DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Data-Driven Procedural Audio: Procedural Engine Sounds Using Neural Audio Synthesis

ANTON LUNDBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Data-Driven Procedural Audio: Procedural Engine Sounds Using Neural Audio Synthesis

ANTON LUNDBERG

Master in Computer Science and Engineering, specializing in Systems, Control and Robotics
Date: June 25, 2020
Supervisor: Bob Sturm
Examiner: Sten Ternström
School of Electrical Engineering and Computer Science
Host company: Ericsson AB
Swedish title: Datadrivna Metoder för Procedurellt Ljud: Procedurella Motorljud Genom Neural Ljudsyntes

Abstract

The currently dominating approach for rendering audio content in interactive media, such as video games and virtual reality, involves playback of static audio files. This approach is inflexible and requires management of large quantities of audio data. An alternative approach is procedural audio, where sound models are used to generate audio in real time from live inputs. While providing many advantages, procedural audio has yet to find widespread use in commercial productions, partly due to the audio produced by many of the proposed models not meeting industry standards.

This thesis investigates how procedural audio can be performed using data-driven methods. We do this by specifically investigating how to generate the sound of car engines using neural audio synthesis. Building on a recently published method that integrates digital signal processing with deep learning, called Differentiable Digital Signal Processing (DDSP), our method obtains sound models by training deep neural networks to reconstruct recorded audio examples from interpretable latent features. We propose a method for incorporating engine cycle phase information, as well as a differentiable transient synthesizer. Our results illustrate that DDSP can be used for procedural engine sounds; however, further work is needed before our models can generate engine sounds without undesired artifacts and before they can be used in live real-time applications. We argue that our approach can be useful for procedural audio in more general contexts, and discuss how our method can be applied to other sound sources.

Sammanfattning

Det i dagsläget dominerande tillvägagångssättet för rendering av ljud i interaktiva media, såsom datorspel och virtual reality, innefattar uppspelning av statiska ljudfiler. Detta tillvägagångssätt saknar flexibilitet och kräver hantering av stora mängder ljuddata. Ett alternativt tillvägagångssätt är procedurellt ljud, vari ljudmodeller styrs för att generera ljud i realtid. Trots sina många fördelar används procedurellt ljud ännu inte i någon vidare utsträckning inom kommersiella produktioner, delvis på grund av att det genererade ljudet från många föreslagna modeller inte når upp till industrins standarder.

Detta examensarbete undersöker hur procedurellt ljud kan utföras med datadrivna metoder. Vi gör detta genom att specifikt undersöka metoder för syntes av bilmotorljud baserade på neural ljudsyntes. Genom att bygga på en nyligen publicerad metod som integrerar digital signalbehandling med djupinlärning, kallad Differentiable Digital Signal Processing (DDSP), kan vår metod skapa ljudmodeller genom att träna djupa neurala nätverk att rekonstruera inspelade ljudexempel från tolkningsbara latenta prediktorer. Vi föreslår en metod för att använda fasinformation från motorers förbränningscykler, samt en differentierbar metod för syntes av transienter. Våra resultat visar att DDSP kan användas till procedurella motorljud, men mer arbete krävs innan våra modeller kan generera motorljud utan oönskade artefakter samt innan de kan användas i realtidsapplikationer. Vi diskuterar hur vårt tillvägagångssätt kan vara användbart inom procedurellt ljud i mer generella sammanhang, samt hur vår metod kan tillämpas på andra ljudkällor.

Contents

Acknowledgments **vii**

1 Introduction **1**
1.1 Problem Formulation 3
1.2 Aim 3
1.3 Objectives and Limitations 4
1.4 Structure Outline 5

2 Background **6**
2.1 Procedural Audio 6
2.1.1 Methods for Procedural Audio 7
2.1.2 Procedural Engine Sounds 8
2.2 Sound Analysis/Synthesis 10
2.2.1 Spectral Modeling Synthesis 10
2.2.2 Transients 12
2.3 Neural Audio Synthesis 13
2.3.1 Autoregressive and Flow-Based Models 13
2.3.2 Differentiable Digital Signal Processing 14
2.3.3 The DDSP Autoencoder 15

3 Methodology **18**
3.1 Sounds and Data 18
3.1.1 Recording Process 19
3.1.2 Preprocessing 20
3.2 Models 25
3.2.1 Original DDSP Settings (vanilla-DDSP) 26
3.2.2 Multi-Scale Mel Loss (mel-DDSP) 27
3.2.3 Inharmonic Partials (cylinder-DDSP) 28
3.2.4 Engine Cycle Phase as Input (phase-DDSP) 29
3.2.5 Higher-Dimensional GRU (large-DDSP) 29

3.2.6 Synthesizing Transients (transient-DDSP) 30
3.3 Training and Loss Functions 30
3.3.1 Multi-Scale STFT Loss 30
3.3.2 Multi-Scale Mel Loss 31
3.4 Evaluation 32

4 Results 34
4.1 Recordings and Data 34
4.2 Models and Training 35
4.3 Overfitting to a Small Dataset 38
4.4 Reconstruction From Recorded Inputs 44
4.5 Control by Synthetic Inputs 54

5 Discussion 64
5.1 Audio Synthesis Performance 64
5.2 Model Inputs 65
5.3 Synthesizers 66
5.4 Evaluation Metrics and Loss Functions 68
5.5 Fine-Tuning 69
5.6 Other Sound Sources 69
5.7 Possible Applications and Relevance 70
5.8 Notes on Sustainability and Ethics 71

6 Conclusions & Future Work 73
6.1 Conclusions 73
6.2 Future Work 74

Bibliography 76

A Model Details 80
A.1 Decoder Networks 80
A.2 Derivation of the Number of Mel Bins 81

Acknowledgments

This thesis work was conducted at the Audio Technology section of Digital Representation & Interaction (DRI) at Ericsson Research in Kista. I would like to express my deepest appreciation to Tommy Falk, my industry supervisor and senior researcher at DRI, for reaching out to me with an incredibly interesting research topic and for all our constructive discussions throughout my work. I also want to thank Harald Pobloth, head of section Audio Technology at DRI, for putting his confidence in me and letting me perform my thesis work at his department.

I would like to extend my deepest gratitude to Bob Sturm, my academic supervisor and associate professor at KTH, for guiding me through my thesis and for your helpful advice while writing this report. I also want to thank my examiner Sten Ternström, professor at KTH, for reviewing my work.

Lastly, I want to express my warmest gratitude to my wife Linnéa Håcansdotter for supporting me throughout my studies, especially during my thesis work. You are my other half and without your support I would not have been able to carry this through.

Chapter 1

Introduction

This thesis studies the problem of performing procedural audio using data-driven methods by investigating how neural audio synthesis can be applied to procedural engine sounds. Our method involves obtaining models for synthesis of engine sounds by training neural networks to reconstruct recorded audio examples from interpretable latent features. Such models could provide flexible alternatives to the currently dominating method of using playback of static recordings in interactive media.

The primary focus of interactive media tends to fall on the graphics. This holds true both for more traditional applications such as video games, and for more recent applications such as virtual reality. While the graphics are rendered procedurally based on user interaction and the state of the environment, the audio content usually consists of prerecorded material which is played back when triggered by events in the virtual environment. While ensuring high quality of the audio material, this approach exhibits a range of drawbacks including lack of flexibility and requires management of large quantities of audio data [1].

As immersive media such as virtual and augmented reality are gaining more attention in both academic research and commercial productions, efforts are being made to standardize their representations. The Moving Picture Expert Group (MPEG) is currently working on a new era of standards for immersive applications, referred to as MPEG-I [2]. Apart from containing video coding standards, MPEG-I also contains standards for immersive audio. To be able to benefit from its advantages, procedural audio has been discussed as a possible future feature of the standard.

In procedural audio, sound is generated in real time by audio objects using live inputs from the virtual environment [1]. These objects contain models of

their target sounds, and these models can be used to generate infinite variations of that sound. In theory, procedural audio brings an array of potential advantages such as increased sound variability at controllable level of detail (LOD) and cost, without the need for managing large quantities of recordings.

Despite its potential, procedural audio has still not found widespread use in commercial productions. Stated reasons for this within the video game industry are usually related to aesthetics or the workflow of sound designers and programmers [3]: Many of the proposed procedural audio models do not produce audio that meets industry standards, and there is still a lack of reliable tools for procedural audio in larger commercial productions.

Proposed models for procedural audio often use physical modeling in an attempt to simulate the physical system that produces the sound of interest, without directly making use of recordings of the target sound [4]. The resulting models produce audio that often does not meet industry standards. An attractive alternative is to use data-driven approaches where recorded sounds are analyzed and used to build an audio model capable of reproducing similar sounds. Such data-driven methods could use recent advances in machine learning to obtain suitable models.

Neural audio synthesis is a recently emerged research field which uses deep learning to obtain sound models by training deep neural networks to generate audio similar to the target sound. State-of-the-art methods for speech synthesis and text-to-speech (TTS) use neural audio synthesis [5, 6, 7], and a recent paper introduces a software library for neural audio synthesis, called Differentiable Digital Signal Processing (DDSP) [8], which integrates digital signal processing (DSP) modules into deep neural networks. The authors illustrate the potential of DDSP by presenting results from synthesis of monophonic musical signals, while encouraging research into what other applications the framework can be used in.

This thesis investigates how procedural audio can be performed with neural audio synthesis, specifically by building upon the findings of the DDSP authors. We focus our attention on generating car engine sounds, while keeping in mind how the general framework may be used to synthesize other sounds. We present a method for recording engine sounds and for using these recordings to obtain models that can be controlled to synthesize sounds similar to the recordings.

We adapt the DDSP authors' model to engine sounds by, among other things, experimenting with different inputs, modifying the loss function and proposing a method for synthesizing transients that is compatible with DDSP. Our results illustrate that neural audio synthesis using DDSP can be used in procedural audio for engine sounds and potentially other essentially harmonic

sounds. We discuss what future work is needed to adapt our models for live real-time applications and to generate audio free of the undesired artifacts found in our generated audio.

Our source code is publicly available on GitHub[1]. To allow readers to listen to the generated audio examples we present, we have published online supplementary material[2].

1.1 Problem Formulation

The problem this thesis investigates is how to perform data-driven procedural audio. This is a very broad problem, and we try to answer it in part by investigating the more specific problem of how to perform procedural engine sounds using neural audio synthesis.

We would like to emphasize that the purpose of this thesis is not to make an attempt at proposing a method for state-of-the-art procedural engine sounds. Instead, we keep in mind how our methods can be used for data-driven procedural audio in general and try not to focus on methods too engineered to engine sounds to be used for other sound sources.

The research conducted in this thesis has been guided by the question

- *how can procedural audio be performed using data-driven methods?*

To partially answer the above question, we further refine it to:

- *can essentially harmonic sound sources other than monophonic musical instruments, such as car engines, be modeled using methods based on DDSP;*
- *how can the periodic nature of the engine cycles be taken into account, and to what extent can that improve the quality of generated audio;* and
- *how can the harmonic-plus-noise synthesizer in DDSP be extended to synthesize transients, and to what extent can that improve the quality of generated audio?*

1.2 Aim

While a variety of methods for performing procedural audio for different sound sources already exist, these often produce low-quality audio or are manually

[1] `https://github.com/antberg/thesis`
[2] `https://antberg.github.io/thesis`

engineered for a particular sound. An attractive alternative approach would be to use a more general data-driven method which takes examples of a target sound to build a model which is able to synthesize audio similar to the target.

This is in line with the methodology of neural audio synthesis. While neural audio synthesis has been used for high-fidelity speech synthesis and synthesis of musical signals, little research has been done into how these methods can be applied to perform procedural audio for other sounds.

The overarching aim of this thesis is to give insight into whether neural audio synthesis can be used to obtain improved models for procedural audio, and, more broadly, to give insight into how deep learning can be used in audio applications in general.

1.3 Objectives and Limitations

To answer the questions in section 1.1, we first propose a method for preparing training data by recording and processing engine sounds. We then suggest modifications to the method proposed by the DDSP authors and train corresponding models to reconstruct our training data. Lastly, we assess our methods by evaluating generated audio from our models.

From our original research question, we want to learn how procedural audio can be performed using data-driven methods. Whether or not the resulting models perform data-driven procedural audio is simply a matter of definition, but somebody asking this question is implicitly expecting that the generated audio is of reasonable quality. We partly evaluate the quality of the generated audio by computing objective metrics for the similarity between recorded and generated audio. Since all objective metrics for perceived audio quality have imperfections, we also listen to the generated audio and discuss its quality in terms of similarity to the recordings and overall audio quality (presence of artifacts, etc.).

Since our evaluation of the generated audio is limited to objective metrics and our own opinions from listening to the audio, our methodology for evaluating our methods has its limitations. Perceived audio quality is inherently subjective, and a better alternative would be to use formal listening tests where a group of experts assign opinion scores to the performance of the models. However, due to the time constraints of this thesis, formal listening tests could not be performed but is something that could be done in future work.

The sensitivity of neural networks to changes in hyperparameter values gives another limitation. Due to time constraints, we could not perform a large scale hyperparameter search and instead often rely on the default values used

by the DDSP authors. We present our results from using these settings, but it could be that changing the hyperparameter values would alter the results significantly. We encourage future work to investigate the result of using other settings than those we use.

Our findings are also limited to the conditions in which we apply our experiments. We want to generalize our results to car engine sounds in general, and even to other sounds, but we only record and use sounds from a single car engine. We argue why our results may generalize to other sound sources, but the validity of hypotheses based on such inductive inferences would have to be tested in further research.

1.4 Structure Outline

The remainder of this thesis is divided into five chapters: Background, Methodology, Results, Discussion and Conclusions & Future Work.

In the next chapter, Background, we introduce the topics of procedural audio, sound analysis/synthesis methods and neural audio synthesis. We conclude the chapter by focusing on DDSP, which our methods are based on.

In Methodology, we explain our methods for recording and processing engine sounds, as well as the components of our models and how they are trained. We also explain how we evaluate the generated audio.

In Results, we present our resulting training data and computed properties of our models. We also present and describe generated audio from our experiments.

In Discussion, we analyze our results and draw conclusions from them. We also discuss possible applications, the relevance of our work and aspects related to sustainability and ethics.

Finally, in Conclusions & Future Work, we summarize our most important conclusions and propose possible avenues for future research.

Chapter 2

Background

This chapter gives important background to our work. First, we introduce procedural audio along with some general methods and methods specific for engine sounds. We then introduce sound analysis/synthesis methods and explain spectral modeling synthesis (SMS) [9], which the authors of Differentiable Digital Signal Processing (DDSP) [8] base their sound synthesis method on. We also introduce the concept of transients and how Verma and Meng [10] proposed an analysis/synthesis method which extends SMS with an additional transient component. Finally, we introduce the field of neural audio synthesis and DDSP in particular.

2.1 Procedural Audio

The current methodology for audio content in virtual environments revolves mostly around capturing high-quality audio recordings and then using these recordings by playing them back when certain events are triggered in the virtual environment. While some variation of the sound is achieved through switching, sequencing and randomization of individual recordings, this approach is static and inflexible. One-shot sound effects may sound unnaturally repetitive, and the possibility of revising the material later on in the process becomes difficult and expensive. Continuous sounds that are controllable, such as engine noises, are particularly ill-suited for this methodology.

An alternative approach is to use procedural audio. In procedural audio, no audio is stored; instead, the audio is generated in real time by a model which carries a representation of its target sound. Farnell [1] defines procedural audio

as follows.

Definition 1. *Procedural audio is non-linear, often synthetic sound, created in real time according to a set of programmatic rules and live input.*

This allows the sound of procedural alternatives to be determined by the virtual objects themselves, more tightly bound to the physics of the state and relying more on real-time parameters than discrete events. Audio is generated at the point of use instead of being selected from prerecorded sequences. Using procedural audio compared to prerecorded audio is analogous to the rendering and lighting of a 3D scene compared to using static textures. The prior is more flexible, but also more CPU intensive.

Despite its many advantages, procedural audio has yet to see any widespread use in commercial productions. A survey by Böttcher [3] states two main reasons for this: The sound quality of proposed procedural models usually do not meet industry standards, and there is a lack of reliable tools for procedural audio in larger commercial productions.

2.1.1 Methods for Procedural Audio

Procedural audio for a given category of sound can be achieved by different methods. These can be divided into model-based and model-free methods. One category of model-based methods uses information about the physical properties of the system that produces the sound. Some of these physics-based models run a detailed physical simulation of the system to synthesize the target sound. Examples are impact sounds from rigid and deformable bodies, the sound of objects moving rapidly through air and liquid sounds [11]. The sounds of these methods are tightly bound to the graphics of a scene, since both audio and graphics emerge from the same physical simulation. These procedural audio methods are hence flexible and coherent with the graphics; however, they are also computationally expensive and often do not meet the audio quality standards of e.g. a commercial video game production.

Other model-based methods do not model the physical system itself, but focus instead on modeling signal components of the the target sound by combining oscillators and filters. Examples of such physically informed signal-based methods can be found in Andy Farnell's book [12], with examples ranging from nature sounds such as wind, fire and running water, to the sound of machines such as clocks, helicopters and cars. These methods often require a lot of manual design and fine tuning of parameters to sound like their targets, but the resulting models are often computationally cheap.

Among the model-free methods are granular and concatenative synthesis. Instead of using a parametric model, like a physical model or a signal model, these methods rely on audio examples of a target sound and split these examples into small so-called grains. In granular synthesis, different sounds and timbres can be achieved by varying the waveform, envelope, duration and density of the grains. In concatenative synthesis, descriptors are extracted from each grain to build a corpus. During synthesis, the method takes an input signal and weaves together individual grains in a way that minimizes the distance in descriptor space between the path of the grains and that of the input signal. Concatenative synthesis has been used for synthesis of musical instruments [13] and used to be state-of-the-art in speech synthesis [14] before more recent approaches based on deep learning were developed [5, 6].

2.1.2 Procedural Engine Sounds

This thesis focuses on one sound source: car engines. Specifically, we look at four-stroke engines, the most common type of internal combustion engines in production cars, trucks and motor bikes. An internal combustion engine generates kinetic energy from the pressure that builds up by combusting an air-fuel mixture in the engine's cylinders. For a four-stroke engine, this process contains a sequence of repeating cycles with four steps: intake, compression, combustion and exhaust. The sound produced by one cycle is the result of moving machinery and exploding fuel, which is a noisy sound rich in transients. Since each cycle is relatively short, we will perceive the periodic sound as tonal with a definite pitch. Furthermore, engine sounds contain a wide range of frequencies covering most of the human hearing range. Hence, engine sounds are noisy with both tonal and transient components while covering a wide range of frequencies, making them complex and challenging to model. To illustrate this, we present a spectrogram of an engine recording in Figure 2.1. For the first 1.5 s of the recording, the engine is idling; for the remainder, the engine is quickly accelerating and deaccelerating. Notice how the bottom half of the frequency range contains horizontal lines corresponding to the harmonics, while the top half contains cyclic vertical patterns corresponding to the explosions in the cylinders.

We perceive engine sounds as having pitch, therefore, the sound is essentially harmonic [15]. This means that there is a fundamental frequency f_0 which together with its harmonics describe the tonal character of the sound. This fundamental frequency is given by the rate of the explosions that occur in the cylinders of the engine. If we know the speed of the engine f_{RPM} in

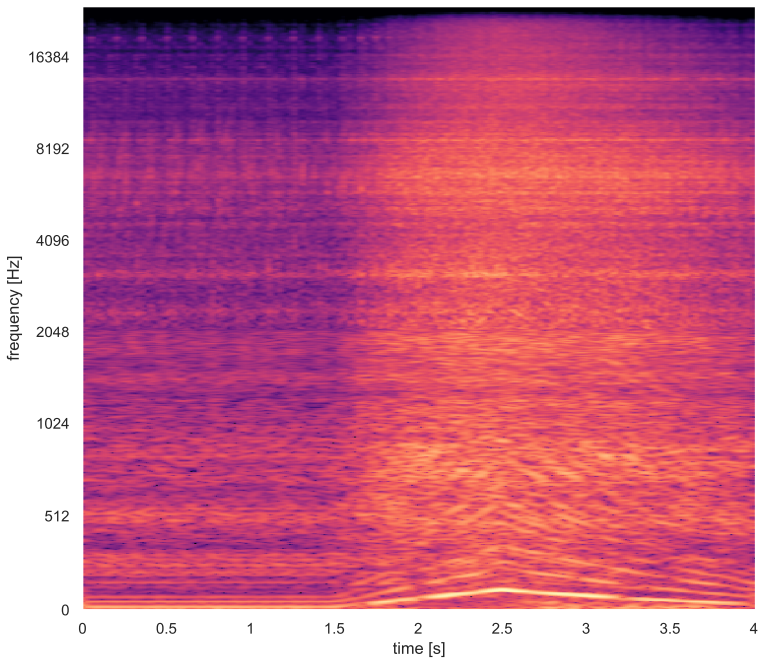

Figure 2.1: Mel spectrogram of an engine recording.

revolutions per minute (RPM), then f_0 is given in Hz by

$$f_0 = \frac{N_c}{2} \frac{f_{\mathrm{RPM}}}{60\,\mathrm{s}} \tag{2.1}$$

where N_c is the number of cylinders in the engine. We divide by 60 s to convert from RPM to Hz and divide by 2 because explosions only occur in a cylinder once every other revolution. Although engine sounds are essentially harmonic, the harmonics may not only be integer multiples of f_0, as they are for ideal harmonic sounds. Since the explosion sounds of the N_c cylinders are slightly different, there are harmonics that are dictated by the frequency f_c of complete engine cycles (which corresponds to two revolutions). The frequency of the complete engine cycles is given by $f_c = f_0/N_c$, and the harmonics will be frequencies that are integer multiples of f_c. With a model where all harmonics are integer multiples of f_c, the pth harmonic is given in terms of f_0 by

$$f_p = \frac{p f_0}{N_c} \tag{2.2}$$

where p is a positive integer. In this case, the strongest harmonic, and what we perceive as the pitch, would be the N_cth partial since $f_{N_c} = f_0$.

Proposed methods in the literature for procedural engine sounds include both physically informed methods and granular approaches. Farnell [12] proposes physically informed models for different engine types based on a DC motor model. Jagla, Maillard, and Martin [15] use a granular method where each grain is an engine firing cycle. The audio is synthesized by putting together grains using an overlap-and-add method which uses a harmonic-plus-noise signal model to minimize phase discontinuities between grains. Although there are published methods for procedural engine sounds, the state-of-the-art methods that have the highest sound quality are likely those owned by video game companies, which are unfortunately not publicly available. Procedural engine sounds in games, such as Need For Speed by Electronic Arts, are often based on granular approaches which involve lots of manual fine tuning [3].

2.2 Sound Analysis/Synthesis

Methods for sound analysis and synthesis have been active research topics for decades. Common applications include processing and compression of audio signals for speech and music. The idea of so-called analysis/synthesis methods is to first, during an analysis stage, break down an audio signal into meaningful components and then, during a synthesis stage, reconstruct the audio signal from these components [16]. In compression algorithms for general purpose audio codecs, the idea is to find a memory efficient representation of the audio during the analysis and then during synthesis be able to construct a signal that is perceptually as similar as possible to the original audio signal. In some audio processing methods, the analysis stage extracts parameters with physical meaning in order to be able to transform these before the synthesis stage. By transforming the physical parameters, these analysis-synthesis methods are able to perform processing such as time-stretching, pitch-shifting and spectral morphing.

2.2.1 Spectral Modeling Synthesis

One analysis/synthesis method that has seen widespread use is spectral modeling synthesis (SMS) proposed by Serra and Smith [9] in 1990. Although originally designed specifically for musical signals, the method has been adapted for processing and synthesis of a wide variety of different sounds. The idea of SMS is to decompose the original audio into a sum with one deterministic

term and one stochastic,

$$x[n] = d[n] + s[n], \tag{2.3}$$

where n is an integer denoting the time step and $x[n]$ denotes the nth time step of a discrete signal x. The deterministic signal d is modeled as a sum of a series of sinusoids,

$$d[n] = \sum_{p=1}^{P} a_p[n] \sin(\phi_p[n]). \tag{2.4}$$

The instantaneous phases ϕ_p of the pth partial are given by numerically integrating the instantaneous frequencies f_p,

$$\phi_p[n] = \frac{2\pi}{f_s} \sum_{j=1}^{n} f_p[j] + \phi_p[0], \tag{2.5}$$

where $\phi_p[0]$ is the initial phase and f_s is the sample rate. Since the signal s is assumed to be stochastic it can be represented as filtered white noise,

$$s[n] = (h * w)[n] \tag{2.6}$$

where $*$ denotes discrete convolution, w is white noise and h is the the impulse response of a time-varying frequency-shaping filter. Intuitively, this sinusoid-plus-noise representation is capable of modelling sounds that are a combination of slowly varying tonal sounds and breath-like noises. Since this covers a large set of sounds, SMS has been used for applications ranging from synthesis of speech, musical instruments and singing voices to general purpose audio codecs [17].

During the analysis stage, SMS estimates the trajectories of the amplitudes a_p and instantaneous frequencies f_p of each partial. a_p and f_p are then used to generate the waveforms of the deterministic component d using (2.4) and (2.5) in a process called additive synthesis. To estimate the impulse response h, SMS uses the envelopes formed by the difference between the short-time Fourier transform (STFT) magnitudes of the original audio and that produced by the additive synthesis. The stochastic component s is then synthesized by estimating (2.6) using an overlap-add technique from the spectral envelopes that represent h in a process called subtractive synthesis.

During the synthesis stage, only a_p and f_p are used to generate waveforms and information about instantaneous phases of the original sound is thus disregarded. Despite not preserving the phases of the original sound, SMS works since human sound perception is not sensitive to relative phase: When two or

more sinusoids are added together, their relative phases have no impact[1] on the perceived sound despite having a significant impact on the shape of the waveform.

For single, monophonic sound sources, a common restriction is to assume that the deterministic term is harmonic. This means that all frequencies at a given time are an integer multiple of a fundamental frequency f_0,

$$f_p[n] = pf_0[n]. \tag{2.7}$$

The fundamental frequency f_0 is what we perceive as pitch. With the harmonic assumption, the instantaneous phases ϕ_p become

$$\phi_p[n] = \frac{2\pi p}{f_s} \sum_{j=1}^{n} f_0[j] + \phi_p[0]. \tag{2.8}$$

This version of the SMS signal model, whose deterministic component is harmonic, is often referred to as the harmonic-plus-noise model.

2.2.2 Transients

Although the signal model of SMS is able to represent a wide variety of sounds, it is not without limitations. The deterministic sinusoidal component works well for capturing slowly varying tonal sounds, whereas everything else is modeled as noise. A certain category of sounds called transients are high-amplitude, short-duration sounds that contain a high-degree of non-periodic components. Transients are present in the attack of musical and percussive sounds such as in the pluck of a guitar string or the hit of a snare, and in other non-musical sounds such as applause and rain. When modeled as noise, transients lose sharpness in their attack and give rise to so-called pre-echo artifacts.

To solve these problems, Verma and Meng [10] proposed in 1998 a method based on SMS which complements the sinusoidal and noise components of the signal model with a transient component. Their key observation is that there is an isomorphic duality between sinusoids and transients. When performing time-frequency analysis on transients modeled as impulses, it turns out that transients are sinusoids in the frequency domain. Conversely, slowly varying sinusoids are impulsive in the frequency domain. This means that sinusoidal modeling can be used for both slowly varying sinusoids and short-duration

[1] While this holds true for most situations, there are rare circumstances—for instance, if the crest factor of the summed waveforms is extreme—where the perceived sound may differ depending on relative phase [18].

transients. Verma and Meng use the discrete cosine transform (DCT) to perform the time-frequency analysis. An impulse in the beginning of a signal frame will occur as a low-frequency sinusoid in the DCT domain. Likewise, if there is an impulse near the end of a frame it will transform to a sinusoid with frequency close to the Nyquist frequency $f_N = f_s/2$.

With this DCT-based representation of transients, the deterministic component d in (2.3) becomes

$$d[n] = d_1[n] + \mathrm{IDCT}(d_2)[n] \tag{2.9}$$

where both d_1 and d_2 are sums of sinusoids as in (2.4), and IDCT is the inverse DCT.

2.3 Neural Audio Synthesis

Neural audio synthesis is the process of synthesizing sound using artificial neural networks. With computational power getting cheaper and artificial intelligence and deep learning becoming increasingly popular research fields, neural audio synthesis has seen a lot of progress only the last couple of years.

2.3.1 Autoregressive and Flow-Based Models

In 2016, Oord et al. [5] proposed WaveNet, an autoregressive model for raw audio synthesis. One of their published experiments show state-of-the-art performance for text-to-speech synthesis, and the other experiments show WaveNet's potential for multi-identity speech synthesis and sound synthesis of musical instruments. However, due to the complex task of synthesizing raw audio waves, WaveNet models require lots of parameters and are cumbersome to train. Furthermore, since autoregressive models generate individual samples serially, synthesis is inherently slow. WaveNet implementations can only generate a couple of hundred audio samples per second even on high-performance GPUs [19], which make them impossible to use for real-time applications.

Other methods for neural audio synthesis inspired by the architectural insights of WaveNet have been suggested. To deal with the slow synthesis of autoregressive models, the authors of WaveNet proposed in 2017 Parallel WaveNet [7]. Parallel WaveNet uses normalizing flows to parallelize the sampling, allowing for significantly faster audio synthesis seemingly without compromising audio quality. Shortly after Parallel WaveNet, more models using normalizing flows to speed up audio synthesis have been proposed, such

as ClariNet [20], WaveGlow [21] and FloWaveNet [19]. Although sampling becomes faster for these flow-based models, this comes at the cost of model complexity and complicated training schemes. Parallel WaveNet and ClariNet require a teacher WaveNet to be trained which is later used to train a student flow network. This training process is sensitive, and both Parallel WaveNet and ClariNet introduce custom-engineered loss terms to avoid specific problems during training. This seems to be in part solved by WaveGlow and FloWaveNet, which use an architecture that can be trained in an end-to-end fashion, without requiring the two-stage training process with a teacher and student network.

2.3.2 Differentiable Digital Signal Processing

Up until now, most methods for neural audio synthesis produce raw audio directly in the time domain, including the autoregressive and flow-based models already mentioned. Some other models, including Tacotron [6], instead generate Fourier coefficients in the frequency domain. Due to the Fourier inversion theorem, there is a one-to-one correspondence between these Fourier coefficients and the time-domain representation, so this representation is equally expressive. However, an important issue is that these representations are, in a sense, overly expressive—they distinguish between sounds that sound the same to humans. Human sound perception is not sensitive to the absolute phase of individual frequency components. Models not taking this into account must learn how to synthesize audio that is phase-coherent with the training material, even though there are an infinite number of perceptually equivalent audio signals. Hence, these models need a lot of parameters and training data to perform well.

Earlier this year (2020), Engel et al. [8] proposed Differentiable Digital Signal Processing (DDSP) which integrates digital signal processing (DSP) techniques with TensorFlow [22], a popular software for automatic differentiation commonly used in deep learning applications. DDSP includes differentiable modules for oscillators and filters that can generate and process audio with methods such as additive synthesis, wavetable synthesis, subtractive synthesis and reverberation.

In their published experiments, they present a relatively simple framework for training an autoencoder network to generate the synthesis parameters of a harmonic-plus-noise model. Owing to the interpretability of each DSP component, the authors show that models using DDSP can perform realistic extrapolation to pitches not seen during training, blind dereverberation and timbre transfer. Perhaps most interestingly, the authors show that neural audio

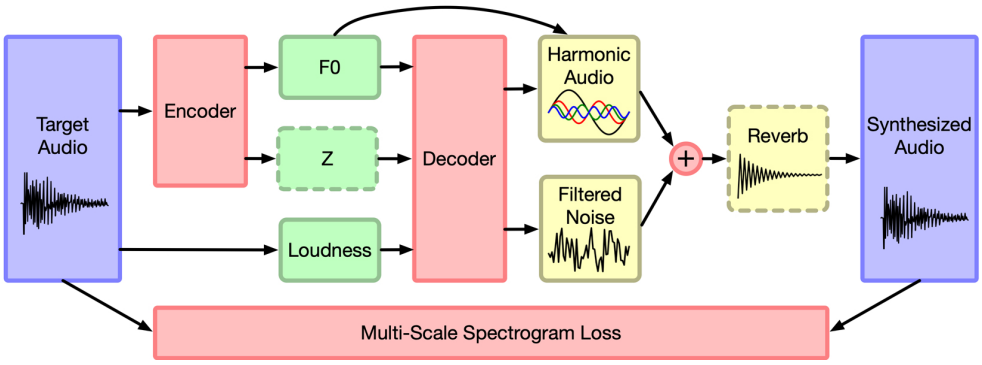

Figure 2.2: The DDSP Autoencoder from [8].

synthesis can be performed using relatively small models and simple training schemes by using interpretable synthesizers and a perceptually motivated loss function. In all of their experiments, they use the harmonic-plus-noise model based on SMS [9] and a loss function based on STFT magnitude spectra.

The authors perform experiments where they train a model on the NSynth dataset [23] to generate single notes of different instruments. They also train their model on recordings of solo violin performences and show that the network can be controlled to synthesize violin sounds given sequences of fundamental frequency f_0 and loudness ℓ. The authors illustrate how the network can be used for timbre transfer by extracting f_0 and ℓ from recordings of a singer and using these features on the model trained on solo violin to synthesize a violin playing the sung melody.

2.3.3 The DDSP Autoencoder

Throughout their experiments, the DDSP authors use variants of an autoencoder network which they call the DDSP Autoencoder. A schematic illustration of the network is presented in Figure 2.2. The input to the full network is a four-second frame of the original audio sampled at 16 kHz, and the output is the network's attempt at reconstructing the original audio frame. The idea of the architecture is to encode the original audio into latent interpretable features and decode these features into parameters used in DSP modules during synthesis of the reconstructed audio. The latent features in the bottleneck of the model are estimated fundamental frequency f_0, loudness ℓ and a feature vector z related to the timbre of the original sound.

Encoder and decoder

The encoder part of the model consists of three parts: one fundamental frequency f_0 tracker, one loudness ℓ tracker and an encoder network that extracts the timbral feature vector z. The z-encoder computes the mel-frequency cepstral coefficients (MFCCs) of the input audio and computes z from these by passing them through a gated recurrent unit (GRU). The loudness tracker computes a perceived loudness ℓ represented by a log-scaled A-weighted power spectrum, and is hence a fixed, non-trainable operation. The authors experiment with both supervised and unsupervised learning of the f_0 feature. In the supervised approach, the model uses an existing algorithm called CREPE [24] for fundamental frequency estimation. In the unsupervised approach, an encoder network based on deep residual networks [25] is trained to extract f_0 while training the full network end-to-end.

While the sample rate of the input audio is 16 kHz, the decoder network operate on frames with a slower rate of 250 Hz. The decoder network consists of multi-layer perceptrons (MLPs) and a GRU, and is trained to decode the latent features f_0, ℓ and z into synthesis parameters.

Audio synthesis

The audio synthesis method in the DDSP Autoencoder is a harmonic-plus-noise method. The model uses 60 partials in the sinusoid model of the harmonic additive synthesizer and represents the transfer function of the noise model's impulse response by a finite impulse response (FIR) filter of order 65. The decoder network outputs the amplitudes a_p for each partial and the coefficients H of the FIR filter at each frame. To increase interpretability, a_p is factored into

$$a_p[m] = a[m]c_p[m] \tag{2.10}$$

where a controls the global loudness and c_p is a normalized amplitude distribution over the partials such that $\sum_p c_p[m] = 1$ and $c_p[m] \geq 0$ for all frames m. Before synthesizing audio, a_p and f_0 are upsampled from the frame rate 250 Hz to the audio sample rate 16 kHz. For f_0, the model uses bilinear interpolation between frames whereas for a_p an overlap-add technique is used. During synthesis, the harmonic part uses the upsampled a_p and f_0 to perform additive synthesis as in (2.4) and (2.5). The noise component multiplies a windowed version of $H[m]$ and a the discrete Fourier transform (DFT) of generated white noise $W[m]$ for each frame m. The resulting frames are combined using overlap-add and the synthesized noise signal is transformed to the time domain by performing an inverse DFT (IDFT).

Loss function and training

Instead of relying on complex autoregressive or adversarial losses, the DDSP Autoencoder is trained on a loss based on the difference between the STFT magnitudes of the original and reconstructed audio. The harmonic-plus-noise synthesizer does not preserve phases of the original sound; however, it does preserve magnitudes and frequencies [9]. Therefore, the STFT magnitudes of the original and reconstructed audio are comparable. The DDSP Autoencoder uses a multi-scale STFT loss defined as

$$\mathcal{L}_{\mathrm{STFT}} = \sum_i \|S_i - \tilde{S}_i\|_1 + \|\log S_i - \log \tilde{S}_i\|_1 \tag{2.11}$$

where i enumerates each FFT scale and $\|\cdot\|_1$ denotes the L_1 norm[2]. For a given FFT size, S_i and $\tilde{S}_i$ denote the STFT magnitudes of the original and reconstructed audio, respectively. The DDSP authors use FFT sizes 64, 128, 256, 512 and 1024, and the neighboring frames in the STFT overlap by 75 %. By using STFT magnitude spectrums with different FFT sizes, the loss terms cover differences between the target and reconstructed audio at different time-frequency scales.

Since all DDSP components are differentiable, the full DDSP Autoencoder is differentiable end-to-end. Hence, any popular variant of the stochastic gradient descent (SGD) algorithms can be applied during training. The DDSP authors use an Adam [26] optimizer with learning rate 0.001 and an exponential learning rate decay of 0.98 every 10 000 steps.

[2]This is a slight abuse of notation, because the DDSP authors do not define the L_1 norm conventionally as the sum of the absolute values of each element, $\|\mathbf{x}\|_1 = \sum_i^n |x_i|$. Instead, they use the mean absolute error (MAE) by defining $\|\mathbf{x}\|_1 = \frac{1}{n}\sum_i^n |x_i|$. In an optimization setting, this does not matter since any loss function will have the same optima after scaling by any positive number.

Chapter 3

Methodology

In this thesis, we consider a data-driven method for procedural engine sounds using neural audio synthesis based on Differentiable Digital Signal Processing (DDSP) [8]. We construct a sound model by training a neural network to reconstruct recorded examples of a target sound.

The first step in our method is to construct a dataset with examples of a target sound together with corresponding network inputs. We focus specifically on car engine sounds, and section 3.1 details our method for recording and processing the data we use during training and reconstruction experiments. The models we consider are based on the DDSP Autoencoder illustrated in Figure 2.2. We experiment with different inputs, different components of the network and different sound synthesis methods, and the details of each model we use in our experiments are given in section 3.2. In section 3.3, we outline the training procedure and explain the loss functions the models use during training. Finally, in section 3.4 we present metrics we use to quantify the quality of reconstructed audio.

3.1 Sounds and Data

The initial step of our approach to finding a model is to record audio of a target sound. We associate each audio recording with corresponding model inputs and split all the audio-input pairs into smaller examples that are used during training. In general, the specific process will differ depending on the target sound, but they will all use a microphone to capture the audio and some way of either recording the model inputs simultaneously or to later extract them from the recorded audio. This thesis focuses on car engine sounds, and this section covers how we record engine sounds along with on-board diagnostics

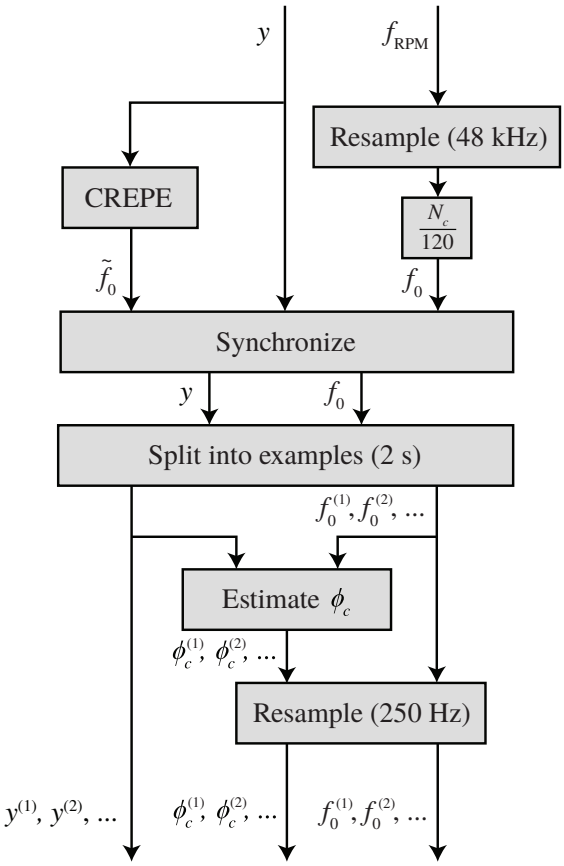

Figure 3.1: Overview of our process of going from recorded audio y and OBD recording f_{RPM} to two-second examples used during training and reconstruction. The audio-f_0 synchronization is illustrated in Figure 3.2 and the ϕ_c estimation is illustrated in Figures 3.3 and 3.4.

(OBD) data, and how we process these to estimate the fundamental frequency f_0 of the sound and the phase ϕ_c of the engine cycles. See Figure 3.1 for an illustration of our full process of going from audio and OBD recordings to training examples.

3.1.1 Recording Process

To capture the sound of a car engine, we mount a microphone inside the engine compartment and record audio while varying the gas pedal position. The audio is recorded in 48 kHz WAV format with 16-bit PCM. We restrict ourselves by letting the transmission be in neutral gear. This lets us perform the recordings with a stationary car and also simplifies the dynamics of the engine by removing

complex behaviors related to driving in different gears with different loads.

In theory, the engine speed in RPM corresponding to an engine sound can be extracted from the recorded audio. Methods have been suggested within the field of vibrational analysis, but automatic and unsupervised RPM reconstruction from vibration signals is still an open problem [27]. Since the RPM is proportional to the fundamental frequency f_0 of the produced sound, a possibility is to use methods developed for f_0 estimation. One such method is CREPE [24], a state-of-the-art f_0 estimation method for musical signals. However, in our experience, these methods are noisy and susceptible to octave errors, especially for engine sounds which have relatively low f_0 values—idle car engines have f_0 around 20 Hz which is barely audible to humans.

We avoid these unreliable methods by directly recording the RPM using an on-board diagnostics (OBD) tool. Since 2003, all passenger vehicles of type M1 fueled by petrol or diesel registered within the European Union are enforced by law [28] to have a plug on the drivers side with a diagnostics link connector using the signal protocol described by the SAE J 1962 standard [29]. This makes using a single method for recording the RPM via an OBD connection applicable to most production cars. Furthermore, there are many quantities other than RPM that can be recorded using the same protocol—such as vehicle speed, engine load, and gas pedal position—which may be used as inputs to a more complex model. We use a USB interface with an ELM327 OBD-II adapter to connect to the car's OBD system. To perform the communication, we use the python-OBD [30] library.

3.1.2 Preprocessing

RPM preprocessing

The OBD signal protocol provides no way to guarantee that the time between each sample is constant. Therefore, we store a timestamp with each RPM sample. We perform linear interpolation between the recorded RPM data points to resample the signal so it has same sample rate as the recorded audio, 48 kHz. Each point of the resampled RPM is then scaled according to (2.1) to convert the RPM to the fundamental frequency f_0. Note that we from here on in this section use f_0 specifically to denote the fundamental frequency calculated from the RPM recording.

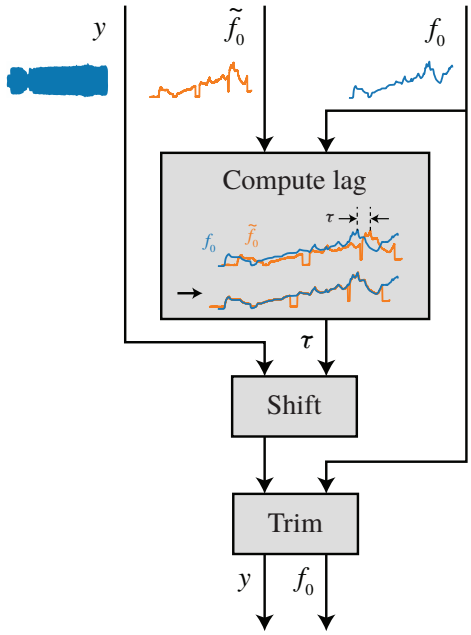

Figure 3.2: Our process for synchronizing audio y and f_0 signals, using a rough estimate $\tilde{f}_0$ computed with CREPE [24].

Audio-f_0 synchronization

We record the audio and the RPM simultaneously but not in a synchronized way. To synchronize them, we first perform a crude fundamental frequency estimation on the recorded audio using CREPE [24]. Let $\tilde{f}_0$ denote the fundamental frequency estimated using CREPE. We then calculate the time lag between $\tilde{f}_0$ and f_0 via cross-correlation. The obtained time lag is then used to shift the audio to align it with f_0. Finally, f_0 and the audio are cropped to have equal length. The process is illustrated in Figure 3.2. For exact details of how we implement this process, we refer to our publicly available source code[1].

Splitting into training examples

With synchronized audio-f_0 pairs of equal sample rate and length, we split these into two-second examples. We use these audio-f_0 examples to estimate the phase ϕ_c of the engine cycles, as explained in the next section. Before these audio-f_0-ϕ_c examples are stored into a dataset, we downsample f_0 and ϕ_c to 250 Hz, the frame rate of the neural networks. Finally, to be able to read

[1]The source code for our audio-f_0 synchronization process is found in the script `preprocess_obd.py` located in the `data` directory.

data efficiently during training, we seralize each example and store them into TensorFlow's TFRecord format.

Engine cycle phase estimation

The engine sound is composed of short cycles whose periods are determined by the RPM, which is proportional to the fundamental frequency f_0. If we would only use f_0, we would not use any information about the phase ϕ_c of the engine cycles. Since ϕ_c determines when an explosion takes place—or any other event in the firing cycle for that matter—ϕ_c is likely a good predictor for when certain transients in the sound will appear. We propose a method for estimating ϕ_c using the recorded audio and the f_0 values corresponding to the RPM.

In mechanical terms, ϕ_c corresponds to the camshaft position. If there were a standardized way of measuring the signal from the camshaft position sensor of the engine, such as through the car's OBD system, that would be a direct way of measuring ϕ_c. However, there is no such standardized way to our knowledge.

Our method of estimating ϕ_c is inspired by the following observations. Consider the phase ϕ describing the explosion cycles[2]. During one explosion cycle, one explosion occurs. If we knew the initial phase ϕ_0 of the explosion cycles at the start of the audio recording, and if we had a perfect f_0 estimate, then the unwrapped explosion phase ϕ would be given, as in (2.5), by

$$\phi[n] = \frac{2\pi}{f_s} \sum_{j=0}^{n} f_0[j] + \phi_0 \tag{3.1}$$

where f_s is the sample rate. There are two main reasons we cannot use this expression explicitly. First, we do not know ϕ_0. Second, our f_0 estimate is not perfect and the error will accumulate in the sum. Consider instead the sinusoid

$$u[n] = \sin(\phi[n]). \tag{3.2}$$

This is the sinusoid generated by f_0 and ϕ_0. If we could filter out all frequencies from the engine sound except f_0 we would be left with $u[n]$. Hence, there is information about ϕ embedded in the audio recording. We use this information by estimating u, and from u we estimate ϕ and in turn ϕ_c. The process of estimating ϕ from the corresponding audio-f_0 training example is illustrated in Figure 3.3. The process of estimating ϕ_c from ϕ is illustrated in Figure 3.4.

[2]Here, ϕ should not be confused with ϕ_c. Remember that ϕ_c describes the engine cycles, and that during one such cycle N_c explosions occur, where N_c is the number of cylinders.

1. Compute low-pass filtered version y_{LPF} of audio y with cut-off frequency $f_c = 1.5 \max f_0$.

2. Split y_{LPF} and f_0 into 5 ms frames.

3. For each frame i, generate a sinusoid $u^{(i)}$ from $f_0^{(i)}$.

4. Synchronize $y_{\mathrm{LPF}}^{(i)}$ and $u^{(i)}$ using cross-correlation.

5. Enforce continuity between $u^{(i)}$ and $u^{(i+1)}$, and $(-1, 1)$ range by generating a sinusoid between the last extremum of $u^{(i)}$ and the second of $u^{(i+1)}$. Stack $u = \left(u^{(1)}, u^{(2)}, \ldots\right)$.

6. Compute the phase ϕ of u by forming the analytic signal ζ using the Hilbert transform, as in (3.3).

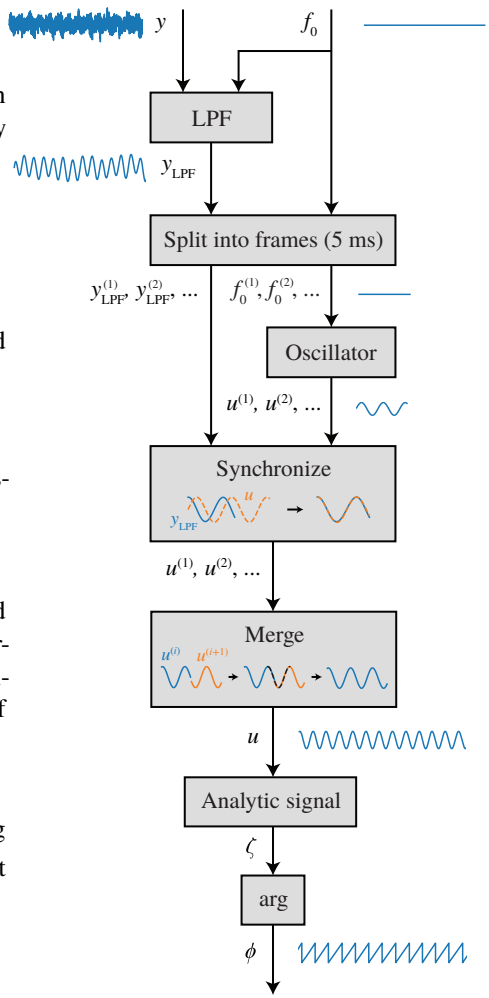

Figure 3.3: Our process for estimating the phase ϕ of the engine explosion cycles from recorded audio y and fundamental frequency f_0.

For exact details on how we implement this process we refer to our publicly available source code[3].

First, we low-pass filter the audio and divide it into 5 ms windows. For each window, we generate a sinusoid based on our f_0 estimate from the RPM. We align the generated sinusoid with the low-pass filtered audio using cross-correlation. When adding together windows to form the full signal, we use a custom cross-fade method to ensure continuity between windows and to

[3]The source code of our ϕ_c estimation process is found in the function `get_synchronized_osc` within the script `prepare_tfrecord.py` located in the `data` directory.

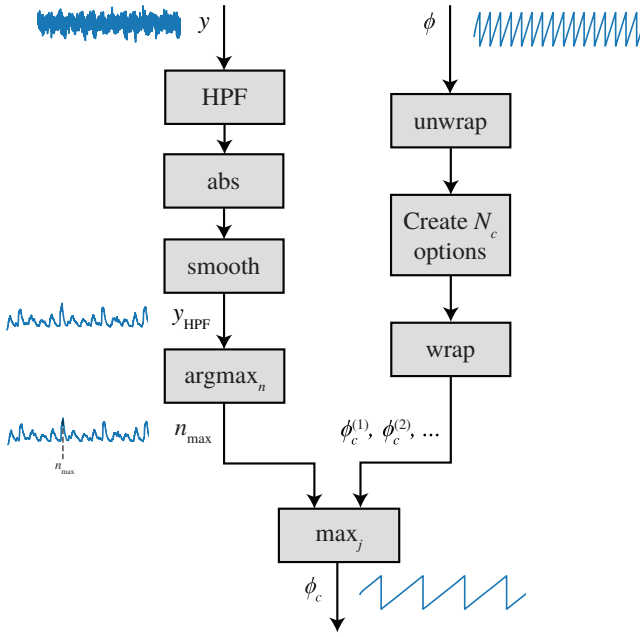

Figure 3.4: Our process for estimating the phase ϕ_c of the engine cycles from recorded audio y and estimated phase ϕ of the engine explosion cycles. The cut-off frequency in the high-pass filter (HPF) is 15 kHz and the smoothing factor is 0.995. The N_c options $\phi_c^{(1)}, \phi_c^{(2)}, \ldots$ are defined by (3.5) and the $\max_j$ operator corresponds to eqs. (3.6) and (3.7).

make sure the resulting sinusoid always oscillates between -1 and 1. With the windows added together, we have an estimated version of u in (3.2) that is phase-aligned with the explosion cycles. Hence, we can estimate ϕ by estimating the phase of the sinusoid u. We do this by forming the analytic signal

$$\zeta[n] = u[n] + i\mathcal{H}(u)[n] \tag{3.3}$$

where i is the imaginary unit and $\mathcal{H}(u)$ denotes the discrete Hilbert transform of u. The wrapped phase ϕ is now given as[4]

$$\phi[n] = \arg(\zeta[n]). \tag{3.4}$$

If we unwrapped this phase, we would get a good estimate of $\phi[n]$ in (3.1).

Recall that during one engine cycle, whose phase is given by ϕ_c, N_c explosions occur. However, it is not as easy to get ϕ_c from ϕ as to simply calculate

[4]Technically, this gives the phase of a cosine and not a sine as in (3.2); i.e., the phase is shifted by $\pi/2$. However, this is irrelevant for our purposes as long as we consistently associate one point in time of an engine cycle to a unique phase value.

$\phi_c[n] = \phi[n]/N_c$. If we enumerate each cylinder $1, 2, ..., N_c$, we want cylinder 1 to always be the first cylinder to fire in an engine cycle, cylinder 2 to be the second, and so on. We make sure this is the case by using our observation that one of the N_c explosions consistently results in the loudest peak in the audio.

We create N_c options defined by

$$\phi_c^{(j)}[n] = \frac{\phi[n]}{N_c} + \frac{2\pi j}{N_c}, \qquad j = 0, 1, ..., N_c - 1 \tag{3.5}$$

and choose ϕ_c such that

$$\phi_c[n] = \phi_c^{(j_{\max})}[n] \tag{3.6}$$

where

$$j_{\max} = \arg\max_j \left(\phi_c^{(j)}[n_{\max}]\right) \tag{3.7}$$

at time step $n_{\max}$ where a smoothed, high-pass filtered version of the audio recording has its maximum[5].

3.2 Models

This thesis focuses on neural networks based on the DDSP Autoencoder [8] illustrated in Figure 2.2. Although the DDSP Autoencoder was developed for synthesis of monophonic musical signals, its harmonic-plus-noise model of sound is a useful representation for any sound that has an essentially harmonic component. Here, we experiment with using it for the synthesis of engine sounds.

Our goal is to be able to control the synthesis using a low-level control parameter corresponding to the RPM of the engine. Since a fundamental frequency f_0 is present in all harmonic sounds, and since it is proportional to the RPM of an engine, we use f_0 as control input. See Figure 3.5 for a schematic view of our process of controlling the synthesis of engine sounds.

All of our models are based on the version of the DDSP Autoencoder that the authors use for synthesis of solo violin sounds. Hence, we do not use the latent timbre vector z, neither do we use the reverb component in the synthesizer. Instead of the relatively low 16 kHz sample rate, all our models use 48 kHz sample rate in order to preserve high-frequency content necessary for high-fidelity audio. Using 48 kHz means that we consider frequency information in the 8–24 kHz range which the DDSP authors do not.

[5]Choosing the maximum $\phi_c^j[n_{\max}]$ is equivalent to choosing j which has $\phi_c^j[n_{\max}]$ closest to π. The choice π is arbitrary; as long as we consistently choose the same out of the N_c options.

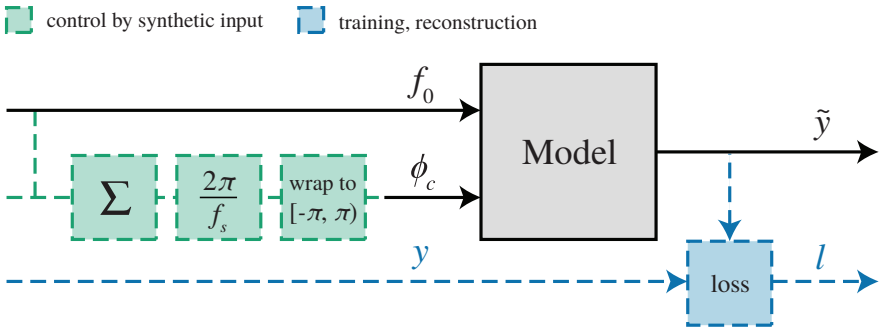

Figure 3.5: Diagram of components involved during training, reconstruction and control by synthetic input. During training and reconstruction, fundamental frequency f_0, engine cycle phase ϕ_c and audio signal y are given as an example from the dataset. During control by synthetic input, a synthetic f_0 is given and ϕ_c is generated through numerical integration of f_0 and wrapping it to $[-\pi, \pi)$. $\tilde{y}$ denotes the synthesized audio.

The following sections specify the models we use in our experiments. For a schematic illustration of all models and their components, see Figure 3.6. To make the difference between each model clear, we present their differences in Table 3.1.

3.2.1 Original DDSP Settings (vanilla-DDSP)

Since engine sounds are essentially harmonic, we first experiment with a model that is more or less the same as the decoder network and synthesizer of the DDSP Autoencoder. Call this model vanilla-DDSP. This model acts as a decoder that takes a latent feature as input, in this case a sequence of fundamental frequencies f_0, and outputs synthesized audio samples.

As shown in Figure 3.6, the first block of vanilla-DDSP is a preprocessing block. This block converts f_0 to a normalized MIDI scale by applying

$$\hat{f}_0^{\text{MIDI}} = \frac{1}{127}\left(12\log_2\left(\frac{f_0}{440}\right) + 69\right). \tag{3.8}$$

The next block, the decoder network, takes $\hat{f}_0^{\text{MIDI}}$ as input and generates parameters for the synthesizer block. The synthesizer block consists of a harmonic-plus-noise synthesizer which generates the samples of the output audio.

A detailed description of the decoder network of vanilla-DDSP, and all other models, is provided in section A.1. Just like the original DDSP Autoencoder, vanilla-DDSP uses a 512-unit GRU in the decoder network, 60 partials in the

Table 3.1: Configurations that differ between the models. The presented f_0 scale states how the f_0 input is scaled before being passed through the decoder network. The scales n-MIDI and n-mel refer to the normalized MIDI and mel scales defined in (3.8) and (3.9), respectively. GRU dim is the dimensionality of the hidden state in the decoder network's GRU. The H input refers to the value each harmonic is an integer multiple of in the harmonic synthesizer. Under synthesizer, H means harmonic, N means filtered noise, and T means transients.

	inputs	f_0 **scale**	**GRU dim**	**H input**	**synthesizer**	**loss function**
vanilla-DDSP	f_0	n-MIDI	512	f_0	H+N	multi-scale STFT
mel-DDSP	f_0	n-mel	512	f_0	H+N	multi-scale mel
cylinder-DDSP	f_0	n-mel	512	f_0/N_c	H+N	multi-scale mel
phase-DDSP	f_0, ϕ_c	n-mel	512	f_0	H+N	multi-scale mel
large-DDSP	f_0, ϕ_c	n-mel	2048	f_0	H+N	multi-scale mel
transient-DDSP	f_0, ϕ_c	n-mel	512	f_0	H+N+T	multi-scale mel

harmonic synthesizer and a 65th order finite impulse response in the filtered noise synthesizer.

There are essentially two differences between vanilla-DDSP and the solo-violin DDSP Autoencoder. One difference is that vanilla-DDSP does not take a loudness feature as input, only the fundamental frequency f_0. In synthesis of musical instruments, loudness corresponds to the velocity of the note—for instance, how hard you strike a piano key. Since there is no obvious corresponding control input for engine sounds, we do not use the loudness feature.

The other difference is that vanilla-DDSP is adapted to use an audio sample rate of 48 kHz instead of 16 kHz. Due to this increased sample rate, we changed the FFT sizes in the multi-scale STFT loss that is used when training vanilla-DDSP. Details on this loss are given in subsection 3.3.1.

3.2.2 Multi-Scale Mel Loss (mel-DDSP)

The multi-scale STFT loss uses a linear frequency scale, meaning that all frequencies are weighted equally. However, we know that human sound perception puts less importance on higher frequencies than on lower. Therefore, it might be beneficial to use a perceptually motivated frequency scale instead of a linear, especially after having introduced high-frequency information in the 8–24 kHz range by increasing the sample rate to 48 kHz. Furthermore, from preliminary experiments, we noted that the multi-scale STFT loss is rather noisy during

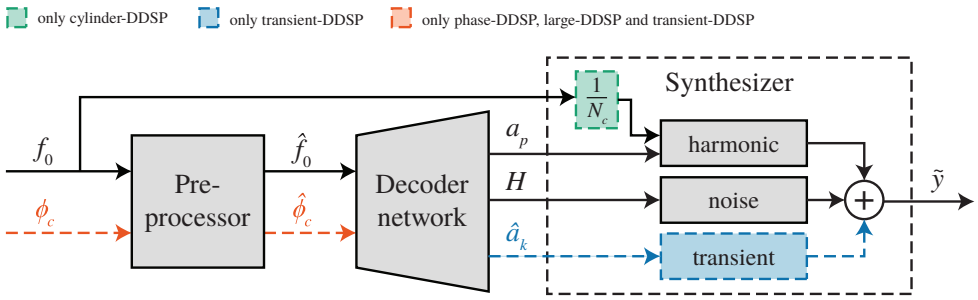

Figure 3.6: Schematic view of our models and their components. See Table 3.1 for a tabular representation of the differences between the models.

training. To take both of these observations into account, we introduce a model, mel-DDSP, which has the same structure as vanilla-DDSP but uses a different loss function based on the mel frequency scale during training. This loss function is explained in detail in subsection 3.3.2.

Another change we make in mel-DDSP is the normalization of f_0 in the preprocessing block. Instead of relying on a musical scale such as the MIDI scale, and to ensure that the decoder input lies in the interval[6] $(0, 1)$, we instead use a normalized mel scale by computing

$$\hat{f}_0^{\text{mel}} = \frac{\log(1 + f_0/700)}{\log(1 + f_0^{\max}/700)} \tag{3.9}$$

where $f_0^{\max} = 138.43$ is the maximum f_0 value in our dataset.

3.2.3 Inharmonic Partials (cylinder-DDSP)

Although car engine sounds are essentially harmonic, they contain inharmonic partials; i.e., some partials are not integer multiples of the fundamental frequency f_0. This is explained in detail in subsection 2.1.2.

Since the explosions in each cylinder sound a little different, the harmonic spectrum is enriched with partials that are integer multiples of the frequency of the full engine cycles $f_c = f_0/N_c$, where N_c is the number of cylinders. We use this observation in a model we call cylinder-DDSP by simply substituting f_0 with f_0/N_c in the harmonic synthesizer. Apart from this, cylinder-DDSP has the same structure as mel-DDSP.

[6]Using the normalized MIDI scale in (3.8), all frequencies between 0 Hz and $440 \times 2^{-69/12}$ Hz ≈ 8.18 Hz are mapped to the interval $(-\infty, 0)$.

3.2.4 Engine Cycle Phase as Input (phase-DDSP)

When listening to engine sounds or looking at their mel spectrograms, it is apparent that for low speeds—especially for idling engines—engine sounds have periodic pulses corresponding to the churning and clicking from the engine cycles (see e.g. the top-left corner in Figure 2.1). Learning where in time to place these pulses is a challenging task for the network that might become simpler by adding features that are aligned with the pulses.

Since the periodicity of the pulses is determined by the phase of the engine cycles ϕ_c, adding ϕ_c alongside f_0 as input to the model may help. We do this in our model phase-DDSP, where we estimate ϕ_c from the audio and f_0 signals as described in subsection 3.1.2.

In phase-DDSP, ϕ_c is shifted and scaled in the preprocessing block to lie in $(0, 1)$ before being sent through the decoder network. Preprocessed versions of f_0 and ϕ_c are both used as inputs to the decoder network by using a shared-bottom architecture in the same way the DDSP Autoencoder uses f_0, loudness and the z feature as inputs. See section A.1 for more details on the decoder network.

Apart from the additional ϕ_c input, phase-DDSP has the same structure as mel-DDSP. Due to the individual MLP for ϕ_c before the shared bottom in the decoder network, phase-DDSP will have more trainable network parameters than the three models yet introduced.

3.2.5 Higher-Dimensional GRU (large-DDSP)

From our preliminary experiments, it seems like our engine sounds might be more challenging to model than the violin sounds used by the DDSP authors to train the solo-violin DDSP Autoencoder. This may be due to a combination of the increased sample rate from 16 kHz to 48 kHz and the complex nature of engine sounds. Therefore, the models may benefit from having more expressivity than the original DDSP Autoencoder.

A simple way to increase the expressivity is to increase the dimensionality of the hidden state in the decoder network's GRU. We experiment with this in our model large-DDSP, where we extend the original 512-unit GRU to a 2048-unit GRU. Apart from the larger GRU, large-DDSP has the same structure as phase-DDSP. The increased dimensionality of the hidden state in the GRU will increase the number of trainable parameters significantly compared to phase-DDSP.

3.2.6 Synthesizing Transients (transient-DDSP)

Engine sounds are rich in transients, especially so at low engine speeds in our recordings. In the harmonic-plus-noise synthesizer of the DDSP Autoencoder, transients have to be modeled as fast changes in the amplitudes of the harmonics or the in time-dependent filter of the noise.

Modeling transients separately could be helpful, which is why we experiment with a synthesizer inspired the model proposed by Verma and Meng [10]. Similar to Verma and Meng, we keep the harmonic and noise components and add a transient component which performs additive synthesis in the discrete-cosine transform (DCT) domain. This means that the transient component uses an additive synthesizer whose output is transformed via an inverse DCT (IDCT) to generate the transients in the time domain. Hence, the decoder network outputs the amplitudes of the sinusoids of an additive synthesizer, in addition to the parameters for the harmonic and noise synthesizers.

Instead of using a harmonic distribution based on f_0, as in the harmonic synthesizer, the additive synthesizer of the transient component uses a constant frequency grid with equidistant frequencies on the interval $[0, f_N)$, where f_N is the Nyquist frequency. After taking the IDCT of the additive synthesizer's output, this corresponds to placing transients equally spaced in time over the audio frame. We use 160 DCT frequencies in the additive synthesizer of our transient component, which for our two-second windows corresponds to 80 transients per second.

3.3 Training and Loss Functions

To obtain parameter values in the neural networks of our models, we train the models to reconstruct examples of a dataset we create using the methods described in section 3.1. A schematic illustration of the training and reconstruction processes is presented in Figure 3.5.

Just as the DDSP authors, we train our models using an Adam [26] optimizer with learning rate 0.001 and an exponential learning rate decay of 0.98 every 10 000 steps.

3.3.1 Multi-Scale STFT Loss

All loss functions we use are based on the multi-scale spectral loss in (2.11) proposed by the DDSP authors. Our vanilla-DDSP model the same loss function, only with different FFT sizes. Since our experiments use 48 kHz audio

sample rate we increase the FFT sizes to powers of 2 from 128 to 8192. This ensures we look at time scales that are similar to those in the original loss when using the lower sample rate of 16 kHz originally used by the DDSP authors.

3.3.2 Multi-Scale Mel Loss

All models other than vanilla-DDSP use a slightly different loss function. Since we use a three times higher sample rate than the DDSP authors, we get three times as much frequency information in our audio signals. All of this new information ends up in the high-frequency range, specifically 8–24 kHz.

Using STFT directly, which is done in the original loss, means using a linear frequency scale (Hz) and gives equal weighting on all frequencies. However, we perceive frequencies as logarithmically spaced. Therefore, instead of using a linear frequency scale, we use the mel scale.

The mel scale [31] scale is a perceptual scale of pitches judged by listeners to be equal in distance from one another, and is defined by

$$m = 1127 \log\left(1 + \frac{f}{700}\right). \tag{3.10}$$

This scale gives more weight to lower frequencies and, conversely, less weight to higher frequencies in the sense that lower frequencies are spaced wider apart and higher frequencies closer together compared to a linear scale.

To be able to use the mel scale, we need a way to determine the number of mel frequency bins n_{mel} to use given an FFT size n_{FFT}. The way we do this is by using the equation

$$n_{\mathrm{mel}} = \left\lfloor \frac{100}{161} \frac{m_N n_{\mathrm{FFT}}}{f_s} \right\rfloor, \tag{3.11}$$

where m_N is the Nyquist frequency expressed in mels and f_s is the sample rate in Hz. We derive (3.11) using the uncertainty principle of signal processing and the definition of the mel scale in (3.10). For a detailed derivation, see section A.2.

To make sure we have high enough resolution in both frequency and time, we use FFT sizes that are all powers of 2 from 128 to 16384. FFT size 128 ensures a time resolution that corresponds to the input sample rate of $f_f = 250\,\mathrm{Hz}$ since

$$2^{\lfloor \log_2(f_s/f_f) \rfloor} = 2^{\lfloor \log_2(48\,\mathrm{kHz}/250\,\mathrm{Hz}) \rfloor} = 128 \tag{3.12}$$

while 16384 ensures a frequency resolution of about 2 Hz since

$$2^{\lfloor \log_2(48\,\mathrm{kHz}/2\,\mathrm{Hz}) \rfloor} = 16384. \tag{3.13}$$

The choice of 2 Hz frequency resolution is rather arbitrary, but 2 Hz is roughly a semitone's distance at 24 Hz which is the lowest f_0 value of our engine recordings.

Apart from using the mel frequency scale, this loss also only uses the logarithm of the spectrograms. Our new multi-scale mel loss can be expressed

$$\mathcal{L}_{\text{mel}} = \sum_i \|\log S_i - \log \hat{S}_i\|_1 \tag{3.14}$$

where S_i and $\hat{S}_i$ are the mel-frequency magnitude spectra of the original and reconstructed audio, respectively, for FFT sizes $128, 256, ..., 16384$. As in the original STFT loss, the spectra in the multi-scale mel loss use a 75 % overlap between neighboring frames.

3.4 Evaluation

There are no ideal methods for objectively evaluating sound synthesis performance. Since our aim is to generate audio that to a human listener sounds similar to a target sound, a good option would be to use mean opinion scores from a formal listening test where a group of experts assign scores to generated audio examples based on their opinion of the quality. However, due to time constraints we cannot conduct any formal listening tests. Instead, we propose objective metrics and discuss the synthesized audio based on our own opinions after listening to it.

Simply computing the mean absolute error (MAE) between the time series of the original and reconstructed audio would not yield a good metric, since two audio signals that have very different time representation can sound the same to a human listener [16]. Instead, we look at perceptually motivated metrics that disregard the phase information of the audio—just like we do for the loss functions used during training of the models.

We use two metrics to quantify the similarity between the target audio and the reconstruction. The first metric is the multi-scale mel loss described in subsection 3.3.2. Since this is the loss function used during training of all models except one (vanilla-DDSP), we propose a second independent metric based on the constant-Q transform (CQT). Call this metric the CQT loss. We define the CQT loss as

$$\mathcal{L}_{\text{CQT}} = \|S_{\text{CQT}} - \hat{S}_{\text{CQT}}\|_1 \tag{3.15}$$

where S_{CQT} and $\hat{S}_{\mathrm{CQT}}$ are the CQT magnitudes in dB of the target audio and the synthesized audio, respectively, and $\|\cdot\|_1$ denotes the L_1 norm[7].

[7]Like the DDSP authors, we commit the slight abuse of notation by letting the L_1 norm to denote the mean absolute error (MAE).

Chapter 4

Results

In this chapter, we present data resulting from using the recording and processing methods in section 3.1, as well as results from training the models introduced in section 3.2. To present the generated sounds, we include spectrograms as visual representations. Since an image can only say so much about a sound, we encourage the reader to listen to the audio in our online supplementary material[1] while inspecting the corresponding spectrograms. Our implementation is written in Python using TensorFlow 2.0 [22] together with DDSP [8] and the source code is publicly available on GitHub[2].

We begin this chapter by taking a closer look at our recordings and the resulting dataset in section 4.1. Computed properties of our models and aspects of our experiments related to training are presented in section 4.2. In section 4.3, we show results from training a collection of models to reconstruct only 6 s of training data. This is followed by reconstruction results from models trained on the full dataset in section 4.4. Finally, in section 4.5, we present results from audio generation using synthetic input signals.

4.1 Recordings and Data

The data we use in our experiments consist of audio and input signals captured and processed as described in section 3.1. For our audio recordings, we use a Røde M5 condenser microphone. Our engine sound source is a diesel engine with four cylinders ($N_c = 4$), specifically a 2.0 liter TDI engine inside a Ford S-Max 2010. We record audio while varying the gas pedal position with the transmission in neutral gear.

[1] `https://antberg.github.io/thesis`
[2] `https://github.com/antberg/thesis`

In total, our diesel-engine dataset consists of about 32 min of recordings: 30 min of mixed gas-pedal variation and 2 min of idling. Instead of using the rather low sample rate of 16 kHz that the DDSP authors use, our recordings are sampled at 48 kHz. This means that our data has high-frequency information in the 8–24 kHz range which the DDSP authors do not.

We split our recordings into two-second examples, which results in a total of 957 examples. Between these examples, we make a 70 %-15 %-15 % train-validation-test split, resulting in 675 examples for training and 141 examples each for validation and testing.

To illustrate what the examples in our dataset look like, we present in Figure 4.1 time-domain plots of the audio and inputs together with spectrograms of three training examples. The first spectrogram type uses the mel frequency scale. To be able to see the low-frequency content better, we also include constant-Q transform (CQT) spectrograms. The examples in Figure 4.1 are selected so that the first is of an idling engine, the second is at a mid-range RPM and the third is of a revving engine at a high-range RPM. Similarly, we have selected three examples from the test split which are presented in Figure 4.2. Remember that the f_0 input is estimated independently of the audio recording, using the method described in section 3.1, and to verify that the estimation is correct we plot f_0 on top of the spectrograms. We use the training and test examples in Figures 4.1 and 4.2 to illustrate the performance of our models in the experimental results presented in sections 4.3 and 4.4.

To see how the f_0 inputs in our dataset are distributed, we present normalized histograms in the top row of Figure 4.3. Our f_0 inputs vary between 24–138 Hz, corresponding to 720–4200 RPM. The histograms use 30 bins of equal width in this interval. Notice how the recordings of the engine at idle show up as a peak in the left-most bin while the spread of f_0 has a skewed bell shape for when the gas pedal is varying. In an attempt to illustrate how the f_0 inputs vary between time steps, we present normalized histograms of differences between adjacent f_0 samples in the bottom row of Figure 4.3. The y-axis is logarithmic to easier see the edges of the distributions. Notice that f_0 mostly varies slowly; i.e., $\Delta f_0 \approx 0$ for most time steps. Also, the distribution is asymmetrical with the maximum acceleration being greater than the maximum deacceleration.

4.2 Models and Training

In all experiments that follow, we use the models introduced in section 3.2. The training procedure was explained in section 3.3 and is illustrated in Figure 3.5.

We train our models by distributing the computations over two Nvidia

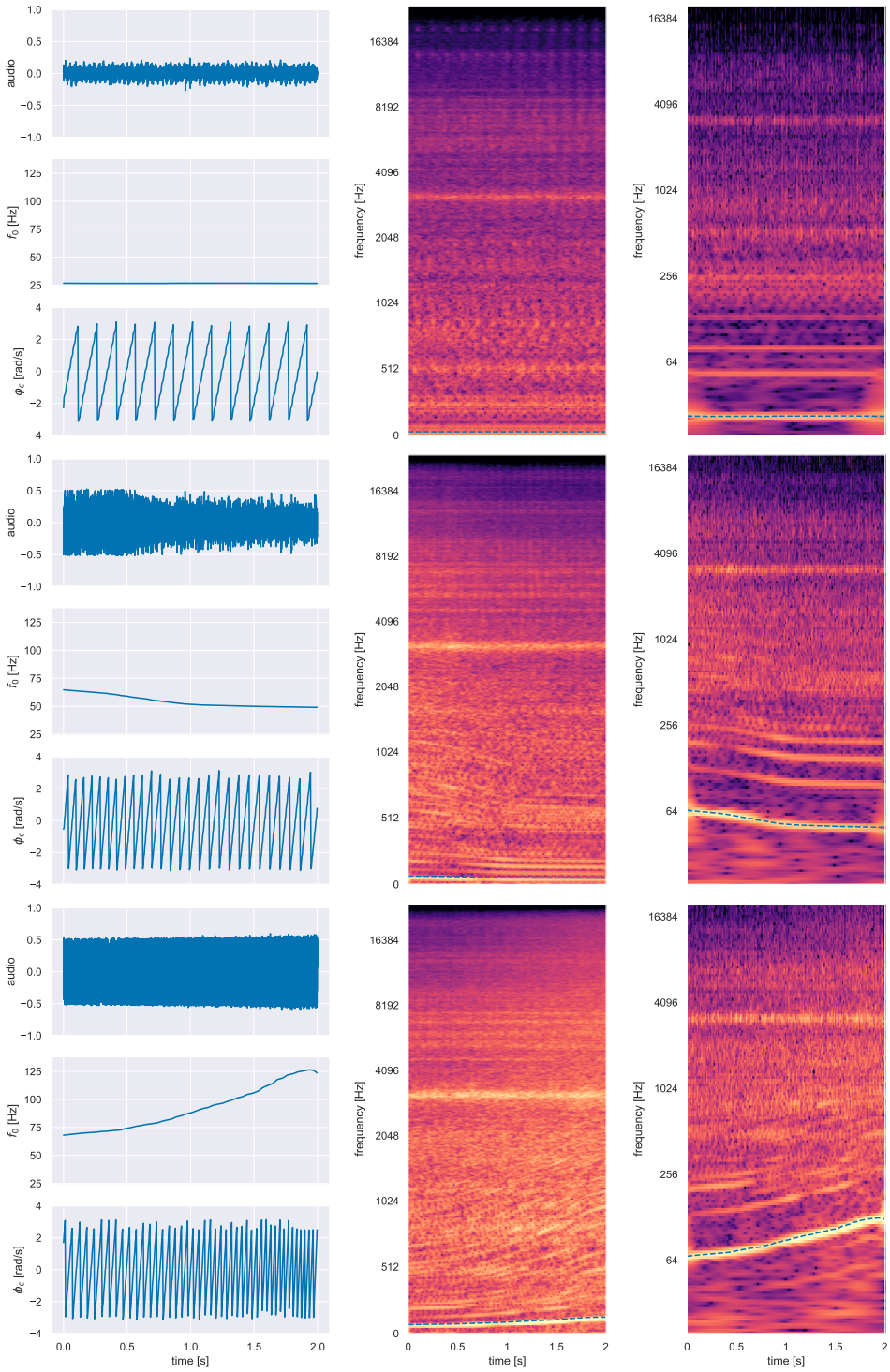

Figure 4.1: Time-domain plots of audio and inputs (f_0, ϕ_c) for three training examples (left), mel spectrograms of the audio (middle), and CQT spectrograms of the audio (right). To verify that our estimated f_0 inputs are correct, we plot them on top of the spectrograms (dashed line). The colors cover a 75 dB range for all spectrograms.

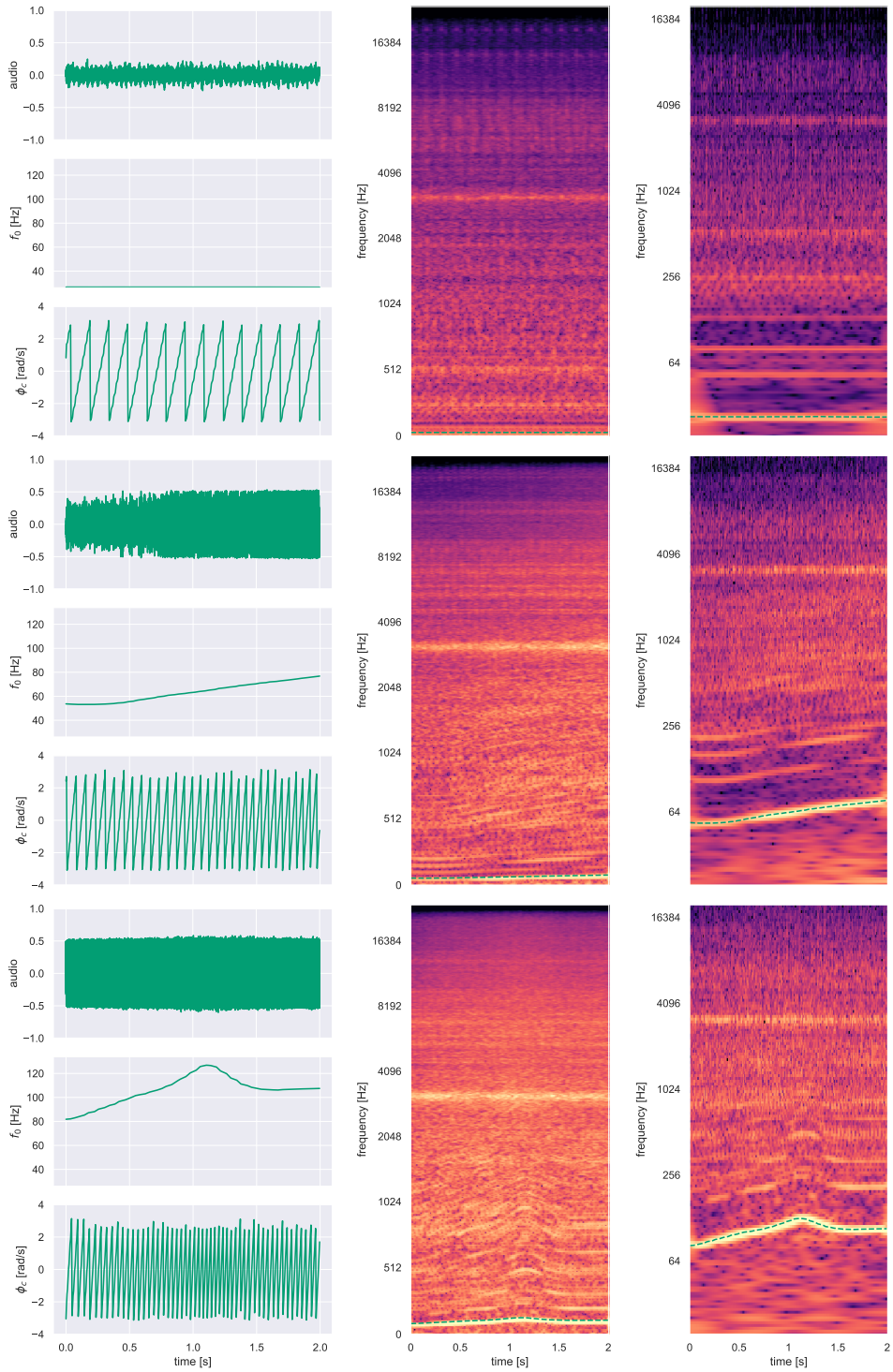

Figure 4.2: Time-domain plots and spectrograms for three test examples. See caption of Figure 4.1 for details.

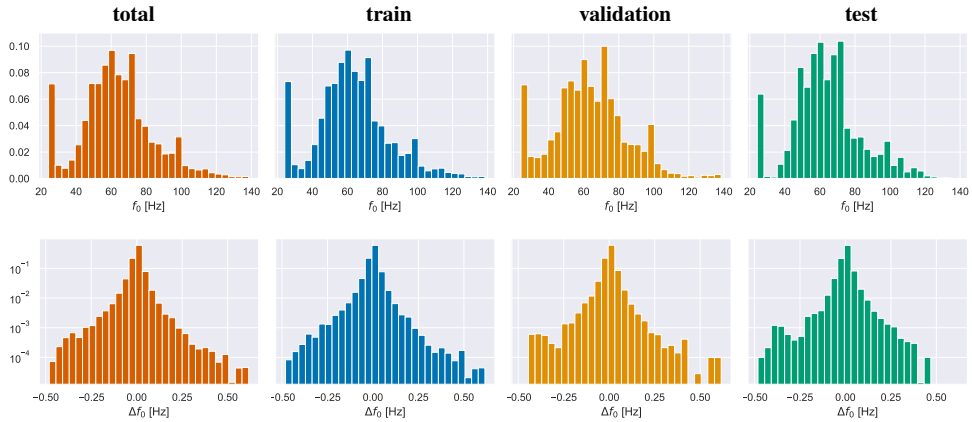

Figure 4.3: Normalized fundamental frequency f_0 histograms for each split in our dataset (top) and normalized histograms of the difference Δf_0 in fundamental frequency between time steps for each split in our dataset (bottom).

GeForce GTX TITAN X 12 GB GPUs. The models are trained to reconstruct the training examples of two datasets: the full 32 min dataset described in the previous section, and a subset which only has the three training examples of Figure 4.1. The results of these two training processes are presented later in section 4.4 and section 4.3, respectively.

For a specification of some of the models' properties—specifically, their number of parameters, training time and inference time—see Table 4.1. The presented training time is the time elapsed after 105 468 training steps, which with our used batch size of 32 corresponds to 5000 epochs for our dataset. The inference time measures the time it takes for a model to do one forward pass through the full network, and the presented value is the mean and standard deviation from reconstructing the 675 training examples in our dataset. In addition to presenting inference times on our GPUs, we also present inference times when performing the computations on a CPU. The CPU we use is an Intel Core i7-8650U.

4.3 Overfitting to a Small Dataset

As a first experiment, we train a collection of models using a very small subset of the 32 min dataset containing only the three training examples illustrated in Figure 4.1. This way, we ignore the generalization capabilities of the models and instead focus on each model's sound synthesis method and what difference it makes to add the phase ϕ_c of the engine cycle to the inputs. The models we

Table 4.1: Number of parameters, training time and inference time for each model. The presented training time is the time elapsed after 105.5k training steps on our GPUs. The inference time measures the time it takes for a model to do one forward pass through the full network, and the presented value is the mean and standard deviation from reconstructing the 675 training examples in our dataset.

			inference time [s]	
	parameters	**training time [h]**	GPU	CPU
vanilla-DDSP	3 223 166	25.3	0.059 ± 0.010	0.85 ± 0.08
mel-DDSP	3 223 166	25.3	0.070 ± 0.011	0.84 ± 0.07
cylinder-DDSP	3 223 166	24.7	0.070 ± 0.065	1.06 ± 0.11
phase-DDSP	4 801 150	26.0	0.083 ± 0.070	0.96 ± 0.09
transient-DDSP	4 883 743	42.2	0.098 ± 0.015	1.86 ± 0.15
large-DDSP	22 111 870	69.0	0.088 ± 0.017	3.48 ± 0.17

train are vanilla-DDSP, mel-DDSP, cylinder-DDSP, phase-DDSP and transient-DDSP. Since we are looking only at the sound synthesis methods and what difference ϕ_c makes, we ignore large-DDSP as it is the same as phase-DDSP only with more units in the GRU. In this experiment, we train each model for 30 000 epochs with a batch size of 3; i.e., one batch is the whole dataset. For plots of the losses during training, see Figure 4.4. Note that vanilla-DDSP uses a different loss function (multi-scale STFT) than the others (multi-scale mel), which is why they are plotted on different axes.

To objectively evaluate the models on reconstruction of the three training examples, we present reconstruction losses in Table 4.2. To illustrate that these models do not generalize well, we include losses for the test split of the full dataset alongside losses for the three training examples of this experiment. Judging from the numbers in Table 4.2, vanilla-DDSP performs worst on reconstructing the training examples in the sense that it gives the highest losses. Using integer multiples of f_0/N_c in the harmonic synthesizer of cylinder-DDSP gives about the same as or slightly higher reconstruction losses than using integer multiples of f_0 as in mel-DDSP. Adding ϕ_c to the input, as in phase-DDSP and transient-DDSP, results in lower reconstruction losses. Adding the transient component to the sound synthesis method, as in transient-DDSP, seems to further lower the reconstruction loss.

To be able to compare audio generated by each model in this experiment,

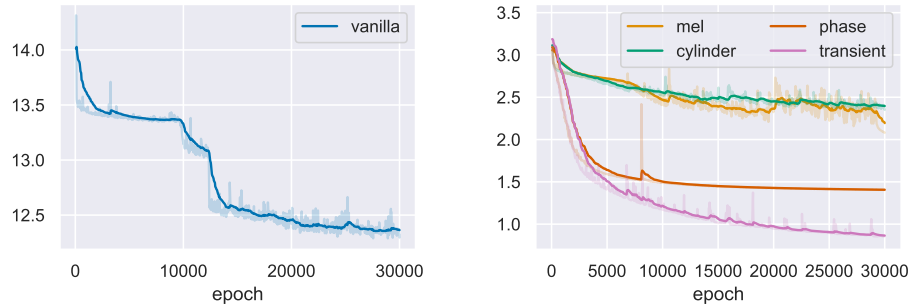

Figure 4.4: Multi-scale STFT (left) and mel (right) losses during training in the experiment described in section 4.3. The low-opacity lines are the computed losses at each iteration, while the opaque lines are smoothed versions with a 0.99 smoothing factor.

Table 4.2: Mean multi-scale mel and CQT losses for audio reconstructed by the models in the experiment described in section 4.3. Minimum mean losses for each column are boldfaced.

	multi-scale mel		**CQT**	
	train	test	train	test
vanilla-DDSP	2.90 ± 0.23	4.47 ± 1.49	5.78 ± 0.33	$\mathbf{7.53} \pm 1.72$
mel-DDSP	2.08 ± 0.32	4.58 ± 1.87	4.42 ± 0.20	7.87 ± 2.05
cylinder-DDSP	2.34 ± 0.45	4.43 ± 1.43	4.29 ± 0.62	8.10 ± 1.30
phase-DDSP	1.40 ± 0.09	$\mathbf{4.42} \pm 0.71$	2.85 ± 0.59	8.96 ± 0.95
transient-DDSP	$\mathbf{0.87} \pm 0.03$	6.97 ± 1.59	$\mathbf{2.03} \pm 0.56$	9.31 ± 0.88

we present audio examples in our online supplementary material as well as mel spectrograms in Figures 4.5 to 4.7. In the figures, for each model there is a spectrogram of the reconstructed audio next to that of the original audio. We also present a difference between the original and reconstructed spectrograms, which is computed by taking the absolute difference between the spectrogram amplitudes. The remaining spectrograms show the output from each of the sound synthesis components: the harmonic synthesizer, the filtered noise synthesizer and the transient synthesizer (where applicable).

Listening to the reconstructed audio, all models capture the frequency characteristics of all three training examples rather well. However, looking only at the idling example, neither of vanilla-DDSP, mel-DDSP or cylinder-

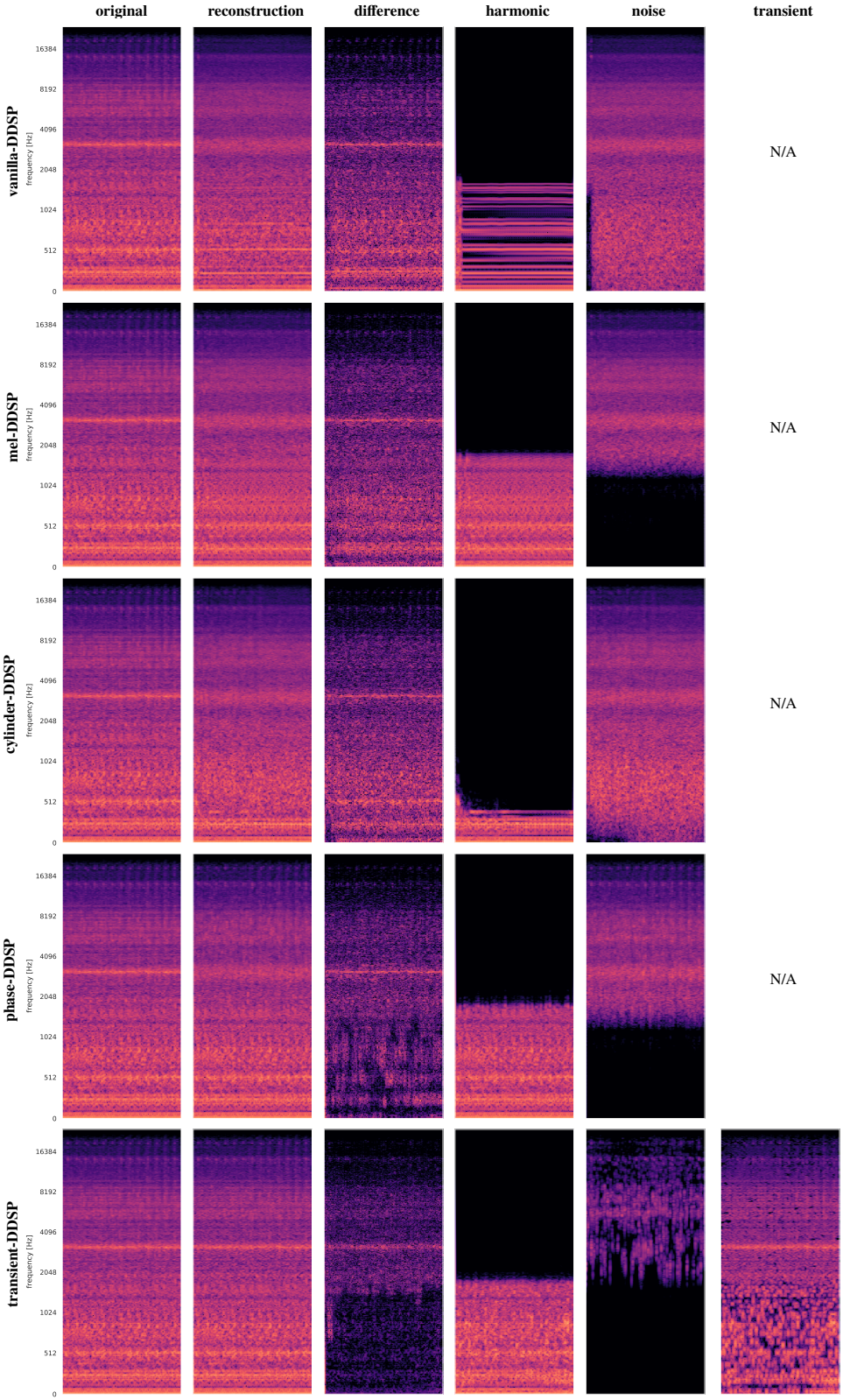

Figure 4.5: Reconstruction of the idle training example in Figure 4.1 by the models in the experiment described in section 4.3. All spectrograms use the mel scale and the colors cover a 75 dB range.

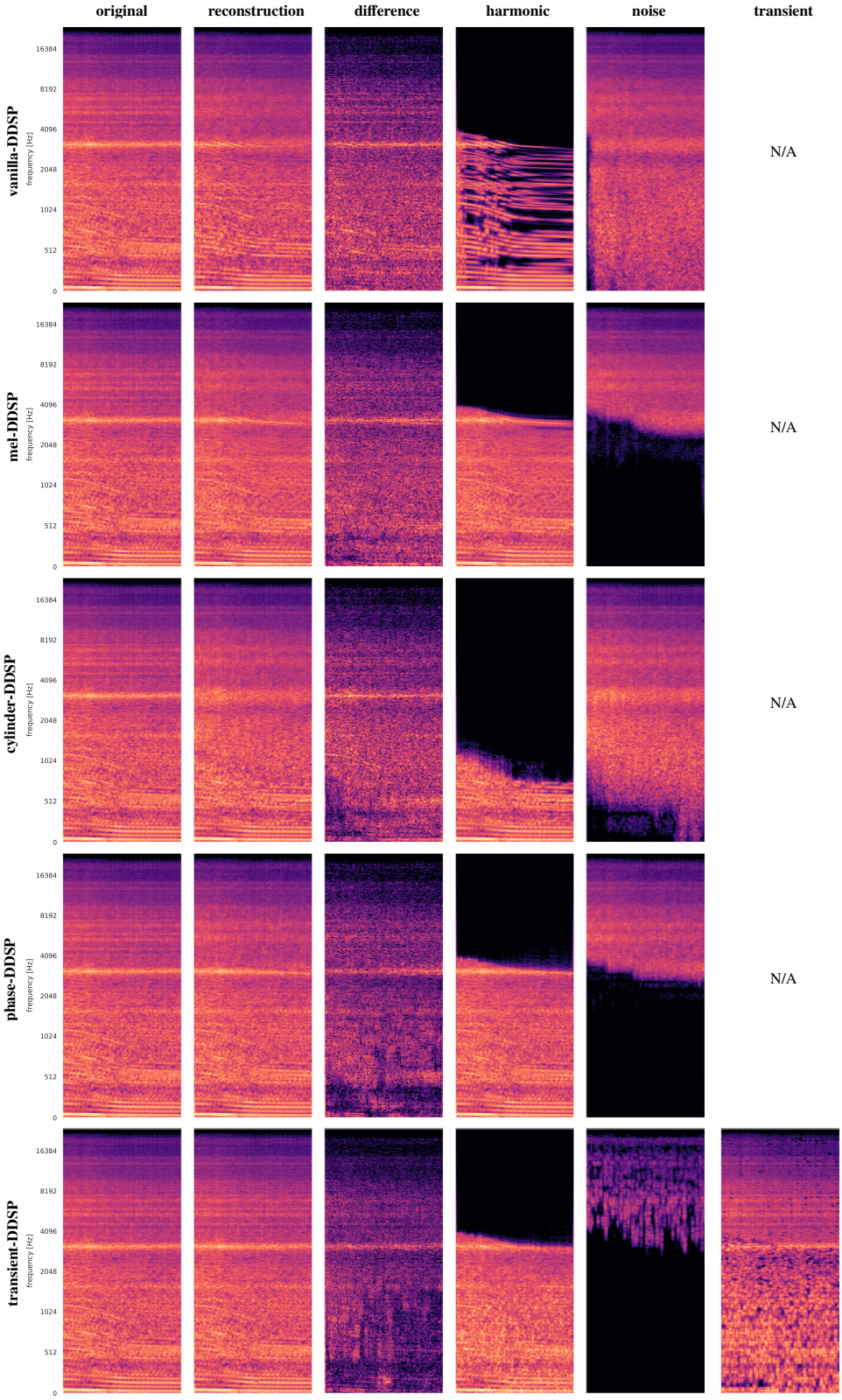

Figure 4.6: Reconstruction of the mid-range RPM training example in Figure 4.1 by the models in the experiment described in section 4.3. All spectrograms use the mel scale and the colors cover a 75 dB range.

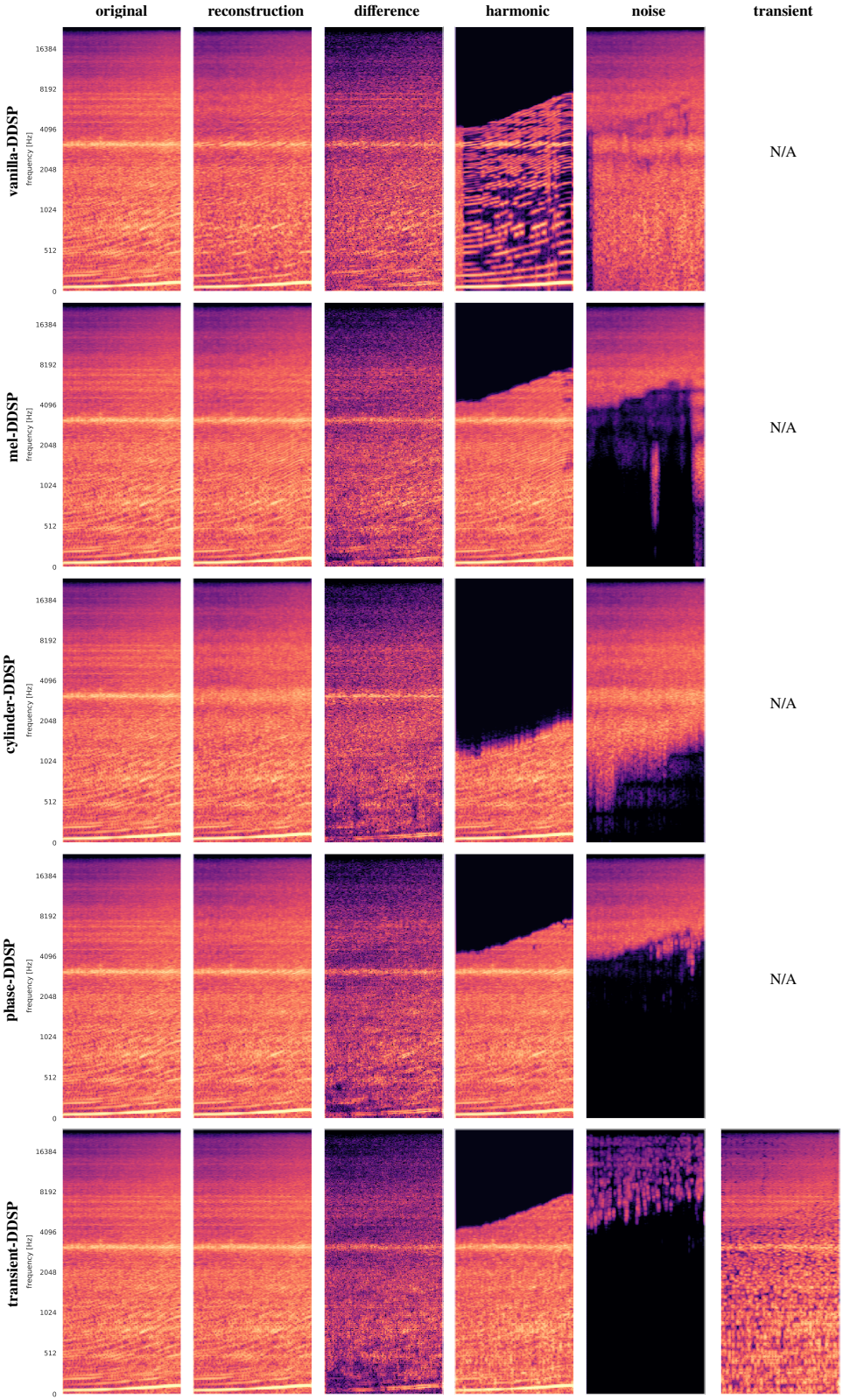

Figure 4.7: Reconstruction of the high-range RPM training example in Figure 4.1 by the models in the experiment described in section 4.3. All spectrograms use the mel scale and the colors cover a 75 dB range.

DDSP—the models who only have f_0 as input—capture the periodic pattern of the engine cycles. Instead of sounding pulsating like the original, they sound smeared out. The models who have both f_0 and ϕ_c, phase-DDSP and transients-DDSP, manage to reproduce the engine cycles better. Looking at the corresponding spectrograms in Figure 4.5, we can see how the engine cycles appear more clearly for phase-DDSP and transient-DDSP compared to vanilla-DDSP, mel-DDSP and cylinder-DDSP. The reconstructed idling example by phase-DDSP and transient-DDSP both sound similar to the original, but differ slightly in different ways. The phase-DDSP audio has audible pre-echo artifacts that sound watery, and the sharp transients in the high-frequency register are not present. The transient-DDSP audio is sharper but has audible artifacts in the form of grainy clicks.

For the mid-range and high-range RPM examples, the vanilla-DDSP reconstructions sound smeared out like with the idle example. However, mel-DDSP and cylinder-DDSP seem to capture the engine cycles better as long as f_0 is varying—as soon as f_0 becomes more or less constant, the cycles disappear. This is most noticeable in the mid-range example. The phase-DDSP and transient-DDSP reconstructions of the mid-range and high-range RPM examples sound again like the originals, but with the same artifacts described earlier (pre-echo and grainy clicks, respectively).

Inspecting the audio generated by each synthesizer component, the decoder networks of mel-DDSP, cylinder-DDSP and phase-DDSP seem to have learned to split up generation of low-frequency and high-frequency audio between the harmonic and noise components, respectively. This is true for transient-DDSP as well, but the new transient component seems to fill in with transients covering the entire frequency spectrum. Only vanilla-DDSP uses the noise component together with the harmonic for low-frequency audio. However, comparing the original and reconstruction spectrograms—especially for the idling example—the harmonic component of vanilla-DDSP does not seem to reproduce a harmonic distribution that is as rich as that of the original sound.

4.4 Reconstruction From Recorded Inputs

In this section, we present the results from training a collection of our models to reconstruct the training examples in the full 32 min dataset. We now look at the generalization capabilities of the models and consider reconstruction from recorded inputs of test examples not seen by the models during training. We consider all models presented in section 3.2, namely vanilla-DDSP, mel-DDSP, cylinder-DDSP, phase-DDSP, transient-DDSP and large-DDSP. All models are

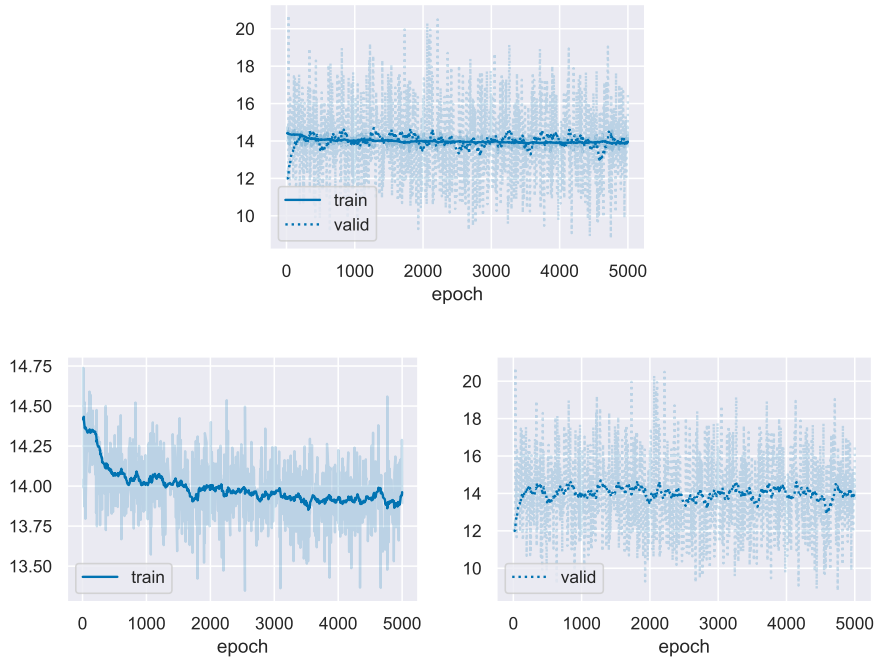

Figure 4.8: Multi-scale STFT losses on training and validation batches during training of the vanilla-DDSP model in the experiment described in section 4.4 (top). To make it easier to inspect the trends in the training and validation losses, we plot these separately (bottom). The low-opacity lines are the computed losses at each iteration, while the opaque lines are smoothed versions with a 0.99 smoothing factor.

trained for 5000 epochs with a batch size of 32. For plots of the losses during training, see Figures 4.8 and 4.9.

To objectively evaluate the models on their reconstruction performance, we present reconstruction losses in Table 4.3. On average, large-DDSP has the lowest reconstruction losses on the training examples. However, looking at the test examples, phase-DDSP and transient-DDSP have the lowest multi-scale mel loss, while cylinder-DDSP has the lowest CQT loss.

To be able to compare the audio generated by each model, we present audio examples in our online supplementary material as well as mel spectrograms in Figures 4.10 to 4.15. Figures 4.10 to 4.12 show spectrograms from reconstruction of training examples, while Figures 4.13 to 4.15 show those of test examples. The figure columns show spectrograms of the original audio, reconstructed audio, difference and individual sound synthesis components in the same way as already explained in section 4.3.

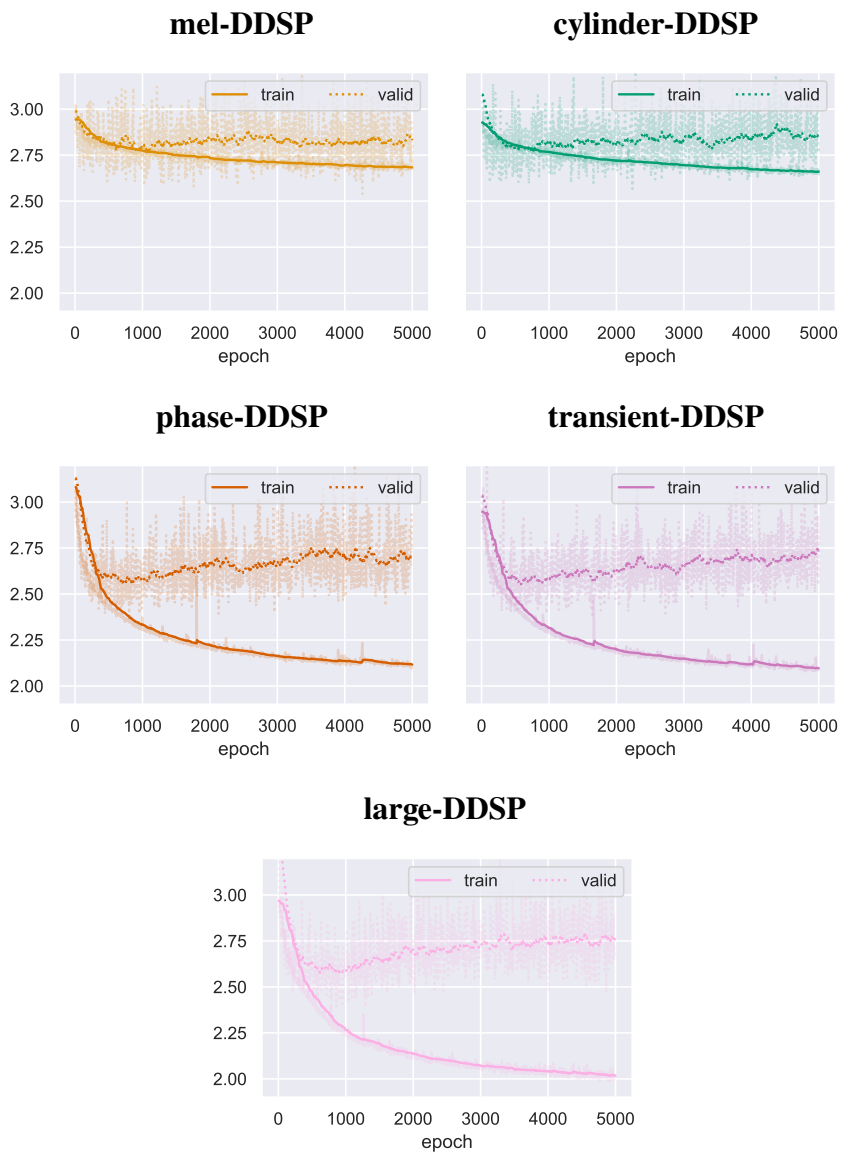

Figure 4.9: Multi-scale mel losses on training and validation batches during training of each model (except vanilla-DDSP) in the experiment described in section 4.4. The low-opacity lines are the computed losses at each iteration, while the opaque lines are smoothed versions with a 0.99 smoothing factor.

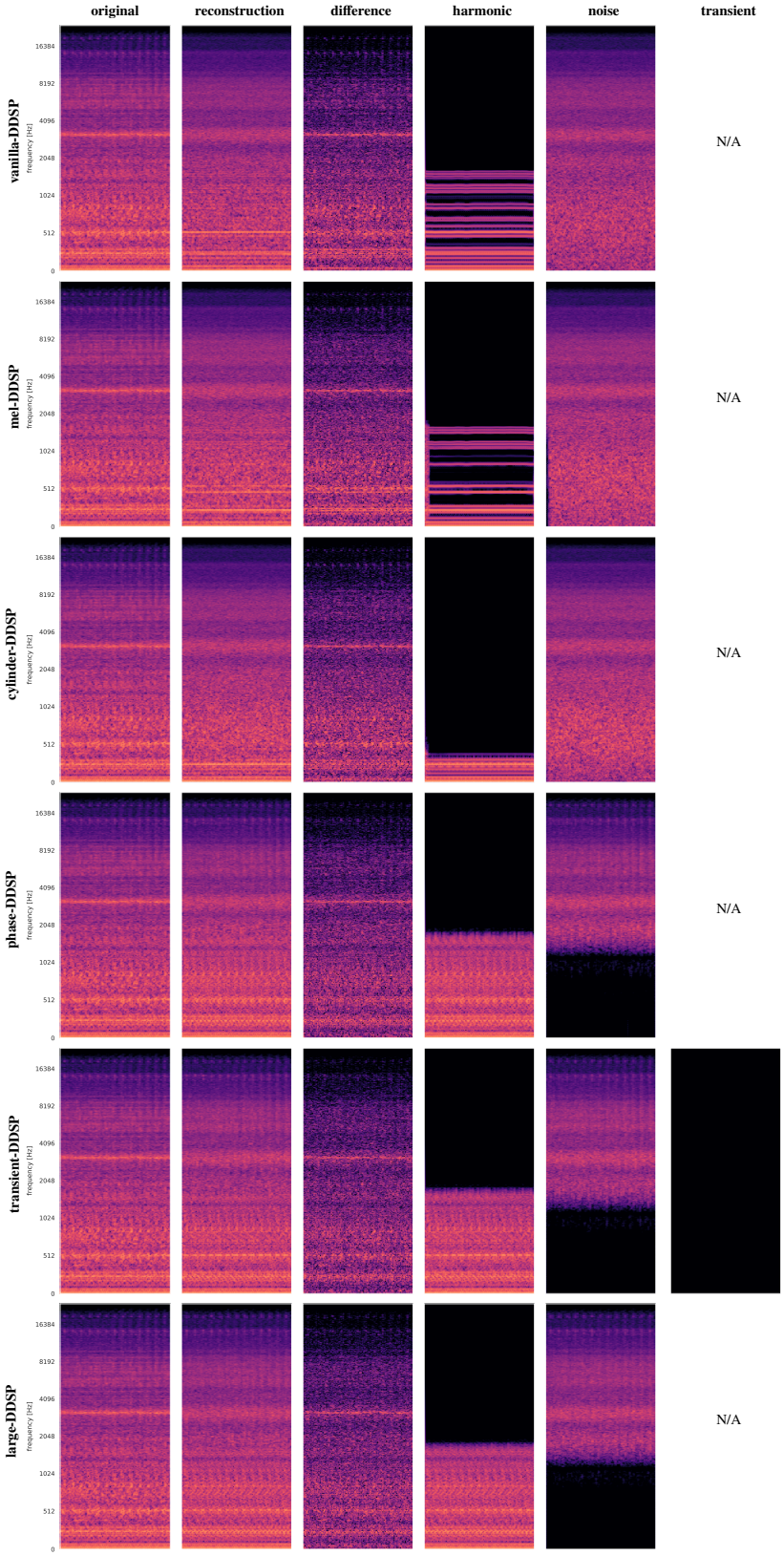

Figure 4.10: Reconstruction of the idle training example in Figure 4.1 by the models in the experiment described in section 4.4. All spectrograms use the mel scale and the colors cover a 75 dB range.

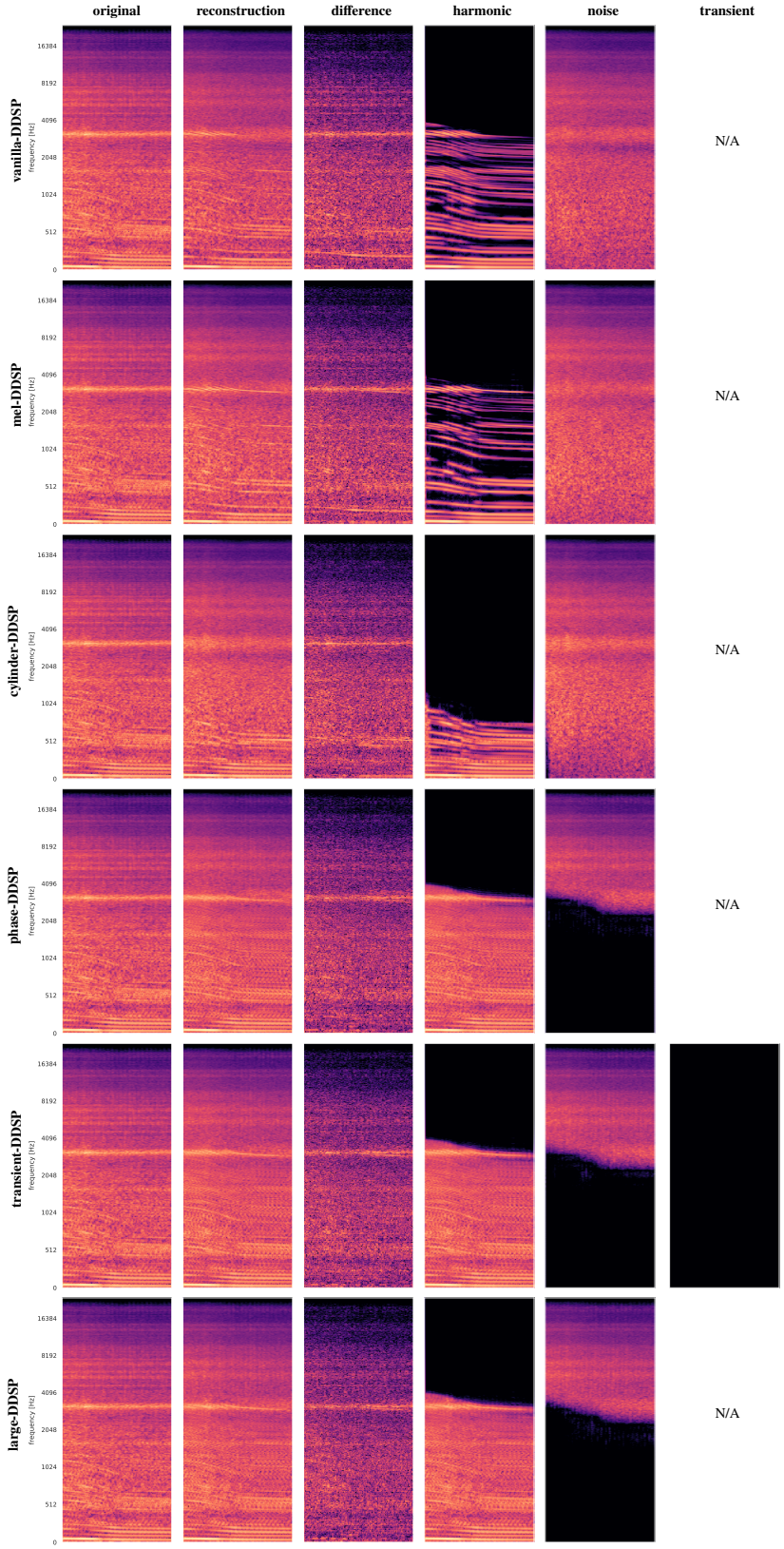

Figure 4.11: Reconstruction of the mid-range RPM training example in Figure 4.1 by the models in the experiment described in section 4.4. All spectrograms use the mel scale and the colors cover a 75 dB range.

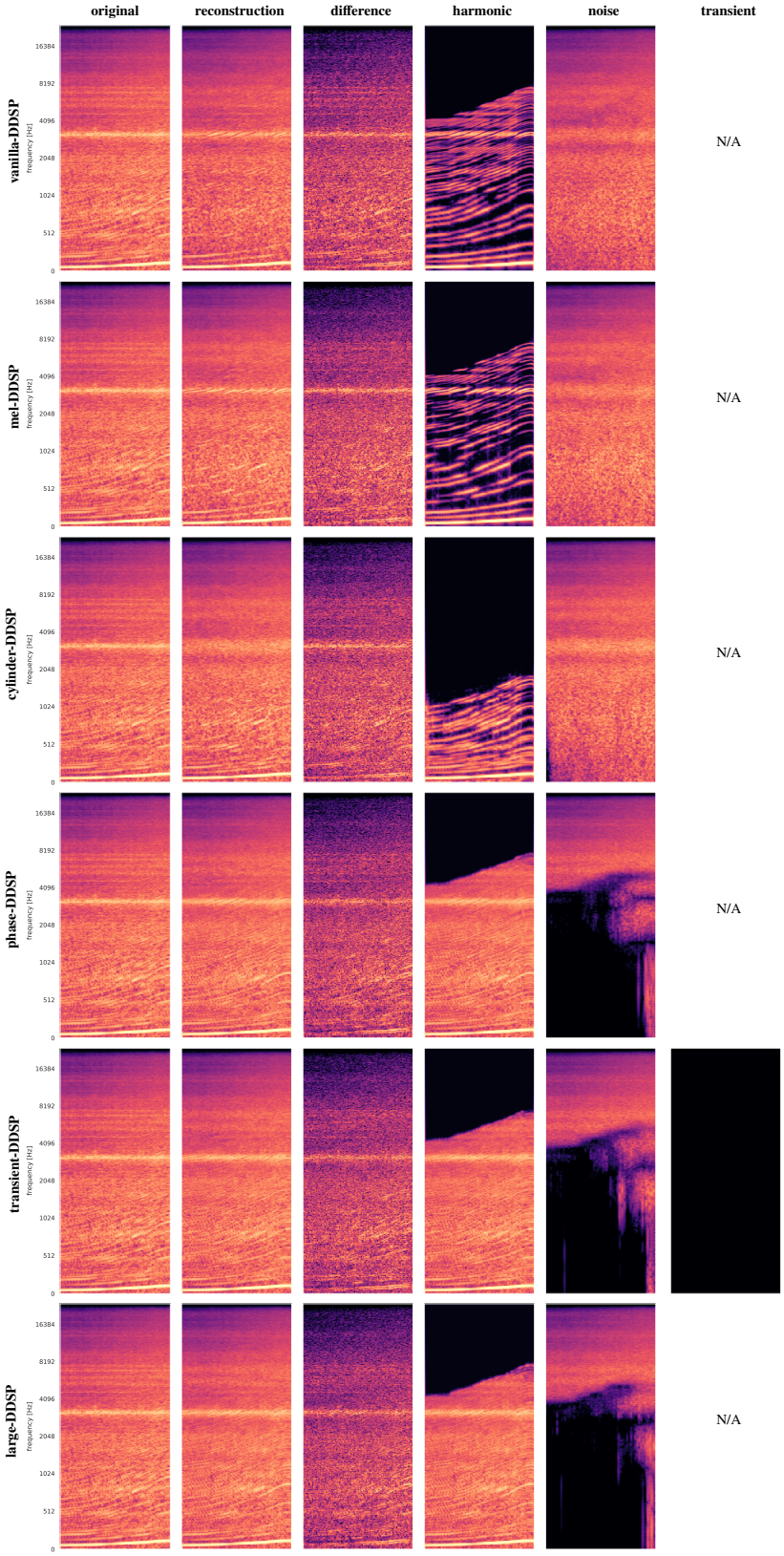

Figure 4.12: Reconstruction of the high-range RPM training example in Figure 4.1 by the models in the experiment described in section 4.4. All spectrograms use the mel scale and the colors cover a 75 dB range.

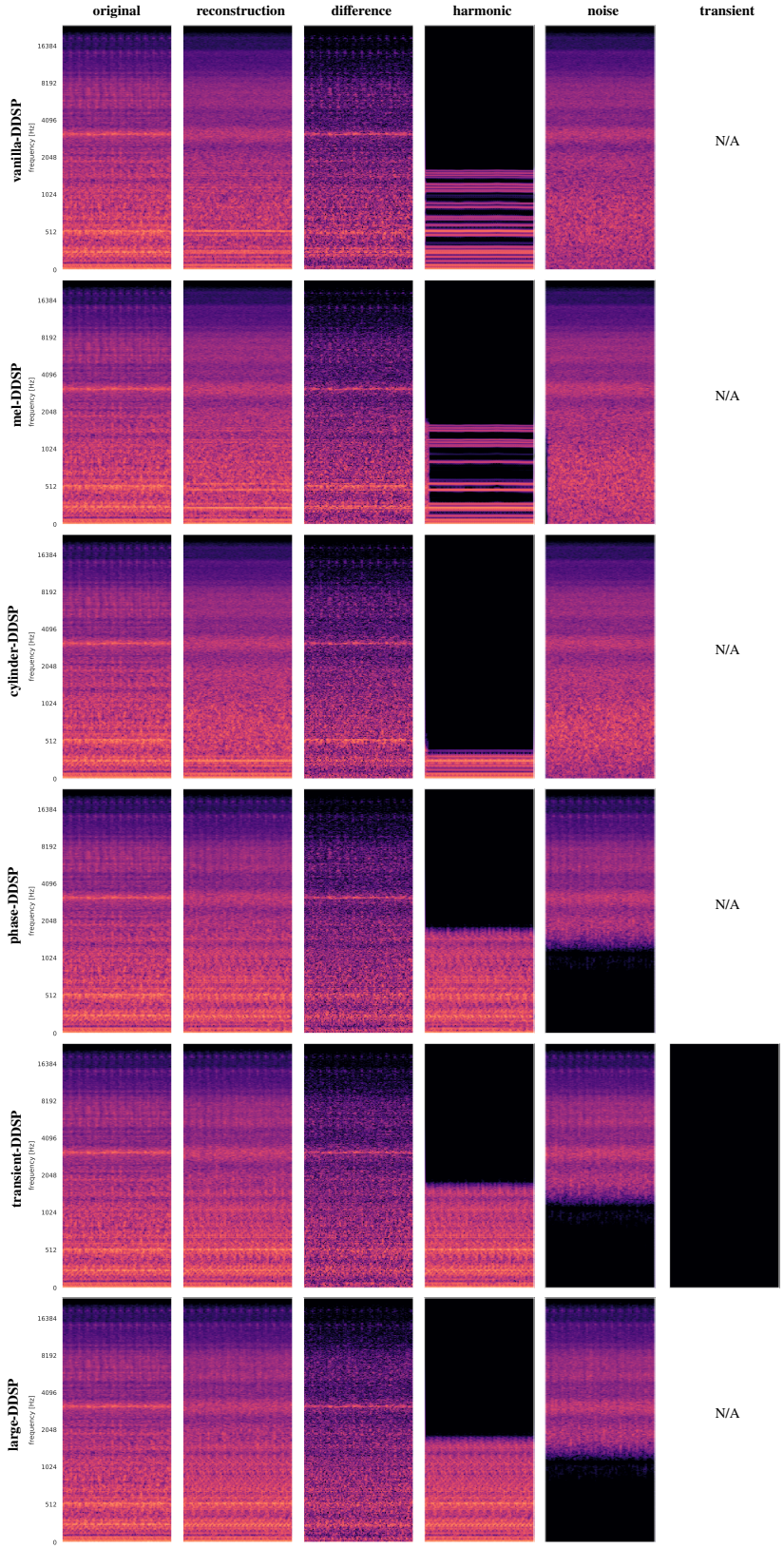

Figure 4.13: Reconstruction of the idle test example in Figure 4.2 by the models in the experiment described in section 4.4. All spectrograms use the mel scale and the colors cover a 75 dB range.

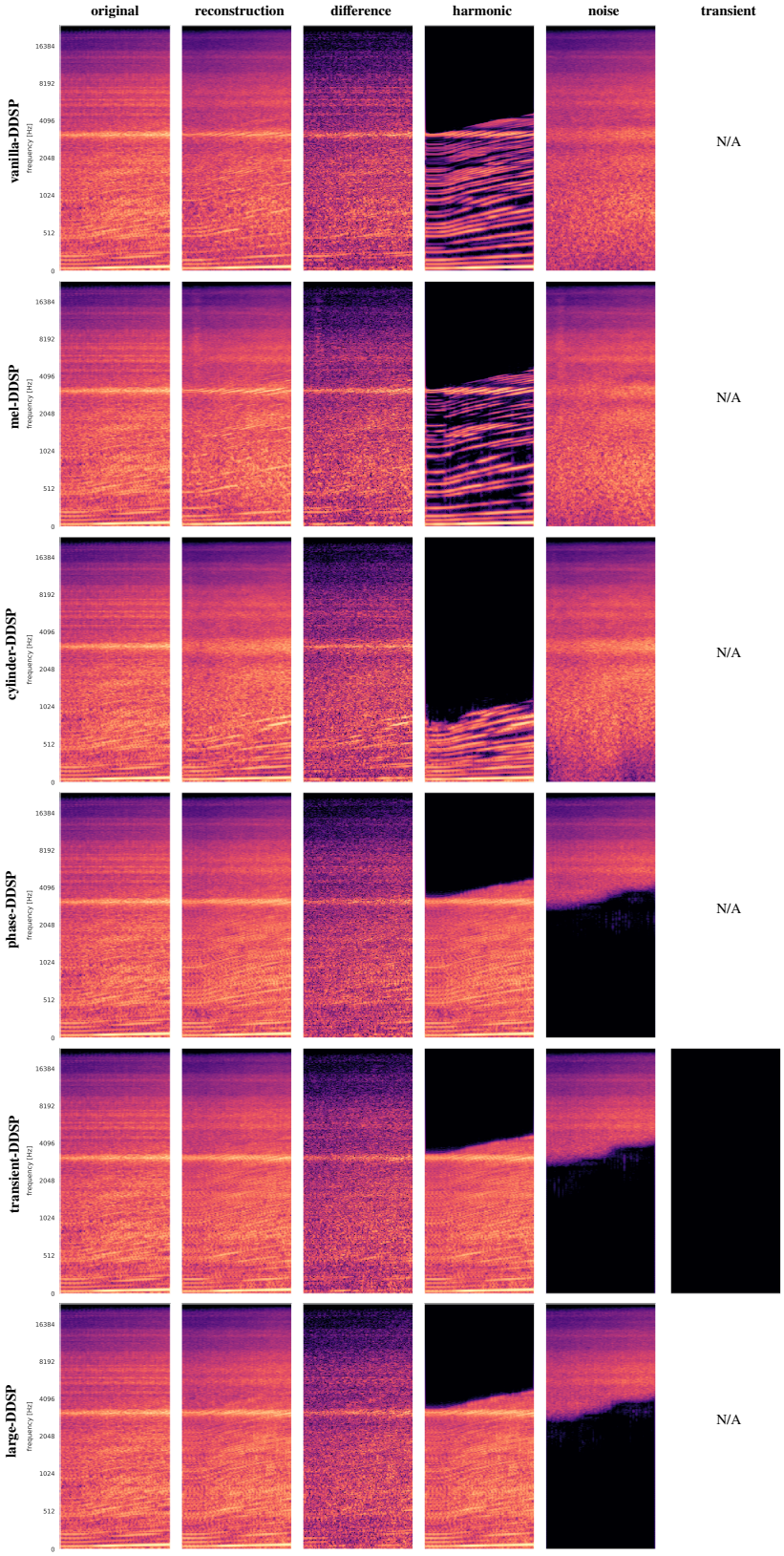

Figure 4.14: Reconstruction of the mid-range RPM test example in Figure 4.2 by the models in the experiment described in section 4.4. All spectrograms use the mel scale and the colors cover a 75 dB range.

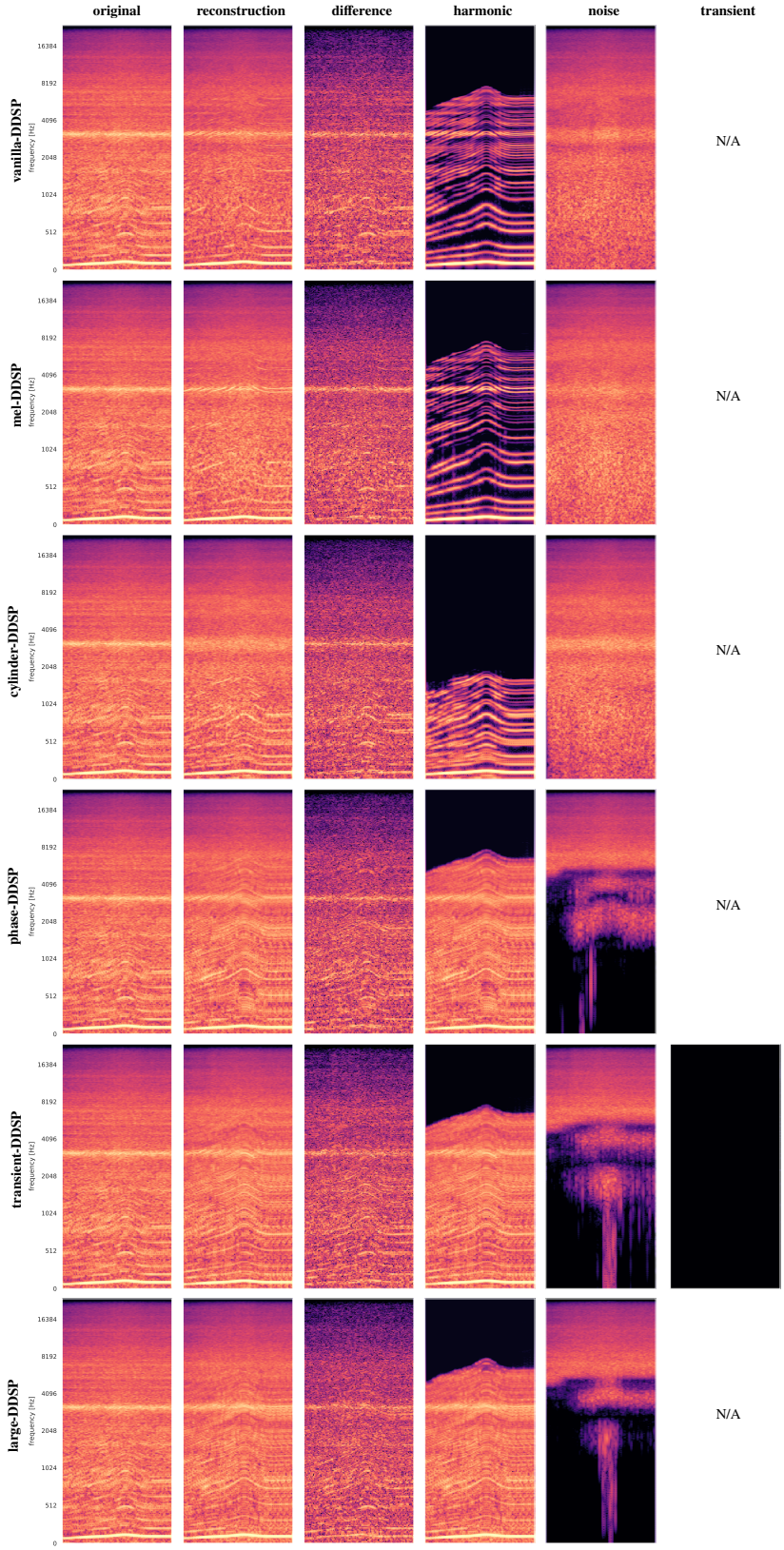

Figure 4.15: Reconstruction of the high-range RPM test example in Figure 4.2 by the models in the experiment described in section 4.4. All spectrograms use the mel scale and the colors cover a 75 dB range.

Table 4.3: Mean multi-scale mel and CQT losses for audio reconstructed by the models in the experiment described in section 4.4. Minimum mean losses for each column are boldfaced.

	multi-scale mel		**CQT**	
	train	test	train	test
vanilla-DDSP	3.03 ± 0.15	3.16 ± 0.29	6.12 ± 0.20	6.24 ± 0.23
mel-DDSP	2.69 ± 0.20	2.84 ± 0.26	5.91 ± 0.23	6.09 ± 0.27
cylinder-DDSP	2.66 ± 0.20	2.87 ± 0.26	5.48 ± 0.22	$\mathbf{5.90} \pm 0.23$
phase-DDSP	2.11 ± 0.10	$\mathbf{2.70} \pm 0.28$	4.68 ± 0.16	5.94 ± 0.32
transient-DDSP	2.09 ± 0.10	$\mathbf{2.70} \pm 0.29$	4.61 ± 0.17	5.93 ± 0.33
large-DDSP	$\mathbf{2.00} \pm 0.14$	2.74 ± 0.27	$\mathbf{4.42} \pm 0.20$	5.97 ± 0.35

Looking at the spectrograms, one difference compared to when only training on three examples, as in the previous section, is that mel-DDSP and cylinder-DDSP now use their noise component for low frequencies and their harmonic components produce a sparser harmonic distribution. This is similar to how vanilla-DDSP behaved in the previous section and again does here.

Another difference is that the transient-DDSP model has learned not to use the transient component in its synthesizer. Instead, it seems to behave much like phase-DDSP.

Listening to the reconstructed training examples, the audio from vanilla-DDSP, mel-DDSP and cylinder-DDSP sound again smeared out lacking the sharp attacks from the engine cycles in the original sound. In contrast, the reconstructed audio from phase-DDSP, transient-DDSP and large-DDSP clearly reproduce the engine cycles. Although neither of them are very similar to the original audio in terms of capturing the fine details, they still sound convincingly like engines.

After listening to the reconstructed test examples, we can say something about the generalization capabilities of the models. For all models, the reconstructed idling and mid-range test examples have about the same properties as the corresponding reconstructed training examples. However, for the high-range test example, the reconstructed audio from phase-DDSP, transient-DDSP and large-DDSP contain audible undesired artifacts.

4.5 Control by Synthetic Inputs

In this section, we present results from using synthetic inputs to the same models used in the reconstruction experiment in section 4.4. Our procedure of controlling the models with synthetic inputs is illustrated in Figure 3.5. The different input signals we use are presented together with spectrograms of synthesized audio in Figures 4.16 to 4.23. To allow the reader to listen to the audio generated by each model, we have included the audio examples corresponding to each spectrogram in our online supplementary material.

Figures 4.16 to 4.18 show three input signals with constant f_0—one low-range signal at 24 Hz (720 RPM) in Figure 4.16, one mid-range signal at 81 (2430 RPM) in Figure 4.17, and one high-range signal at 138 Hz (4140 RPM) in Figure 4.18. These f_0 values are the minimum, center and maximum f_0 in our dataset, and by looking at these we should get an idea of how the models sound at constant speeds.

Figures 4.19 to 4.21 show three input signals with varying f_0—one linear ramp going between 24–138 Hz (720–4140 RPM) in Figure 4.19, a single period of a sinusoid oscillating between 42–138 Hz (720–4140 RPM) in Figure 4.20, and ten periods of a sinusoid oscillating between 52–110 Hz (1560–3300 RPM) in Figure 4.21. The ramp signal gives an idea of how the models sound at constant acceleration, while the oscillating signals help to show how the models sound at (unnaturally) fast variation in speed.

Figures 4.22 and 4.23 show two inputs with f_0 outside the range of the training examples. Figure 4.22 shows a linear ramp between 2–22 Hz (60–660 RPM), which corresponds 10–90 % of the minimum f_0 in the dataset, 24 Hz (720 RPM). Figure 4.23 shows a linear ramp between 152–277 Hz (4560–8310 RPM), which corresponds to 110–200 % of the maximum f_0 in the dataset, 138 Hz (4140 RPM). By looking at these signals, we get an idea of how well the models can extrapolate beyond the f_0 range of the training dataset.

Listening to the generated audio, the low-range f_0 input in Figure 4.16 produces audio similar to that of the idling examples in the reconstruction experiment for all models. The audio from vanilla-DDSP, mel-DDSP and cylinder-DDSP sounds like continuous noise lacking the distinct engine cycles, while phase-DDSP, transient-DDSP and large-DDSP produce clear engine cycles. These observations remain true for the mid-range f_0 input in Figure 4.17.

For the high-range f_0 input in Figure 4.18, the audio from vanilla-DDSP, mel-DDSP and cylinder-DDSP sound similar to an engine at constant high speed, albeit quite noise-like and lacking sharpness. The audio from phase-DDSP, transient-DDSP and large-DDSP contain evident undesirable artifacts,

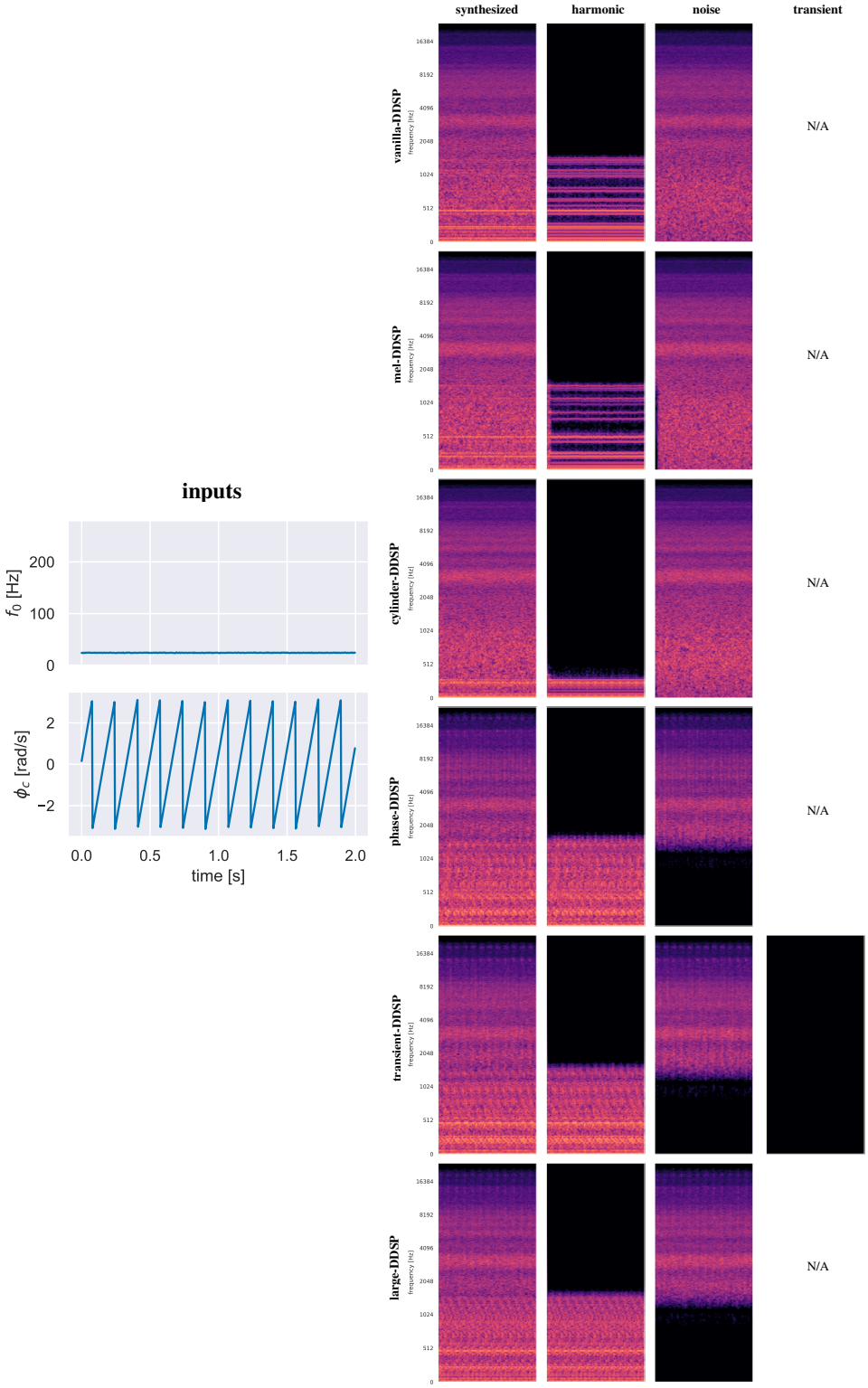

Figure 4.16: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with constant f_0 at 24 Hz (720 RPM). The colors cover a 75 dB range for all spectrograms.

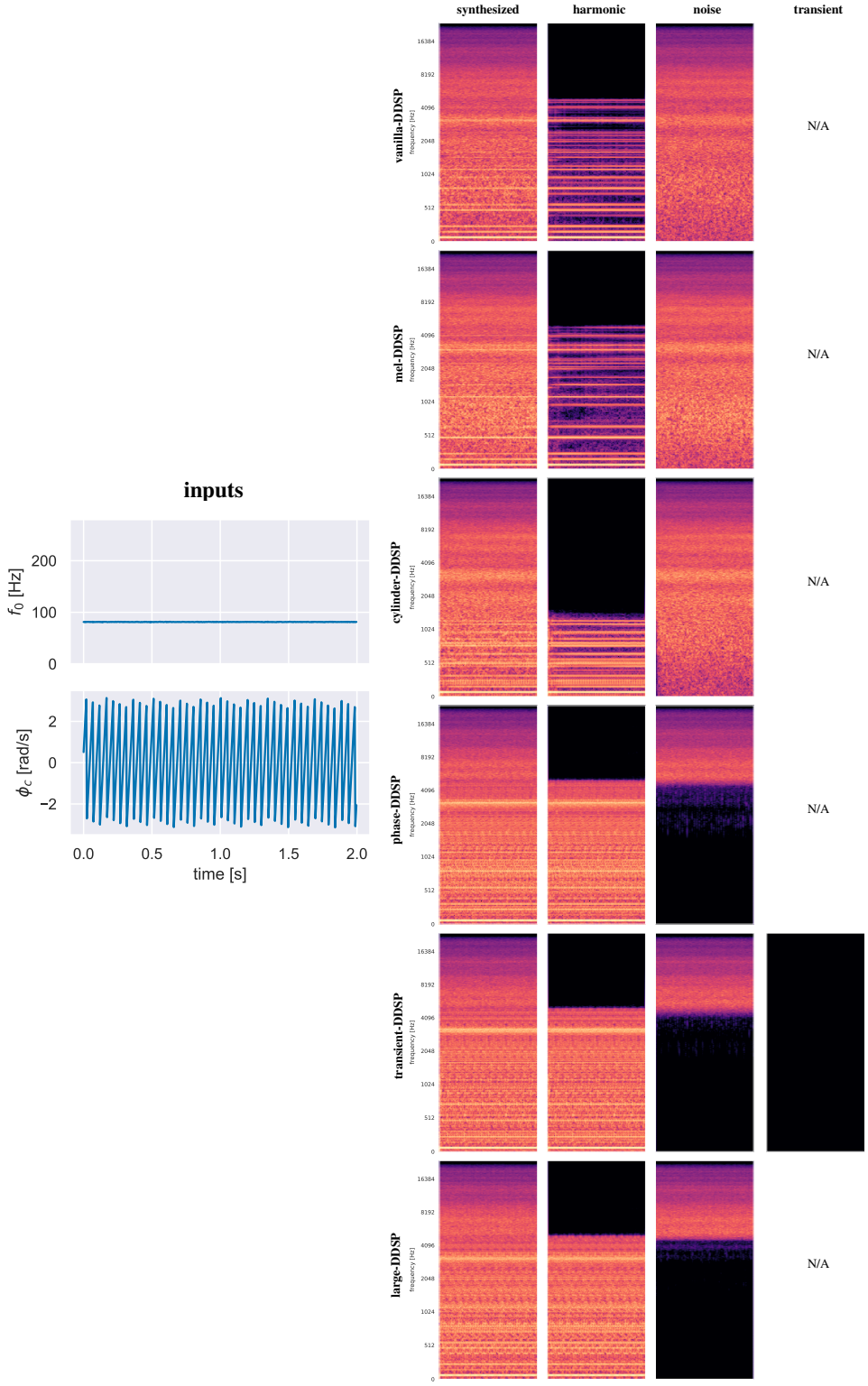

Figure 4.17: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with constant f_0 at 81 Hz (2430 RPM). The colors cover a 75 dB range for all spectrograms.

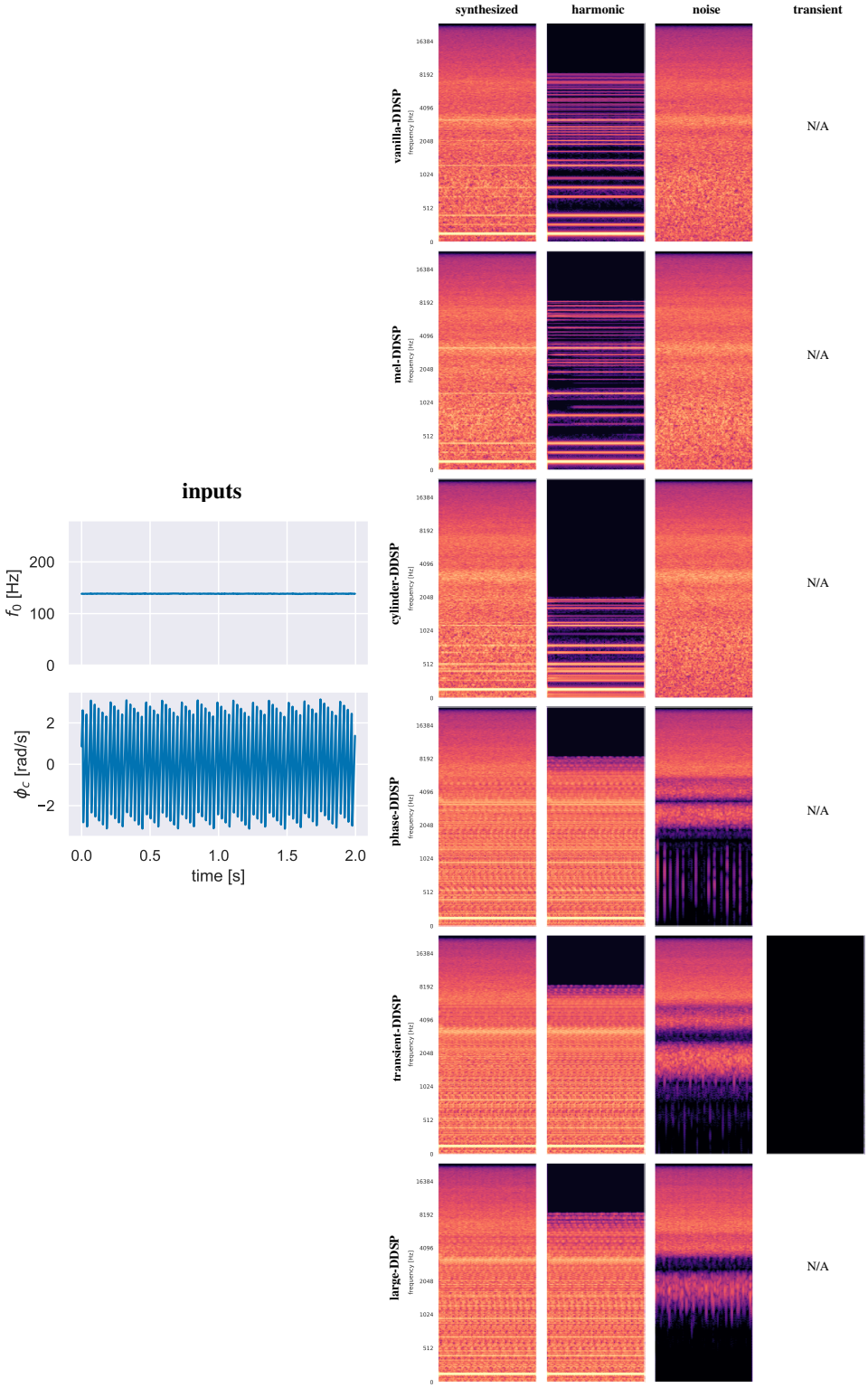

Figure 4.18: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with constant f_0 at 138 Hz (4140 RPM). The colors cover a 75 dB range for all spectrograms.

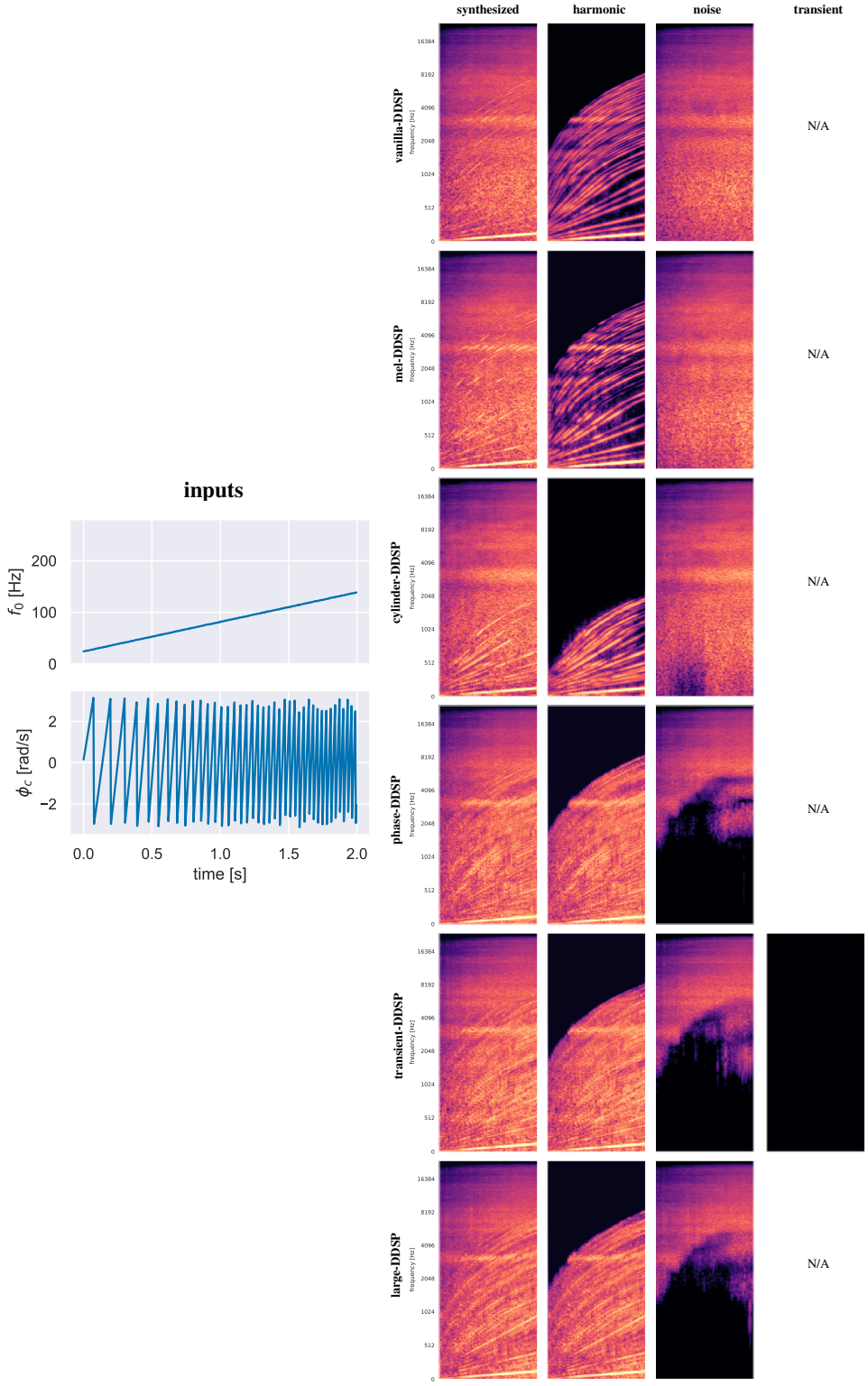

Figure 4.19: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with f_0 increasing linearly between 24–138 Hz (720–4140 RPM). The colors cover a 75 dB range for all spectrograms.

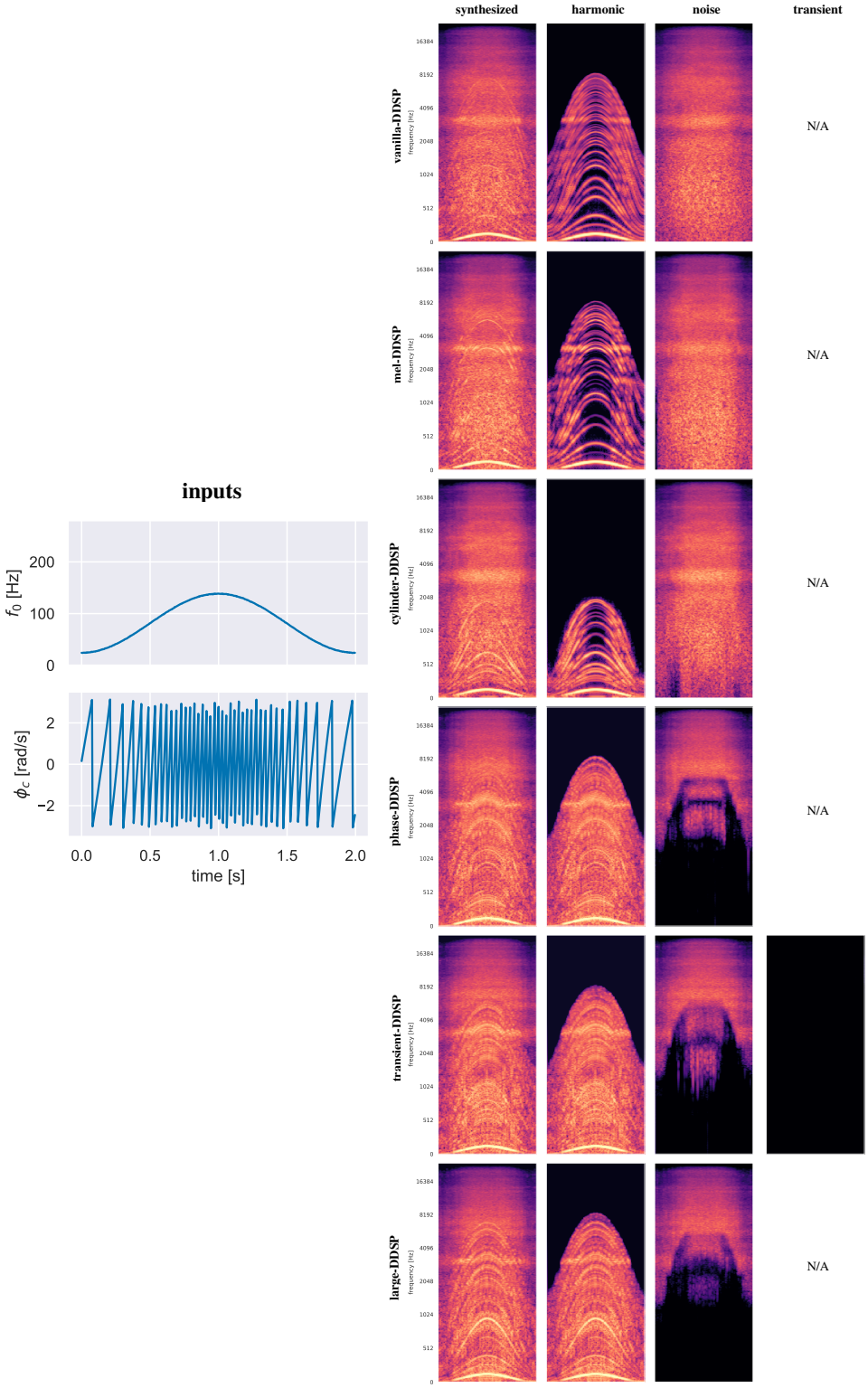

Figure 4.20: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with f_0 being one period of a sine with mean 81 Hz (2430 RPM) and amplitude 57 Hz (1710 RPM). The colors cover a 75 dB range for all spectrograms.

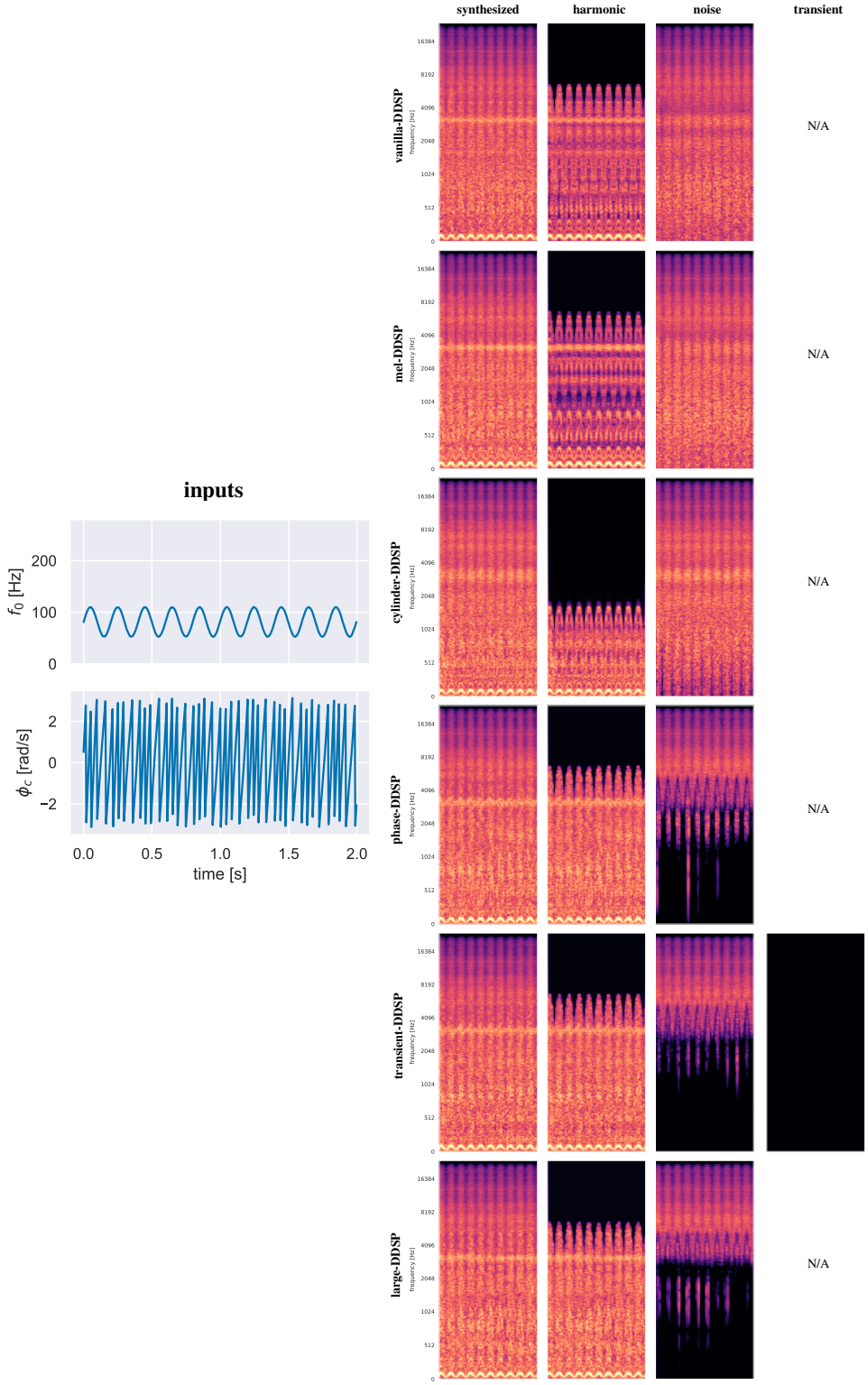

Figure 4.21: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with f_0 being ten periods of a sine with mean 81 Hz (2430 RPM) and amplitude 29 Hz (870 RPM). The colors cover a 75 dB range for all spectrograms.

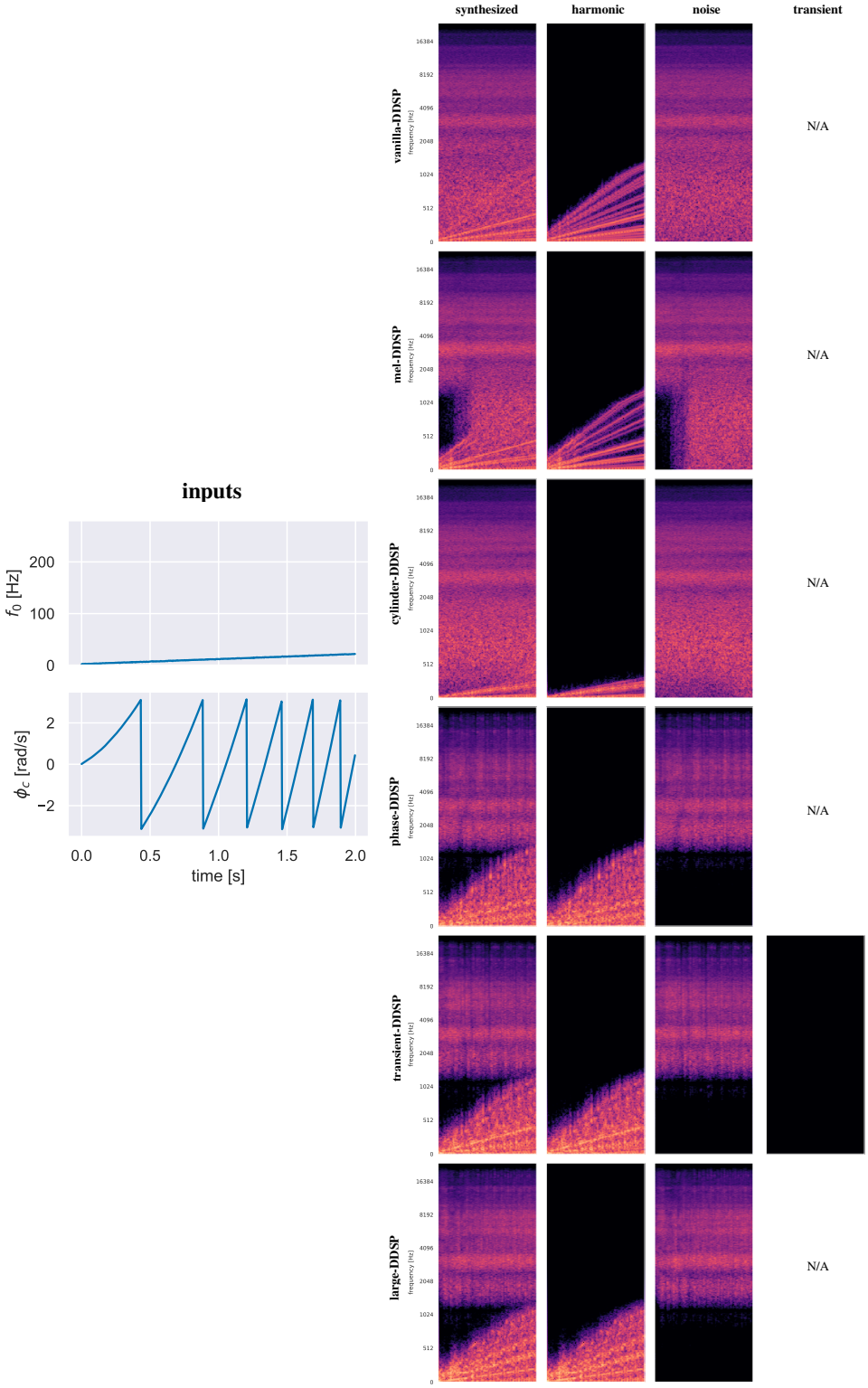

Figure 4.22: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with f_0 being below those in the training data, going linearly between 2–22 Hz (60–660 RPM). The colors cover a 75 dB range for all spectrograms.

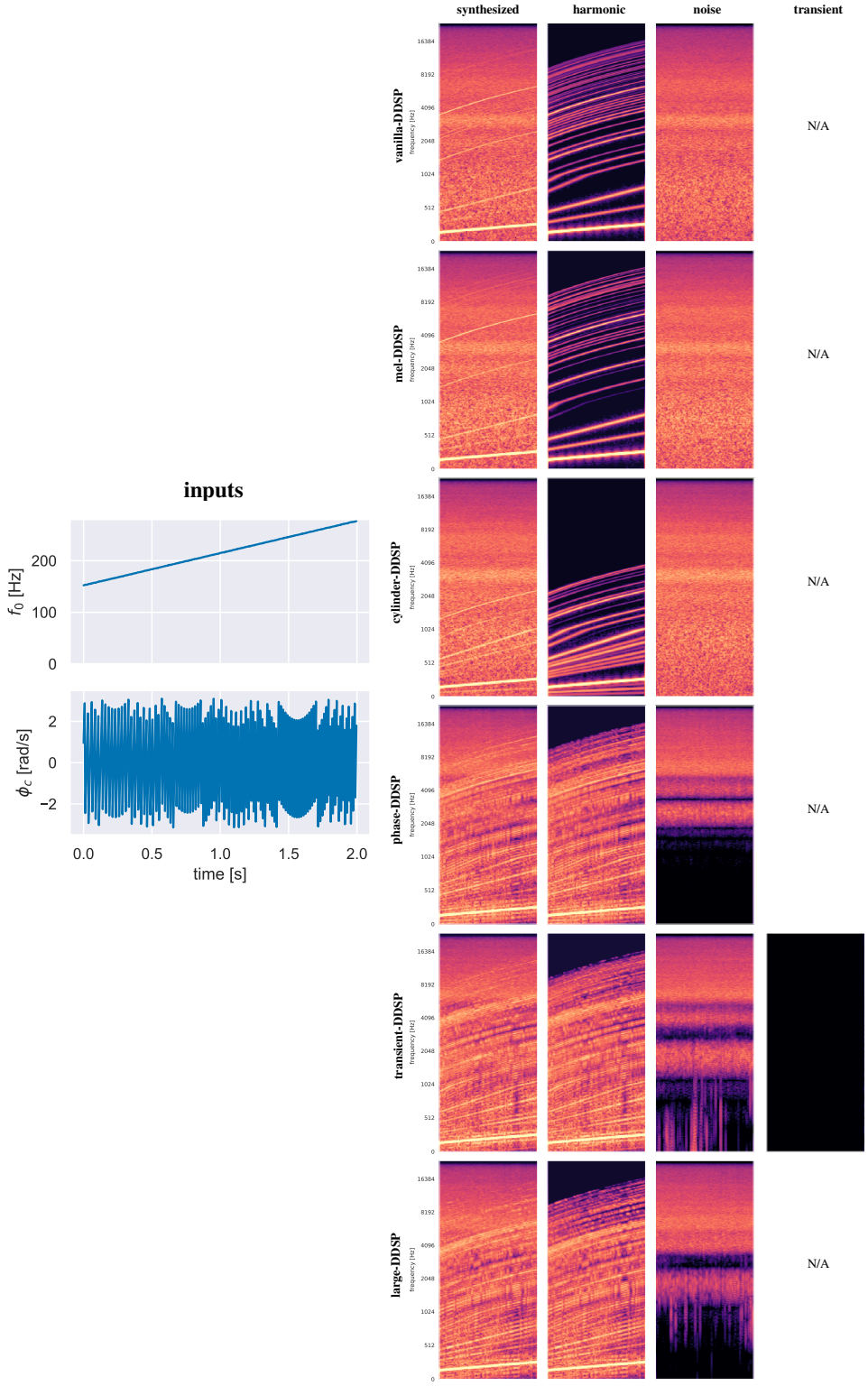

Figure 4.23: Mel spectrograms of synthesized engine sounds and time-domain plots of inputs with f_0 being above those in the training data, going linearly between 152–277 Hz (4560–8310 RPM). The colors cover a 75 dB range for all spectrograms.

making them sound less like realistic car engines for this high-range input.

The artifacts in the audio produced from using the high-range f_0 input in Figure 4.18 with phase-DDSP, transient-DDSP and large-DDSP remain present for all inputs with high-range f_0 values. In contrast, vanilla-DDSP, mel-DDSP and cylinder-DDSP generate engine-like sounds from high-range f_0, even with the input in Figure 4.23 with f_0 above the range of the training examples.

Chapter 5

Discussion

In this chapter, we discuss our results and possible directions for future work. First, we discuss the performance of our models based on our quantitative results and from listening to the synthesized audio. We continue by discussing how the choice of model inputs, sound synthesis method and loss function affects the resulting audio. After discussing how the framework could be used for different sound sources, we consider possible applications and the relevance of our work. We conclude with a note on sustainability and ethics.

5.1 Audio Synthesis Performance

Based solely on the multi-scale mel loss, phase-DDSP and transient-DDSP perform reconstruction of the test examples best. Based on the CQT loss, cylinder-DDSP performs better than the other models. However, since these metrics have imperfections—like all objective metrics for perceived audio quality—results from them regarding how the models compare in performance will always remain to some extent inconclusive. Furthermore, these metrics are very specific to how our models are built and generalizing from them is questionable.

When listening to the reconstructed test examples from section 4.4, our impression is that all models get the general characteristics of the original engine sound right: How the energy is distributed across frequencies over time sounds similar, and the spectrograms of the original and reconstructed audio look similar. However, none of the models are able to generate audio that truly sounds like it came from the recorded engine and without any undesirable artifacts. How these artifacts depend on model inputs and components in the synthesizer are further discussed in section 5.2 and section 5.3.

We get the above mentioned observations further verified when listening to the results from control by synthetic input presented in section 4.5. Although some models synthesize audio that sounds engine-like for some inputs, all of them also produce undesired artifacts for other inputs.

It should be mentioned that our original plan with the experiment in section 4.5 was to create a simple model of the dynamical system that takes a sequence of gas pedal positions as input and outputs a resulting engine RPM sequence. With such a model, we could have experimented with more realistic f_0 inputs. However, due to time constraints we settled for using synthetic inputs that do not necessarily correspond to RPM sequences realistic for a car engine. As a consequence, the resulting audio may sound dissimilar to an engine mostly because the inputs are not realistic in the first place.

5.2 Model Inputs

Comparing the models that only use f_0 as input (vanilla-DDSP, mel-DDSP and cylinder-DDSP) against those that use both f_0 and ϕ_c (phase-DDSP, large-DDSP and transient-DDSP), our impression is that the models using only f_0 are more robust to fast variations in f_0 and f_0 values outside the training domain. We mostly base this on listening to the audio presented in Figure 4.15 under section 4.4 and in Figures 4.20, 4.21 and 4.23 under section 4.5.

On the other hand, our impression is that the models using both f_0 and ϕ_c produce a lot more convincing engine sounds for low to mid-range f_0 inputs. For such inputs, these models are able to generate what sounds like the distinct pulses coming from individual engine cycles, while audio from the other models sounds more like continuous noise. We mostly base this on listening to the audio presented in Figures 4.13 and 4.14 under section 4.4 and in Figure 4.16 under section 4.5. Hence it seems like adding the phase of the engine cycles ϕ_c improves reconstruction of the periodic pulses from the explosion cycles for low to mid-range engine speeds (low to mid-range f_0 inputs). However, when listening to the reconstructed audio in Figure 4.15 and the generated audio from synthetic inputs in Figures 4.20, 4.21 and 4.23, it appears that using ϕ_c as input introduces undesirable artifacts for high-range engine speeds.

Looking at the time-domain plots of ϕ_c for the inputs with high-range f_0 in section 4.5, there are some visible aliasing effects. This aliasing is especially prominent for the synthetic input with f_0 above the training range in Figure 4.23. This may contribute to why the audio generated by models with ϕ_c as input contains undesirable artifacts at high-range f_0 inputs. Increasing the 250 Hz

frame rate may reduce these artifacts, but would come at the cost of increased inference time.

5.3 Synthesizers

The idea of the experiment in section 4.3 was to train the models to reproduce a small set of audio examples partly to see how well the sound synthesis methods possibly can perform when disregarding any expectations on the decoder network's generalization capabilities. The reconstructed training examples by phase-DDSP probably show how well the harmonic-plus-noise synthesis method of the DDSP Autoencoder can perform on our engine sounds. They sound similar to the originals, only lacking a bit of sharpness. Adding the transient component to the transient-DDSP model adds the sharpness to the reconstructed audio that phase-DDSP is lacking, but introduces some graininess.

The number of transients we use in the transient synthesizer, 80 per second, is admittedly rather low. We chose this value to be able to use the same batch size as for the other models during training with the full dataset. We did a small experiment after those presented in this report and managed to train a model to reconstruct a training example with almost inaudible difference from the original only by increasing to 100 transients per second. More experiments would have to be performed and documented, but increasing the number of transients might reduce the grainy clicks, although at the cost of increased time and memory cost.

Interestingly, if you listen separately to the audio produced by the harmonic and transient components of transient-DDSP in section 4.3 you will hear metallic-sounding artifacts—especially in the high-range f_0 example—that are not present in the full reconstruction. This implies that the model learned how to use interference between the harmonics and transients to generate the final audio.

Although the reconstructed audio from the transient-DDSP model in section 4.3 sounds more like the original, transient-DDSP learns to not use the transient component when trained on the full 32 min dataset in section 4.4. We cannot give any obvious reason to why, but one explanation could be that many of the transients in car engine sounds occur to some extent randomly. If this were true, then the model would find it difficult to correctly place the transients in time when trying to reconstruct examples with similar inputs but with transients at random times. If the times of the transients on average happen within a given time interval, the model would on average obtain lower losses by using the noise model instead which smears out the sharp peaks. Our best

guess is that this is what is happening.

To be able to use the transient component better when synthesizing engine sounds, given the assumption that the transients in engine sounds occur at stochastic times, we cannot rely on a deterministic model such as the DDSP Autoencoder. It would be interesting to look at methods using variational inference, such as variational autoencoders (VAEs), or adversarial losses, which is used in generative adversarial networks (GANs). Models using such methods learn to generate new data with the same statistics as the training data, without being incentivized to generate examples identical to training examples. In such a framework, reconstruction performance would be meaningless since each generated example for a given input would be different, and we would have to rely entirely on evaluating performance by listening. Although such methods could improve perceived audio quality, they are likely more difficult to train and may require bigger networks with increased time and memory complexity.

The sound synthesis method of the DDSP Autoencoder based on the harmonic-plus-noise model seems adequate for synthesizing car engine sounds. Although the generated audio lacks some of the sharp transients of our engine recordings, they capture the essential aspects of the sound.

All models that use ϕ_c as input learned to generate harmonic and noise sounds in disjoint frequency bands. The harmonic synthesizer became responsible for the low frequencies while the filtered noise synthesizer dealt with high frequencies. We see this as an illustration of the expressivity of additive synthesis. Additive synthesis was used in analysis/synthesis applications before Serra and Smith [9] added the noise component in their spectral modeling synthesis (SMS) method. We expect that using only a harmonic synthesizer would be enough to generate relatively realistic engine sounds when using enough partials. However, it is also more computationally expensive than the noise model, which together with the fact that people do not perceive high frequencies in as high resolution as low frequencies makes the noise model suitable for generating the high-frequency content of the audio in applications where computational resources are limited.

Counter to what we expected, using integer multiples of f_0/N_c instead of f_0 in the harmonic synthesizer to enrich the spectrum did not make any noteworthy improvement. If anything, there is a small degradation of the synthesized sound when using the same number of harmonics due to the narrower range of the resulting harmonic frequency distribution. It could be that the specific four-cylinder diesel engine we recorded has cylinders that produce very similar and consistent sounds so that the most prominent harmonics are indeed integer multiples of f_0. This does not rule out the possibility that a denser distribution

from using integer multiples of f_0/N_c would work better for some other engines.

5.4 Evaluation Metrics and Loss Functions

As discussed in section 3.4, there are no perfect objective metrics for perceived audio quality. This, along with the fact that our evaluation metrics are closely related to how the models are trained, should be kept in mind when inspecting the losses for the reconstructed examples by each model in Tables 4.2 and 4.3. Algorithms for objective measurement of perceived audio quality have been suggested, such as Perceptual Evaluation of Audio Quality (PEAQ) [32]. Although our metrics also use time-frequency analysis with perceptually motivated frequency scales, using PEAQ or a similar method could give an automated evaluation with results more similar to a formal listening test.

Looking at the multi-scale mel losses during training in Figure 4.9, it appears as if the loss reaches a sweet spot for the validation data at around 800 epochs for each model. However, when comparing the audio generated by the models after 800 to audio generated after 5000 epochs, we would say that the latter sounds more similar to the target. We think this illustrates a weakness in these multi-scale spectral losses when applied to sounds containing transients. If the aim is to generate audio that is *perceptually similar* to the original sound, we do not want a loss function that favors generated audio that is *identical* to the training examples. The multi-scale spectral losses do this by using time-frequency representations (magnitude spectrograms) that disregard the perceptually irrelevant phase information. However, for many sounds rich in transients such as our engine sounds, the transients could have slightly different energies and timing and the listener would still perceive the sound as coming from the same sound source. This way, even though the models seem to overfit to the training data by the look of the validation losses in Figure 4.9, the generated audio sounds better because the model learns how to place the transients consistently with the training examples—even though it is inconsistent with the validation examples.

An alternative to using multi-scale spectral losses, as discussed earlier, could be to use adversarial losses as in GANs. Such models would learn how to synthesize audio with the same statistics as the training examples without necessarily being identical or having identical spectrograms.

5.5 Fine-Tuning

It should be mentioned that we put no effort into fine-tuning certain aspects of our methods. It is possible that our results could be improved by performing large-scale searches for optimal parameter values, but this would be a very time-consuming effort.

Our choice to increase the sample rate from the DDSP authors' 16 kHz to 48 kHz probably affects how much training data is needed and the optimal hyperparameters for the training procedure. The DDSP authors say they did a large-scale hyperparameter search, but we simply used their values directly in our experiments even though we use a different sound source and a different sample rate. For their solo-violin model, the DDSP authors use around 14 min of training material. We use around 22 min of engine audio for training. It is difficult to know what amount is optimal, and how the distributions of the training data—such as that of the f_0 values—affect the performance of the final models.

Another aspect we do not fine-tune is the filtered noise component in the harmonic-plus-noise synthesizer. We use the same settings as the DDSP authors, but they mention that the window function applied to the impulse response during synthesis makes a difference. The shape and size of the window function can be used to control the time-frequency resolution trade-off. It is possible that a different shape and size giving higher time resolution could make transients in the reconstructed audio sound sharper and less smeared out.

5.6 Other Sound Sources

Our results illustrate that the approach proposed by the DDSP authors—performing neural audio synthesis by training neural networks to decode latent features to get sound synthesis parameters—can be adapted to other sound sources than musical instruments. The process is conceptually very similar to traditional analysis/synthesis methods, where training the neural networks is analogous to the analysis stage and audio generation from features is analogous to the synthesis stage. Due to the generality of this approach, many more sound sources may be possible to model this way.

Our method contains a highly engineered method for associating car engine recordings with f_0 values. To be able to use this framework on a different sound source, you would have to come up with a new method that is tailored to the new target sound or propose a more general method that could work for a variety of

sounds. The DDSP authors experiment with an unsupervised learning scheme where a part of the encoder in the DDSP Autoencoder is trained to extract f_0 from the original audio. Something similar could be useful for any harmonic sound source.

Some sounds may not need a harmonic, noise or transient component to be modeled well, and inputs to the decoder network may not be limited to perceptual features such as f_0. We initially planned on experimenting with friction sounds, such as the sounds from sliding objects on different surfaces or the sound of a pencil against a paper. In these experiments, we would have tracked the speed of the object using a sensor and later used the resulting speeds as input to the decoder network. Since these friction sounds are not harmonic we would have experimented with using only the filtered noise and transient synthesizer components, possibly together with a non-harmonic additive synthesizer. We did not perform these experiments due to the time constraints of this thesis, but they would be interesting to look into in future work.

5.7 Possible Applications and Relevance

A relevant question is whether our models can be used in their current form for procedural engine sounds. By Andy Farnell's definition in Definition 1, procedural audio is created in real time according to live input. Since all models have inference times below 2 s (the length of an example) on our GPUs, as shown in Table 4.1, they do operate in real time. In other words, all models have real-time throughput on our GPUs. On our CPU, all models have real-time throughput except large-DDSP. Furthermore, there is no limitation in the models that rules out the use of live input. Therefore, the models can perform procedural audio.

The models cannot be used in their current form for *live* real-time applications[1]. The models are trained to operate on 2 s audio frames, and applying them directly in a live setting would hence result in a latency of at least 2 s. Tolerable limits for latency in live applications is a matter of debate, but stated limits are usually below 50 ms [33], so 2 s is clearly not tolerable. The models could be adapted to operate on shorter audio frames, maybe even variable frame lengths, but such an adaptation would likely be non-trivial. The models are currently trained on 2 s examples and each generated 2 s frame is treated

[1] We use the following distinction between live and real-time signal processing. In real-time signal processing, the time to process a signal must be shorter than the signal duration. Live signal processing requires both real-time operation and acceptably low latency.

independently. Only shortening the training examples to below 50 ms would not work since the neural network cannot recognize patterns on longer time scales than the duration of individual training examples, and an engine cycle can be longer than 50 ms. Furthermore, since the frames are treated independently, methods more sophisticated than cross-fading would be necessary to have seamless transitions between frames.

Our experiments use 100 % of the available computational resources, while audio processing is usually only allocated a fraction of the resources in applications such as VR and video games. For models similar to ours to be useful in such applications they would have to be more memory and time efficient.

Since the models cannot be used in live applications and do not consistently generate engine sounds without undesirable artifacts, their current form are arguably of little practical use. However, the research behind them is part of an ongoing effort within the fields of procedural audio and neural audio synthesis to find flexible methods for synthesis of target sounds, and to get a better understanding of how deep learning can be used in audio applications. Anybody interested in research into procedural audio and neural audio synthesis, or deep learning in general, may therefore find the methods and results of this thesis relevant.

5.8 Notes on Sustainability and Ethics

Sustainability and ethics are important topics that every research project should reflect upon. However, since this thesis involves no live subjects and its results will have no direct implications for ecological, economical or social sustainability, they are topics of no direct relevance to this work.

Looking more broadly, this work is part of a body of research in data-driven sound synthesis. While being able to control the synthesis of engine sounds does not create obvious ethical problems, being able to synthesize, for instance, somebody's voice would be different. Speech synthesis methods capable of realistic voice cloning give rise to a host of ethical issues since they could be used for unethical applications such as identity theft, spread of misinformation and propaganda.

This work is also part of a body of research into artificial intelligence. While artificial intelligence can be used for benevolent causes to increase quality and productivity of business as well as health and wellbeing of people, it could become a double edged sword if data is collected unethically or if all resources end up in the hands of a few, leading to social and economical inequity.

In this thesis, we use our own recordings of engine sounds, but other data-driven methods within artificial intelligence may need personal data to create models, such as images of faces for facial recognition or recordings of voices for speech synthesis. With many people publishing personal data on their social media accounts, perhaps ignorant of what terms and conditions they have agreed to, the ownership and use of personal data is still a legal and ethical grey area.

The above are all examples of crucial issues that must be considered when conducting research within these topics.

Chapter 6

Conclusions & Future Work

In this final chapter, we first summarize our most relevant conclusions. We end by proposing directions for future work.

6.1 Conclusions

This thesis investigated how data-driven procedural audio can be performed by looking at synthesis of engine sounds using neural audio synthesis. We specifically investigated how differentiable digital signal processing (DDSP) [8] can be applied to generate engine sounds from a sequence of fundamental frequencies corresponding to the engine speed in RPM.

We proposed a method for recording audio and engine speed simultaneously and for synchronizing them, as well as a method for estimating the phase of the engine cycles based on the recorded audio and engine speed. We also proposed a set of models based on the DDSP Autoencoder with adaptations for engine sounds. Our synthesized audio illustrates that the DDSP Autoencoder, originally designed for synthesis of monophonic musical instrument sounds, can be adapted to synthesize other essentially harmonic sounds such as those of car engines.

For the models to be able to reproduce the periodic chugging and clicking sounds from the engine cycles, we find it helpful to use our estimated engine cycle phase together with the fundamental frequency as model input. Although this helps capturing the sound of the engine cycles for low to mid-range engine speeds, it introduces undesired artifacts for high-range engine speeds and fast engine speed variations.

Our models using only the fundamental frequency as input are more robust to high-range engine speeds and fast engine speed variations. However, these

models do not capture the periodic sounds from the engine cycles at low to mid-range engine speeds, but instead sound smeared out like continuous noise.

Since the harmonic-plus-noise synthesizer in the DDSP Autoencoder has its limitations, especially with respect to modeling transients, we introduced a transient component inspired by Verma and Meng [10]. We showed that our harmonic-noise-transient synthesizer can generate audio which has sharper attacks. However, when the model is trained on a larger set of recordings, the model learns to silence the transient component. We argue that this may be because the energies and timing of the transients in engine sounds are stochastic, although further study is needed to verify this. If the transients in the engine sounds are stochastic, then the DDSP Autoencoder may not be able to model them well since it is a deterministic model.

We proposed a multi-scale spectral loss using the mel frequency scale as an alternative to the loss based on the short-time Fourier transform (STFT) used by the DDSP authors. When inspecting the loss curves for our validation data during training, we noticed that the validation loss reaches a minimum and then increases despite the generated audio becoming increasingly similar to the target sound. We argue that this illustrates a weakness in the multi-scale spectral losses when used on sounds rich in transients such as those from engines.

The resulting inference times of our models show that most of them have real-time throughput, even on a CPU. This allows the possibility to adapt them for live real-time applications, although such adaptations would likely be non-trivial.

6.2 Future Work

To further increase the performance of our models on engine sounds, we think it would be useful to investigate whether the energy and timing of the transients in the engine sounds are stochastic. If they are, then adapting our deterministic models to use adversarial losses, as in generative adversarial networks (GANs), could result in synthesized sound perceptually more similar to the target sound, but may be more difficult to train and less memory efficient.

For our models to be useful in live real-time applications, such as virtual reality and video games, they would have to be adapted to operate on shorter audio frames. This would likely require non-trivial modifications, and we would like to see more work into how such adaptations could be made.

To generalize our method for fundamental frequency estimation, which is engineered specifically for engine sounds, we would like to see work into automated methods. These methods could take the form unsupervised learning

by using an encoder network to estimate the fundamental frequency, possibly together with variational inference.

By performing adaptations specifically for synthesis of engine sounds, we show that methods based on the DDSP Autoencoder can be used to synthesize other sounds than monophonic musical instruments. If time permitted, our plan was to experiment with friction sounds, such as those from sliding objects on different surfaces and pencils against paper. Since we did not find time for this during this thesis work, we would like to see more work into how these methods can be used for friction sounds—and other sounds as well.

Bibliography

[1] Andy Farnell. "An Introduction to Procedural Audio and Its Application in Computer Games". In: *Audio Mostly Conference*. 2007, pp. 1–31.

[2] The Moving Picture Experts Group (MPEG). *MPEG-I ISO/IEC 23090 Coded Representation of Immersive Media*. URL: https://mpeg.chiariglione.org/standards/mpeg-i.

[3] Niels Böttcher. "Current problems and future possibilities of procedural audio in computer games". In: *Journal of Gaming & Virtual Worlds* 5.3 (Sept. 2013), pp. 215–234.

[4] Christian Floisand. *An Interview with Nicolas Fournel*. Creating Sound: Sound Design Resources. Mar. 2013. URL: http://creatingsound.com/2013/03/an-interview-with-nicolas-fournel.

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

[6] Yuxuan Wang et al. "Tacotron: A Fully End-to-End Text-To-Speech Synthesis Model". In: *CoRR* abs/1703.10135 (2017). arXiv: 1703.10135. URL: http://arxiv.org/abs/1703.10135.

[7] Aäron van den Oord et al. "Parallel WaveNet: Fast High-Fidelity Speech Synthesis". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018.

[8] Jesse Engel et al. "DDSP: Differentiable Digital Signal Processing". In: *International Conference on Learning Representations (ICLR)*. 2020.

[9] Xavier Serra and Julius Smith. "Spectral Modeling Synthesis: A Sound Analysis/Synthesis System Based on a Deterministic plus Stochastic Decomposition". In: *Computer Music Journal* 14.4 (Winter 1990), pp. 12–24.

[10] Tony S. Verma and Teresa H. Y. Meng. "An analysis/synthesis tool for transient signals that allows a flexible sines+transients+noise model for audio". In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*. Vol. 6. 1998, pp. 3573–3576.

[11] William Moss et al. "Sounding Liquids: Automatic Sound Synthesis from Fluid Simulation". In: *ACM Transactions on Graphics* 29.3 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1805964.1805965. URL: https://doi-org.focus.lib.kth.se/10.1145/1805964.1805965.

[12] Andy Farnell. *Designing Sound*. MIT Press, 2010.

[13] Diemo Schwarz. "Corpus-Based Concatenative Synthesis". In: *IEEE Signal Processing Magazine* 24.2 (Mar. 2007), pp. 92–104.

[14] Xavi Gonzalvo et al. "Recent Advances in Google Real-time HMM-driven Unit Selection Synthesizer". In: *20th Annual Conference of the International Speech Communication Association (INTERSPEECH)*. 2016.

[15] Jan Jagla, Julien Maillard, and Nadine Martin. "Sample-based engine noise synthesis using a harmonic synchronous overlap-and-add method". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012, pp. 373–376.

[16] James W. Beauchamp. *Analysis, Synthesis, and Perception of Musical Sounds*. Springer, 2007.

[17] Heiko Purnhagen and Nikolaus Meine. "HILN-the MPEG-4 parametric audio coding tools". In: *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 3. 2000, pp. 201–204.

[18] Alexander Galembo et al. "Effects of relative phases on pitch and timbre in the piano bass range". In: *The Journal of the Acoustical Society of America* 110.3 (2001). URL: https://doi.org/10.1121/1.1391246.

[19] Sungwon Kim et al. "FloWaveNet: A Generative Flow for Raw Audio". In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. PMLR 97. 2019, pp. 3370–3378.

[20] Wei Ping, Kainan Peng, and Jitong Chen. *ClariNet: Parallel Wave Generation in End-to-End Text-to-Speech*. arXiv preprint: https://arxiv.org/abs/1807.07281. 2018.

[21] Ryan Prenger, Rafael Valle, and Bryan Catanzaro. "WaveGlow: A Flow-based Generative Network for Speech Synthesis". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 3617–3621.

[22] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

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

[24] Jong Wook Kim et al. *CREPE: A Convolutional Representation for Pitch Estimation*. 2018. arXiv: 1802.06182 [eess.AS]. URL: https://arxiv.org/abs/1802.06182.

[25] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[26] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference for Learning Representations (ICLR)*. 2015. arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

[27] Jacek Urbanek, Tomasz Barszcz, and Jerome Antoni. "A two-step procedure for estimation of instantaneous rotational speed with large fluctuations". In: *Mechanical Systems and Signal Processing* 38.1 (July 2013), pp. 96–102. DOI: 10.1016/j.ymssp.2012.05.009. URL: https://doi.org/10.1016/j.ymssp.2012.05.009.

[28] The European Parliament and the Council of the European Union. *Directive 98/69/EC of the European Parliament and of the Council*. 1998. URL: https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31998L0069:EN:HTML.

[29] Society of Automotive Engineers. *SAE J1962: Diagnostic Connector Equivalent to ISO/DIS*. 1995. URL: https://law.resource.org/pub/us/cfr/ibr/005/sae.j1962.1995.pdf.

[30] Brendan Whitfield. *python-OBD: a library for handling data from a car's On-Board Diagnostics port (OBD-II)*. URL: https://python-obd.readthedocs.io/en/latest/.

[31] S. S. Stevens, J. Volkmann, and E. B. Newman. "A Scale for the Measurement of the Psychological Magnitude Pitch". In: *The Journal of the Acoustical Society of America* 8.3 (1937).

[32] International Telecommunication Union (ITU). *Method for objective measurements of perceived audio quality*. Recommendation ITU-R BS.1387-1. 2015. URL: https://www.itu.int/rec/R-REC-BS.1387-1-200111-I/en.

[33] Michael Lester and Jon Boley. "The Effects of Latency on Live Sound Monitoring". In: *Audio Engineering Society Convention 123*. Oct. 2007. URL: http://www.aes.org/e-lib/browse.cfm?elib=14256.

Appendix A

Model Details

A.1 Decoder Networks

The decoder networks of all our models have the same structure as that of the DDSP Autoencoder proposed by the DDSP authors. A diagram of their structure is shown in Figure A.1, and a diagram of their multi-layer perceptron (MLP) components is shown in Figure A.2.

In our models, the decoder network have only f_0, or f_0 and ϕ_c, as inputs instead of f_0, z and loudness. The dimensionality of the inputs corresponds to 500 time steps. The outputs of the decoder network are the parameters required by the synthesizers. In our setup, these parameters are the amplitudes a_p for each partial p in the harmonic synthesizer, and the transfer function H of the FIR filter in the noise synthesizer. For transient-DDSP, among the outputs are also the amplitudes $\hat{a}_k$ for each DCT frequency k in the additive synthesizer of the transient component.

The decoder network, as seen in Figure A.1, has a shared-bottom structure

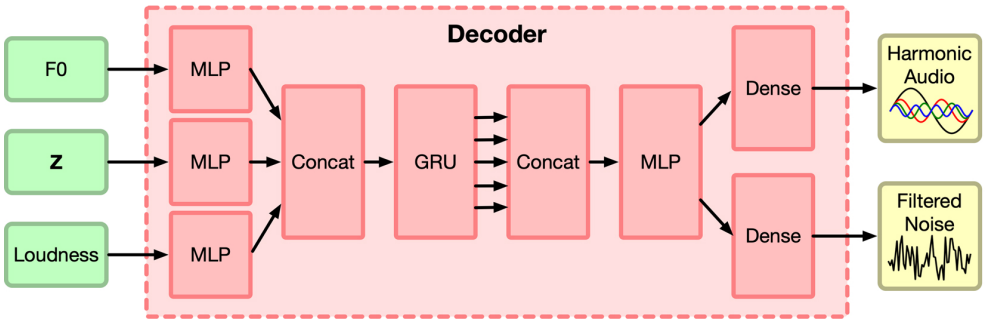

Figure A.1: Structure of the decoder network in the DDSP Autoencoder from [8].

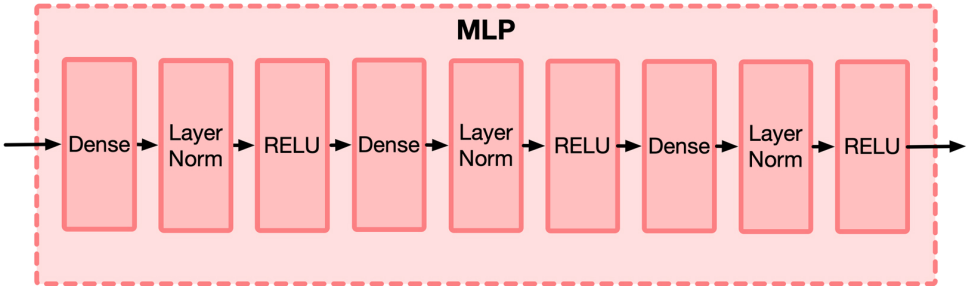

Figure A.2: Structure of the MLP components of the decoder network in the DDSP Autoencoder from [8].

that computes a shared embedding from the inputs—f_0 and possibly ϕ_c—and have one head for each output—a_p, H and possibly $\hat{a}_k$—corresponding to a synthesizer component.

The outputs of the MLPs are concatenated and passed through a gated recurrent unit (GRU) with 512-dimensional hidden state (2048-dimensional for large-DDSP). The GRU outputs are concatenated with the outputs of the f_0 and ϕ_c MLPs in the channel dimension. The concatenated output is passed through a final MLP followed by an individual dense layer for each synthesizer component to get the outputs.

The MLP architecture, as seen in Figure A.2, consists of three dense layers in sequence, with a layer normalization and a rectified linear unit (ReLU) activation function at the end of each layer.

A.2 Derivation of the Number of Mel Bins

We want an expression for the number n_{mel} of bins to divide our mel frequency band into. Intuitively, we have

$$n_{\text{mel}} \Delta m = m_{\text{max}} - m_{\text{min}} \tag{A.1}$$

where m_{max} and m_{min} are the upper and lower limits, respectively, of our frequency band in mels, and Δm is the width of each bin in mels. Since n_{mel} must be an integer, we let

$$n_{\text{mel}} = \left\lfloor \frac{m_{\text{max}} - m_{\text{min}}}{\Delta m} \right\rfloor . \tag{A.2}$$

Since our frequency band goes from 0 to the Nyquist frequency f_N, we have $m_{\text{min}} = 0$ and $m_{\text{max}} = m_N$, where m_N is the Nyquist frequency expressed in

mels. This gives

$$n_{\text{mel}} = \left\lfloor \frac{m_N}{\Delta m} \right\rfloor . \tag{A.3}$$

Now, all we need is an expression for Δm in terms of known variables. Treating Δm as a differential, we get

$$\Delta m = \frac{\mathrm{d}m}{\mathrm{d}f} \Delta f \tag{A.4}$$

$$= \frac{\mathrm{d}}{\mathrm{d}f} \left(1127 \log \left(1 + \frac{f}{700} \right) \right) \Delta f \tag{A.5}$$

$$= \frac{1127}{700 + f} \Delta f \tag{A.6}$$

where f is the frequency in Hz and $\mathrm{d}m/\mathrm{d}f$ is the derivative of the definition of mel frequency in (3.10) treated as a function $m(f)$. We need the highest resolution at the lowest frequencies, and evaluating the above expression at the lower limit $f = 0$ of our frequency band gives

$$\Delta m = \frac{161}{100} \Delta f. \tag{A.7}$$

The time resolution of a spectrogram is given by

$$\Delta t = \frac{n_{\text{FFT}}}{f_s} \tag{A.8}$$

where n_{FFT} is the FFT size—the number of samples in a window—and f_s is the sample rate in Hz. From the uncertainty principle of signal processing, we have the inequality[1]

$$\Delta t \Delta f \geq 1. \tag{A.9}$$

Assuming equality, solving for Δt and inserting into (A.8) gives

$$\Delta f = \frac{f_s}{n_{\text{FFT}}} \tag{A.10}$$

which inserted into (A.7) gives

$$\Delta m = \frac{161}{100} \frac{f_s}{n_{\text{FFT}}}. \tag{A.11}$$

The above inserted into (A.3) yields our final expression

$$n_{\text{mel}} = \left\lfloor \frac{100}{161} \frac{m_N n_{\text{FFT}}}{f_s} \right\rfloor \tag{A.12}$$

which is the one presented in (3.11).

[1] It is possible to derive the tighter bound $\Delta t \Delta f \geq 1/(4\pi)$, but we are only interested in the correct order of magnitude here.