings
urmp_train urmp_test	04:01:19 00:34:24	Violin, viola, cello, double bass, flute, oboe, clarinet, bassoon soprano saxophon, tenor saxophon, trumpet, horn, tromboe and tuba
combined_train combined_test	06:06:10 01:28:44	All above datasets combined

Table 3.1: Datasets

tor with 16 dimensions. The wavefiles were shuffled and split into training and test sets that remained fixed across all experiments. In a preprocessing step, the audio was resampled to 16 kHz, loudness and pitch were computed and snippets of four seconds were extracted.

3.2 Multi-Instrument Model

By z aggregation and some other tweaks, a single model can learn to perform timbre transfer on all kinds of instruments - as long as they are in the train set. This more general model serves as the starting point for fine-tuning on unseen instruments in the next step. In this section, the steps that led to a single model handling multiple instruments as well as multiple specialized baseline models are described.

3.2.1 Time constant z enforces z to be a timbre vector

A DDSP autoencoder with a z-encoder can learn to reconstruct monophonic recordings of instruments it has seen during training. However, its usage as a multi-instrument synthesizer is limited by the fact that it is unclear how to obtain z if the target audio is unknown. The naive approach is to simply encode some audio z of a target instrument and keep it fixed for new melodies. A timbre transfer that synthesizes the melody $f_0(A), ld(A)$ with the timbre infered

by the audio sample B is then given by:

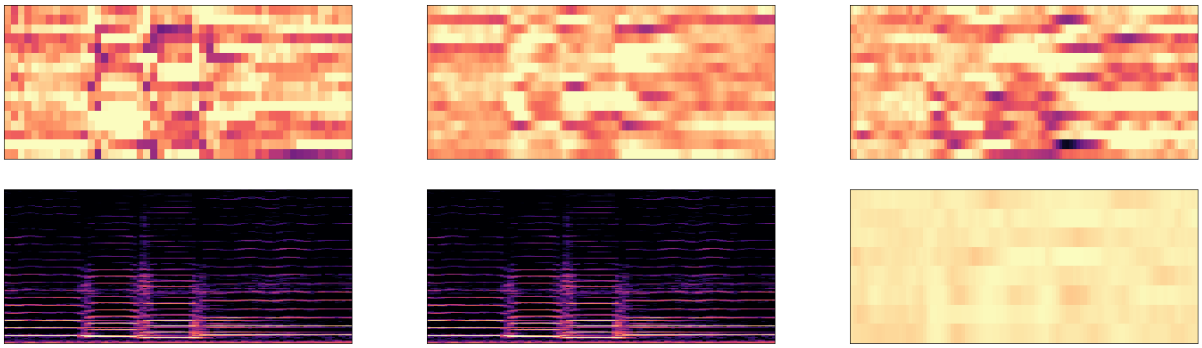
$$\begin{aligned} \hat{z}(B) &= encoder(B) \\ z(B) &= \frac{1}{T}\sum_{t=1}^{T}\hat{z}(B)_t \\ out &= decoder(f_0(A), ld(A), z(B)) \end{aligned} \tag{3.1}$$

As Figure 3.2 shows, this is very untypical input for the decoder and therefore yields unnatural output. Audio samples are shown Figure 3.3 and can be listened to on nielsrolf.github.io?thesis. If we integrate such an aggregation step into the encoder, it has to learn to extract features of the audio signal that describe not only the current time step and instead generalize along the complete time axis. In clean audio recordings without reverb, this information is essentially the timbre of an instrument. Additionally, the decoder has to learn to deal with noisy z estimates and sees input of much lower entropy. Using this adjustment, the model indeed improves in terms of timbre transfer and cycle reconstruction loss, but this comes at the cost of slightly worse performance on pure reconstruction loss.

One reason for this is that the model has to achieve a much higher compression factor: as $z \in \mathbb{R}^d$ and $\hat{z} \in \mathbb{R}^{(T,d)}$, the model uses only 2 vs $d+2$ parameters for each additional timestep, namely the current pitch and loudness values.

Surprisingly, it is still possible for a time-constant z model to achieve a comparable reconstruction error as the baseline by introducing a few additional tweaks. One of them directly addresses the problem of noisy z estimates. During a recording, there may be silent parts that should not be taken into account when computing z, and different parts of the audio signal can contain different information about the timbre.

So rather than simply averaging along the time axis, the confidence-masked z model predicts a



Baseline | Time-averaged z model | Confidence-masked z model

Figure 3.2: The latent variable z varies over time

In the top row, the decoder output for z is displayed. The baseline model (left) passes z as a time distributed variable to the decoder, which only learns to synthesize natural audio for melodies if the corresponding z is presented, leading to poor timbre transfer performance. The time-averaged z model (middle) uses the mean of all z values as the input to the decoder, which leads to a much better timbre transfer performance as the decoder learns to deal with noisy z inputs, which can be obtained from a single recording of the target instrument. The confidence-masked z model (right) predicts a confidence mask (bottom row) for each z value and computes a weighted mean. This enables the model to infer certain dimensions of z from different parts of the audio.

confidence mask for each $\hat{z}$ value and computes a weighted mean.

$$
\begin{aligned}
x &= \alpha_{enc}(A) \\
\hat{c} &= sigmoid(\alpha_c(x)) \\
c &= \hat{c} / \sum_t \hat{c}_t \\
\hat{z} &= \alpha_z(x) \\
z &= \sum_t (\hat{z} * c)_t
\end{aligned}
\tag{3.2}
$$

Here, α_{enc} is a neural network with the same architecture as the baseline encoder, and α_c and α_z are time distributed linear layers. This enables the model to infer certain dimensions of z from different parts of the audio. The additional changes explained in the following sections were also used for the confidence-masked model.

3.2.2 Loudness

In psychoacoustics, loudness describes the perceived intensity of a sound. Unlike the spectral power or amplitude, loudness is not defined in a precise mathematical way. There are algorithms

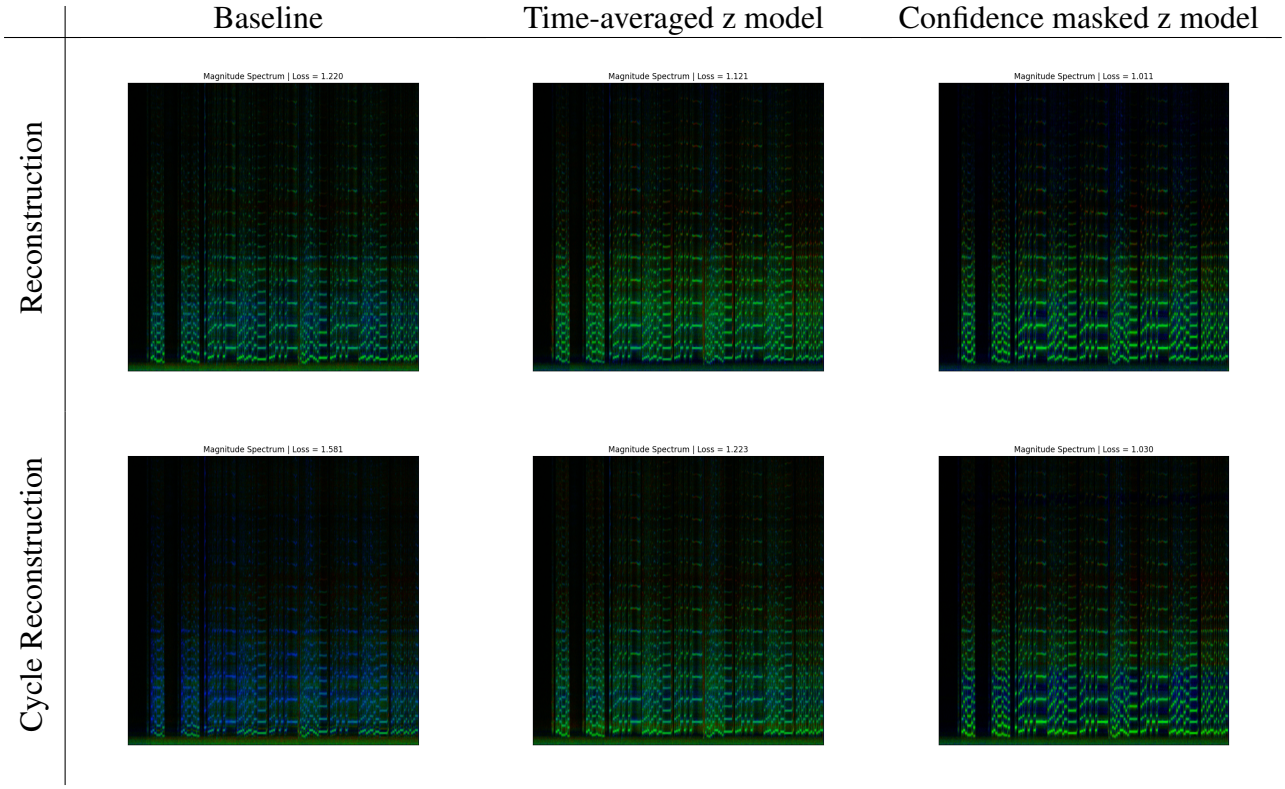


Figure 3.3: A time-constant z model outperforms the baseline in terms of cycle reconstruction loss

These error heatmaps visualize the difference between the spectrogram of some original audio and its reconstructed version. Frequencies present in both the original and the synthesized audio are shown in green, red shows where the synthesized audio is too loud and blue highlights, where the synthesized audio is missing frequencies.

for approximating loudness that are used in the DDSP autoencoder. To compute loudness, a power spectrum is computed and converted into a decibel scale. In order to take into account that for most people the loudness of a sound depends not only the power but also on the frequency at which it appears, A-weighting is applied. A-weighting scales the activations of the power spectrogram by a constant factor depending on the frequency. The weighted power spectrum is then meaned along the frequency axis to obtain the loudness curve. A visualization of this process is displayed in Figure 3.4.

The official DDSP implementation uses A-weighting by adding a per-frequency bias to the power decibel spectrogam. The resulting spectrogram is then averaged over the frequency axis, such that the A-weighting becomes ineffective: it only shifts the complete loudness curve by its mean value.

The oscillation of the loudness curve that can be seen in Figure 3.4 are due to another issue. During the signal, the frequency of the loudness varies, but the spectrogram only has a discrete number of frequency bins. Whenever the instantaneous frequency is not close to the center of a frequency bin, the spectrogram gets a bit blurred along the frequency axis. This phenomenon is called spectral leakage. If we now transform the power spectrum into the decibel scale, we apply a concave function. As is known from Jensen's inequality, the mean of these values will be much larger than if all the power at the timestep would fall into a single frequency bin. Therefore, loudness oscillates as the instantaneous frequency moves between frequency bins.

To fix these bugs, I computed loudness by applying a-weighting as a multiplication in the power spectrum, which is equivalent to adding a bias in the log-scale. Then, I applied a mean over the frequency axis to obtain the loudness curve and applied the logarithm as the last step. A detailed discussion of this algorithm can also be found in the corresponding github issue [1]. A model trained with the adjusted loudness algorithm improves particularly in silent parts, where the model otherwise puts an unpleasant squeaky noise as it does not have reliable information on the true loudness. Such an example is shown in Figure 3.5.

[1] github.com/magenta/ddsp/issues/361

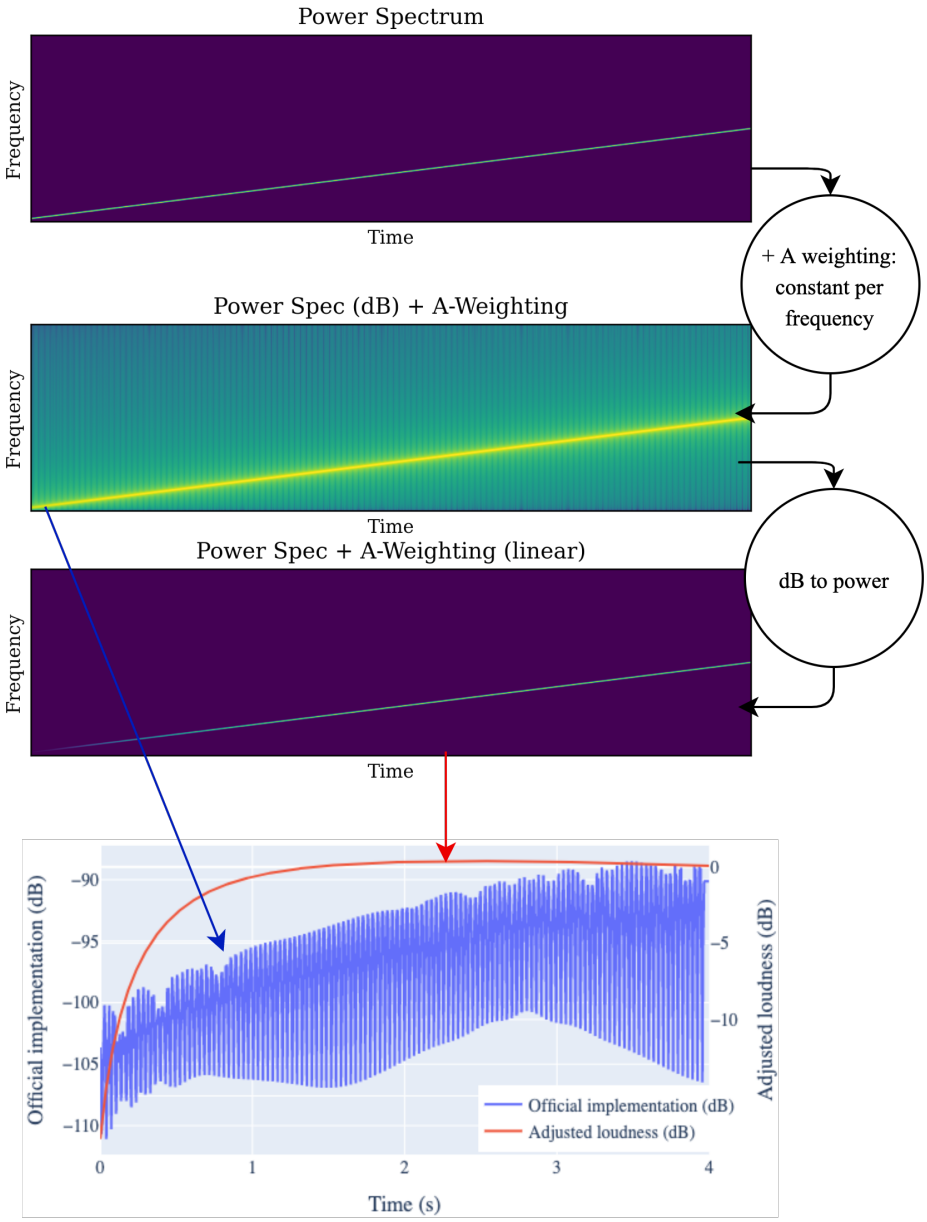
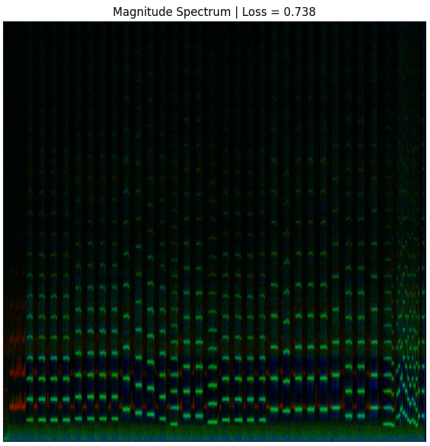
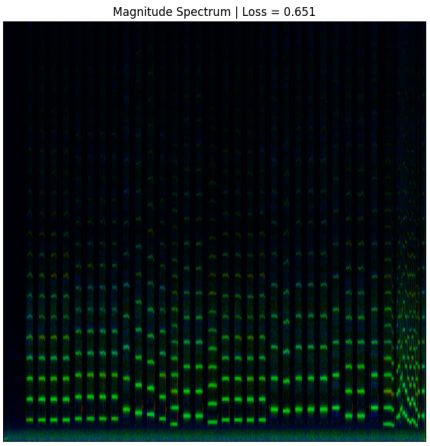


Figure 3.4: Computing loudness

Two loudness curves for the same audio signal are shown here: the signal is a single sine wave with a frequency starting at 100Hz and going to 4000Hz. A bug in the algorithm for computing loudness makes the loudness curve sensitive to artifacts of the spectrogram. The resulting loudness curve for the pitch sweep oscillates strongly as frequency leakage occurs in the spectrogram. The loudness algorithm used for the improved baseline fixes this and produces a smooth loudness curve that is shaped by the A-weighting.



Time average z + old loudness



Time average z + adjusted loudness

Figure 3.5: A reconstruction using the official vs the adjusted loudness algorithm
Two models reconstruct a 30 second audio recording of a flute playing. The models only differ in the loudness algorithm used, all other parameters are identical. In the reconstruction using the old loudness (left) the model creates an unpleasant squeaky noise in the silent beginning of the audio. A model trained with the adjusted loudness algorithm (right) does not have this problem.

3.2.3 Multi-Scale Spectrogram Loss

Two very different waveforms can sound the same, for example, if they are overlayed sine waves with the same frequencies in different phases. Therefore, it is problematic to use a mean-squared error loss function of the waveform in order to evaluate a reconstruction. Some models like WaveNet follow this approach, but the task is more complex than it needs to be: instead, the difference between the two spectrograms can be used to compute the loss.

A common approach to do so is via a multi-scale spectrogram (MSS) loss. Since the phase information is lost in the spectrogram, the MSS loss is invariant to phase shifts, like the human perception. For the MSS loss, multiple spectrograms of the audios are computed with different window sizes. A spectrogram with a large window size has a high frequency resolution but a low temporal resolution, whereas a spectrogram with a small window size has a low frequency resolution but a high temporal resolution. The MSS loss is then computed by summing the absolute difference between the two spectrograms. It therefore has a high resolution in both frequency and temporal dimensions.

[1] and other work use magnitude spectrograms and log-magnitude spectrograms with window sizes ranging from 4ms to 128ms. These spectrograms are shown in Figure 3.7.

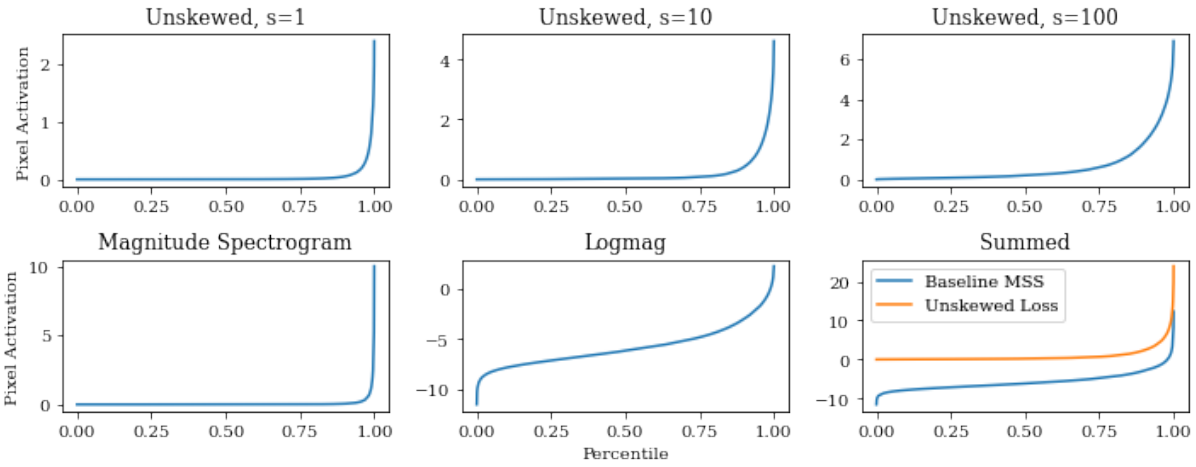


Figure 3.6: Pixel activation distribution of differently scaled spectrograms
Looking at the distribution of activations in the spectrograms, it can be seen that the magnitude spectrograms have a highly skewed distribution: only a few pixels are activated a lot, and most pixels have a very low activation. Unskewing with a smoothing parameter s as described in Equation (3.3) therefore makes patterns in the spectrogram visible that have comparably small activations. The logmag-spectrogram also achieves this, but it is also sensitive to differences in the least activated pixels, which correspond to very quiet frequencies that are likely not audible to most humans.

Ideally, the spectrogram magnitudes are scaled such that the resulting loss function aligns with the human perception. A good indicator for this is if audible mistakes are also visible in the spectrogram difference. When playing around with visualizing the difference between two audios, I found the scaling of the baseline implementation not to be ideal.

The logmag-spectrograms look noisy, but the amplitude spectrograms have such a skewed distribution that many perceptually relevant features are hardly visible. As a solution, I unskewed the activation distribution of the magnitude spectrogram with the function

$$u(M, s) = log(1 + s * M) \tag{3.3}$$

where M is the magnitude spectrogram, and s is a scalar smoothing factor. High values for s also highlight differences in low-energy parts of the spectrogram but are less sensitive to differences in parts of the spectrogram that are loud and contain high energy. Example spectrograms with different smoothing factors are shown in Figure 3.7.

If we evaluate all trained models for this thesis [2], we observe as expected that the two loss functions produce correlated scores, see Figure 3.8. However, models also tend to perform

[2] A full list is given in Table C.1

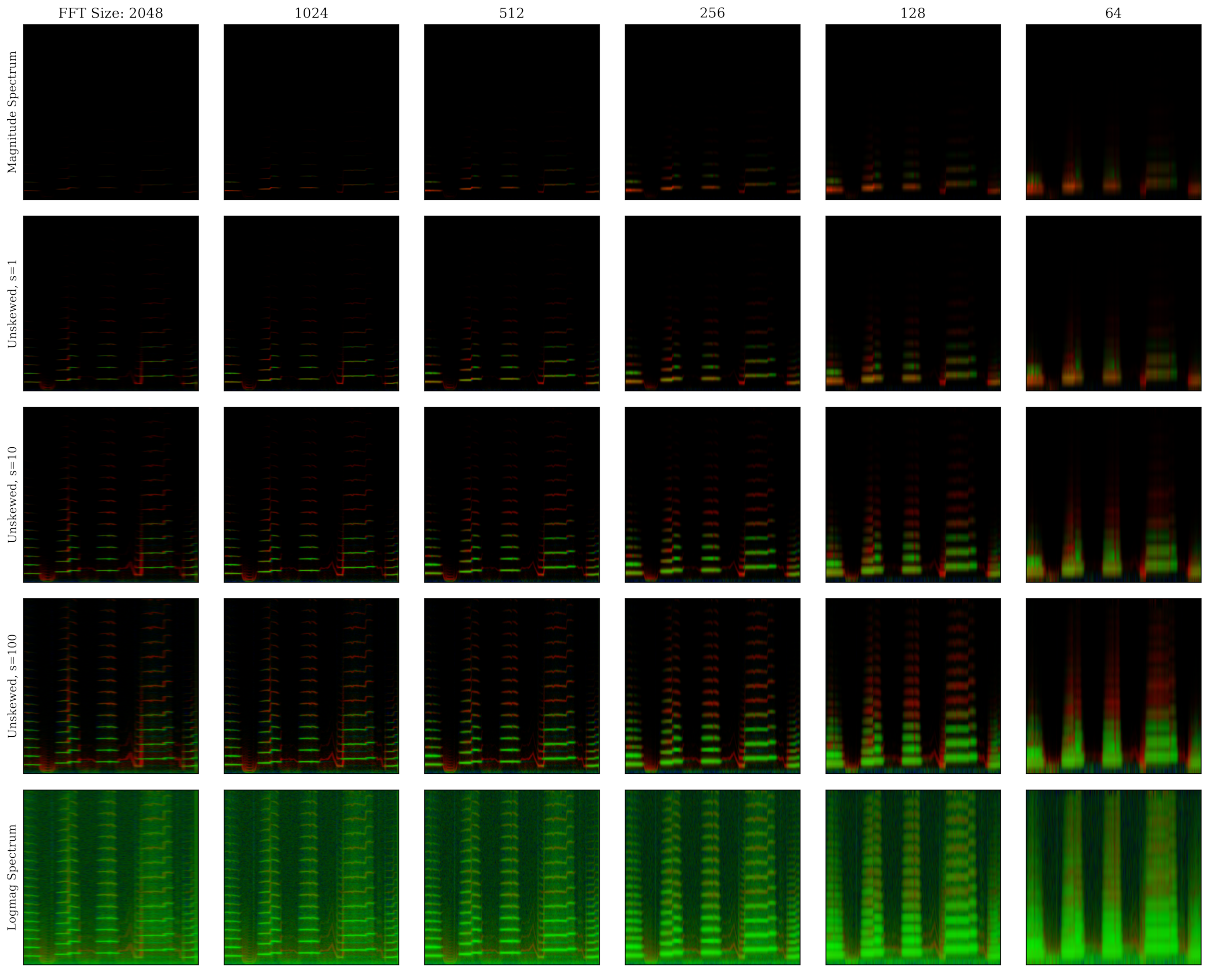


Figure 3.7: Spectrograms used to compute the MSS loss
The baseline MSS uses only the magnitude and log-magnitude spectrograms (first and last row). The unskewed loss uses the magnitude spectrums with different smoothing factors s. These spectrograms visually highlight more differences that can also be heard.

comparably better on the loss function they have been trained to optimize. A fair comparison of models trained with different loss functions is therefore difficult. Whether or not the unskewed loss really reflects human perception better is not entirely clear and would require confirmation by multiple human listeners. For the evaluations in this thesis, the unskewed loss was chosen as the default metric.

3.2.4 Bringing it all together

To validate the ideas proposed in the previous sections, I trained a number of models with different configurations. They differ in how they aggregate z, which loss function they optimize, and how they compute loudness. These models were then evaluated on all test datasets separately to compute the reconstruction loss. Additionally, on a subset of this test set timbre transfer has

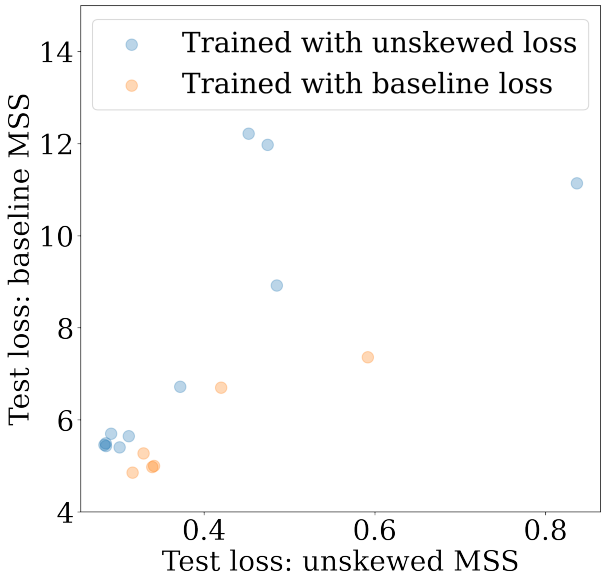


Figure 3.8: Test loss evaluated with unskewed MSS vs baseline MSS

been evaluated using the cycle-reconstruction loss. As intermediate melodies, a sample of a guitar playing and a person singing are used. A summary of the results for the URMP test set is shown in Figure 3.9. The reason to use only a subset of the test set for evaluating the cycle reconstruction loss lies in the amount of data produced: the total size of produced artifacts at the time of writing this is 14.7GB already.

It can be seen that aggregating z via a mean over the time axis improves timbre transfer but hurts reconstruction. However, using the u-MSS and the adjusted loudness algorithm improve the performance such that the improved baseline beats the baseline in both timbre transfer and reconstruction. The architectural design composed of the best individual choices is therefore the one that

- Uses confidence-masked z-aggregation in the encoder,
- is trained with the unskewed MSS loss and
- uses the fixed loudness algorithm.

Models using all these settings are now referred to as improved baseline models.

The baseline and the improved baseline models have not only been trained and evaluated on the URMP dataset but also all other datasets listed in Table 3.1. The improved baseline has additionally been trained on a composition of all other datasets. This single model compares

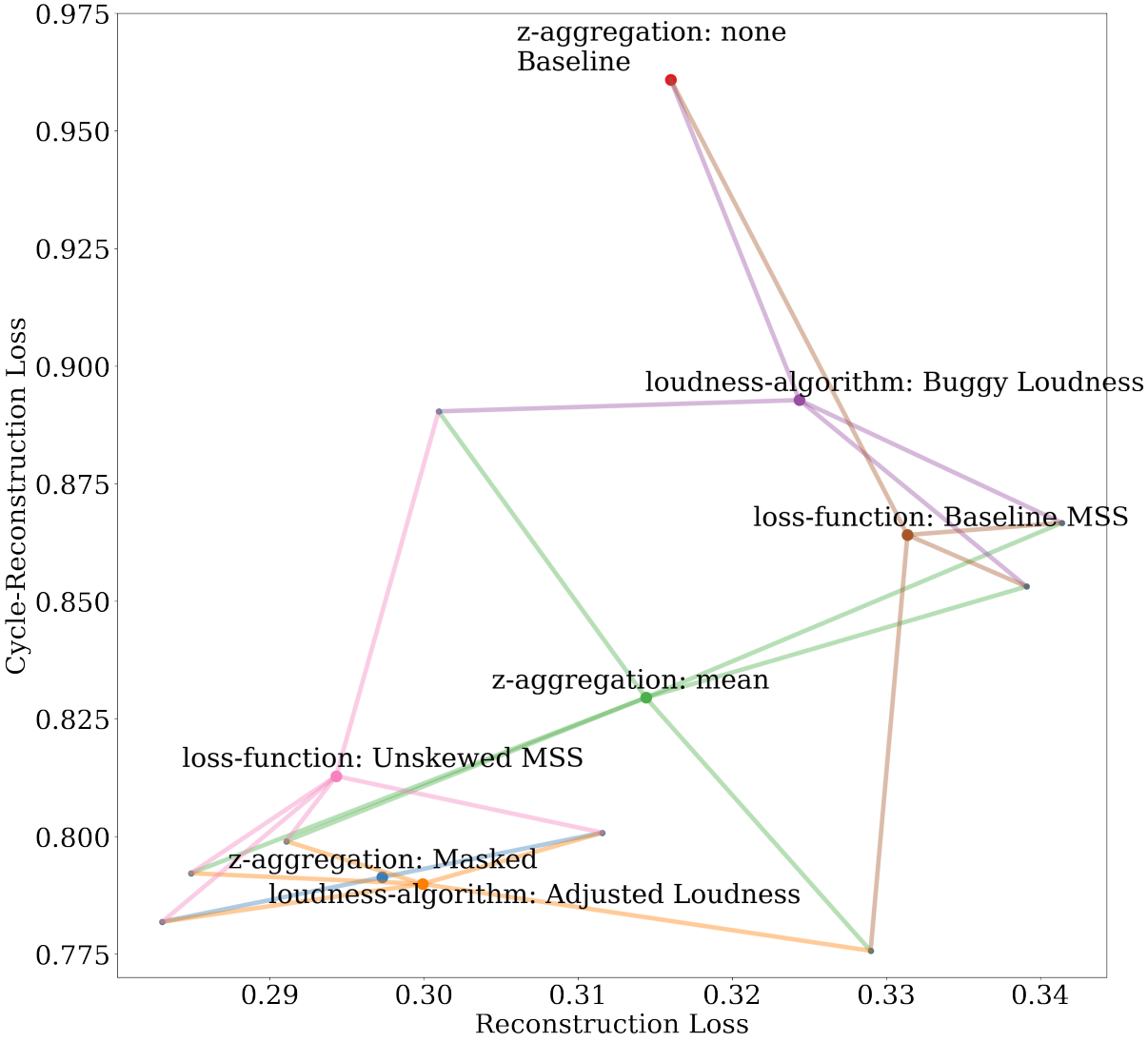


Figure 3.9: Comparison of Test Losses on URMP

This figure summarizes all metrics of the ablation studies shown in Table 3.2. Each small blue point corresponds to one model trained on the URMP train dataset. The colored points correspond to the mean performance of all models trained with a certain configuration. Models and configurations are connected to if the model uses this configuration. The baseline model uses the configrations in the upper right, showing that all proposed changes improve the model's reconstruction and timbre transfer performance as measured by cycle reconstruction error.

Baseline loss, baseline loudness

z-aggregation	Reconstrunction u-MSS	Cycle-Reconstruction u-MSS
None	0.316*	0.961
Mean	0.341	0.867*

z-aggregation: mean, baseline loudness

Training objective	Reconstrunction u-MSS	Cycle-Reconstruction u-MSS
MSS (Baseline)	0.341	0.867*
u-MSS	0.301*	0.890

z-aggregation: mean, Training objective: u-MSS

Loudness algorithm	Reconstrunction u-MSS	Cycle-Reconstruction u-MSS
Loudness (Baseline)	0.301	0.890
Loudness (fixed)	0.285*	0.792*

Training objective: u-MSS, Loudness algorithm: fixed

z-aggregation	Reconstrunction u-MSS	Cycle-Reconstruction u-MSS
Mean	0.291	0.799
Confidence-masked	0.283*	0.781*

Table 3.2: Ablations on URMP

well on all datasets with a baseline model that has only been trained on the corresponding train set, as Figure 3.10 shows. Again, it is highly recommended to listen to the examples of the dashboard, as the loss seems to be better for the base model on some data sets like the sh101 data set. However, listening to the examples shows that the timbre transfer and cycle reconstruction of the improved model sound much better than those of the baseline model. The aim to create a single model that can handle a large number of instruments is thereby partly achieved - yet, the sound quality is not perfect, and the cycle reconstruction loss is consistently much above the reconstruction loss, showing that the models cannot perfectly preserve timbre in a transfer task. Another finding is that a mixture of experts has slightly better performance than a single model trained on all datasets. Some of this may be explained by the fact that the mixture of experts trained for five times more steps than the single model and equally has five times more parameters. It might therefore be interesting to train a larger model for more steps on the combined dataset, but this is beyond the scope of this project.

If the dataset consists of a single instrument, the baseline architecture is still a good choice for performing timbre transfer. If the train dataset consists of multiple instruments, as it is the case

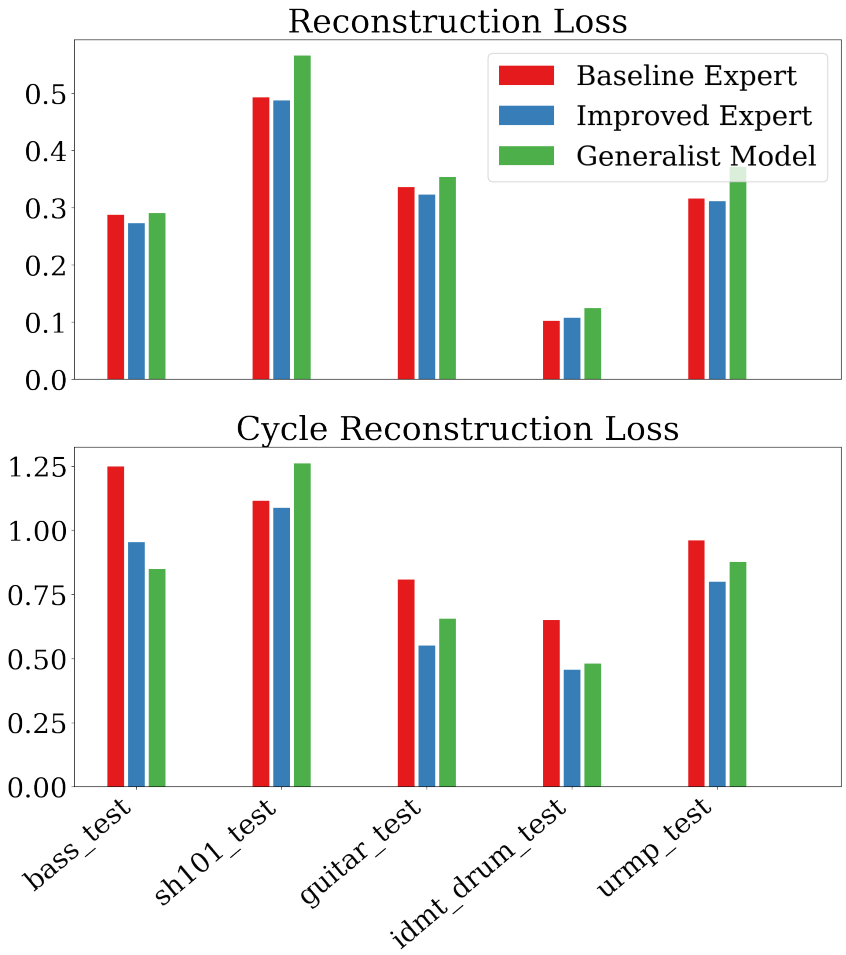


Figure 3.10: Improved Baseline: Comparison on all datasets
For each dataset, three models are compared: two experts (with new and old architecture) that were trained on the corresponding train data set containing similar sounds, and one generalist that was trained on the combined train set. It seems that the generalist model can handle a large number of instruments, but a mixture of experts trained on sounds similar to the target sound still outperforms the generalist model. In some cases, the metrics differ greatly from subjective listening tests: the reconstructions of the improved model on sh101 actually sound good compared to the baseline.

in the URMP dataset, the improved baseline model is a better choice for timbre transfer.

3.3 Few-Shot Transfer to a New Instrument

In the previous sections, the model learned to perform one-shot timbre transfer for a melody and a timbre audio sample, but with the limitation that the target timbre has to be contained in the train dataset. Other instruments sound unnatural - a reconstruction of a piano sample with the generalist model sounds as if it is somewhat influenced by the sh101 data in the train set.

For this section, I attempt to perform timbre transfer to a piano sample with only one audio sample of 96 seconds length to learn from. The expert guitar models were trained on only 38:10

minutes of audio data - so the question is whether a pre-trained model can be used to learn from much less data.

A common form of performing transfer learning is to use a pre-trained model that performs well on a related task - such as modeling various other instruments if the new task is modeling a piano - and continue training the model on data of the new task. Often, the first layers of the pre-trained model remain fixed during this fine-tuning step. Natural language models such as BERT [24] add task-specific output heads to the pre-trained model - this is especially required if the target task requires a differently shaped output compared to the pre-trained model.

In the first experiment of this section, the complete model is fine-tuned on the piano sample ("Für Elise"). After every step, reconstruction error and cycle reconstruction error are evaluated on the target timbre - another piano sample, but not the one used for fine-tuning ("Twinkle twinkle little star"). As the cycle-reconstruction loss is the relevant performance measure here, experiments in Section 3.3.2 will integrate this into the training objective. Additional melodies of the train dataset are used as intermediate melodies to compute and minimize the cycle reconstruction loss.

In Section 3.3.3, only the decoder is fine-tuned. Consider again the idea used in Jukebox, where an autoencoder is used as a part of a larger model. The autoencoder produces a latent code that serves as the ground truth for a sequence model. If a DDSP-based autoencoder were to be used in such a setting, it would be desirable to have a fixed encoder as otherwise, the sequence model would need to adapt to the new latent code. However, an improved decoder can easily be installed in an existing model - so it is desirable to fine-tune only the decoder.

This brings us to the next question: can multitask fine-tuning be used to give the model additional capabilities without compromising performance on the original task? It is possible that if the encoder has not been trained to distinguish between a piano and a guitar, there are some limits to the fine-tuning that can be achieved with the decoder-only approach.

The experiments show that each variant works to some extent - in particular listening tests show that a fine-tuned model can mimic the sound of a piano better. However, the losses do not converge to a much better value than the pretrained model. An overview of the metrics is given in Figure 3.11.

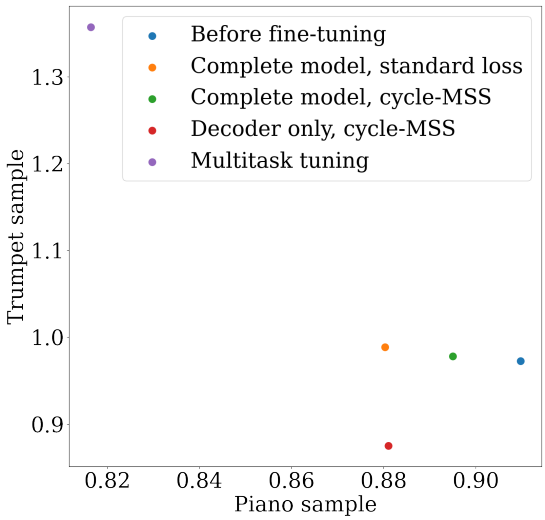


Figure 3.11: Metrics for piano and trumpet test sample for all fine-tuning strategies
Each variant leads to improved cycle-reconstruction metrics on the test sample, even though it is not the same piano. Surprisingly, the multitask fine-tuned model is the least stable and ends up having the worst performance for the trumpet test sample. The loss does not correspond perfectly to the perceived audio quality.

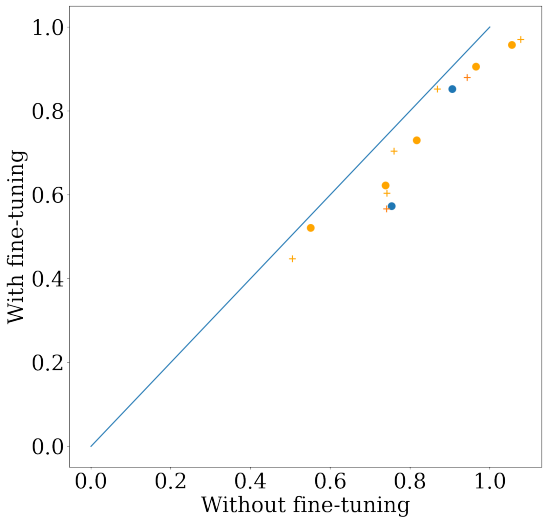


Figure 3.12: Complete model
Here, the loss is compared for various samples of the URMP-test set (orange) and the piano sample (blue). The complete model was fine-tuned on the second half of each test sample individually and evaluated on the first half. Round dots are used for the cycle-reconstruction loss, and the + markers show reconstruction loss.

One reason for this is that minor errors of the f_0 curve cannot be fixed by the model and lead to a large u-MSS error, especially in the upper harmonics, where the pitch difference is multiplied by the harmonic index. As a way to overcome this, I tried using u-MSS with mel-spaced frequency bins. The mel-scale attempts to be a perceptually uniform scale and has a lower frequency resolution in the upper frequencies. The experiments in Section 3.3.5, however, do not solve the issue of fluctuating loss during fine-tuning.

3.3.1 Fine-Tuning the Complete Model

For the first experiment, training of the autoencoder is simply resumed for 100 steps with snippets of the fine-tuning data. The best performance occurs after seven steps of fine-tuning. Indeed, at this point, the cycle reconstruction loss and the reconstruction loss have improved - however, the reconstruction loss is surprisingly higher than the cycle-reconstruction loss. Subjectively, the synthesized audio sounds very much like a piano - even after only seven steps of

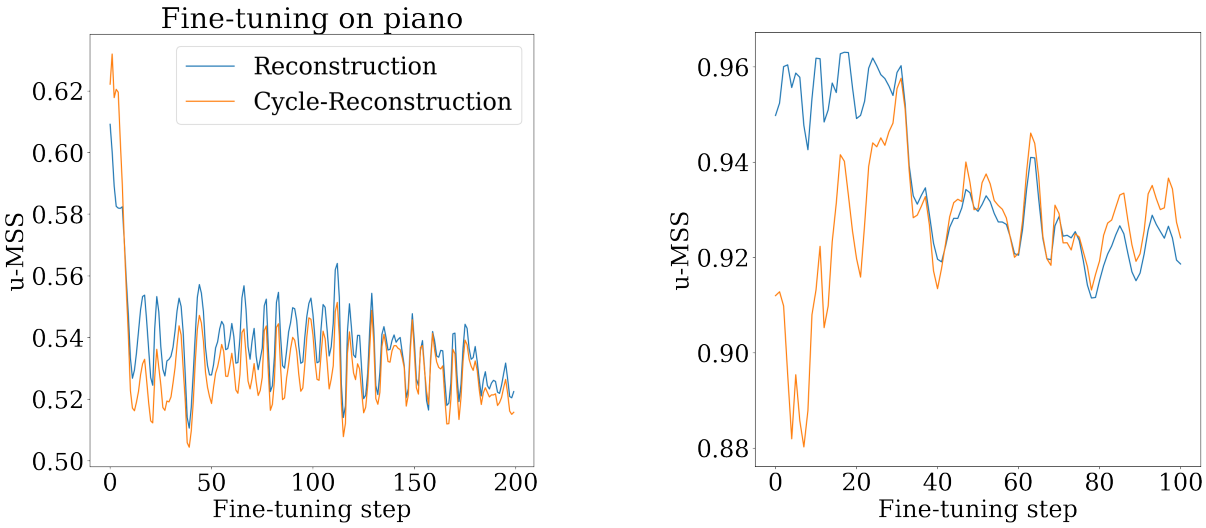
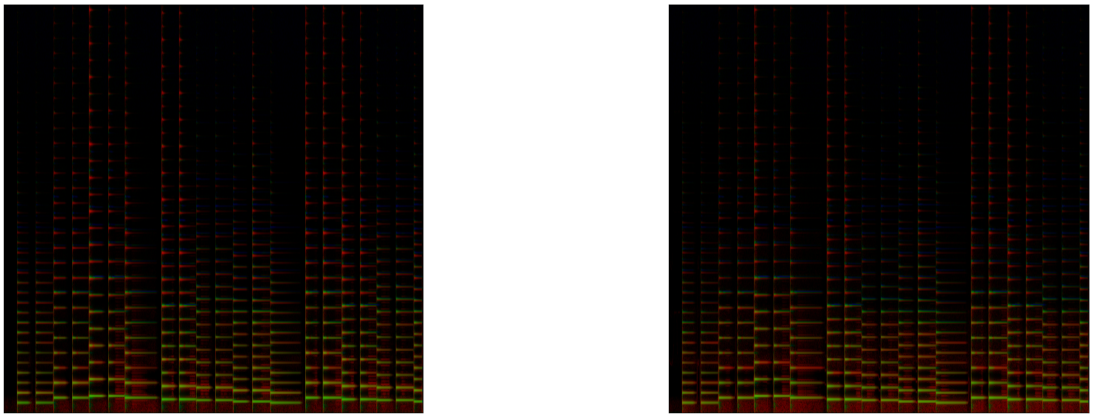


Figure 3.13: Fine-Tuning the Complete Model



Before fine-tuning

After fine-tuning

Figure 3.14: Cycle reconstruction of a piano sample before and after fine-tuning
Even though the heatmaps look similar, a clear difference can be heard between the two reconstructions. After fine-tuning for only seven steps, the audio sounds clearly like a piano.

fine-tuning. The complete model with fine-tuning can be used to synthesize new melodies with the timbre of any monophonic instrument in this colab notebook.

A poor property of this approach is the fact that fine-tuning does not seem to converge reliably. Instead, both reconstruction and cycle reconstruction losses are going up and down after the first few iterations. This issue persists during the rest of these experiments.

Considering that the loss on the train sample does go down quite reliably, the larger error on the test sample can also be due to differences between the sound of the two piano samples. This also explains why the loss seems to be much worse on the test sample, even while the audio sounds like a very convincing piano.

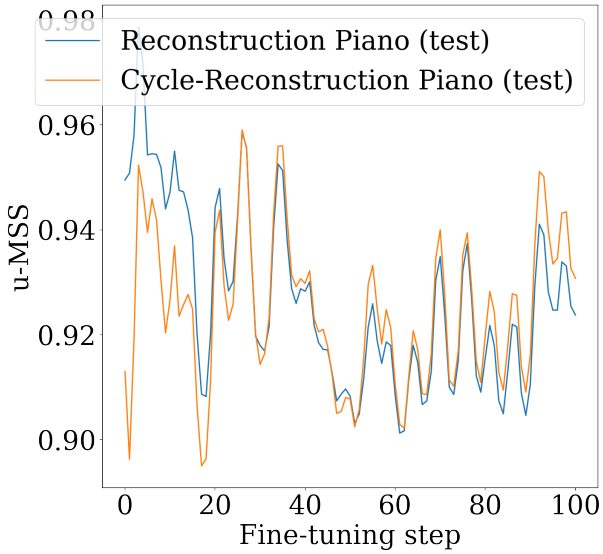


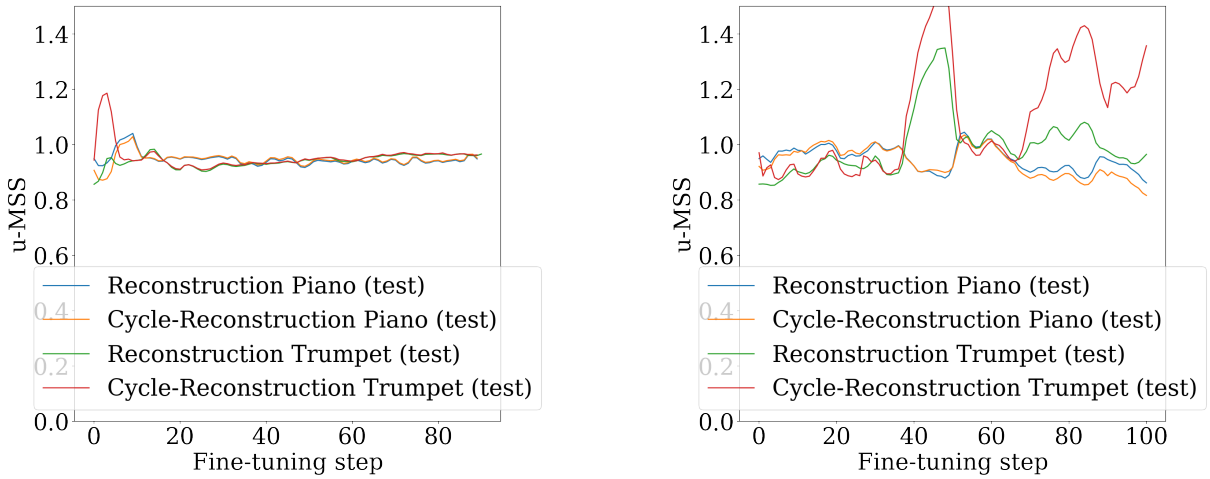
Figure 3.15: Fine-Tuning the Complete Model, Cycle-Reconstruction Loss as Training Objective

3.3.2 Cycle-Reconstruction Loss as Training Objective

In the previous experiments, the cycle reconstruction loss was used as the main performance measure. It is therefore a natural choice to use it also as a training objective. The complete loss used to train this model is

$$
\begin{aligned}
z_x, f_{0,x}, ld_x &= encode(x) \\
z_y, f_{0,y}, ld_y &= encode(y) \\
rec &= decode(z_x, f_{0,x}, ld_x) \\
transfer &= decode(z_x, f_{0,y}, ld_y) \\
\hat{z_x}, \hat{f_{0,y}}, \hat{ld_y} &= encode(transfer) \\
cycled &= decode(\hat{z_x}, f_{0,x}, ld_x) \\
loss &= MSS(x, rec) + MSS(x, cycled)
\end{aligned}
\tag{3.4}
$$

where x is a sample of the txarget timbre, and y is sound of any intermediate melody - here taken from the combined train set. The best performance is reached after 17 steps of fine-tuning and sounds a bit better than the model trained without cycle-reconstruction loss as objective. It is therefore kept for the following experiments.



Only piano

Piano + combined train

Figure 3.16: Fine-Tuning the Decoder, single task vs multitask

3.3.3 Fine-Tuning the Decoder Only

In the next experiment, the encoder remains fixed, and the decoder is fine-tuned. Now, in addition to the metrics for the target timbre, consider the performance of the fine-tuned model for a trumpet sample. We can observe that the model does not unlearn its ability to model other instruments, but the performance varies in a similar way as the performance on the piano.

The overall sound quality is not significantly improved, and the minimal loss is reached after two steps of fine-tuning already.

3.3.4 Multitask Fine-Tuning

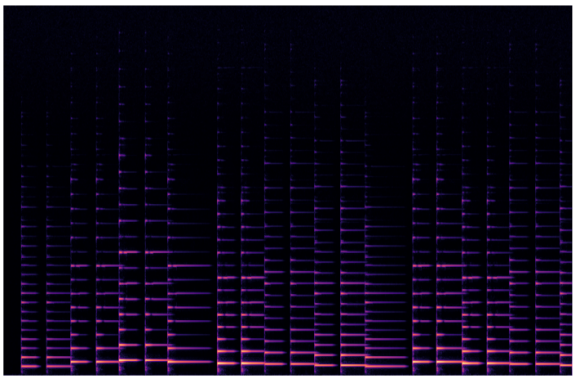
In the previous experiments, the model did not unlearn to model other instruments to a large degree. This suggests we can even fine-tune it on the original training data and the new data combined. For this experiment, exactly the same setup as in the previous setup has been used with the only change, that now only two out of four samples in the batch are of the piano, while the other two samples are taken from the combined train set.

As can be seen in Figure 3.16, the model behaves in unexpected ways during training. The best cycle-reconstruction loss is reached at the very end of the 100 steps of training, but it sounds worse than samples generated in the previous experiments. For potential use cases, this means that rather than continuously fine-tuning a model on more and more instruments, it is better to adapt the model each time it is used for an instrument that the pretrained autoencoder does

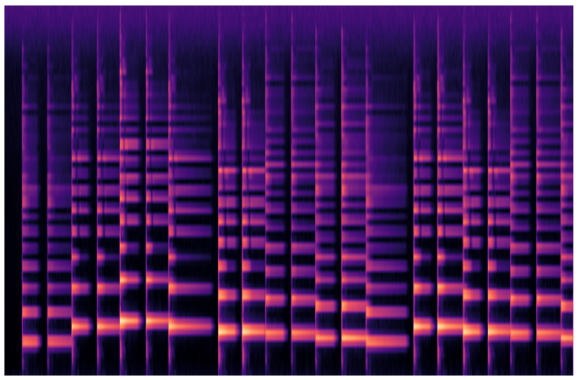
not handle sufficiently well. Due to the small number of update steps required to fine-tune the model, fine-tuning and inference combined can still be done in a matter of minutes.

3.3.5 Mel-MSS

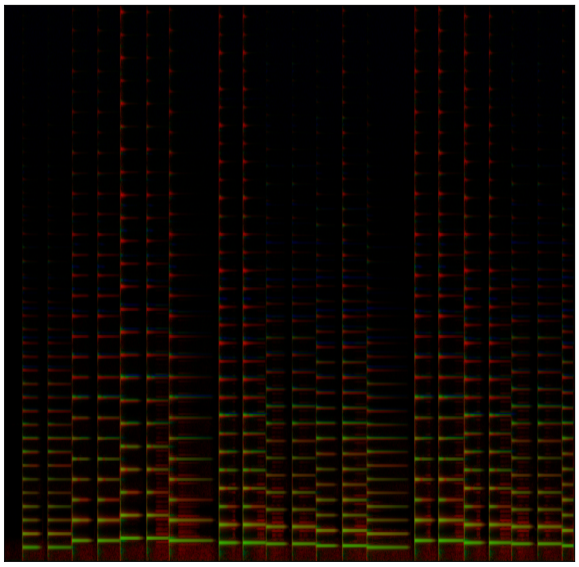
If we look closely at the error heat maps for a reconstruction of the test sample, we can see that there are many small errors at the high frequencies, simply because f_0 seems to be slightly off. This can be observed in Figure 3.17 in the form of red areas with a blue area above them. Since the f_0 extraction is not learned by the autoencoder, we do not necessarily want to penalize this type of error. This motivates the use of mel-spaced spectrograms for computing the MSS, which don't suffer from this problem due to their lower frequency resolution. Fine-tuning with this objective, however, does not lead to a better convergence, as Figure 3.18 shows. Fine-tuning was otherwise done as in Section 3.3.1 - i.e. the complete model was trained including the encoder, and no cycle-reconstruction loss or multitask fine-tuning was used. The resulting audios sound similar to the variant without mel-spectrograms. Therefore this approach does not add to the final performance and hence has been discarded.



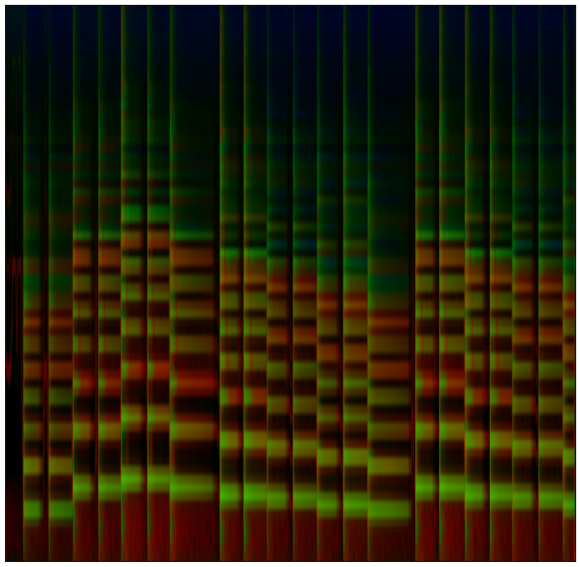
Spectrogram



Mel-Spectrogram



Spectrogram Error



Mel-Spectrogram Error

Figure 3.17: Mel-Spectrogram Error
A mel-spectrogram has a lower frequency resolution in the upper part, allowing for minimal errors of the predicted f_0-curve. This motivates to use them in the multi-scale spectrogram loss.

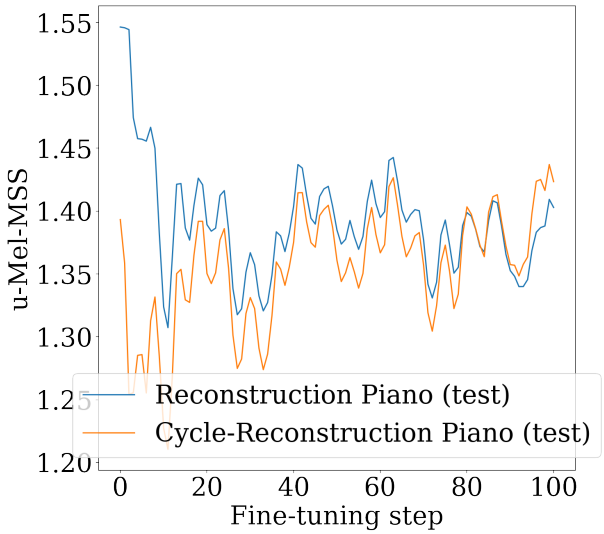


Figure 3.18: Fine-Tuning with mel-MSS

CHAPTER 4

CONCLUSION & FUTURE WORK

In the previous section, a number of changes have been proposed that allow training a single autoencoder that can model a wide range of sounds from different instruments. While the baseline DDSP model with z-encoder can also reconstruct sounds of different instruments, it is not straightforward to synthesize new realistic sounds with it, since it is unclear how to obtain a good sequence of z-encodings. We can solve this problem by using a single timbre vector extracted from a recording rather than time-distributed features. The time-constant z models produce better audios for timbre transfer that sound less noisy, and this can be measured using the cycle-reconstruction loss.

Two other modifications, namely the adapted way of calculating loudness and the use of u-MSS as a loss function, have been shown to further improve the performance of the model and allow the synthesis of near-realistic sounds.

Even better quality can be achieved by fine-tuning the model on the target timbre. A pretrained general model quickly learns to adapt to a specific instrument, requiring only a minute of training data even for completely new instruments. How to perform fine-tuning has implications on how the model can be used for other tasks besides reconstruction or timbre transfer.

In this chapter, a number of potential applications and future directions are discussed. A huge constraint of the models considered so far is that they only work on clean monophonic audio. To overcome this, Section 4.1 suggests an idea on how to perform audio source separation jointly with encoding each source into a similar latent representation as it is done in the monophonic case.

Another exciting direction is building models to obtain higher-level abstractions of audio, such as MIDI. As MIDI is a widely used format for music, models that decode MIDI can easily be used with existing tools to create music. Other works like Huang, Vaswani, Uszkoreit, Shazeer, Simon, Hawthorne, Dai, Hoffman, Dinculescu, and Eck [25] use transformers to generate music as a sequence of MIDI events. A combination of a language model for MIDI sequences with

another model that decodes MIDI to realistic waveforms can therefore be used to generate new music in an interpretable and modifiable way.

Abstractions like MIDI necessarily discard information on details. If neural networks can freely choose their latent encodings, it relies on the loss function to guide the network towards representations that preserve the relevant information. In the case of compressing audio, the relevant information is determined by human perception and approximated using loss functions like the MSS loss. If a model cannot freely choose the structure of the latent representation - for example, if we train it to encode to MIDI or the f_0, ld, z representation - then this representation can limit the information that can be recovered. In order to synthesize singing, more information than MIDI can encode is required: lyrics or details of pronunciation are beyond the scope of MIDI. An idea on how to combine the advantages of human informed high-level abstractions with the flexibility of unconstraint latent representations is discussed in Section 4.3.

4.1 Polyphonic Audio

Monophonic audio rarely occurs in the wild. It would therefore be useful to build an autoencoder that can separate different audio sources from a mixed signal such that each source can be modeled separately in a latent code like z, f_0, ld. This problem could be approached in various ways, with the most obvious being to use one encoder that controls k decoders by outputting k latent vectors. The decoders can be pretrained on monophonic audio as their role is to synthesize single monophonic sources of the mixed signal.

Another approach is to feed the input repeatedly to the model and extract monophonic audio sources iteratively. In this case, the encoder would have two inputs: one for the mixed signal and one for the summed previous outputs of the decoder.

One obstacle to training these two approaches in an unsupervised manner is that loudness and pitch cannot be extracted as easily as they can be with monophonic audio. Even the monophonic autoencoder cannot learn to extract f_0 in an unsupervised way, which is why CREPE is used. An interesting analysis of reasons for this is done by Turian and Henry [26] and additional experiments regarding this are described in Section A.2.

On the other hand, it is easy to train a supervised model that can extract f_0 of a monophonic audio sample. Therefore, one idea to teach the model to deal with polyphonic audio is to use a constructed dataset of mixed monophonic sounds for training, such that f_0 and loudness can be learned in a supervised way, while z is learned in an unsupervised way.

If it succeeds, the resulting autoencoder can then separate the individual audio sources by considering each decoder's output individually. The findings of Section 3.3 suggest that fine-tuning on each sample can help to improve the reconstruction accuracy. This can also be evaluated in the context of polyphonic audio: here, the f_0 encoder would be fixed as we know that the gradients provide a poor learning signal, but the z encoder and the decoders can adapt to the specific sample.

4.2 Higher-level Abstractions

Unconstrained latent representations obtained by autoencoders are difficult to manipulate in a targeted way. The power of the DDSP autoencoder discussed in this thesis arises from the latent code that follows a human-designed structure, such that we can easily modify only the melody or the timbre.

Certain combinations of melody and z-vector, however, sound poor. A melody extracted from singing transferred to the timbre of a piano is an example of this phenomenon. The reason lies in the fact that the pitch and loudness curves are also typical of certain instruments: a piano note is always played loudest shortly after the key is hit - this is called the attack of the amplitude envelope - and then decays. The f_0 and loudness curves of singing, on the other hand, are much less constrained.

To overcome this limitation, higher-level abstractions can be explored. A widely used format in music production is MIDI. MIDI are control sequences for electrical keyboards that control synthesizers. It assumes a discrete set of notes that can be pressed with velocities of a certain range. Different synthesizers can decode the same MIDI to waveforms with different timbre. The pitch and loudness of the generated audio also depend on the specific synthesizer used because these are lower-level descriptors than MIDI.

Therefore, an interesting direction of research is to build an additional autoencoder that learns to extract a MIDI-like encoding on top of the f_0, ld, z representation. The new decoder would then learn to predict a realistic f_0, ld curve from the MIDI sequence and the z vector. Therefore, the model can learn to adjust the details of the melody to the instrument it tries to model and hopefully produce a more realistic sound for timbre transfer.

Additionally, this can be combined with existing work on modeling MIDI sequences to generate music [27] [28]. Artists can use it to co-create music and modify either the MIDI level or dimensions of z.

4.3 Combining Human-Informed Abstractions with Unconstrained Latent Representations

While it is an advantage to abstract away details sometimes, it can also be a limitation. Models discussed so far all lack the flexibility to generate complex audio like singing, which has important time-varying features besides pitch and loudness. Models like VQ-VAEs with a less constrained latent space have this power but lack controllability.

As another potential line of future work, it can be explored how to combine the advantages of human-informed abstractions with the flexibility of unconstrained latent representations. One idea is to reintroduce a time-varying latent variable in addition to the time-constant timbre vector and use regularization techniques to reward the model for putting as much information as possible into the most abstract latent representation - i.e. to reward using the timbre vector to encode information over using the time-distributed latent variable. In the case of singing, a desirable outcome would be to disentangle the voice, which corresponds to the time-constant part, from the lyrics.

Appendix

APPENDIX A
WHY WE NEED CREPE

A natural question in the DDSP autoencoder setup is why CREPE is needed. Deep learning, in most cases, benefits from learning end to end compared to composing multiple models that specialize on subtasks of a complex problem. Reasons for this can be that human designed subtasks are not the ideal way to break down a problem for neural networks, and that errors made by different components accumulate.

We observe this problem in the DDSP autoencoder as well: if the predicted f_0 is not exactly the same as the true f_0, the autoencoder has no chance of synthesizing the upper harmonics correctly, which is also discussed in Section 3.3.5.

A.1 Spectral Losses Provide Poor Gradients for Learning Pitch

Instead of using a pretrained model to predict f_0, we can also let the encoder output the f_0 itself. Surprisingly, this does not work, and the reason is not an architectural limitation of the encoder. To proof this, I first evaluated different encoder architectures and found some that can learn to extract the pitch in a supervised fashion using the CREPE output as the ground truth. More details on this can be found in Section A.2. Then, I used the successful architecture in an unsupervised DDSP autoencoder with the same decoder architecture and loss as in the improved baseline. Note that as f_0 is a direct input to the harmonic synthesizer, the meaning of this latent variable as pitch is strictly enforced. The trained model does not learn to extract the pitch, and hence cannot learn to reconstruct an audio signal.

This agrees with the findings of Turian and Henry [26], who show that spectral based losses provide a poor learning signal for unsupervised f_0 learning. Concrete, they focus simplified setting of pure sine waves

$$x_t(f_0, \Phi) = \sin(t f_0 + \Phi) \tag{A.1}$$

MSS	0.482
u-MSS	0.52
Spectral Centroid Loss	0.514

Table A.1: Gradient Accuracy of Various Loss Functions for Pitch Learning
For each loss, 1000 pairs of $f_0, \hat{f}_0$ are computed to evaluate the gradient accuracy.

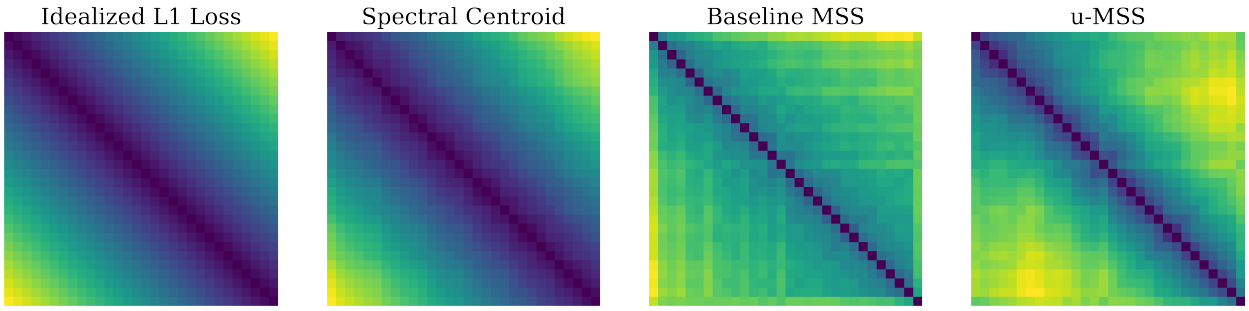


Figure A.1: Global Loss Landscapes for Pitch Learning
For each loss, a heatmap shows the error of two sine waves with frequencies according the x- and y-coordinates ranging from 40Hz-100Hz. The ideal loss landscape is plotted simple computes $|x - y|$. This is recovered by the L1 distance of the spectral centroids. The MSS loss and the u-MSS loss have a different global loss landscape, which may hinder pitch learning.

and analyze the gradient accuracy, which is defined as the percentage of the times that the gradient points into the correct direction if random values for $f_0, \Phi, \hat{f}_0, \hat{\Phi}$ are used:

$$P_{f_0,\Phi,\hat{f}_0,\hat{\Phi}\sim\mathcal{U}}[sign(\frac{\delta}{\delta_{\hat{f}_0}} MSS(x(f_0,\Phi), x(\hat{f}_0,\hat{\Phi}))) = sign(f_0 - \hat{f}_0)] \tag{A.2}$$

Many spectral based losses like pure L_1 distance of a single (not mulitscale) spectrogram or log spectrogram losses have a gradient accuracy of less than 55%. The MSS performs a bit better with a gradient accuracy of 0.771 if $|f_0 - \hat{f}_0| \to 0$ and scores above 0.9 if $|f_0 - \hat{f}_0|$ is large. I repeated these experiments with the MSS, u-MSS and the spectral centroid. Table A.1 shows the gradient accuracies, and they are significantly worse than the results reported in Turian and Henry [26] - it is unclear why this is the case, but it supports their main finding even stronger.

For the first two, even the global loss landscape that can be seen in Figure A.1 looks quite different than the ideal L_1 distance $|f_0 - \hat{f}_0|$. Since the spectral centroid of a pure sine wave recovers the true f_0 exactly up to numerical errors, its global loss landscape looks perfect for learning the pitch.

However if we plot the gradient $\delta/\delta_{\hat{f}_0} J(x(f_0,\Phi), x(\hat{f}_0,\hat{\Phi}))$, we observe that the gradient oscillates strongly which explains the poor gradient accuracy, see Figure A.2. We know that

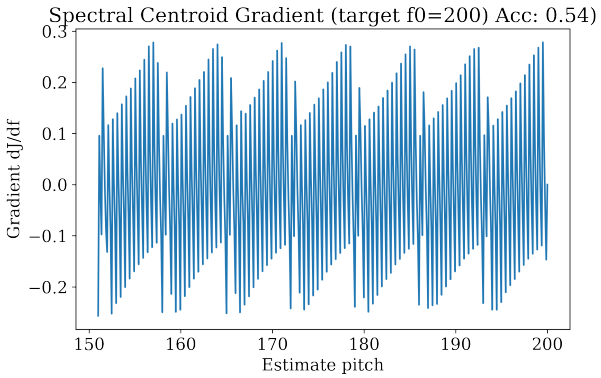


Figure A.2: Oscillating Gradient of f_0

For a target signal with $f_0 = 200Hz$, the gradient of the spectral centroid based loss is plotted for varying $\hat{f}_0$ (x-axis). The gradient oscillates strongly and points into the wrong direction almost half the time.

the spectral centroid is exactly correct whenever the frequency can be represented by a single basis vector of the STFT, that is if the resulting spectrogram has a single frequency activation. If we would only consider these points, no oscillation could be observed and the gradient of the spectral centroid would point into the right direction 100% of the time, as it appears to be the case in Figure A.1. However, if the true frequency is somewhere between two basis frequencies, spectral leakage occurs and introduces an error on the gradient. This explains the oscillation observed in Figure A.2 and therefore offers the first explanation for the poor gradient accuracy of spectral losses described in Turian and Henry [26].

A potential solution that is left for future work is to override the analytical gradient of f_0 with a numerically evaluated slope of the loss over a small interval close to the true f_0, evaluated only at those frequencies that make up the STFT basis.

A.2 Supervised f_0 Learning

In Engel, Hantrakul, Gu, and Roberts [1], an autoencoder that jointly learns to predict f_0 and z is trained using a perceptual loss that uses activations of the pretrained CREPE model. As this approach requires CREPE for training, the encoder can also be trained as a student while CREPE acts as the teacher.

To validate this idea, I searched for an architecture that is able to learn f_0 in a supervised manner. This new model first computes a multiscale spectrogram representation of the input signal as well as MFCCs. The spectrograms are stacked into channels and fed into a stack of

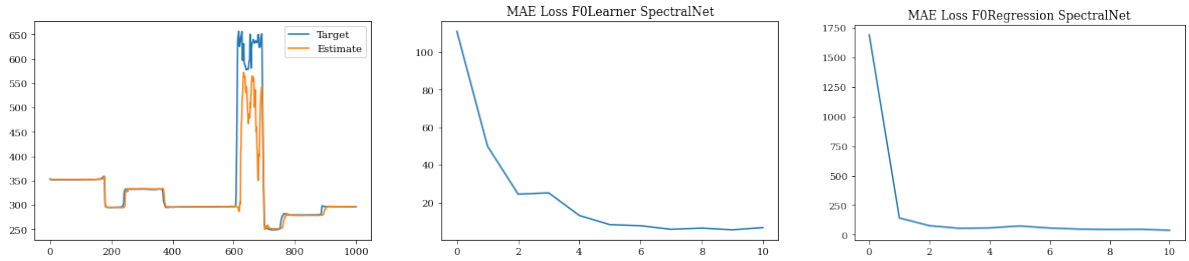


Figure A.3: Supervised f_0 Learning

Left: The prediction of a model trained to predict the correct bin of f_0. In the area where the predicted pitch does not align with the true pitch, no audible sound is played. *Center:* Convergence of the Mean Absolute Error (MAE) of the predicted pitch and the true pitch of the same model. *Right:* Convergence of a model with the same architecture, trained directly to optimize the MAE

convolutional layers. Then, the output is concatenated with the MFCCs and fed to a GRU layer. The new part of the computational graph involving spectrograms rather than MFCCs is added because MFCCs are designed to be pitch invariant.

CREPE frames the regression problem as a classification problem, where f_0 is binned into 360 classes. Indeed, this helps convergence compared to training the model directly with a regression loss like MSE. The classifier is trained using a cross entropy loss, and to evaluate the MAE a weighted mean of the bin centers is computed to obtain f_0. Figure A.3 shows results for both approaches.

In order to jointly learn f_0 and z, this model and its training objective can be integrated into the DDSP autoencoder and trained, while the decoder uses the true f_0 as an input. During inference, CREPE is then not needed anymore and can be replaced by the estimated f_0. Training a DDSP autoencoder in this supervised manner is currently work in progress.

APPENDIX B

INTERACTIVE DASHBOARD

For this thesis, a large number of models have been trained and tested on a number of datasets. Since subjective evaluation by listening tests remains an important part of finding perceptually good models, a systematic way to find and explore artifacts was needed. The interactive dashboard is a web application that allows the user to select a model and a dataset, and then to listen to the model's synthesized audio. For this, I implented github.com/nielsrolf/pandas_db, which is a key-value-artifact store combined with a web UI to explore data. Generated audios, images and metrics can then be searched by their meta data such as model hyperparameters. Components of this generic UI are used for the dashboard on nielsrolf.github.com?thesis.

It is structured into sections accompaning the methods part of this thesis, and each section displays a comparison of a subset of the trained models. In the top part, the user can specify model parameters such as the dataset it has been trained on or which z-aggregation has been used, an example is shown in Figure B.1. Since the models have been evaluated on multiple test datasets, the user can additionally select for which one to display metrics and artifacts.

Below this selection, plots that display the unskewed MSS test losses and the cycle-reconstruction loss are displayed. In these plots, different model parameters can be compared by selecting multiple configurations in the dropdown menus.

Below these plots, the user can listen to the synthesized audio and look at generated plots associated with these models and some audio inputs, this is shown in Figure B.2. The filters work such that all selected filters must be true and some selections lead to an empty screen. In that case, the user should remove some of the applied filters.

Section 3.2.4: Baseline vs improved baseline on all datasets

here, you can check out how the baseline vs the improved baseline sounds.Models have been trained on different datasets, note how a model trainedon guitar makes everything sound like a guitar. The baseline works well for timbre transfer in these cases, if the target timbre is also a guitar.An interesting weakness of the loss function can be seen when you look into the cycle reconstruction error of sh101 sounds.Clearly, the improved model sounds much better than the baseline model.However, the loss of the baseline model is smaller than that of the improved model!This is, why manual eevaluation remains important for tasks that target human perception.

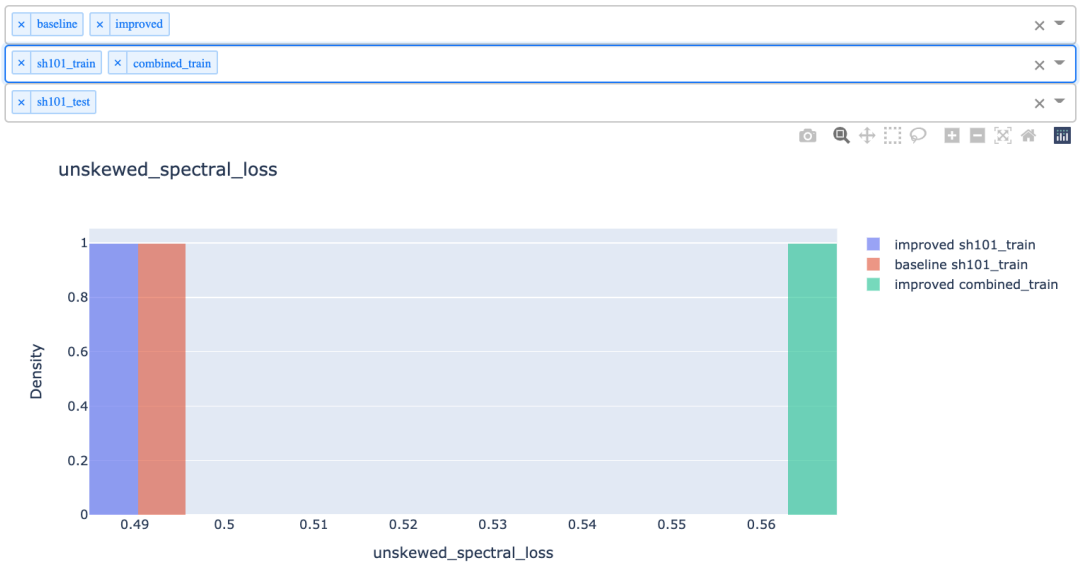


Figure B.1: Select parameters of the model and possibly the test dataset

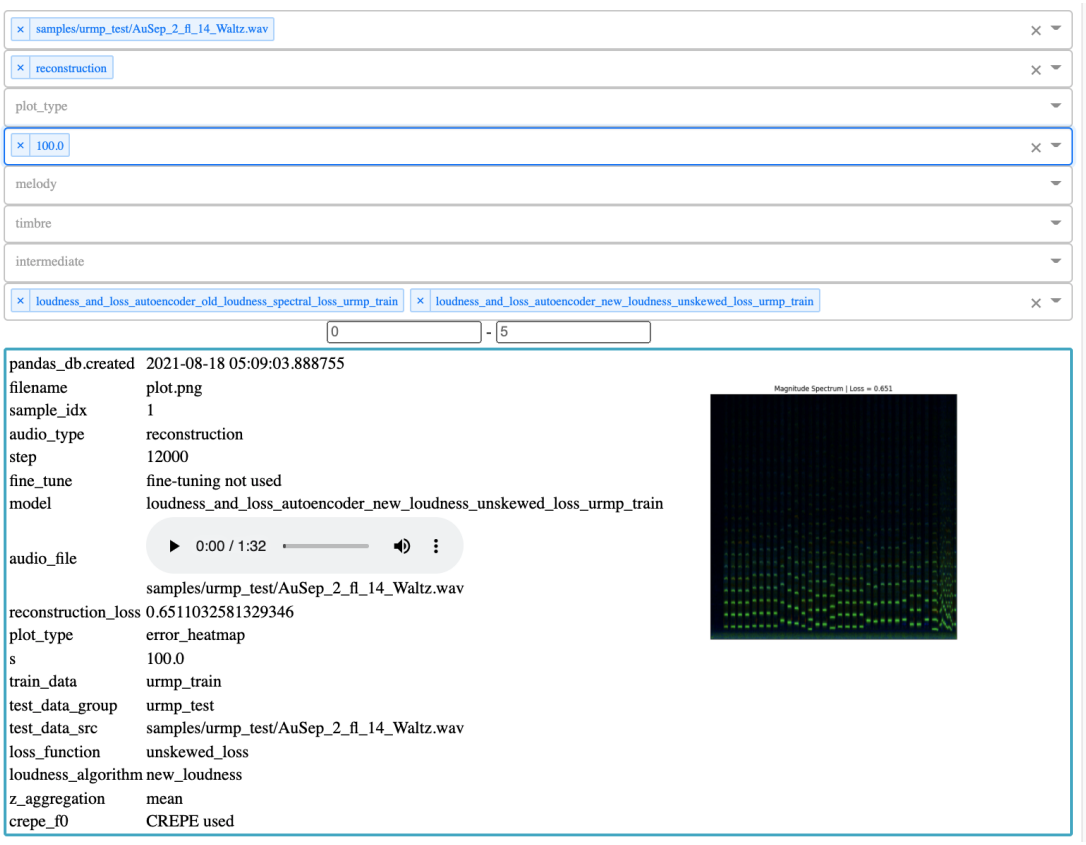


Figure B.2: Find reconstructions, timbre transfer audios and cycle reconstructions and compare models

APPENDIX C

TRAINING DETAILS

This section describes the training settings including the model architecture and hyperparameters. Models have been trained with various settings for the following parameters:

- the z-aggregation in the encoder
- the loudness algorithm
- the training loss
- the train dataset

If not explicitly mentioned, the hyperparameters and architectures described here are therefore used by all models.

Preprocessing As the very first step, all audio is resampled to 16kHz and converted to mono. Then, the pretrained CREPE model [29] of the official implementation on github.com/marl/crepe is used to extract the pitch contour f_0, and the two different versions of the loudness algorithm are applied independently on spectrograms of the audio. For training, the audio, the pitch contour and the two loudness curves are then cut into slices of 4 seconds with a hop size of 1 second. During inference, the model can be built for any audio length input and is therefore applied at once to the audio without this additional step.

Loudness Algorithm The baseline loudness algorithm first computes a spectrogram with a window size of 2048 and 75% overlap between the windows. Then, it computes:

$$\begin{aligned} s &= stft(audio) \\ amp &= |s| \\ power_{db} &= 20 \quad log_{10}(max(\epsilon, amp)) \\ loudness &= mean_T(power_{db} + A_{db}) \end{aligned} \tag{C.1}$$

The adjusted loudness algorithm uses a smaller fft size of 512 to achieve a better time-resolution. It computes:

$$\begin{aligned} s &= stft(audio) \\ power_{amp} &= |s|^2 \\ power_{amp}^{+A} &= power_{amp} * A_{amp} \\ loudness &= 10 \quad log_{10}(max(\epsilon, mean_T(power_{amp}^{+A}))) \end{aligned} \tag{C.2}$$

A refers to a constant adjustment factor per frequency - once used in the decibel scale and once used in the amplitude scale. The adjustment factor is taken from the librosa library.
The difference between the two is, that the adjusted version takes the mean in the linear domain and then converts to decibels, while the baseline version takes the mean in the log domain where it is highly sensitive to spectral leakage.

Encoder

The encoder architecture can be divided into a feature extractor and a feature aggregator. The feature extractor follows the same architecture as the encoder of [1]: in the first step, MFCCs are extracted from the audio signal. Then, instance normalization is applied on the MFCCs and in the second step, the features are passed through a recurrent neural network consisting of one GRU layer and one time-distributed fully connected layer. In the baseline architecture, this is directly passed to the decoder. The encoders with mean aggregation simply compute the mean over the time axis without any additional parameters. The confidence masked z-aggregation encoders uses two additional learnable dense layers to predict $\hat{z} \in \mathbb{R}^{(B,T,d)}$ and a confidence mask $c \in \mathbb{R}^{(B,T,d')}$ from the feature extractor output x. Note that the confidence masks dimension d' can be smaller than the dimension of d of z - in early experiments, I experimented with a single confidence value per time step. These models are referenced as 1-d masked in table Table C.1.

The feature extraction model has 843.852 trainable parameters. If no z-aggregation happens or the mean is used, this equals the total number of trainable parameters in the encoder. The confidence masked z-aggregation module comes with extra 4104 trainable parameters, or only 513 is the 1d-masked version is used.

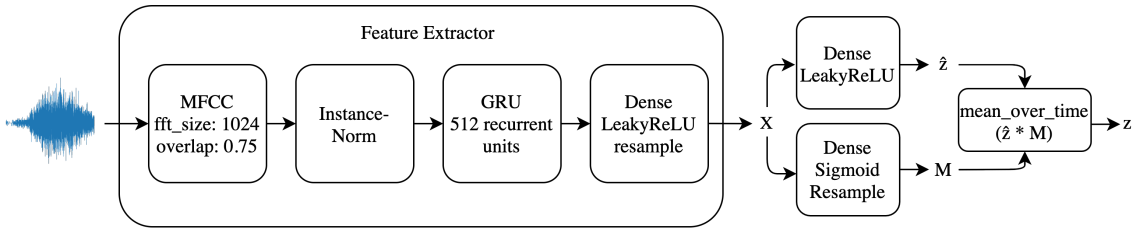


Figure C.1: Encoder architecture

Decoder The decoder gets a normalized loudness and pitch curve as an input and the additional variable z and computes controls for the harmonic synthesizer and the filtered noise synthesizer. It computes:

$$
\begin{aligned}
&(ld, f_0, z) \mapsto (a, h, n) \\
&\quad ld, f_0 \in \mathbb{R}^{(B,T,1)} \qquad \text{for us the latent frame rate} \quad T = 1000 \\
&\quad\quad z \in \mathbb{R}^{(B,T,d)} \qquad \text{in all experiments} \quad d = 16 \\
&\quad\quad a \in \mathbb{R}^{(B,T,1)} \qquad \text{amplitude envelope for the harmonic synthesizer} \\
&\quad\quad h \in \mathbb{R}^{(B,T,100)} \qquad \text{harmonic distribution} \\
&\quad\quad n \in \mathbb{R}^{(B,T,65)} \qquad \text{noise envelopes for the filtered noise synthesizer}
\end{aligned}
\tag{C.3}
$$

On these three inputs, it applies separate stacks consisting of dense layers and instance normalization. The outputs are then concatenated and passed through a GRU-layer followed by another stack of dense layers with instance normalization. The output is then split into parts that control the filtered noise and the harmonic synthesizer. In total, it has 6.407.334 trainable parameters.

MSS-Loss To compute the MSS-loss, magnitude spectrograms with sizes 2048, 1024, 512, 256, 128, 64 have been computed. The baseline MSS-loss computes the L_1-distance of those and the corresponding logmag spectrograms. The adjusted MSS-loss also computes the L_1-distance of the magnitude spectrograms and the same spectrograms after passing it through the unskewing function Equation (3.3) with $s \in \{1, 10, 100\}$. Before taking the L_1-distance, the inputs are scaled to the range $[0, 1]$.

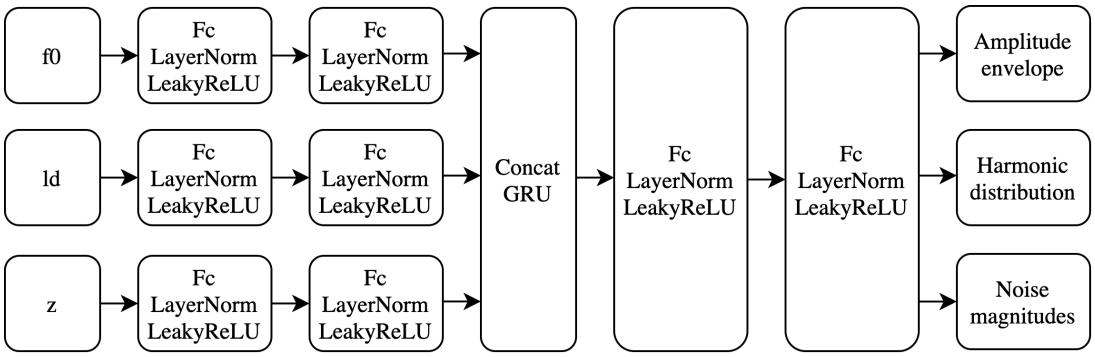


Figure C.2: Decoder architecture

C.1 Ablation Study Results

Table C.1 shows the full table of results for the ablations studies concerning the changes proposed in Section 3.2. A discussion of the results can be found in Section 3.2.4

					Reconstruction	Cycle-Reconstruction
Test Data	Train data	Train loss	loudness	z_aggregation		
bass	bass	baseline	baseline	none	0.29	1.25
		u-MSS	adjusted	masked	0.27	0.95
	combined	u-MSS	adjusted	masked	0.29	0.90
	drums	baseline	baseline	none	0.31	1.00
	guitar	baseline	baseline	none	0.45	1.08
		u-MSS	adjusted	masked	0.39	1.02
	idmt_drum	baseline	baseline	none	0.41	1.17
		u-MSS	adjusted	masked	0.60	2.29
	sh101	baseline	baseline	none	0.47	0.96
		u-MSS	adjusted	masked	0.31	0.95
	urmp	baseline	adjusted	mean	0.40	0.91
			baseline	mean	0.40	0.97
				none	0.34	0.97
		u-MSS	adjusted	1d-masked	0.36	0.92
				masked	0.36	0.88
				mean	0.35	0.92
			baseline	mean	0.34	0.93
combined	bass	baseline	baseline	none	0.55	
		u-MSS	adjusted	masked	0.49	
	combined	u-MSS	adjusted	masked	0.26	

	drums	baseline	baseline	none	0.46	
	guitar	baseline	baseline	none	0.46	
		u-MSS	adjusted	masked	0.40	
	idmt_drum	baseline	baseline	none	0.44	
		u-MSS	adjusted	masked	0.59	
	sh101	baseline	baseline	none	0.38	
		u-MSS	adjusted	masked	0.30	
	urmp	baseline	adjusted	mean	0.42	
			baseline	mean	0.47	
				none	0.43	
		u-MSS	adjusted	1d-masked	0.36	
				masked	0.39	
				mean	0.36	
			baseline	mean	0.39	
guitar	bass	baseline	baseline	none	0.67	0.97
		u-MSS	adjusted	masked	0.58	0.84
	combined	u-MSS	adjusted	masked	0.35	0.71
	drums	baseline	baseline	none	0.62	0.91
	guitar	baseline	baseline	none	0.34	0.83
		u-MSS	adjusted	masked	0.32	0.66
	idmt_drum	baseline	baseline	none	0.79	1.02
		u-MSS	adjusted	masked	1.28	2.10
	sh101	baseline	baseline	none	0.46	0.81
		u-MSS	adjusted	masked	0.43	0.77
	urmp	baseline	adjusted	mean	0.49	0.65
			baseline	mean	0.62	0.77
				none	0.56	0.89
		u-MSS	adjusted	1d-masked	0.39	0.65
				masked	0.43	0.67
				mean	0.39	0.65
			baseline	mean	0.52	0.79
idmt_drum	bass	baseline	baseline	none	0.34	0.76
		u-MSS	adjusted	masked	0.23	0.67
	combined	u-MSS	adjusted	masked	0.12	0.50
	drums	baseline	baseline	none	0.19	0.53
	guitar	baseline	baseline	none	0.41	0.77

		u-MSS	adjusted	masked	0.21	0.59
	idmt_drum	baseline	baseline	none	0.10	0.66
		u-MSS	adjusted	masked	0.11	0.49
	sh101	baseline	baseline	none	0.39	0.65
		u-MSS	adjusted	masked	0.22	0.56
	urmp	baseline	adjusted	mean	0.26	0.62
			baseline	mean	0.27	0.73
				none	0.24	0.66
		u-MSS	adjusted	1d-masked	0.24	0.65
				masked	0.24	0.65
				mean	0.23	0.63
			baseline	mean	0.23	0.73
sh101	bass	baseline	baseline	none	1.39	1.57
		u-MSS	adjusted	masked	1.27	1.62
	combined	u-MSS	adjusted	masked	0.57	1.58
	drums	baseline	baseline	none	1.06	1.46
	guitar	baseline	baseline	none	1.02	1.44
		u-MSS	adjusted	masked	0.97	1.27
	idmt_drum	baseline	baseline	none	1.02	1.64
		u-MSS	adjusted	masked	1.26	2.42
	sh101	baseline	baseline	none	0.49	1.15
		u-MSS	adjusted	masked	0.49	1.28
	urmp	baseline	adjusted	mean	1.03	1.25
			baseline	mean	1.15	1.31
				none	1.09	1.48
		u-MSS	adjusted	1d-masked	0.87	1.54
				masked	0.95	1.51
				mean	0.85	1.47
			baseline	mean	0.91	1.33
urmp	bass	baseline	baseline	none	0.58	1.31
		u-MSS	adjusted	masked	0.49	1.10
	combined	u-MSS	adjusted	masked	0.37	0.91
	drums	baseline	baseline	none	0.42	1.05
	guitar	baseline	baseline	none	0.55	1.25
		u-MSS	adjusted	masked	0.45	0.86
	idmt_drum	baseline	baseline	none	0.66	1.24

	u-MSS	adjusted	masked	0.84	2.09
sh101	baseline	baseline	none	0.59	1.01
	u-MSS	adjusted	masked	0.47	1.00
urmp	baseline	adjusted	mean	0.33	0.81
		baseline	mean	0.34	0.90
			none	0.32	0.98
	u-MSS	adjusted	1d-masked	0.28	0.81
			masked	0.30	0.83
			mean	0.29	0.83
		baseline	mean	0.30	0.93

Table C.1: Training setting of all models trained

REFERENCES

[1] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "DIFFERENTIABLE DIGITAL SIGNAL PROCESSING," p. 19, 2020.

[2] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, and I. Sutskever, "Jukebox: A Generative Model for Music," *arXiv:2005.00341 [cs, eess, stat]*, Apr. 2020, arXiv: 2005.00341.

[3] X. Wang, S. Takaki, and J. Yamagishi, "Neural source-filter waveform models for statistical parametric speech synthesis," *arXiv:1904.12088 [cs, eess, stat]*, Nov. 2019, arXiv: 1904.12088.

[4] L. Hantrakul, J. Engel, A. Roberts, and C. Gu, "FAST AND FLEXIBLE NEURAL AUDIO SYNTHESIS," p. 7, 2019.

[5] M. Michelashvili and L. Wolf, "Hierarchical Timbre-Painting and Articulation Generation," *arXiv:2008.13095 [cs, eess]*, Sep. 2020, arXiv: 2008.13095.

[6] R. Yamamoto, E. Song, and J.-M. Kim, "Parallel WaveGAN: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram," *arXiv:1910.11480 [cs, eess]*, Feb. 2020, arXiv: 1910.11480.

[7] B. Hayes, C. Saitis, and G. Fazekas, "NEURAL WAVESHAPING SYNTHESIS," p. 8,

[8] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit Neural Representations with Periodic Activation Functions," *arXiv:2006.09661 [cs, eess]*, Jun. 2020, arXiv: 2006.09661.

[9] Z. Liu, K. Chen, and K. Yu, "Neural Homomorphic Vocoder," in *Interspeech 2020*, ISCA, Oct. 2020, pp. 240–244.

[10] O. McCarthy and Z. Ahmed, "HooliGAN: Robust, High Quality Neural Vocoding," *arXiv:2008.02493 [cs, eess]*, Aug. 2020, arXiv: 2008.02493.

[11] A. v. d. 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," *arXiv:1609.03499 [cs]*, Sep. 2016, arXiv: 1609.03499.

[12] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient Neural Audio Synthesis," in *International Conference on Machine Learning*, ISSN: 2640-3498, PMLR, Jul. 2018, pp. 2410–2419.

[13] A. v. d. Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. v. d. Driessche, E. Lockhart, L. C. Cobo, F. Stimberg, N. Casagrande, D. Grewe, S. Noury, S. Dieleman, E. Elsen, N. Kalchbrenner, H. Zen, A. Graves, H. King, T. Walters, D. Belov,

and D. Hassabis, "Parallel WaveNet: Fast High-Fidelity Speech Synthesis," *arXiv:1711.10433 [cs]*, Nov. 2017, arXiv: 1711.10433.

[14] A. v. d. Oord, O. Vinyals, and K. Kavukcuoglu, "Neural Discrete Representation Learning," *arXiv:1711.00937 [cs]*, May 2018, arXiv: 1711.00937.

[15] A. Razavi, A. v. d. Oord, and O. Vinyals, "Generating Diverse High-Fidelity Images with VQ-VAE-2," *arXiv:1906.00446 [cs, stat]*, Jun. 2019, arXiv: 1906.00446.

[16] P. Dhariwal and A. Nichol, "Diffusion Models Beat GANs on Image Synthesis," *arXiv:2105.05233 [cs, stat]*, Jun. 2021, arXiv: 2105.05233.

[17] Z. Kong, W. Ping, J. Huang, K. Zhao, and B. Catanzaro, "DiffWave: A Versatile Diffusion Model for Audio Synthesis," *arXiv:2009.09761 [cs, eess, stat]*, Mar. 2021, arXiv: 2009.09761.

[18] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "GANSynth: Adversarial Neural Audio Synthesis," *arXiv:1902.08710 [cs, eess, stat]*, Apr. 2019, arXiv: 1902.08710.

[19] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," *arXiv:1412.3555 [cs]*, Dec. 2014, arXiv: 1412.3555.

[20] J. Abeßer, C. Dittmar, P. Kramer, and G. Schuller, "Idmt-smt-bass-single-track," *16th International Conference on Digital Audio Effects (DAFx)*, 2013.

[21] C. Kehling, J. Abeßer, C. Dittmar, and G. Schuller, "Automatic tablature transcription of electric guitar recordings by estimation of score- and instrument-related parameters," *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx, 2014)*, 2014.

[22] C. Dittmar and D. Gärtner, "Idmt-smt-drums," *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx, 2014)*, 2014.

[23] B. Li, X. Liu, K. Dinesh, Z. Duan, and G. Sharma, "Creating a Multitrack Classical Music Performance Dataset for Multimodal Music Analysis: Challenges, Insights, and Applications," *IEEE Transactions on Multimedia*, vol. 21, no. 2, pp. 522–535, Feb. 2019.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, May 2019, arXiv: 1810.04805.

[25] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, "Music Transformer," *arXiv:1809.04281 [cs, eess, stat]*, Dec. 2018, arXiv: 1809.04281.

[26] J. Turian and M. Henry, "I'm Sorry for Your Loss: Spectrally-Based Audio Distances Are Bad at Pitch," p. 16,

[27] I. Simon and S. Oore, *Performance rnn: Generating music with expressive timing and dynamics*, https://magenta.tensorflow.org/performance-rnn, Blog, 2017.

[28] A. Roberts, J. Engel, C. Raffel, C. Hawthorne, and D. Eck, "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music," *arXiv:1803.05428 [cs, eess, stat]*, Nov. 2019, arXiv: 1803.05428.

[29] J. W. Kim, J. Salamon, P. Li, and J. P. Bello, "CREPE: A Convolutional Representation for Pitch Estimation," *arXiv:1802.06182 [cs, eess, stat]*, Feb. 2018, arXiv: 1802.06182.