The Synthesizer Programming Problem: Improving the Usability of Sound Synthesizers

by

Jordie Shier
B.Sc., University of Victoria, 2017

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in Interdisciplinary Studies

We acknowledge with respect the Lekwungen peoples on whose traditional territory the university stands and the Songhees, Esquimalt, and W̱SÁNEĆ peoples whose historical relationships with the land continue to this day.

The Synthesizer Programming Problem: Improving the Usability of Sound Synthesizers

by

Jordie Shier
B.Sc., University of Victoria, 2017

Supervisory Committee

Dr. George Tzanetakis, Co-Supervisor
(Department of Computer Science)

Prof. Kirk McNally, Co-Supervisor
(School of Music)

Abstract

The sound synthesizer is an electronic musical instrument that has become commonplace in audio production for music, film, television and video games. Despite its widespread use, creating new sounds on a synthesizer – referred to as synthesizer programming – is a complex task that can impede the creative process. The primary aim of this thesis is to support the development of techniques to assist synthesizer users to more easily achieve their creative goals. One of the main focuses is the development and evaluation of algorithms for inverse synthesis, a technique that involves the prediction of synthesizer parameters to match a target sound. Deep learning and evolutionary programming techniques are compared on a baseline FM synthesis problem and a novel hybrid approach is presented that produces high quality results in less than half the computation time of a state-of-the-art genetic algorithm. Another focus is the development of intuitive user interfaces that encourage novice users to engage with synthesizers and learn the relationship between synthesizer parameters and the associated auditory result. To this end, a novel interface (Synth Explorer) is introduced that uses a visual representation of synthesizer sounds on a two-dimensional layout. An additional focus of this thesis is to support further research in automatic synthesizer programming. An open-source library (SpiegeLib) has been developed to support reproducibility, sharing, and evaluation of techniques for inverse synthesis. Additionally, a large-scale dataset of one billion sounds paired with synthesizer parameters (synth1B1) and a GPU-enabled modular synthesizer (torchsynth) are also introduced to support further exploration of the complex relationship between synthesizer parameters and auditory results.

Contents

List of Tables

List of Figures

Related Publications

A portion of the work presented as a part of this thesis has been published and/or presented at international conferences and venues.

Chapter 4

Chapter 4 is based on a conference paper presented at the Audio Engineering Society (AES) Virtual Vienna 2020 conference. This paper was entitled "SpiegeLib: An automatic synthesizer programming library" and was co-authored by the author's supervisors George Tzanetakis and Kirk McNally, who provided feedback during the development of the software library and helped with editing of the resulting paper [124]. Chapter 4 presents the details of the open-source software library (SpiegeLib) that was the main contribution of that work.

Chapter 5

Chapter 5 builds on the example experiment that was included as a component of "SpiegeLib: An automatic synthesizer programming library". This experiment has since been extended, the details of which are documented in chapter 5. The author presented an in-progress version of this experiment that focused on convolutional neural networks for inverse synthesis at the 2020 AES Virtual Symposium: Applications of Machine Learning in Audio.

Chapter 6

Chapter 6 is a version of a paper entitled "One Billion Audio Sounds from GPU-enabled Modular Synthesis" that was published in the Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx20in21) [138]. It should be noted Dr. Joesph Turian was the first author of this paper and the author of this thesis was

the second author (noted as contributing equally to Joseph Turian). The other co-authors of this paper were George Tzanetakis, Kirk McNally, and Max Henry. Joseph Turian held the overall vision for the research project, contributed to the design and development of synth1B1 and torchsynth, designed and conducted the main research experiments, and led the writing of the paper. The author of this thesis contributed to the design and development of the synth1B1 dataset and the torchsynth synthesizer, helped design and conduct experiments, helped write the paper, and presented the work at the DAFx conference. Max Henry contributed to the design and development of synth1B1 and torchsynth, helped design the experiments, helped write/edit the paper and provided feedback on the presentation. George Tzanetakis and Kirk McNally provided feedback on the experimental design, participated in informal listening experiments, and helped edit the paper.

Chapter 7

Chapter 7 builds on work that the author conducted at the end of his undergraduate degree and completed during the first year of his master's. This work, entitled "Manifold Learning Methods for Visualization and Browsing of Drum Machine Samples" was published in 2021 in the Journal of the Audio Engineering Society (JAES) [123]. It focuses on the development of techniques for visualizing drum samples in two-dimensions using audio feature extraction and manifold learning, and builds off of the author's previously published work on the topic [121, 122]. It was co-authored with Kirk McNally, George Tzanetakis, and Ky Grace Brooks. For a summary of this work please refer to appendix D. Chapter 7 of this thesis draws from this work and uses a similar technique to visualize synthesizer sounds in two-dimensions based on sound similarity.

Acknowledgements

I'd like to thank:

George Tzanetakis for suggesting masters research on synthesizers at UVic and for encouraging me to apply for an NSERC scholarship. I am thoroughly grateful for my time researching under your supervision and appreciate all the guidance I have received from you over the years.

Kirk McNally for encouraging me to apply for a JCURA during my last year of my undergrad and for supporting me to present work at my first international conferences. Those experiences opened me up to wanting to pursue graduate research. Thank you for all the support and guidance you've provided through my masters.

The Natural Sciences and Engineering Research Council of Canada, the British Columbia Provincial government, and the University of Victoria for the financial support provided to conduct this research.

My parents for their endless encouragement and support. Thank you Mom for encouraging me to pursue music and for helping me get Ableton during my first year of undergrad – that is how this whole synthesizer business started. Thank you Dad for all the chats and support, and for helping with my writing during this thesis.

My loving partner Rachel for all the support and encouragement through my masters. Thank you for putting up with me when I was grumpy and for celebrating all the small victories with me along the way. And thank you for all the writing help.

Carson Gant for encouraging me to pursue music and for all the time you spent trying to guide me as I tried program that Flume sound into Massive synth. Much of this work is inspired by our time working on music together.

Joseph Turian, Max Henry, and Grace Brooks for your collaboration and support on the research presented as a part of this thesis.

Colin Malloy and Keon Lee for being great lab partners.

And thank you to everyone else I couldn't list. There are a lot of you!

Chapter 1

Introduction

The sound synthesizer is a familiar tool for many musicians and audio practitioners working in music, film, video games, and other industries related to audio production. Despite its widespread use, the task of programming new sounds into a synthesizer is complex and requires a thorough understanding of sound design principles and technical details. It is not uncommon for a software synthesizer to have thirty or more parameters displayed on a user interface and labelled using technical names specific to the particular device [108]. Synthesizer programming involves manually adjusting these parameters to achieve a desired sound. The task is further complicated by the fact that modifications to these parameters are often not intuitively reflected in the end sonic result. The implication of these constraints is that there is a large learning curve associated with becoming an effective synthesizer programmer and even experienced musicians and audio practitioners can find the task of programming a synthesizer disruptive to their creative process [79]. This is referred to as the "synthesizer programming problem."

The central goal of this thesis is to support the development of methods that address the challenges of synthesizer programming. These methods have the potential to make synthesizers more accessible and enable more individuals to be creative more often. The benefits of engaging in the creative process on a regular basis have been proven [26] and music technology has the power to enable individuals to express themselves in domains that were previously unavailable to them [134]. Developing methods that lower the barrier to entry can allow novices to work with synthesizers in ways that would have previously been unavailable to them. It would also enable them to gain experience more quickly and easily, and remove impediments to their creative process.

Automatic synthesizer programming is the field of research that involves the development of techniques and tools to address the challenges of synthesizer programming. It is inherently interdisciplinary and draws from fields including digital signal processing, creativity support, music interaction, artificial intelligence, and machine learning. Work in automatic synthesizer programming dates back to the 1970s [74]. A large body of research that explores a variety of approaches for improving and automating the task of synthesizer programming has followed. One central focus of automatic synthesizer programming has been *sound matching*, or *inverse synthesis*, which involves predicting synthesizer parameters that will recreate a target sound as closely as possible [64]. A number of other methods have also been proposed such as using descriptive words [118] or vocal imitations [16]. Despite the breadth of work on the topic, the task of automatic synthesizer programming remains a complex problem with many open questions. The continued desires expressed by synthesizer users for improved methods of synthesizer programming points to the need for further work in this field [79].

The main goal of this thesis is to explore and contribute to the algorithmic and user interaction aspects of the synthesizer programming problem. The challenge with synthesizer programming is essentially a human-computer interaction (HCI) problem; the conceptual gap between synthesizer parameters and the associated auditory result is large and users are forced to bridge that gap themselves, which involves a steep learning curve. Novel user interaction paradigms help users communicate ideas to synthesizers in ways that support their creativity, such as using example sounds [64, 153], vocal imitations [16, 155], or descriptive words [118]. Algorithmic techniques involving artificial intelligence and machine learning, including deep learning, have been used to create new user interfaces for synthesizer programming and are also a focus of this thesis. Inverse synthesis, which synthesizer users have identified as helpful [79], has especially benefited from advancements in deep learning in recent years. In this thesis, several techniques for inverse synthesis are evaluated, and a new approach that combines the strengths of evolutionary programming and deep learning is introduced. A novel interface for exploring synthesizer sounds using two-dimensional visualizations is presented in chapter 7, along with a design framework for automatic synthesizer interactions.

A secondary goal of this thesis is to support continued development in the field of automatic synthesizer programming and to encourage collaboration and reproducible research [142]. A related field of study, music information retrieval (MIR), has ben-

efited from open-source software, open datasets, and shared evaluations. However, there are few examples of similar shared resources and collaboration in the field of automatic synthesizer programming. The open-source software and datasets that were developed by the author as a part of this thesis are inspired by the contribution of similar resources to the field of MIR. Chapter 4 presents an open-source library – named *spiegelib* after American electronic music composer Laurie Spiegel – for sharing and evaluating inverse synthesis algorithms. Chapter 6 presents a large-scale multi-modal synthesizer sound dataset called *synth1B1* and an open-source GPU-enabled modular software synthesizer called *torchsynth* that is used to generate synth1B1. synth1B1 and torchsynth are designed to assist in the investigation of the complex relationship between synthesizer parameters and associated auditory output.

1.1 Research Questions

The main challenges in synthesizer programming arise from the disconnect between synthesizer parameters and the associated auditory output. The result of this disconnect is that synthesizers are difficult to use, have a high-barrier to entry, and impede the creative process of creating music and producing audio. The purpose of this thesis is to address these challenges. The main research question is:

> How can designers of synthesizer programming interfaces enable more people to be more creative more often?

Fundamental to answering this question is developing an understanding of the complex relationship between synthesizer parameters and the associated auditory result. Automatic synthesizer programming seeks to develop this understanding and bridge the conceptual gap between synthesizer parameters and the associated auditory output. More specific sub-questions that have motivated the work presented in this thesis are listed in the sections below.

1.1.1 Automatic Synthesizer Programming

What is the field of automatic synthesizer programming? What makes synthesizer programming so challenging and how do those challenges inform research focused on improving and automating it? A significant amount of previous work has been conducted in this field covering a large number of different approaches. How can

we organize the body of previous work to help understand the problem and the approaches that have been taken?

1.1.2 Inverse Synthesis

A substantial portion of previous work in automatic synthesizer programming has focused on algorithmic techniques for inverse synthesis. Many early approaches used evolutionary programming and more recent techniques use deep learning. How do evolutionary approaches compare to deep learning? What are the strengths and weaknesses of each of these approaches? What types of deep learning models are best suited for inverse synthesis? How can we support open evaluation and reproducibility of these approaches?

1.1.3 Representing Synthesized Sounds

A key component of developing automated synthesizer programming systems is being able to effectively answer the question: how similar is sound X to sound Y? How do we represent audio computationally and how do we define a metric that can help in answering this question?

1.1.4 Generating Synthesized Sounds

Datasets of synthesized audio and associated parameters are important for research on understanding the relationship between synthesizer parameters and the resulting audio, as well as for development of machine learning methods for synthesizer programming. The simplistic approach is to uniformly sample the parameters of a particular synthesizer. However, is uniformly sampling parameters the best approach to generating parameter selections? The associated sounds in many cases do not represent sounds that a human would have programmed in a musical context. How can we sample synthesizer parameters to generate sounds that sound as if a human programmed them?

1.1.5 Developing Supportive Tools

How do we create effective and intuitive user interfaces to support synthesizer programming? To what extent should the tool automate the process of synthesizer pro-

gramming? What interaction paradigms best support creativity? Which algorithmic methods for automatic synthesizer programming can best support creativity?

1.2 Summary of Contributions

The main contributions of this thesis are the following:

1. A survey and taxonomy of related work in automatic synthesizer programming from a user interaction perspective.
2. An open source software library designed to support the development, sharing, and evaluation of automatic synthesizer programming techniques.
3. An evaluation of several techniques for inverse synthesis conducted on a benchmark frequency modulation (FM) synthesis task
4. A novel inverse synthesis technique that combines deep learning and a multi-objective genetic algorithm.
5. Three open datasets, including one large-scale billion sound+parameter dataset designed to support further research in synthesizer programming and deep learning training / pre-training.
6. An open-source GPU-enabled modular synthesizer for efficiently generating the billion sound dataset on-the-fly and for supporting efficient research on synthesis algorithms.
7. A design framework for developing tools to support synthesizer programming.
8. A prototype automatic synthesizer programming tool designed to support exploration using two dimensional visualization of sounds based on sound similarity.

1.3 Thesis Structure

- Chapter 2 provides background information and historical context for audio synthesizers. The challenges and opportunities associated with synthesizer programming are outlined.

- Chapter 3 contains an overview of the field of automatic synthesizer programming. Synthesizer programming is framed as a human computer interaction problem and the various interaction paradigms and algorithmic techniques proposed in previous work is reviewed.

- Chapter 4 describes a software library the author has created to support research in automatic synthesizer programming and to serve as a repository for sharing and comparing approaches. This library was used for the experiments discussed in chapter 5.

- Chapter 5 describes an inverse synthesis experiment that compares recent approaches as presented in the literature. Several deep learning models and genetic algorithms were compared on a baseline FM synthesis problem. A novel hybrid approach is also proposed.

- Chapter 6 introduces a large-scale synthesizer dataset and a GPU-accelerated modular synthesizer called torchsynth that were designed to support further research in automatic synthesizer programming.

- Chapter 7 outlines a novel automatic synthesizer programming application that utilizes two-dimensional visualization of sounds to support exploration of synthesizer sounds. The application was developed with novice users in mind and is based on a set of design principles proposed as a framework to support the design of assistive synthesizer programming tools.

Chapter 2

Audio Synthesizers

Martin Russ introduces the synthesizer as any device that generates sound [115]. Even the human voice can be thought of as a synthesizer. However, sound synthesizers have become more broadly accepted as an electronic musical instrument that produces synthetic sounds. A synthesizer may do this through playback and recombining preexisting audio or through generating and shaping raw audio waveforms. Numerous types of synthesis techniques exist and are capable of producing a significant variety of sounds.

This chapter serves as background for audio synthesizers to provide context and motivation for research focused on improving their usability. Section 2.1 describes the evolution of synthesizers and provides historical context for developments in synthesizer technology relevant to this thesis. Section 2.2 overviews some of the main components of a typical synthesizer and introduces two of the most common types of synthesis: subtractive and frequency modulation (FM) synthesis. Section 2.3 introduces the topic of synthesizer programming in more depth and discusses the associated challenges and opportunities for improvement.

2.1 Evolution of Synthesizers

2.1.1 Analog Synthesizers

Until the late 1950s all synthesizers were analog. Analog synthesizers are defined by their use of continuous-time signals, as opposed to digital synthesizers, which use discrete-time signals (or signals consisting of separate, discrete pieces of information). Early analog synthesizers can be broken down into two broad categories based on their

approach to sound generation: (1) sounds are generated directly by electric circuits or oscillating vacuum tubes, or (2) sounds are generated by rotating or vibrating physical systems that are controlled by electronic sources [111]. The first – and the largest – sound synthesizer ever built was developed in the early 1900s by Thaddeus Cahill. On September 26, 1906, an audience of 900 individuals gathered to view the massive electronic instrument, called the Telharmonium, that was capable of producing pure sinusoidal waves, which was sound that had never been heard before. Other early synthesizers include the Theremin, built by Leon Theremin in 1920, which produced a pure tone with a varying pitch and amplitude that is controlled by a performer moving their arms in relation to two antennas. Versions of the theremin have been used in popular music by musicians including The Beach Boys, Led Zeppelin, and The Rolling Stones.

In the 1960s, two companies emerged on opposite sides of America and released synthesizers that shaped the modern landscape of audio synthesis. Around 1964, Don Buchla, who lived in the San Francisco area, released the the Buchla 100 Series Modular Electronic Music System. At the same time, in New York, Robert Moog released the R.A. Moog Modular System [94]. Both synthesizers were modular systems containing individual processing units called *modules* that could be interconnected using patch cables. Connecting together synthesizer modules is known as creating a *synth patch*, or simply a *patch*. Both the Buchla and Moog systems introduced Voltage-Controlled Oscillators (VCOs) which created electronic waveforms at musical pitches and could be controlled using an input signal called a control-voltage (CV). The Moog Modular System also featured a Voltage Controlled Filter (VCF) that was an early version of Moog's famous ladder filter design. This filter could resonate at a controllable frequency and was responsible for creating some of the most iconic synthesizer sounds that are still heard in contemporary music.

Important philosophical differences between Moog and Buchla Synthesizers lead to two distinct schools of thought: East Coast and West Coast synthesis. The development of the Buchla synthesizer by Don Buchla was guided by Morton Subotnick and other experimental composers working out of the San Francisco Tape Music Center. Subotnick explicitly requested that the synthesizer was not to be controlled by a traditional keyboard interface as he was worried that it would trap him into creating traditional tonal music. Instead, Buchla synthesizers are controlled using a set of touch plates and sequencers. At the same time, on the East coast, Robert Moog was developing the Moog Modular, which featured a traditional keyboard interface. This

allowed the Moog Modular Systems to be integrated more easily with Western music and was one of the reasons that Moog synthesizers became much more popular and commercially successful compared to Buchla synthesizers.

Another reason that Moog synthesizers were launched into the public eye was due to their use in recorded music. In 1968 Wendy Carlos used a Moog Modular synthesizer to orchestrate, perform, and record a selection of Johann Sebastian Bach pieces. The collection of music, called *Switched On Bach*, went on to become the best selling classical recording of all time. Other musicians had recreated classical music pieces on synthesizers, but none had reached the same level as *Switch On Bach*. The success of the release was in part attributed to Carlos' ability to design synthesizer sounds that worked synergistically with Bach's compositions [70]. Building on this success, Carlos went on to score synthesized soundtracks for movies including Stanley Kubrick's *A Clockwork Orange*. Another exceptional example of classical music recreated using Moog synthesizers is Japanese composer Isao Tomita's *Snowflakes Are Dancing*, which was released in 1974. The use of synthesizers in music and film extends into almost all genres of music and was the cornerstone in the development of new genres including techno and other electronic music genres. For more information, see Mark Jenkins' overview of the use of synthesizers throughout different genres of music in his book *Analog Synthesizers: Understanding, Performing, Buying* [70].

2.1.2 Digital Synthesizers

The first experiments with digital synthesis were conducted by Max Mathews on an IBM 704 computer in 1957 [110]. These experiments consisted of programming and synthesizing melodies using simple waveforms. The Music III program was developed by Mathews in 1960 and introduced an important concept called the *unit generator*, which was used to define basic components of a synthesizer that could be connected together in a similar way to how one one would "patch" a modular synthesizer. Mathews describes this concept as being developed in parallel, but separate from similar concepts in the analog synthesizer world (e.g., modular synthesizers). He described this as "an advantage because a musician who knew who to patch together Moog synthesizer units would have a pretty good idea how to put together unit generators in the computer."

In 1973 John Chowning, a researcher at Stanford, released landmark work on Frequency Modulation (FM) synthesis [24]. The patent for FM synthesis was licensed

to Yamaha who developed the Yamaha DX7 synthesizer using the technology. After being released in 1983, the Yamaha DX7 became one of the best selling synthesizers of all time. One of the major benefits of FM synthesis is that it can produce complex audio waveforms at low computational cost. Additionally, the Yamaha DX7 was a fully polyphonic synthesizer, which means that it was capable of producing multiple tones simultaneously (i.e., able to play chords), whereas most analog synthesizers at that time were monophonic (or only capable of playing one note at a time). The Yamaha DX7 was also difficult to program, though it came preloaded with a large selection of quality parameter settings, or presets, that allowed users to play the synth without having to learn how to program it. The evidence for the difficulties in using the DX7 have been primarily anecdotal, but "allegedly, nine out of ten DX7s [that went into] workshops for servicing still had their factory presets intact" [119].

As digital technology improved and computers became more powerful, new synthesis techniques, such as sampling synthesis [94], physical modelling [69], and digital emulations of analog synthesizers, or virtual analog (VA) synthesis, emerged. The development of more powerful computers also enabled recording workflows to be transferred into software and professional recording studios started to transition to digital with the release of Digidesign ProTools in the early 1990s. This shift has democratized music technology and more people than ever before have been able to start producing music [134].

2.1.3 Audio Plugins

In 1996, Steinberg[1] released the Virtual Studio Technology (VST) interface, which allowed third-party software including audio effects to be integrated into host applications, including digital audio workstations (DAWs) such as ProTools. Third-party audio software that integrates into host applications like this are more broadly referred to as *audio plugins*. The second version of VST was released in 1999, which added support for the Musical Instrument Digital Interface (MIDI) [114], a communication protocol enabling musical hardware and software to exchange information and control signals. The addition of MIDI to the VST interface opened the doors for VSTi, VST instruments, including software synthesizers. Other audio plug-in architectures have been developed in addition to VSTs, popular examples including Apple's Audio Units (AU) and Avid's Avid Audio eXtension (AAX). Audio plug-ins

[1] `https://www.steinberg.net`

are a platform for software developers to create and distribute unique audio effects and synthesizers, and an industry dedicated to their development has blossomed over the last three decades. At the time of writing, there are over 500 different synthesizer plug-ins available on the KVR[2] database of audio products.

2.2 Anatomy of a Synthesizer

Synthesizers can be viewed as comprising two major components: the **synthesis engine**, which is where sound is generated, and a **control interface**, which allows a user to control the synthesis engine [115]. Audio synthesis can be a complex process, resulting in a high-level of abstraction between the synthesis engine and the control interface. The role of the control interface is to present a conceptual model of the synthesizer to a user, which allows the user to express their ideas and modify the synthesis engine. The parameters on the control interface are mapped to components within the synthesis engine – often in non-linear ways. Figure 2.1 shows a diagram of the general components of a synthesizer and the control interface abstraction layer. The following sections provide more detail on the two major components of a synthesizer.

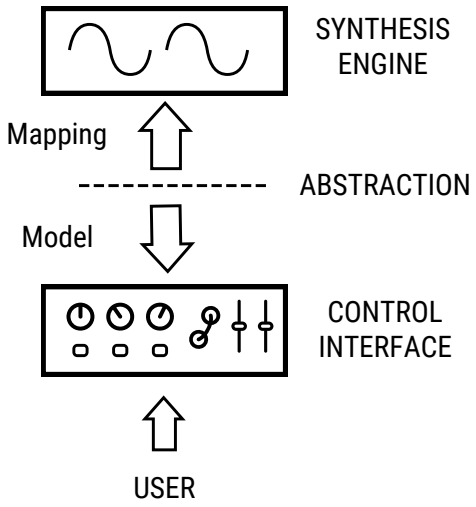


Figure 2.1: Abstraction between the synthesis engine and control interface. A control interface on a synthesizer is responsible for presenting a conceptual model of the underlying synthesis engine to a user. The parameters on the control interface are mapped back to the synthesis engine to modify audio generation.

[2] https://www.kvraudio.com/plugins/softsynth-virtual-instruments

2.2.1 Synthesis Engine

The synthesis engine is at the heart of sound generation in any synthesizer, whether it is an analog modular synth or a software audio plugin. Sam McGuire and Nathan Van der Rest provide an overview of the more popular synthesis methods in their book *The Musical Art of Synthesis* [94]. Popular synthesis methods include the following: subtractive, sample-based, modulation (e.g. FM), additive, wavetable, granular, vector, and physically modelling.

The exact technique that each synthesis method uses may differ; however, there are common components across many different techniques. It is useful to take a modular perspective when thinking about different synthesizer components, similar to the *unit generator* concept introduced by Max Mathews [110]. From this perspective, different components of a synthesizer are broken down into functional units (modules) that can be interconnected in various ways to build up a full synthesizer. We can generalize modules as both producing some output signal and having an optional input signal. Modules may also have parameters that can be mapped to a control interface for user control. We can broadly categorize the signals that are output by modules as either audio signals or control signals. Audio signals are generated by the synthesizer and are ultimately output as a sound. Control signals are used to modulate the parameters of other modules within a synthesizer.

Types of Modules

We can further categorize modules into two different types based on the type of signal they output: 1) **audio modules**, generate or process audio signals, and 2), **control modules**, generate or process control signals. In analog synthesis, audio signals are commonly generated by voltage-controlled oscillators (VCOs). The frequency of the oscillator in a VCO is controlled by the voltage of an input control signal. While voltages only exist in analog circuits, the concept has been extended into digital synthesizers as well, with the digital equivalent of a VCO sometimes being referred to as a digitally controlled oscillator (DCO). Other common audio modules are voltage-controlled filters (VCFs), voltage-controlled amplifiers (VCAs), and noise sources. Filters accept audio signals as input and attenuate, or boost, specific frequencies, VCAs are essentially an automated volume knob, and noise sources generate different types of noise, such as white noise.

Two common control modules are envelope generators (EGs) and low frequency

oscillators (LFOs). Envelope generators are generally triggered in response to an event, for example, a keyboard note being pressed. Once an EG has been initiated, it produces a control signal that evolves over time. The most common EG is an attack-decay-sustain-release (ADSR) envelope, which was designed to emulate the temporal evolution of instrumental sounds. LFOs function the same as regular oscillators, but at a frequency below the threshold of hearing. One common way that LFOs are used is to modulate the pitch on a VCO to create vibrato.

Subtractive Synthesis

Subtractive synthesis was one of the earliest methods and is used in Moog synthesizers. The basic idea behind subtractive synthesis is to start with a harmonically rich waveform and subtract from it using filters. This method is associated with the east coast synthesis philosophy.

Frequency Modulation Synthesis

FM synthesis engines are capable of producing a huge array of complex waveforms using a relatively simple structure, which makes them powerful; however, they are more conceptually challenging to understand compared to subtractive methods. The basic unit of an FM synthesizer is referred to as an operator, which generally contains a single simple sine wave oscillator and an amplitude gate controlled by an envelope generator. The simplest FM synthesizer consists of two operators that are connected together so that one of the operators controls the frequency of the second operator. The operator that does the modulating is referred to as the *modulator* and the operator that is modulated is referred to the *carrier*. Contrary to subtractive synthesis, FM synthesizers start with simple waveforms and build up to a final timbre using modulation. This method is more associated with west coast synthesis philosophy.

2.2.2 Control Interfaces

The control interface of a synthesizer allows a user to build up a conceptual model of the underlying synthesis engine so that they can exert control over the sound being generated. Most synthesizers have interface components such a knobs and sliders that allow users to control the pitch, loudness, and timbre of a generated sound. Keyboard type interfaces and MIDI keyboard controllers provide a direct and easily understood method for controlling pitch for those who are familiar with Western music

traditions. Volume controls also provide a relatively direct method for controlling the loudness. The rest of the parameters on a synthesizer control interface are dedicated to controlling the timbre. Seago [119] conducted an analysis on synthesizer interfaces and describes three types:

1. Parameter selection on a fixed architecture
2. Architecture specification and configuration
3. Direct specification of physical characteristics of sound

Parameter selection interfaces present the user with an organization of synthesizer modules that have been wired together in a fixed arrangement. Generally parameters are arranged in a hierarchical or structured way so as to represent the signal flow of the synthesizer architecture. These are the most common types of interfaces and were the type used on some of the early commercially successful units including the Moog Minimoog (see figure 2.2). A majority of software synthesizers emulate fixed architecture synthesizers and many software synthesizers directly emulate hardware interfaces.

Figure 2.2: MiniMoog. An example of a parameter selection on a fixed architecture interface. Photo attribution [53].

Architecture specification interfaces allow the user to wire together synthesizer modules. Modular synthesizers are a good example of these types of interfaces. Software like VCV Rack[3] provide emulations of Eurorack [67] hardware modular syn-

[3] `https://vcvrack.com/`

thesizer modules in software (see figure 2.3). Cycling 74's Max/MSP[4] and Native Instrument's Reaktor[5] are other examples of architecture specification interfaces implemented in software.

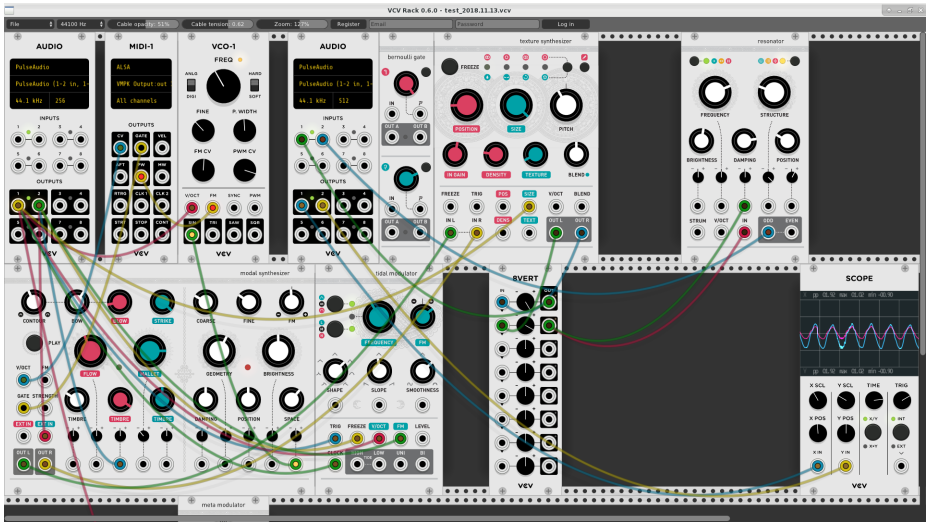


Figure 2.3: VCV Rack interface. Software synthesizer interface that emulates eurorack modular synthesizers. This is an example of an architecture specification and configuration interface. Photo attribution [105].

Direct specification interfaces make an attempt to allow the user to interact with the sound itself, as opposed to interacting with a conceptual model of a sound engine. A visual representation of sonic material is presented, typically as the time-domain waveform or a representation of the frequencies. Users are able to draw-in and shape the output sound through this visual interface. This type of interaction can be challenging to use due to the complex relationship between the visual representation of a sound and its perceptual quality. Some software synthesizers provide users a form of direct specification within a larger more traditional architecture: Xfer Records Serum[6] allows users to draw in custom waveforms for wavetables and Izotope Iris[7] has a visual interface to draw in the frequencies of a spectral filter, shown in 2.4.

[4] `https://cycling74.com/products/max`
[5] `https://www.native-instruments.com/en/products/komplete/synths/reaktor-6/`
[6] `https://xferrecords.com/products/serum`
[7] `https://www.izotope.com/en/products/iris.html`

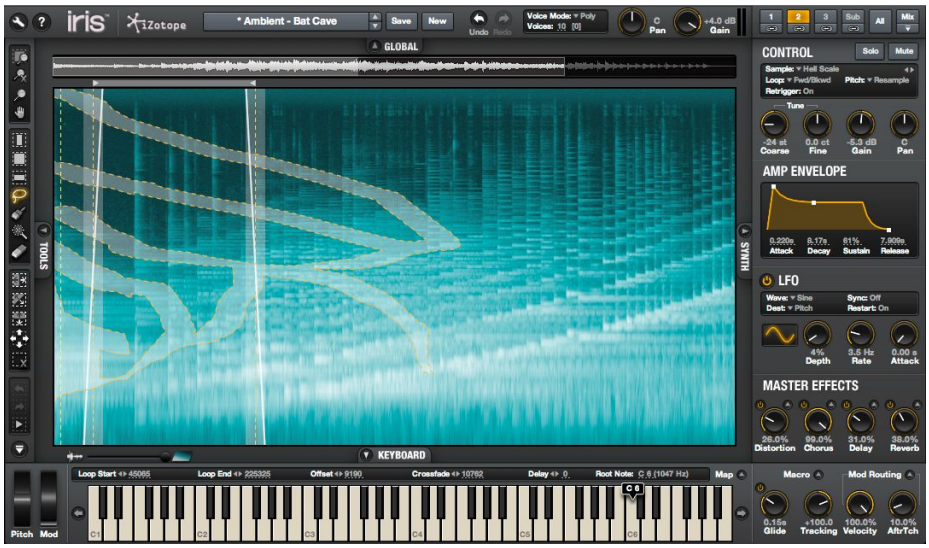


Figure 2.4: Izotope Iris. A sample-playback synthesizer plugin with a visual interface that allows users to draw in frequencies.

Skeumorphism

A common trend in software synthesizer control interface design is the use of skeumorphism. Skeumorphic interfaces are computer user interfaces that attempt to directly mimic their real-world counterpart. Development of these types of interfaces has become common in audio, partly due to nostalgia of analog audio gear [129]. The Arturia Mini V[8] is an example of a software synthesizer plugin that utilizes skeumorphism. The Mini V emulates the previously mentioned Moog Minimoog. Figure 2.5 shows a screenshot of the Arturia Mini V, refer back to figure 2.2 to see the similarities. Many software synthesizer plug-ins use skeumorphic interfaces, despite the ability for developers to create more nuanced and flexible user interfaces using computer graphic user interfaces (GUIs). Researchers have begun to question whether or not these skeumorphic interfaces enhance or hinder usability of audio software [86].

2.3 Synthesizer Programming

When trying to obtain a particular sound using a synthesizer, users generally have two options: they can try to build up the sound from scratch by adjusting parameters, or they can hope that someone else has gone through that process for them and search through a database of presets to find a sound that fits their criteria. Many users use

[8] `https://www.arturia.com/products/analog-classics/mini-v/media`

Figure 2.5: Arturia Mini V. A software synthesizer plugin emulation of the classic Moog Minimoog synthesizer. The user interface replicates the hardware version as closely as possible, an example of skeumorphic design.

a preset as a starting point and then manually adjust parameters to reach a desired sound [79]. The act of programming a synthesizer refers to the process of manually adjusting parameters, whether from scratch or using a preset as a starting point. Synthesizer programming is not an easy task, requiring a strong technical understanding of the particular synthesizer. Carlos and Tomita were masters at programming rich sounds that enhanced their music, and is one of the reasons their work achieved critical acclaim [70]. Specific techniques for programming synthesizers have lead to the creation of sounds that have defined genres of music, especially in electronic music, such as the "wobble bass" characteristic of Dubstep and or "squelchy" synth lines of Acid House tracks.

2.3.1 Challenges

The difficulty of synthesizer programming was identified in 1979 by James Justice [74]. Justice identified the complexity of real-world sounds and how challenging it is to specify synthesis parameters to recreate sounds in a satisfying way. Richard Ashley later pointed out that the difficulty of synthesizer programming is "due to the conceptual distance many musicians find existing between their intuitive notions of timbres and the control of synthesis parameter" [5]. In more recent work, Pardo *et al.* also describe the challenges of working with audio production tools as being related to the conceptual distance between intuitive spaces and control parameters.

Conceptual Spaces of Synthesizer Programming

There are three conceptual spaces that a user must navigate when using a synthesizer: 1) the parameter space, 2) the perceptual space, and 3) the semantic space. The parameter space represents the control interface and technical aspects of synthesizer programming, e.g. the specific value in Hertz of a filter cutoff. The perceptual space is related to the actual sound of a synthesizer and the semantic space is related to how one would describe the sound, e.g. that sound is "bright" or "gritty". When programming a synthesizer, a user must learn to relate between the low-level parameter space and the high-level perceptual / semantic spaces. The relationship between the low-level and high-level spaces is often complex. This complexity is responsible for the "conceptual distance" that Ashley was referring to [5]. Figure 2.6 shows an updated version of the synthesizer diagram from 2.1 with the conceptual distance between the parameter space and the perceptual space. Users must translate desired auditory changes (the perceptual space) to modifications in the control interface (the parameter space).

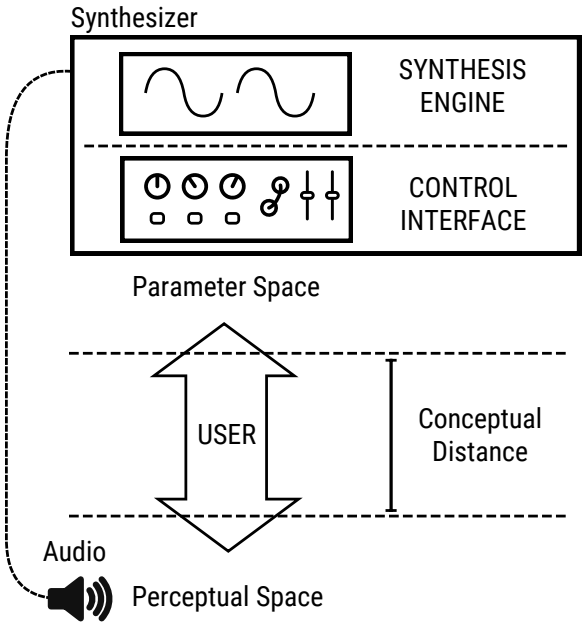


Figure 2.6: Updated figure of the components of a synthesizer (from figure 2.1) showing the conceptual distance between the parameter space and the perceptual space that the user must translate between.

Specifying Timbre

Perception of musical tones can be described as being comprised of three dimensions: pitch, loudness, and timbre. Synthesizers typically have controls for specifying all three of these dimensions. Pitch and loudness are uni-dimensional and have relatively simple mappings to parameters [119]. Therefore, the vast majority of parameters, which can be over hundred, are used to specify timbre. This means that the core of problem of synthesizer programming resides in the mapping between the perceptual and semantic spaces of timbre and the space of synthesizer parameters.

Following this, developing a good understanding of how timbre is defined would be a good place to begin to learn how to build more intuitive synthesizer controls. Unfortunately, we are at once faced with a challenge. The ANSI definition describes timbre as the attribute of auditory sensation that allows one sound be distinguished from other sounds at the same pitch and loudness [4]. This definition doesn't tell us very much about what timbre is, as opposed to what it is not. Understanding precisely what musical timbre *is* has presented itself as challenging problem [81] and significant research has been conducted to try to answer this. McAdams identifies that timbre is a purely perceptual quality of sound and provides a review of the subject [91]. In early research on musical timbre, Grey described timbre as being multi-dimensional and introduced the concept of a three dimensional *Timbre Space* [51]. Risset and Wessel explore timbre in the context of sound synthesis and emphasize the spectro-temporal representation of sound for understanding timbre [109]. This conception of timbre stresses the importance of the temporal aspects of sound and how the various frequency components of a sound evolve over time to our perception of timbre. More recent research based on neuroscience is moving away from the idea that timbre comprises a set of unique dimensions, but is instead a complex high-dimensional quality that must be taken as a whole [91].

All this is to say that deriving a concrete definition for timbre in the context of an audio synthesizer is a complicated problem. The intractable nature of the problem is one of the main reasons that the conceptual distance between the perceptual / semantic space of a synthesizer and the parameter space is so large. Defining parameters in technical terms based on their relation to the synthesis engine is a much more precise and concrete way to design a control interface, so it is not surprising that is what the vast majority of synthesizer developers do. Unfortunately, this means that users are stuck with learning the technical domain language of a particular synthesizer and

learning to relate that to their own perceptual / semantic conception of the associated audio output.

Impediments of Synthesizer Programming

Gordan Kreković recently conducted a study with synthesizer users that provides insight into attitudes towards synthesizer programming [79]. 122 individuals participated in that study, which consisted of answering questions related to their experiences with synthesizers. A majority of users were very experienced with synthesizers, 71% had ten or more years of experience, and only 2.7% were novice users, having less then a few months of experience. Kreković identified four impediments of synthesizer programming and asked the participants how much they agreed with each impediment:

1. it can be time consuming;
2. it can be a distraction from focusing on music;
3. it can be difficult and non-intuitive to learn to use a particular instrument;
4. it rarely leads to desirable results.

Most participants agreed with statements 1-3 and disagreed with statement 4, however, participants with less experience were more likely to agree with statement 4. This indicates that users with more experience programming synthesizers more often felt that they were able to achieve desirable results. The fact that most participants agreed with statements 1-3, especially given that the majority are highly experienced synthesizer users, indicates the extent of the challenges associated with synthesizer programming. Even after ten years of experience, users still feel like synthesizers can be difficult and non-intuitive to use. When given the opportunity to write about their experiences in a more open-ended way, participants generally reported on difficulties with user-interfaces, learning specific synthesizers, limited features, and the creative process.

Differences between Synthesizers

As mentioned earlier, there are a large number of different approaches to synthesis, and even within a particular synthesis type (e.g., subtractive or FM) there could be a nearly infinite number of variations. This is reflected by the hundreds of different

software synthesizers that are currently commercially available on websites like KVR[9]. Some methods such as subtractive synthesis have parameters that are more intuitively understood, whereas methods like FM synthesis "may be viewed as essentially an exploration of a mathematical expression, but whose parameters have little to do with real-world sound production mechanisms, or with perceived attributes of sound" [118]. These challenges and their affect on synthesizer programming is summarized in one of the responses from Kreković's study:

> Different sorts of synthesis require different background knowledge, most of which have steep learning curves that are at least partially exclusive. In other words, there is an enormous investment of time to deeply learn how the different forms of synthesis work. This learning is a prerequisite to effective use of synthesizers.

2.3.2 Opportunities

Based on the pervasiveness of the identified challenges and complexities associated with synthesizer programming, there is opportunity for development of methods that support both novices and experts. In fact, research into approaches that help bridge the conceptual gap between the perceptual / semantic and parameter spaces of synthesizers has been ongoing for over 40 years now. This research broadly falls under the umbrella of automatic synthesizer programming and will be reviewed in-depth in the next chapter.

To inform future work in this area, Kreković asked participants to rate their perceived helpfulness of four proposed systems based on approaches from previous work. The systems proposed were: 1) a system that generates random presets within a category, 2) a user provides a description of a the desired sound and a preset is generated for them, 3) a user provides an example sound and the system generates a presets to sound similar, and 4), more intuitive interactive user interface. Participants thought that proposed systems three and four would be helpful and systems one and two would be slightly helpful.

[9] `https://www.kvraudio.com/plugins/softsynth-virtual-instruments`

2.4 Summary

This chapter provided an overview of audio synthesizers and the task of synthesizer programming, which involves manually adjusting parameters to achieve a desired sound. A brief history of the evolution of synthesizers was provided, starting with analog synthesizers and leading up to software synthesizers that are implemented as audio plugins that work directly within modern digital audio workstations. The core of any synthesizer is the synthesis engine, which generates audio using a variety of different possible techniques. Users can control the synthesis engine through a control interface by modifying a potentially large number of parameters. The majority of this process of adjusting parameters, which is referred to as synthesizer programming, is related to the specification of timbre. Three conceptual spaces are involved in the process of synthesizer programming: the perceptual space, semantic space, and parameter space [102]. The perceptual and semantic space are higher level and are more directly understood by humans, whereas the parameter space is technical and relates directly to a particular synthesis algorithm. The distance between the perceptual / semantic space and the parameter space is large and leads to difficulties in learning how to use synthesizers effectively. The affect of these challenges was reflected by users in a recent user study [79] and opportunities for improvement to current synthesizer programming paradigms was identified. Automatic synthesizer programming has developed to address the challenges associated with synthesizer programming. An overview of automatic synthesis programming is provided in the next chapter.

Chapter 3

Automatic Synthesizer Programming

In the previous chapter, some of the specific challenges in synthesizer programming were identified along with the desire among synthesizer users for improved user interfaces and supportive tools. The field of automatic synthesizer programming emerged from the desire to solve these challenges and to find more intuitive answers for the question "How do I create _____ sound using _____ synthesizer?" James Justice [74] was one of the first to try to answer this question in the late 1970s. Justice's work used analytic methods to estimate the parameters for the FM algorithm [24]. This is an example of inverse synthesis, or sound matching [64], where a system estimates synthesizer parameters to replicate a target sound as closely as possible. Since then a large volume of work in automatic synthesizer programming has been published, exploring a variety of synthesis techniques, algorithmic methods, and user interaction approaches. This chapter provides an overview of the field and surveys some of the more popular methods that have been explored over the more than 40 years that automatic synthesizer programming has been an active area of research.

To the author's knowledge, at the time of writing there are no published works that provide an overview of the field of automatic synthesizer programming. The term *automatic synthesizer programming* was first coined by Matthew Yee-King in work on sound matching using a genetic algorithm [149]. This term has typically been used to refer to approaches that are focused on algorithmic techniques for sound matching or inverse synthesis problems. For the purpose of this thesis, automatic synthesizer programming (ASP) refers to any system that uses technology to support

the process of programming an audio synthesizer. This includes all of the approaches to inverse synthesis, as well as other interaction paradigms, which will be reviewed here.

3.1 Problem Formulation

The synthesizer programming problem is fundamentally a human computer interaction (HCI) problem. Currently, in order to use a synthesizer, users are expected to learn the domain language of the synthesizer they are using, as opposed to communicating ideas to their synthesizer in a way that suits their own creative needs. Creativity support is an emerging field of study that is interested in addressing HCI issues similar to this in creative domains. It is focused on the development of tools that enable and enhance the creative output of an individual or group – both novices and experts. Creativity support tools (CSTs) [125] span a wide array of application domains including visual art, textiles, cooking, and music. A central question that CSTs ask is: "How can designers of programming interfaces, interactive tools, and rich social environments enable more people to be more creative more often?" [125]. Reframing this question in the context of automatic synthesizer programming research results in the following question:

> How can designers of *synthesizer* programming interfaces enable more people to be more creative more often?

Answers to this question are related to the conceptual distance that was identified as one of the central challenges of synthesizer programming in the previous chapter. The distance between the perceptual/semantic space and the parameter space of a synthesizer is large and complex, requiring users to obtain large amounts of domain knowledge to effectively learn how to translate between those spaces themselves. Previous work in automatic synthesizer programming has sought to address this conceptual gap from a variety of different angles. This chapter reviews the main approaches to this problem.

3.2 Approaches

A typical automatic synthesizer programming system is an additional layer of abstraction on top of the control interface of an existing synthesizer. The goal of this

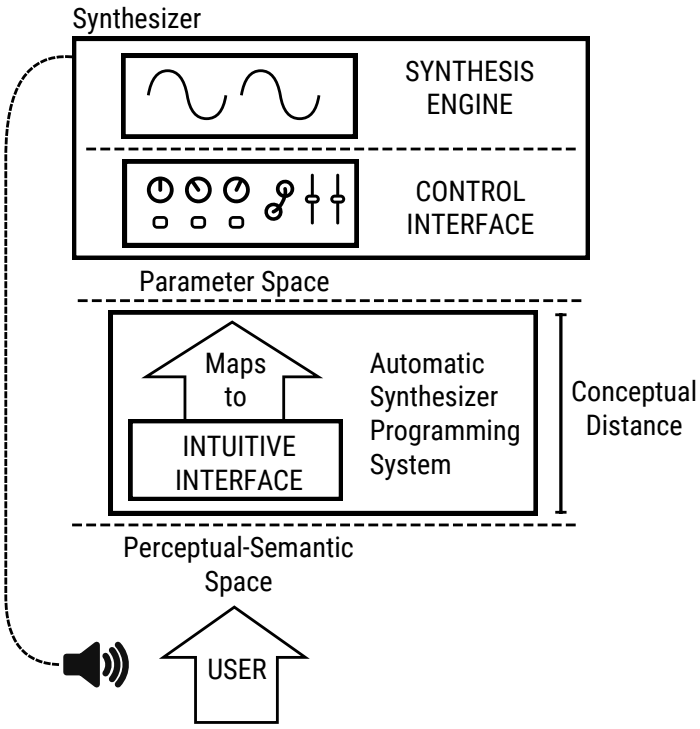


Figure 3.1: Automatic synthesizer programming systems assist in translating between the parameter space and perceptual / semantic space of a synthesizer. This diagram updates figure 2.6 from the previous chapter and shows how an automatic synthesizer programming system fits with an existing synthesizer and provides an intuitive interface that maps to the parameter space.

abstraction is to help bridge the conceptual distance between the parameter space and the perceptual and semantic space. Similar to how synthesizer control interfaces create a mapping from the parameter space to the synthesis engine, interfaces of automatic synthesizer programming systems present an abstracted higher-level model of the parameter space to a user and create a mapping to the parameter space. Figure 3.1 shows an updated diagram of the synthesizer user interaction model from the previous chapter (see figure 2.6) with an automatic synthesizer programming system inserted that assists in the translation from the perceptual/semantic space to the parameter space of a synthesizer.

A number of different interface approaches have been explored in previous ASP research. Pardo *et al.* [102] presented a framework for classifying different interaction paradigms within audio production tools. The development of this framework was based on how musicians and audio engineers communicate auditory concepts and included four different types of interaction styles: 1) evaluation, 2) descriptive words,

3) vocal imitations, and 4) exploration. Pardo *et al.* evaluated four different examples of audio production software with *natural* interfaces that attempted to bridge the gap between the parameter space and perceptual/semantic space. Each system was categorized under one or more of the interaction paradigms. These categorizations are useful for understanding the different approaches to ASP interfaces that have been proposed in previous work. For ASP systems, two additional categories are included: *example-based* interfaces and *intuitive controls*. There are six interaction paradigms for automatic synthesizer programming:

1. example-based interfaces (users provide an audio example of the sound that they want to play);
2. evaluation interfaces (users compare and evaluate results of different parameter settings to help guide the selection process);
3. using descriptive words (semantic description of the desired result);
4. vocal imitations (users imitate the sound that they desire);
5. intuitive controls (higher-level parameters, or macro parameters, are provided that have a complex mapping to to the lower level parameters); and
6. exploration interfaces (a number of different results are produced and provided to the user in a way for them to explore).

An overview of each of the approaches and related work is provided in the following sections. While previous work is introduced as a being a particular interaction style, there is overlap between each of these styles and many applications utilize more than one approach. For example, an evaluation-based interface that presents a user with a set of options for ranking could also be classified as an exploration-based interface because it supports the user in listening to a variety of different solutions. As a result, some work may be introduced as utilizing a specific style, and then repeated in another. Example-based approaches represent the largest portion of related work in automatic synthesizer programming and provide technical context for the other approaches discussed in this thesis.

3.3 Example-Based Interfaces

The example-based paradigm for automatic synthesizer programming has dominated the landscape of previous work. Approaches that fall into this category are typically referred to as **sound matching** or **inverse synthesis**. The goal of these systems is to find a parameter setting for a synthesizer that sounds as close as possible to a given example target sound.

The first research in this vein was conducted out of a desire to develop a deeper understanding of the complex relationship between the perceptual / semantic space and the parameter space of a synthesizer, and to learn more intuitive methods for synthesizer programming. In the late 70s through to the early 90s, several researchers studied analytic methods to attempt to reverse engineer parameter settings for FM synthesis [74, 8, 103] and non-linear synthesis [35]. In 1993, Andrew Horner conducted one of the first synthesizer sound matching experiments for FM synthesis using an artificial intelligence approach [64]. Horner used a genetic algorithm (GA) to search through the parameter space and to find the optimal parameter settings to match target instrumental sounds, including trumpets and guitars.

A typical inverse synthesis system involves the following steps: 1) the system receives a raw audio target, 2) the audio is processed and transformed into a representation for input to an algorithm, and 3) an algorithm receives the input representation and outputs parameter settings to match the audio target.

3.3.1 Audio Representations

Generally, the first step in a typical inverse synthesis system is to convert the raw input audio into a representation that exposes relevant features for the proceeding algorithm. What defines a relevant feature in the context of audio analysis for synthesizer programming is an open question to which a wide variety of approaches have been explored. Most commonly, audio is transformed from a time-domain representation to a temporal-spectral representation that exposes the time-varying frequency components of the target sound. Previous work has used the temporal-spectral representation produced by the short-time Fourier transform [62, 61, 63, 22, 150, 7].

Audio features extracted from temporal, spectral, and temporal-spectral representations are reviewed by Peeters [104] and have been used in inverse synthesizer research [96, 128, 17, 13]. Mel-frequency cepstral coefficients (MFCCs) – initially used for speech processing research – provide a compact representation of the shape

of a spectrum, and have been used in work by Yee-King [149], Heise [56], Roth [113], and Smith [126]. Transforms that attempt to model perceptual attributes of human-hearing have also been explored, including log-scaled Mel-spectrograms [156], which feature loudness and frequency scales that are more reflective of the human auditory system.

A more recent approach to extracting audio representations is to learn them using deep learning networks, a process called representation learning [9]. Barkan *et al.* explored using a convolutional neural network to learn features directly from time-domain audio [7]. Beyond the context of synthesizers, several researchers have built and trained models for generating audio representations [28, 43, 40], which provide promising options for future automatic synthesize programming approaches – both as off-the-shelf solutions or for inspiration for learning custom representations specifically for synthesizer sounds.

3.3.2 Approaches to Inverse Synthesis

Algorithmic approaches that have been used for parameter estimation in inverse synthesis can be loosely categorized into search and modelling methods. Search algorithms, which include genetic algorithms, estimate an optimal parameter settings by performing a structured search of the parameter space. Modelling methods, on the other hand, which have become more popular in recent years with the growth of deep learning, attempt to model the parameter space in order to estimate parameter settings. Other methods beyond search and modelling approaches that have been used in ASP research include fuzzy logic [97, 52], linear coding [96], and query approaches [17].

3.3.3 Search Based Methods

The most popular search methods used in automatic synthesizer programming research are genetic algorithms. A genetic algorithm (GA) is a method for solving an optimization problem using techniques based on the principles of Darwinian evolution, and is part of a broader class of evolutionary algorithms [147]. In a GA, a potential solution (an individual) is represented by its *genotype* and *phenotype*. In biology the genotype of an organism refers to its genetic makeup or set of genes, and the phenotype refers to the observable properties of that organism. In the context of

synthesizer programming, the genotype is the set of parameters, represented by an array of numeric values, and the phenotype is the auditory result.

During optimization with a GA, an initial set of individuals is randomly generated, and then iteratively evolved by subjecting the genotypes to a set of biologically inspired processes including selection, breeding (cross-over), and mutation. Individuals are ranked using an evaluation function that measures the $fitness$ of a given solution, which is calculated on the phenotype. The objective of a GA is to minimize that value (or maximize it, depending on the problem definition). The best candidates are selected for further evolution until either an optimal solution is found or a set number of iterations has been completed.

In the case of sound matching, the *fitness* of a potential solution is determined by measuring the error with regards to the phenotype of that solution and a target. The phenotype is usually represented using a time-frequency representation of the resulting audio; previous solutions have used spectrograms from the STFT [64, 133, 90] as well as mel-frequency cepstral coefficients (MFCCs) [149, 113, 88, 126].

Tatar *et al.* introduced the use of a multi-objective GA (MOGA) for synthesizer sound matching that used three different methods for representing phenotypes: the STFT, Fast Fourier Transform (FFT), and signal envelope [133]. Each phenotype representation was used in a different fitness function for an objective; therefore, the MOGA used by Tatar *et al.* had three objectives. In their work, they sought to automatically program a popular, portable synthesizer called the OP-1[1] developed by Teenage Engineering.

The goal of a MOGA is to find a set of *pareto-optimal* solutions. A solution is *pareto-optimal* when no other solution is better than it for all the fitness values and the set of *pareto-optimal* solutions is called the *pareto-front*. There are multiple different approaches to solving the optimization problem presented by a MOGA, popular solutions include the NSGA II [34] and NSGA III [33] algorithms. Tatar *et al.*'s approach used an NSGA III.

A more recent solution proposed by Masuda and Saito [90] utilized the NSGA II algorithm. The MOGA in their work had two objectives: the first was to minimize the error between the power spectrograms of the phenotypes, and the second was to maximize the diversity of phenotpyes as measured by a *behaviour characteristic*, which was represented by the spectral centroid and spectral flatness. From this dual-objective arises the concept of *quality diversity*: the *pareto-front* should con-

[1] `https://teenage.engineering/products/op-1`

tain solutions that are not only of high-quality (close to the target), but also should represent a diverse selection of solutions. In the context of automatic synthesizer programming, presenting a user with a set of potential solutions based on quality diversity could be beneficial in the sense that it would allow them to evaluate potential candidates from a wider range of possibilities that are close to their target.

In addition to GAs, other search-based techniques that have been used for sound matching include Particle Swarm Optimization (PSO) [56] and Hill-Climbing [113, 87].

3.3.4 Deep Learning Methods

Deep learning is a subset of machine learning that utilizes artificial neural networks to learn patterns in data and make predictions based on those patterns [84]. Deep learning models contain multiple layers composed of simple non-linear modules. Through iterative training, the layers are able to extract features from raw input data and learn intricate patterns in high-dimensional data. These multi-layer models have enabled deep learning models to excel at complex tasks including image recognition, speech recognition, and music related tasks such as audio source separation [57] and pitch detection [75].

In the context of an automatic synthesizer programming inverse synthesis experiment, a deep learning model accepts an audio signal as input and predicts synthesizer parameter settings to replicate that audio signal. Audio signals are often pre-processed using audio feature extraction or transformed into a time-frequency representation, although some approaches use raw time-domain audio [7]. Models are trained using a large set of example sounds generated from a synthesizer and use the parameter settings that generated a particular sound as the ground truth. During training, the loss is used to evaluate how well a model is learning and to optimize the parameters of the model through gradient descent. The loss is calculated using a loss function, which is computed using the error between predicted parameter settings and the actual parameter settings (the ground truth). In deep learning, models are grouped into discriminative models, which learn decision boundaries through observed data, or generative models, which learn the distribution of observed data. Related automatic synthesize programming approaches to the inverse synthesis problem have explored both types of models, and will be reviewed in the following sections.

Discriminative Models

One of the first works on the application of deep learning to the inverse synthesis problem was published by Matthew Yee-King et al. [153] in 2018. The main contribution of their work was an experiment that showed the effectiveness of a type of recurrent neural networks (RNN) called long short-term memory (LSTM) networks at sound matching on an FM synthesizer audio plugin. RNNs were developed to handle time-series data and to receive ordered data which is successively fed into the network architecture. As data is fed into the networks, activation states are stored internally and help to provide temporal context in latter stages of computation. RNNs have been particularly successful for audio generation problems [100, 40].

Yee-King *et al.* also experimented with additional machine learning techniques including genetic algorithm (GA), Hill-climber, and multi-layer perceptron (MLP) methods. They also compared two RNN models: a regular LSTM network as well as a modified LSTM network that had a bi-directional LSTM layer and as several highway layers. They called this network an LSTM++. Their methodology compared a set of algorithms on a series of successively more challenging problems on the open-source FM synthesizer Dexed[2]. Each problem was focused on programming a subset of the parameters in Dexed; a larger subset was used for each successive problem. A dataset of audio samples paired with the parameters used to generate the audio was created for training each of the deep learning models. Mel-frequency Cepstral Coefficients (MFCCs) were used as input for each of the models. The results were evaluated by looking at the error between MFCCs from a target sound and a predicted sound. Results showed that the hill-climber algorithm and the LSTM++ model performed the best. The LSTM++ model showed significant improvements over the other deep learning methods; however, the hill-climber performed the best on a majority of the tasks.

Barkan et al. explored convolutional neural networks (CNNs) applied to the inverse synthesis problem in their work which presented InverSynth [6]. The CNN has been used extensively for image related deep learning tasks and has recently been used successfully in music and audio related tasks, including music genre classification [23] and neural audio generation [37]. A key feature of CNNs is the use of shared filters that perform convolutions and produce representations at various levels of specificity. The shared filters allowed them to process large input data such as images and audio

[2] `https://asb2m10.github.io/dexed/`

with relatively few parameters compared to their fully connected counterparts.

In their work, Barkan et al. experiment with several different CNN architectures and compare them to a few different fully-connected networks. The focus of their research was performing inverse synthesis on a custom four oscillator FM synthesizer. They framed the inverse synthesis problem as a classification problem and quantized each of the 23 continuous synthesizer parameters into 16 discrete states. As input, they experimented with spectrograms from the STFT as well as raw time-domain audio. Because of the size of these inputs, they created different input representations for the fully-connected networks using a selection of hand-picked audio features defined in work by Itoyama *et al.* [68].

Micheltree and Koike introduced an interesting approach to programming Serum[3], a popular VST wavetable synthesizer [98]. They focused on the audio effects processing chain that follows the initial synthesis stage and explored using an ensemble of CNN models that worked together to select and adjust the effects. Each model in the ensemble was responsible for a single effects module (compressor, distortion, equalizer, phaser, or a reverb) and another model was responsible for selecting the ordering of each of the individual effects modules. Micheltree and Koike hypothesized that this approach was similar to how a human might select and program a synthesizer audio effect chain. Their approach also provided insight into the intermediate steps of programming a synthesizer that could be useful for educational purposes.

Generative Models

Esling et al. recently presented a novel application called *FlowSynth* that uses a generative model based on variational auto-encoders (VAEs) and normalizing flows [41]. Building on FlowSynth, Le Vaillant *et al.* also used a generative approach and explored programming a software implementation of the Yamaha DX7 using a VAE with normalizing flows [82]. Their work includes two important insights to the process of training a synthesizer programming model. The first involves the dataset that was used: they identified that previous research generated datasets of synthesizer + preset pairs by randomly sampling the parameter space and introduced a dataset of presets designed by humans. The second important insight related to how input is provided to the model. Their model received a multi-channel input that consisted of six different outputs from the same preset, played using different MIDI inputs. This factor is

[3] `https://xferrecords.com/products/serum`

important because it identifies and takes into consideration that a synthesizer can have the exact same parameter setting, but varying the MIDI pitch and velocity that is used to trigger the sound will result in a different sound. Using this multi-channel input allowed the network to learn how a single synthesizer preset may respond to different MIDI input.

3.4 Evaluation Interfaces

Evaluation interfaces for synthesizer programming allow a user to compare the results from multiple parameter settings to help them navigate and select a desired setting. One approach to evaluation style interfaces is through the use of interactive genetic algorithms (IGAs) [72, 31, 152]. In contrast to the GAs introduced in the example-based systems, the evaluation function in an IGA relies on user feedback during each iteration as opposed to measuring error between a candidate and a target. This allows a user to provide feedback to the system and participate in the process of finding a sound. IGAs also facilitate exploration as they can produce a large variety of novel examples in a short period of time and help a user explore different aspects of the parameter space. Figure 3.2 shows an example an evaluation-based interface called EvoSynth developed by Matthew Yee-King [152]. EvoSynth initially presents a user with a number of randomly selected examples, which they can listen to and iterate upon by selecting multiple examples to "breed" together. EvoSynth is web-based and at the time of writing is still available online[4].

Scurto *et al.* [117] explored interactive machine learning for parameter selection using a deep reinforcement learning (RL) algorithm [130]. They developed *Co-Explorer*, an RL algorithm that was able to accept feedback from the user and generate new synthesizer parameter settings in response. Co-Explorer supported both binary feedback, e.g., "I like / don't like that result", as well as *zone feedback* which could be used to direct the algorithm into or away from a particular zone of the parameter space. Users were also able to guide the system using *state commands* such as tell the system to move to a previous setting, enter into an autonomous exploration mode, or allow users to take over control completely.

One of the benefits of evaluation-based systems is that they re-engage users in the process of searching for parameter settings, as opposed to taking over control of

[4] `http://www.yeeking.net/evosynth/`

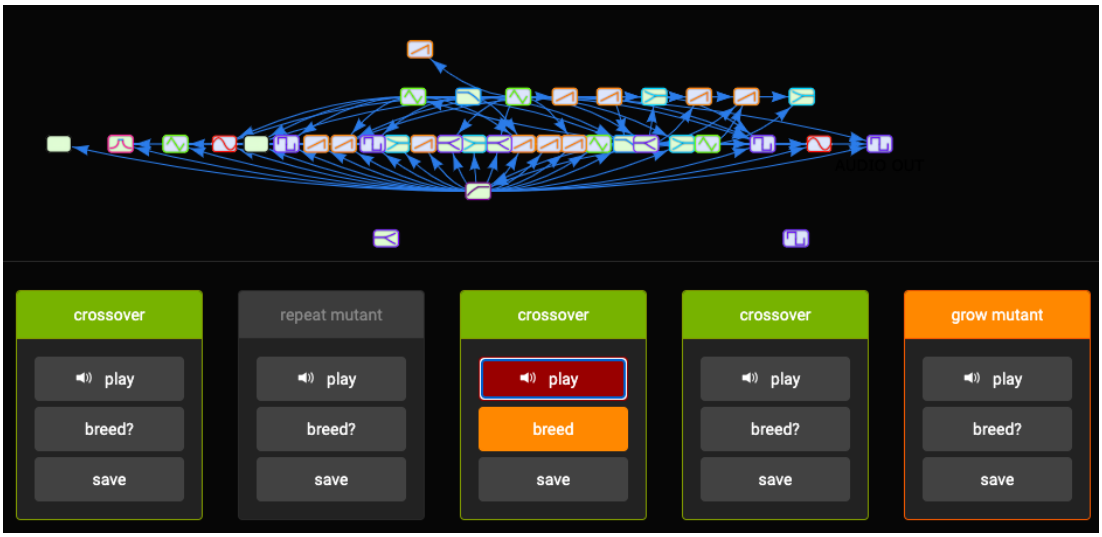


Figure 3.2: EvoSynth. A web-based interface for an interactive evolutionary approach to programming a software modular synthesizer. Users are presented with a selection of potential synthesizer patches generated using concepts inspired by evolutionary biology. They can listen to these patches and refine them through "breeding" which creates mixtures of multiple patches.

entire process as in the case in example-based interaction paradigms. These types of interfaces may be beneficial, especially when the user does not have an example sound for the system or wants to use a more exploratory approach.

3.5 Using Descriptive Words

In 1986 Ashley proposed one of the first examples of a system for programming a synthesizer using semantic descriptions of the desired timbre [5]. Shortly after, Ethington published the SeaWave system which enabled timbral control using predefined adjectives [42]. SeaWave broke the timbre of a sound into three different overlapping temporal segments (attack, presence, and cutoff) and mapped various adjectives within each of these segments to parameters of an additive synthesis engine. Johnson *et al.* proposed a machine learning approach to mapping timbral descriptors to parameter settings [73]. Ross Clement collected a set of human-made Yamaha DX7 presets and extracted keywords from the names of the presets to help develop a preset generation system based on the most commonly found keywords [25]. A novel approach to using descriptive words proposed by Kreković *et al.* [80] first used a combination of an expert-based system that mapped adjectives to numerical values for target audio

features, and then used a GA to search for a parameter that matched those audio features. Figure 3.3 shows a diagram of this system.

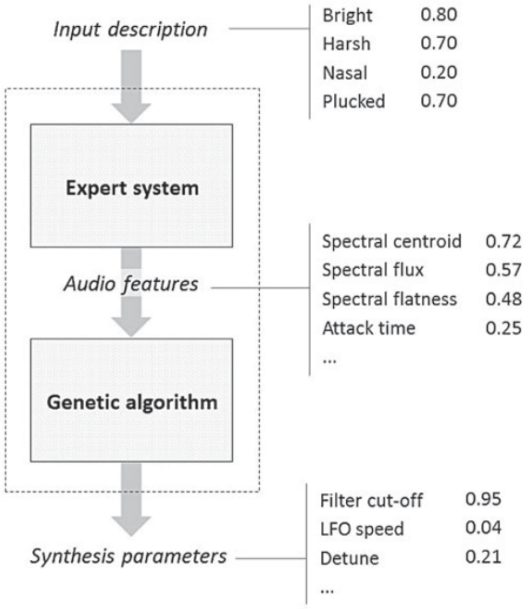


Figure 3.3: Kreković *et al.*'s [80] system for mapping between timbral descriptions and synthesizer parameter settings.

In a recent study, Roche *et al.* explored controlling audio synthesis using eight perceptually motivated parameters [112]. Their work used a neural synthesizer that generated spectrograms using a variational autoencoder (VAE) regularized using perceptual attributes to provide parameters based on timbral descriptions. Although this work does not fit the traditional description of automatic synthesizer programming since the mapping is not made to the parameter space of a regular DSP synthesizer, their method provides a good framework for exploring perceptually motivated controls for audio synthesis and could likely be applied to more traditional forms of synthesis.

In related work, Hayes and Saiti explored the semantic dimensions of a three operator FM synthesizer [55]. They found five major factors related to timbre semantics; the first two factors related to the luminance-texture-mass (LTM) model of timbre semantics [154] and the additional three factors related to clarity, pluckiness, and rawness.

3.6 Vocal Imitations

A method that has been shown as an effective way to communicate sound concepts is by using vocal imitations [85], and is more effective than using a verbalization in the case of unidentified sounds (sounds that have an unclear source). Building on this, Cartwright *et al.* developed the VocalSketch dataset [18] to support research in using vocal imitations to communicate sound concepts to computers. In synthesizer programming, the vocal imitation approach can be viewed as a subset of the example-based interface approaches as vocal imitations are a specific style of examples that could be provided to a system.

Cartwright *et al.* proposed the SynthAssist system for exploring and finding synthesizer sounds through vocal imitations [16]. Using SynthAssist a user can record an imitation of the synthesizer sound they wish to play and be presented with a set of possible solutions. They can then enter into an evaluation-based interaction style where they can rate and iterate through alternative sounds until they find a match. SynthAssist uses a data-driven approach for retrieving candidate synthesizer patches. A large number of patches are pre-generated and a set of time-series audio features are computed and stored in a database along with the parameter settings and audio. Candidate synthesizer settings are queried using dynamic time warping to compare the audio feature time-series.

In related work, Zhang *et al.* recently released *Vroom!*, a sound search engine based on vocal imitations [157]. *Vroom!* builds on previous work by Zhang focused on querying sounds by vocal imitation using CNNs [155, 156]. It implements a siamese CNN stack that extracts audio features from the vocal imitation and sound target in parallel and then uses a fully connected layer to compute similarity.

3.7 Intuitive Controls

The idea of developing more intuitive controls that overlay the more complex low-level parameter space was introduced by Wessel in 1979 [146]. Wessel proposed a system based on Grey's timbre space [51] for controlling an additive synthesizer using two higher-level perceptual parameters arranged on a two-dimensional grid: the vertical axis was related to the spectral energy distribution and the horizontal axis was related to the character of the attack. Adjusting these parameters resulted in smooth perceptual transitions. Wessel suggested a method for achieving more complex forms

of control, based on an efficient computer language.

The descriptive word-based application proposed by Kreković that was previously mentioned [80] also serves as an example of a system that uses intuitive controls. The input to that system is a set of numerical values associated to timbral descriptors, which are then used to program a synthesizer. This input can be thought of as a set of high-level intuitive parameters that operate at the semantic space of the user; the system is responsible for connecting these timbral parameters to the underlying synthesizer parameters.

3.7.1 Learning Controls

Some recent works using generative deep learning models have also proposed systems that expose intuitive *macro* parameters to a user [41, 112, 82]. The auto encoder style networks that have been used for these approaches feature a latent space in the architecture that provides a learned compact representation of the input. New examples can be generated by sampling this latent space, which is why these approaches are called generative. Each variable within the latent space can be thought of as a new higher-level macro parameter for a synthesizer that contains a complex and non-linear mapping to one or more of the underlying synthesizer parameters. One of the issues with these approaches is that the relationship between the latent parameters and synthesizer parameters is challenging to understand, and what each of the latent parameters does is not completely clear. Roche has taken steps towards creating a perceptually relevant latent space by regularizing the parameters to have correlates with timbral descriptors [112].

3.8 Exploration Interfaces

Exploration interfaces support finding potential solutions or alternatives. These types of interfaces have been introduced in related fields including intelligent music production to facilitate finding audio mixing parameters [19] or to help browse for audio samples [46, 123, 139]. Many of the previously mentioned automatic synthesizer programming tools can also be categorized as exploration interfaces. Interactive genetic algorithms help facilitate exploration by iteratively allowing users to listen to and evaluate sets of sounds from a synthesizer's space of sounds [72, 31, 152]. The genetic algorithm approach to inverse synthesis proposed by Masudo *et al.* supported

exploration by emphasizing diversity as well as close matches in the set of possible solutions presented to users [90]. This meant that users would be presented with a set of diverse potential options when searching for a synthesizer setting. The reinforcement learning approach proposed by Scurto *et al.* helps to facilitate a structured approach to parameter exploration [117].

Interfaces that visualize sounds on two or three dimensional interfaces based on sound similarity, such as the timbre space representation proposed by Wessel [146], provide an embodied approach to exploring synthesizer sounds by activating visual and spatial cognition in addition to auditory cognition. The benefit of embodied cognition to creative practices is identified by Davis *et al.* [32] and visualization of synthesizer sounds based on sound similarity is explored in more detail in chapter 7. SynthAssist also uses a visual layout of synthesizer sounds to support users in exploration [16].

Another type of interface that enables exploration are graphical interfaces, which allow users to interpolate between synthesizer presets [48]. These systems generally present a user with a two-dimensional interface that represents a number of different presets for an underlying synthesis engine. Each preset is positioned in a different location on the interface and the user can specify a point on the interface resulting in an interpolation between presets represented at that position. Le Vaillant *et al.* conducted a user evaluation that compared a users ability to recreate a sound using an interface with four parameters that controlled a synthesis engine against a graphical interpolation interface [83]. Expert users were able to use both interfaces equally well; however, novice and intermediate users were able to match the sound more easily and achieve comparable results to expert users with the interpolating interface.

3.9 Conclusions

This chapter has introduced the topic of automatic synthesizer programming and presented a review of related work. This review framed the *synthesizer programming problem* as a human-computer interaction problem and organized the body of automatic synthesizer programming research from this perspective. Six different user interaction methods were identified and include: 1) example-based interfaces, 2) evaluation interfaces, 3) using descriptive words, 4) vocal imitations, 5) intuitive controls, and 6) exploration interfaces. All of these approaches have a common goal of assisting synthesizer users in navigating the disconnect between synthesizer parameters

and the resulting auditory result.

Example-based interfaces allow users to specify a desired auditory output from their synthesizer by providing an example sound to an automatic synthesizer programming interface. These methods utilize inverse synthesis or sound matching to predict synthesizer parameters to match a target sound, and represent a large portion of the automatic synthesizer programming literature. Evolutionary programming has dominated the landscape of related work until recent years, which has seen a rise in deep learning approaches. User studies have identified example-based systems as being considered useful by synthesizer users [79] and as such are a focus of the next two chapters in this thesis. Chapter 4 describes an open-source library that was developed as a part of this thesis to support further research and shared evaluations. Chapter 5 presents an evaluation of several deep learning and genetic algorithm based methods for a baseline FM synthesizer inverse synthesis problem.

The emphasis placed on example-based interactions and inverse synthesis do not necessarily indicate that these approaches are the best solutions to the synthesizer programming problem. Evaluation and exploration-based interfaces and intuitive controls are beneficial as they engage the user in the process of selecting synthesizer sounds as opposed to completely taking over control, as is the case with example-based interfaces. The use of descriptive words and vocal imitations also represent ways that musicians already communicate music ideas to each other and could provide promising approaches for designing intuitive user interfaces [102].

The best interface will likely vary depending on the specific user and their specific needs for a given creative project. Some users may have a specific idea of what they want while others may be searching for inspiration [2]. As a result, developing interfaces that support a variety of different interaction styles will likely be beneficial. Building on these concepts, a prototype for an exploration-based automatic synthesizer programming interface is described in chapter 7, which was designed based on a set of proposed design criteria derived from the fields of creativity support tools [125] and music interaction [60].

-

Chapter 4

SpiegeLib: A framework for automatic synthesizer research

This chapter introduces SpiegeLib, a software framework for automatic synthesizer programming research that was developed by the author as a component of this thesis. SpiegeLib is an open-source software library written in the Python programming language with the goal of promoting collaboration and reproducibility in automatic synthesizer research. The development of the library was based on work published by Yee-King *et al.* [153] that explored automatic synthesizer programming of a VST FM synthesizer using deep learning methods. In their work they focused on the inverse synthesis problem which has the goal of finding synthesizer parameters to match a target sound, also referred to as sound matching. SpiegeLib is designed to support research in inverse synthesizer and provide a platform for sharing and evaluating methods.

Vandewalle *et al.* argue that reproducibility in computational science research increases the impact of a work and they provide a framework for evaluating the quality of reproducibility [142]. The aim of SpiegeLib is to provide a platform for researchers of automatic synthesizer programming to develop, test, and share implementations in a way that promotes reproducibility at the highest level. SpiegeLib stands for Synthesizer Programming with Intelligent Exploration, Generation, and Evaluation Library. The name SpiegeLib was chosen to pay homage to Laurie Spiegel, an early pioneer in electronic music composition. Laurie Spiegel is known for utilizing synthesizers and software to automate certain aspects of the music composition process. Her philosophy for using technology in music serves as a motivation for the SpiegeLib

software library: "I automate whatever can be automated to be freer to focus on those aspects of music that can't be automated. The challenge is to figure out which is which." [59]

4.1 Open Source Research Software

In Vandewalle *et al.*'s paper on reproducibility in computational sciences, they advocate for providing other researchers with "all the information (code, data, schemes, etc.) that was used to produce the presented results"[142]. Several authors of automatic synthesizer programming research have started to make their work open-access with source code available online.

Martin Roth and Matthew Yee-King developed *JVstHost*, a Java-based Virtual Studio Technology (VST) plugin host that was published by Matthew Yee-King [151] and was a component of *SynthBot* [149]. However, the code for *SynthBot* itself was not released. Matthew Yee-King also shared the source code for *EvoSynth*, an application for interactive synthesizer patch exploration [152]. A version of *EvoSynth* is hosted online allowing for immediate experimentation[1]. Kreković *et al.* released source code for their *MightyKnob* system [80]. Esling *et al.* released open-source code and a Max4Live[2] application for *FlowSynth* [41]. Le Vaillant *et al.* released source code for their generative VAE model[3] for performing inverse synthesis with Dexed [82]. Yee-King *et al.* recently took initial steps towards a software framework for automatic synthesizer programming research with the release of source code that provides functionality for generating research datasets and a set of algorithms for parameter estimation [153]. Along with that work they released the *RenderMan*[4] library for programmatically interacting with VST synthesizers using the Python programming language.

SpiegeLib builds upon this work with the goal of supporting and encouraging reproducibility within the automatic synthesizer programming research community. SpiegeLib is inspired by the steps that Yee-King *et al.* took towards creating a software library for automatic synthesizer programming research and extends that work with the inclusion of: an object-oriented API, base classes for customization, more

[1] `http://www.yeeking.net/evosynth/`
[2] `https://www.ableton.com/en/live/max-for-live/`
[3] `https://github.com/gwendal-lv/preset-gen-vae`
[4] `https://github.com/fedden/RenderMan`

robust evolutionary techniques, basic subjective evaluation, complete documentation, and packaging and delivery. It provides a framework for authors to share implementations in an open-access way that allows other researchers to quickly recreate results using a clearly documented set of freely-available tools.

4.2 Design of SpiegeLib

SpiegeLib is designed to be as extensible as possible to allow researchers to develop and test new implementations of components for conducting automatic synthesizer programming research. There are several stages in a typical automatic synthesizer programming experiment:

1) Synthesizer configuration: a synthesizer is selected and a subset of the parameters may be selected for estimation. For example, in work by Yee-King *et al.* [153], several different experiments were conducted using successively larger parameter subsets to increase the difficulty.

2) Dataset generation: for experiments requiring training, such as learning deep learning models, a dataset of synthesized audio and parameter pairs must be generated. Audio features may be extracted at this point too, which will be used as input to a model.

3) Training models: deep learning models are trained using the generated dataset.

4) Sound matching: this is the stage where parameters are estimated to match a target sound. In the case of deep learning models this involves inferring parameters using the trained model. For search techniques, the algorithm is run using the target audio as input.

5 Evaluation: results of the sound matching are evaluated here. Objective evaluation can be performed directly on the parameters as was the case in work by Barkan *et al.* [7], or audio can be rendered using the predicted parameters and evaluation carried out on the audio, which was done by both Barkan *et al.* and Yee-King *et al.* [153]. Subjective evaluation can also be carried out at this point with a user listening experiment.

SpiegeLib contains components to support all stages of this experimental pipeline. Implementation details and the components of the library are detailed in the following section.

Table 4.1: Algorithms currently implemented in `spiegelib`

Algorithms in `spiegelib`		
Audio Representations	**Deep Learning Estimators**	**Search Estimators**
FFT	MLP [153]	Basic GA
STFT	LSTM/LSTM++ [153]	NSGA III [133]
Mel-Spectrogram	Conv6/5 [7]	**Objective Evaluation**
MFCC	Conv6s/5s[2]	MFCC Error [153]
Spectral[1]	**Hybrid Estimators**	LSD [90]
	WS-NSGA[3]	Parameter Error [7]

[1] Spectral bandwidth, centroid, contrast, flatness, and rolloff.
[2] Derivatives of the models from [7] with reduced capacity
[3] Warm-start NSGA. A novel approach that uses a pre-trained deep learning model with a NSGA-III. Introduced in chapter 5 of this thesis.

4.3 Library Components

Base classes with functionality for interacting with software synthesizers, audio feature extraction, parameter estimation, and evaluation provide an API to support development of custom implementations that will work with other components of the library. A number of utility classes are also provided for handling audio signals, generating datasets, and running experiments.

SpiegeLib is written in the Python programming language and utilizes Python packages common in research including `numpy`, `scipy`, `tensorflow`, and `librosa`. SpiegeLib itself is a python package and is available through the Python Package Index (PyPI) with pip[5]. All dependencies, except for `librenderman`, are python packages available through the PyPI and will be automatically installed by pip. For more information on installation, system requirements, and detailed library documentation, please refer to the online documentation.[6]

A summary of the currently implemented algorithms is shown in table 4.1. A brief overview of these components and the main classes and functionalities of SpiegeLib is provided in the following sections.

[5] `https://pypi.org/`
[6] `https://spiegelib.github.io/spiegelib/`

4.3.1 AudioBuffer

The `AudioBuffer` class is used to pass audio signal signals throughout the library. It holds an array of audio samples and sample rate information. Methods of the `AudioBuffer` class provide functionality for loading audio from a variety of file formats, resampling, normalizing, time segmenting, plotting spectrograms, and saving audio as WAV files.

4.3.2 Synthesizers

The `SynthBase` class is an abstract base class that provides an interface for creating programmatic interactions with software synthesizers. `SynthBase` stores information and contains methods required for interaction with other components in SpiegeLib, including getting parameter lists, setting and getting patch configurations, overriding/freezing parameters, triggering audio rendering using MIDI notes, getting audio samples as `AudioBuffer`s, and requesting randomized patch settings. All patch settings are stored as a list of parameter tuples which contain the parameter number and parameter value. All parameter values are expected to be floating point numbers in the range [0.0, 1.0]. No requirement is made on how underlying synthesis engines are implemented, however, inheriting classes must provide parameter descriptions in a class attribute during construction and must provide implementations for four abstract class methods related to loading patches, randomizing patches, rendering audio, and returning an `AudioBuffer` of rendered audio.

`SynthVST` is an implementation of `SynthBase` and provides an interface for interacting with VST synthesizers. `SynthVST` is a wrapper for the *RenderMan* Python library developed by Leon Fedden in conjunction with research by Yee-King *et al.* [153].

4.3.3 Audio Feature Extraction

The abstract base class `FeaturesBase` provides an interface for computing audio representations and extracting features from raw audio samples. The `getFeatures()` abstract method must be overridden in inheriting classes and is where feature extraction algorithms are run. `FeatureBase` also includes functionality for standardizing results. By default, data is standardization is computing by removing the mean and scaling to unit variance. Parameters for standardization can be set based on the

distribution from one dataset, saved, reloaded, and applied to new results to ensure that standardization is carried out using the same parameters. Currently, implemented feature extraction classes utilize the `librosa` library [93] and include Mel Frequency Cepstral Coefficients (`MFCC`), Short Time Fourier Transform (`STFT`), Mel-Spectrograms (`MelSpectrogram`), Fast Fourier Transform (`FFT`), and a set of time summarized spectral features (`SpectralSummarized`).

4.3.4 Datasets

The `DatasetGenerator` class provides functionality for creating datasets of audio samples, feature vectors, and associated parameter settings from a synthesizer. An implementation of `SynthBase` and `FeaturesBase` are passed in as arguments to the `DatasetGenerator` constructor. To generate a dataset, random patches for the synthesizer are created and feature extraction is performed on the resulting audio. In this way, datasets for training and validating deep learning models, as well as datasets for evaluating sound matching experiments can be automatically generated. External datasets can also be used within SpiegeLib and the `AudioBuffer` class provides support for loading folders of audio samples for processing.

4.3.5 Estimators

All parameter estimation classes implement the `EstimatorBase` abstract base class. `EstimatorBase` is a minimal base class with one abstract method, `predict()`, that has an optional input argument. Implementations of estimators are split into deep learning approaches and other approaches including evolutionary algorithms. The included algorithms do not represent a comprehensive set of methods for automatic synthesizer programming research but are meant to cover common methods informed by previous work. Ten estimators are currently implemented and the author plans to add more in the near future including: a hill climbing optimizer [153], a particle swarm optimizer [56], a 1D CNN for raw audio input [7], and recent generative approaches [41, 82].

4.3.6 Deep Learning Estimators

All deep learning models are implementations of the `TFEstimatorBase` abstract base class which utilizes the `tensorflow`[7] and `keras`[8] machine learning libraries. `TFEstimatorBase` implements `EstimatorBase` and provides wrapper functions for setting up data for training and validation, training models, running predictions, and saving and loading model weights. While these methods are designed to help in handling of data typical to a synthesizer parameter estimation problem, all methods for a `tf.keras.Model` can be accessed directly from the `model` class member. Classes that inherit from `TFEstimatorBase` define models in an implementation of the `buildModel()` method which is automatically called during construction in the base class. This allows new models to be quickly designed, switched out, and compared with minimal effort.

The currently implemented deep learning models are based on prior work, specifically on work on Recurrent Neural Networks by Yee-King *et al.* [153] and work on Convolutional Neural Networks (CNN) by Barkan *et al.* [7]. Two modified CNN models with reduced capacity are also included (`Conv6s` and `Conv5s`). These models were created during the experiments conducted in chapter 5 and were found to result in more stable training for synthesizers with less parameters. For a full listing of deep learning models implemented, see table 4.1. An example code listing of sound matching using a trained LSTM model is shown in figure 4.1.

To save training and validation progress, the `TFEpochLogger` class can be passed in as a callback during model training. `TFEpochLogger` stores training accuracy and loss, and validation accuracy and loss over training epochs in a dictionary object which can be plotted after training.

4.3.7 Search-based Estimators

Two search-based estimators are currently implemented and utilize the DEAP python library [45]. A basic GA (`BasicGA`) is included as well as a multi-objective non-dominated sorting genetic algorithm III (`NSGA3`). Both GAs require feature extraction objects, or a list of feature extraction objects in the case of the multi-objective algorithm, which are used in the GA evaluation function.

[7] `https://www.tensorflow.org`
[8] `https://www.tensorflow.org/guide/keras`

```
import spiegelib as spgl
import spiegelib.estimator.TFEstimatorBase

# Load VST and set parameters from JSON file
synth = spgl.synth.SynthVST('./Dexed.vst')
synth.load_state('./dexed_simple_fm.json')

# MFCC Audio Feature Extractor
ftrs = spgl.features.MFCC(normalize=True)

# Load saved normalization parameters
ftrs.load_normalizers('./normalizers.pkl')

# Load LSTM model from saved model file
lstm = TFEstimatorBase.load('./fm_lstm.h5')

matcher = spgl.SoundMatch(synth, lstm, ftrs)

target = spgl.AudioBuffer('./target.wav')
output = matcher.match(target)
output.save('./lstm_predicted_audio.wav')
```

Figure 4.1: Example of SpiegeLib performing a sound match from a target WAV file on a VST synthesizer. A pre-trained LSTM deep learning model is used with MFCC input.

4.3.8 Hybrid Estimator

A parameter estimation technique that is introduced as a part of this thesis is also included in SpiegeLib. This estimator uses a pre-trained deep learning network during the initial population generation for an NSGA-III algorithm. The goal of this approach, which is explored in more depth in the next chapter, is to provide a warm start for the genetic algorithm search with the intention of enabling it find a suitable solution in less generations. As such, this estimator is called a Warm-Start NSGA-III, or WS-NSGA3 (`WSNSGA3`).

4.3.9 Evaluation

Objective evaluation of results can be carried out by measuring error between audio samples. `EvaluationBase` is an abstract base class for calculating evaluation metrics on a set of target and prediction data. A list of target values and lists of predictions for each target are passed into the constructor. `EvaluationBase` provides functionality for calculating statistics on results, saving results as a JSON file, plotting results

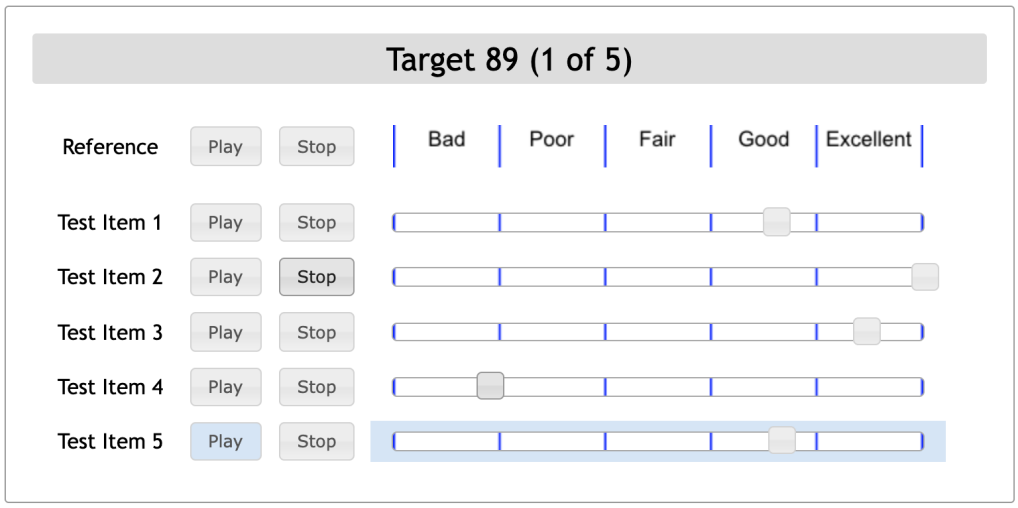


Figure 4.2: Interface for the basic subjective evaluation test using BeaqleJS. This is a MUSHRA style test. Results from four different estimators are being compared. A hidden reference is also included in the test items. The user must rank the quality of each test item against the reference (target used for inverse synthesis).

in histograms, and calculating metrics including mean absolute error, mean squared error, euclidean distance, and manhattan distance. Inheriting classes must implement the `evaluate_target()` method which is called for each target and associated estimations and is expected to return a dictionary of metrics for each estimation. The `MFCCEval` class implements `EvaluationBase` and calculates metrics on MFCC vectors for targets and estimations; the `LSDEval` class calculates the Log Spectral Distance (LSD) between two audio files; and the `ParameterEval` class calculates absolute error on each parameter separately, as well as the mean absolute error across all parameter values.

Functionality for conducting subjective evaluation of results is provided in the `BasicSubjective` class. This class accepts a set of audio files and runs a locally hosted server that generates a simple web interface using BeaqleJS [78]. An image of this interface is shown in figure 4.2. This runs a MUSHRA style listening test, where stimuli are ranked in terms of match quality to a reference. For inverse synthesis experiments, audio targets can be passed in along with a set of predictions for each target, and a sound similarity test will be generated with options for randomizing the ordering of targets and predictions. Results can then be saved as a JSON file.

4.4 Future Work and Conclusion

Development of SpiegeLib is ongoing and a number of expansions to the current library are planned. First, the author would like to continue to expand the number of estimators available and plan on integrating the following: a hill climbing optimizer [153], a particle swarm optimizer [56], a 1D CNN for raw audio input [7], and generative approaches [41, 82]. Second, the author would like to expand on the type of interactions available such as automatic programming from vocal imitations [17] and interactive methods. Finally, the author would like to encourage developers and researchers from the automatic synthesizer programming community to contribute to SpiegeLib. Information on contributing is available online.[9]

This chapter has introduced SpiegeLib, an open-source automatic synthesizer programming library. SpiegeLib is an object-oriented software library that was designed with the goal of supporting development, collaboration, and reproducibility in the field. The library includes implementations of classes for conducting automatic synthesizer programming research. These classes contain functionality for interacting with VST synthesizers, extracting audio features, creating datasets, estimating synthesizer parameters, and evaluating results. Ten implementations of deep learning and evolutionary parameter estimation techniques based on previous work are included, with more planned.

[9] `https://spiegelib.github.io/spiegelib/contributing.html`

Chapter 5

Inverse Synthesis Experiment

This chapter presents an inverse synthesis experiment that was conducted to compare several approaches that have been used in recent work. The methodology for this experiment was modelled after Yee-King *et al.*'s study on deep learning for automatic synthesizer programming [153], but with a simplified synthesizer configuration and a unique set of estimators. In total, eleven different methods for inverse synthesis were compared, including eight deep learning models [7, 153], two versions of a genetic algorithm (GA) [64, 133], and a novel hybrid approach that combines deep learning and genetic algorithms. A VST software emulation of the Yamaha DX7 FM synthesizer called *Dexed* was used with a restricted subset of the parameters. While Dexed has been used in previous work for inverse synthesis [153, 87, 82, 90], each work has used a unique subset of parameters and sounds for their experiments. The difficulty of the problem varies considerably depending on the number of parameters selected. In this experiment, the author proposes a minimal subset of seven parameters to reduce the complexity of the problem and to provide a benchmark for evaluation.

The research questions this experiment seeks to answer are the following:

- **RQ1**: How do deep learning approaches compare to genetic algorithms for FM inverse synthesis?
  - Are GAs able to produce competitive results when constrained to a limited number of generations to improve efficiency?
  - Can GAs be combined with deep learning approaches to improve both efficiency and accuracy?
- **RQ2**: What deep learning approaches are best for FM inverse synthesis?

  - How do recurrent neural networks (RNNs) compare to convolutional neural networks (CNNs)?
  - How does the audio representation affect the deep learning models?
  - Do higher resolution Mel-spectrograms afford a benefit over MFCCs?

- **RQ3**: Are certain synthesis parameters more easily learned? How does the ability to predict parameters correlate with audio results?

The following section describes in more detail the methods used for this experiment, including the precise configuration of the synthesizer, the dataset used for training, and the various techniques compared. Section 5.5 describes the methods used for evaluation and the results of this evaluation. Section 5.6 presents a discussion of the results and highlights some of the challenges associated with the inverse synthesis problem.

5.1 Synthesizer Configuration

The first step in the experiment was defining the synthesizer setup. The *Dexed* VST instrument was selected for this experiment for three reasons: 1) it is an FM synthesizer that is modelled closely after the Yamaha DX7 synthesizer which is both widely used as well as notoriously difficult to program, 2) it is open-source and free to use which supports reproducibility, and 3) it has been used in a number of previous works on inverse synthesis [153, 87, 82, 90]. An image of the *Dexed* interface is shown in figure 5.1. Dexed has 155 parameters that are available for external manipulation and for inverse synthesis. A subset of seven of these parameters were used in this experiment to turn *Dexed* into a simple two-operator FM synthesizer. In this configuration the second operator modulates the frequency of the first operator. A block diagram showing the resulting synth configuration is shown in figure 5.2. The subset of seven parameters control the amplitude envelope and tuning of the modulating oscillator; this simple synthesizer can produce a wide range of timbres that evolve in various ways over time based on the amplitude envelope of the modulating oscillator. An overview of these seven parameters is provided in table 5.1.

Figure 5.1: The Dexed synthesizer interface. Dexed is an open-source emulation of the Yamaha DX7 FM synthesizer and was used in the experiments in this chapter

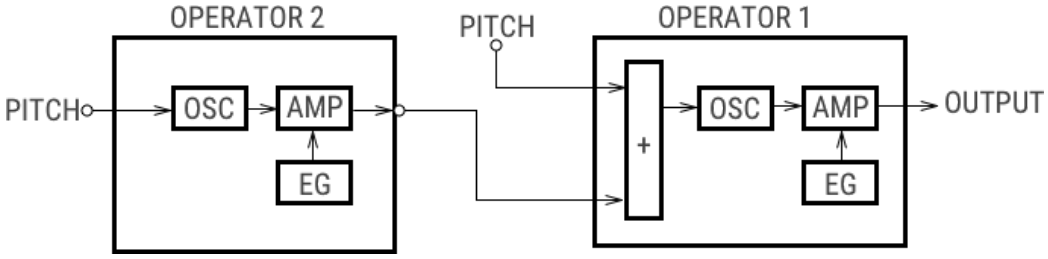


Figure 5.2: Block diagram of a two operator FM synthesizer. Dexed has six independent operators that can be configured in various ways, however for the experiments conducted here only the first two operators were used and were setup in this configuration.

5.1.1 Amplitude Envelope

Each operator in *Dexed* has a complex envelope generator (EG) that is used to modulate the amplitude of that operator. The complex envelope generator has five independent stages that are controlled by a set of parameters that affect the length and amplitude of each stage. See figure 5.3 for a diagram this EG. The EG for the second operator is the only EG that was used for this experiment. Five parameters for the second operator were modifiable: Rate $1-3$, and level 2 and 3. Level 4, the start of the envelope, was locked to zero, and level 1 was locked to one, the maximum value. This meant that the start of the envelope always consisted of an attack starting from zero and rising to one over a duration set by rate 1. The release portion of the

Table 5.1: Synthesis parameters used in experiment

Parameter	Description
OP2 EG RATE 1 OP2 EG RATE 2 OP2 EG RATE 3	Controls the duration of stages 1-3 of the envelope generator applied the the amplitude of operator 2
OP2 EG LEVEL 2 OP2 EG LEVEL 3	Controls level of the envelope generator at stages 2 and 3, which is applied the amplitude of operator 2
OP2 F COARSE OP2 F FINE	Frequency of second operator in relation to the fundamental frequency of the MIDI pitch. Coarse tuning is an integer ratio from $\frac{1}{2}$ to 31. Fine tuning allows for smaller non-integer adjustments.

envelope was not included during audio generation so rate 4 was negligible.

5.1.2 Operator Tuning

Two parameters controlling the tuning of the second operator were also modifiable: coarse tuning and fine tuning. The value of tuning is defined in relation to the fundamental frequency of the midi pitch and first operator. Integer ratios produce harmonic overtones and non-integer ratios produce more dissonant timbres.

5.1.3 Other Parameters

The remainder of the parameters were locked to values such that the first operator would create a static sine wave with no amplitude modulation for the entire duration of a MIDI note. All of the other operators, modulation, and effects processors in Dexed were turned off. This meant that all the variation of possible sounds within this synthesizer configuration were produced through varying the frequency and amplitude of the second operator, which was modulating the first operator.

5.2 Dataset Generation

A dataset of synthesized audio paired with the parameters used to generate those sounds was required for training the deep learning models. Creating this dataset consisted of four steps: 1) sampling the parameter space, 2) playing a single note

Figure 5.3: Diagram of an envelope generator in the Yamaha DX7 and Dexed. The envelope has five independent stages. During the first three stages the envelope moves linearly from level 4 to level 1, then to level 2, then to level 3. Each of these levels is controllable and the length of time taken to move to each level is also definable. The envelope is triggered by a key-on event. Once the envelope has progressed to level 3 it stays at that level until a key-off event is received, at which point the envelope progresses back to level 4.

on Dexed with those parameters, 3) saving the audio and parameters, and 4) transforming the audio into a suitable audio representation. All audio was rendered by playing a single MIDI note on Dexed with a note value of 48 (C3 $\approx$ 130.81Hz) and a length of one second. 80,000 examples were generated by uniformly sampling the seven parameters and rendering audio for one second. As mentioned previously, the release portion of the EG was left out, this is due to the render length and note length being the same.

The audio and parameter values were saved and the dataset was then split into a training and validation set with the training set containing 80% of the samples. Once the audio dataset was created audio representations were generated for each example.

5.2.1 Audio Representations

All the deep learning models received audio that had been transformed from a time-domain representation to a time-frequency representation. Two different representations were compared. The first was Mel-frequency cepstral coefficients (MFCCs), which have been used by Yee-King *et al.* [151, 153] for previous automatic synthesizer programming research. The second representation used in this experiment was log Mel-Spectrograms. Barkan *et al.* [7] used a STFT representation in their experiments with CNNs, however Mel-spectrograms provide a more perceptually relevant frequency scaling and have been used in recent work in audio representations [28, 58].

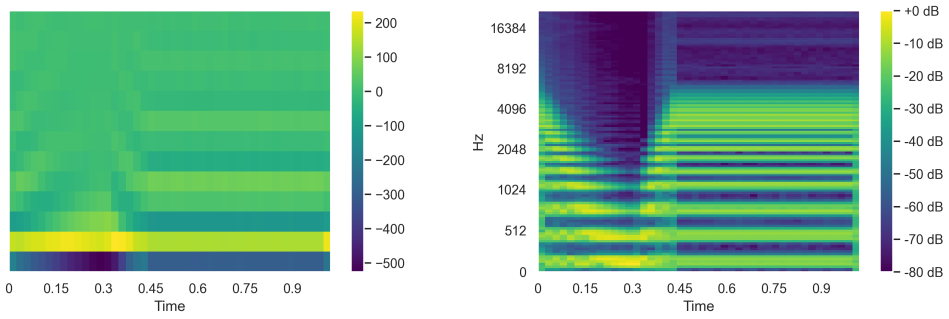
The MFCCs were computed with 13-bands using a frame size of 2048 and a hop size of 1024, this resulted in 44 frames over the 1-second long input audio. To compute the Mel-spectrograms, a STFT was first computed using a frame-size of 2048 samples and a hop-size of 1024 samples. Each frame of the magnitude spectrogram was then converted to a power spectrum and projected onto a 64 component Mel-frequency scale. The resulting Mel-spectrogram was then scaled to a log-scale, an amplitude scaling more reflective of how the human auditory system perceives loudness. Equation 5.1 shows the calculation of the log-scaled Mel-spectrogram from a complex valued spectrogram, $\mathbf{X}$, where $\mathbf{M}$ is a matrix of weights to project each frame in the spectrogram onto 64 Mel-frequency bins.

$$\mathbf{X}_{logmel} = 10 * log_{10}(|\mathbf{X}|^2 \cdot \mathbf{M}) \tag{5.1}$$

Both audio representations were standardized to have zero mean and unit variance. An example of the resulting representations computed on the same audio sample is shown in figure 5.4. Both the MFCC and Mel-spectrogram representations show an envelope in the signal starting at the beginning of the sound and lasting until about 0.45 seconds. The Mel-spectrogram gives a much higher resolution perspective of the frequencies present in the signal, whereas the MFCC only captures the overall shape of the spectral envelope and provides a much more compact representation. Pitch information is lost with MFCCs, which was identified by Masuda *et al.* as an issue for synthesizer sound matching applications [90]. However, Yee-King *et al.* achieved good results using MFCCs, so they are were included for comparison to the higher-resolution Mel-spectrograms.

5.3 Deep Learning Models

Four different models were used in this experiment. Two recurrent neural networks (RNNs) derived from work by Yee-King *et al.* [153], a convolutional neural network (CNN) derived from work by Barkan *et al.* [7], and a baseline multi-layer perceptron (MLP) network, also from Yee-King *et al.* [153]. Two versions of each of these models was created and optimized for either the MFCC or the Mel-spectrogram representation input. Therefore, eight models were trained in total. All models output floating point value estimations for the seven synthesizer parameters. Because these output



(a) MFCC (b) Mel-Spectrogram

Figure 5.4: Audio representations generated for a single audio example in the dataset. The left figures shows a 13-band MFCC, and the right shows a log-scaled 128-band Mel-Spectrogram.

values can be any 32-bit floating point number (outputs are clipped to a $[0-1]$ range), this means that the models are solving a regression problem. The loss function used during training was the mean squared error (MSE) between the target parameter values $\mathbf{y}$ and the predicted parameter values $\hat{\mathbf{y}}$. The MSE is calculated as follows, where N is the number of parameters:

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\sum_{i \in N} (y_i - \hat{y}_i)^2}{N} \tag{5.2}$$

Ideally the loss would be calculated on audio produced by Dexed using the estimated parameters, however there are currently no available solutions for rendering audio using VST instruments within a deep learning model training loop. Recent work by Ramírez *et al.* presented a solution for including audio effect plugins within a deep learning network [107], which could be modified for VSTis, however that is left for future suggested work.

Models were trained using an *Adam* optimizer [76] and hyperparameters for each model were optimized using a Tree-structured Parzen Estimator (TPE), which has been shown to be an effective method for hyperparameter selection [10]. Early stopping was used during training for all models; this halted training if the validation loss stopped decreasing for over ten epochs.

The following subsections describe each of the models that were included in this experiment. To find specific details on the implementation of each model see appendix A and for details on the hyperparameters used for training each model see appendix B.

5.3.1 Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) models, also referred to as a feedforward neural network, were the first and most simple types of neural networks. They contain one or more hidden layers with neurons that are connected to each neuron in the preceding and proceeding layers. These layers are also referred to as dense layers. An MLP was included in this experiment as a baseline model to benchmark the other models against. The architecture for the MLP was derived from Yee-King *et al.* [153] and has three hidden layers containing 256, 128, and 64 neurons respectively. Each neuron utilizes a ReLu activation. Dropout is included after the last hidden layer.

5.3.2 Recurrent Neural Networks

Two different Recurrent Neural Networks (RNNs) derived from work by Yee-King *et al.* [153] were used in this experiment:

- *LSTM:* The first is an RNN with three LSTM layers followed by a dropout layer and a fully connected linear output layer. The models used with MFCC input contained 64 units in each LSTM layer and the model used with Mel-spectrogram input had a larger capacity with 128 units in each LSTM layer.
- *LSTM++:* The LSTM++ is a novel architecture proposed by Yee-King *et al.* that performed the best on their sound matching experiment [153]. It features a bi-directional LSTM layer followed by dropout, then a dense layer with an ELU non-linearity prior to several highway layers. In this experiment, the size of the LSTM layers and the size and number of highway layers were selected using TPE. The models for both MFCCs and Mel-spectrograms were the same and used LSTM layers with 128 units and seven highways layers each of size 128.

5.3.3 Convolutional Neural Networks

Barkan *et al.* experimented with seven different CNN models for inverse synthesis that used a log STFT spectrogram input [7]. They found that the model with the most capacity, a model with 6 CNN layers and 2.3M trainable parameters, performed the best in their experiments. All the other spectrogram based models that they experimented with had 1.2M trainable parameters and between one to six convolutional

layers. They also use strided convolutions as opposed to the more traditional max pooling approach to downsample between layers [49].

The 6-layer and 5-layer networks proposed by Barkan *et al.* were implemented for this experiment. Early tests showed that these models were prone to overfitting and a derivative model with less capacity was developed. This model had five convolutional layers with fewer filters in each of the layers to reduce the number of trainable parameters, which was found to help mitigate overfitting.

The CNN architecture that was selected for MFCC input included batch normalization between each of the convolutional layers and two fully-connected layers before the output, each with 512 neurons and ReLu activation. The Mel-Spectrogram CNN did not use batch normalization and had three fully-connected layers before the output, each with 128 units and ReLu activation. Both the CNNs had dropout before the fully-connected hidden layers. Figure 5.5 shows a diagram of the Conv5s architecture used for the Mel-spectrogram input, excluding the batch normalization and dropout layers.

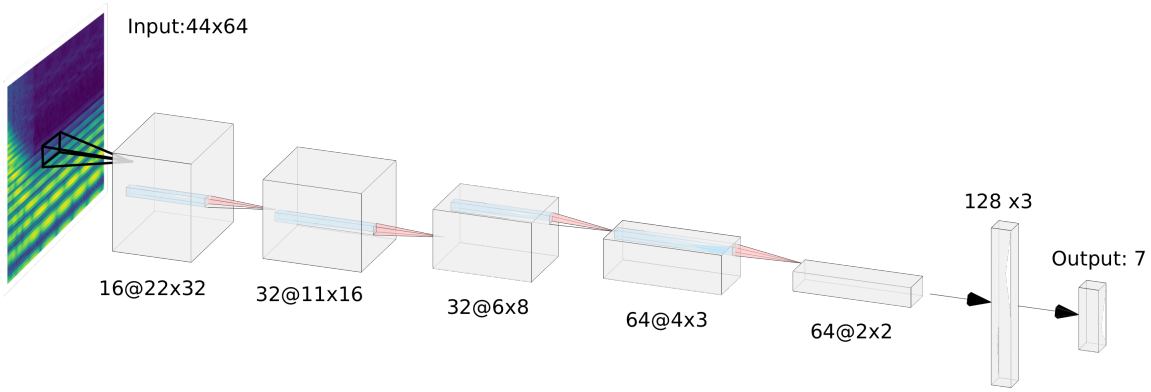


Figure 5.5: Network diagram of the CNN. This model accepts a Mel-spectrogram as input and contains five 2D convolutional layers followed by three dense layers. The output layer is predicted synthesizer parameters.

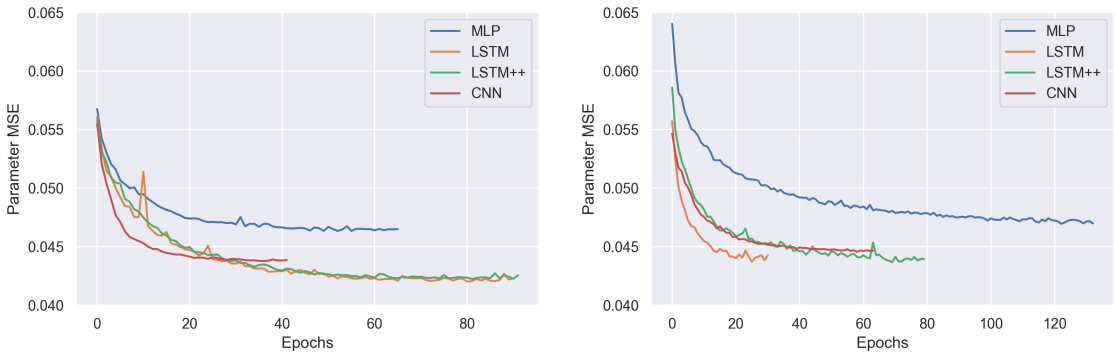
Initial experiments showed that even the reduced capacity CNN was prone to overfitting. An inverse time-decay learning rate scheduler was introduced in an attempt to address this to allow the models to train for longer. The learning rate at each training step was defined by the following equation:

$$\eta_i = \frac{\eta_0}{1 + \lambda \frac{i}{T}} \tag{5.3}$$

Where η_i is the learning rate at training step i, η_0 is the initial learning rate, λ is the decay rate, and T is the decay steps. For this experiment $\eta_0 = 0.001$ and T was set to be equivalent to 25 epochs. $\lambda = 3$ for the MFCC CNN and $\lambda = 4$ for the Mel-Spectrogram CNN.

5.3.4 Training Results

All the four models were trained with both MFCC and Mel-Spectrogram inputs, resulting in a total of eight trained models. The MFCC models will be referred to as MFCC-X and the Mel-Spectrogram models will be referred to as Mel-X where X is one of MLP, LSTM, LSTM++, or CNN. All the models were allowed to train until the early-stopping criteria was triggered. The number of training epochs ranged from 31 epochs for the Mel-LSTM to 133 epochs for the Mel-MLP. Validation loss at each training epoch is shown in figure 5.6 and plots for all models plotted against their respective training loss are provided in appendix C. The MFCC-LSTM and MFCC-LSTM++ models achieved the lowest validation loss during training, ≈ 0.042. All the MFCC versions of the models achieved better final validation loss values when compared to their Mel-Spectrogram counterparts, although only by a small amount; the average final loss for MFCC models was 0.044 whereas the average final loss for the Mel models was 0.045.



(a) MFCC Models

(b) Mel-Spectrogram Models

Figure 5.6: Validation loss during training for all the deep learning models.

5.4 Genetic Algorithms

Two different GAs were used: a basic single-objective GA and a multi-objective NSGA III. The multi-objective GA was derived from work conducted by Tatar *et al.* [133] that used the algorithm to automatically tune the parameters of a Teenage Engineering OP-1[1] synthesizer. In the case of the GAs, the $fitness$ is easily computed directly on the audio rendered from *Dexed*. During each iteration of the algorithm, individuals of the population (whose genotypes are synthesizer parameters) are rendered using Dexed, and the phenotype is produced by generating an audio representation from the resulting audio.

5.4.1 Fitness

In the case of the basic GA, the $fitness$ is computed as the mean absolute error (MAE) between 13-band MFCCs from the target and the individual. The MFCCs were calculated using a frame-size of 2048 samples and a hop-size of 1024 samples. MAE is calculated as follows, where N is the number of features, y is the target, and $\hat{y}$ is the predicted individual:

$$\text{MAE} = \frac{\sum_{i \in N} |y_i - \hat{y}_i|}{N} \tag{5.4}$$

Three different metrics were used for evaluating the $fitness$ of an individual for the NSGA-III algorithm, MAE between: 1) a 13-band MFCC, 2) magnitude spectrum from an FFT, and 3) a set of spectral features. The MFCCs were calculated using a frame-size of 2048 samples and hop size of 1024 samples. The FFT was calculated over the entire input audio, 1 second at 44,100 samples/second. For the spectral features, five different features were calculated: centroid, bandwidth, contrast across seven subbands, flatness, and rolloff. Each feature was calculated using a frame-size of 2048 samples and a hop-size of 1024 samples. The time series of audio features was summarized using the mean and variance. This resulted in a feature vector of size 22.

5.4.2 Generations

The quality of the result that can be produced by a GA is generally related to the amount of time that the algorithm is allowed to run for. While there is no guarantee

[1] `https://teenage.engineering/products/op-1`

that the optimal solution will be found, and the optimization may get stuck in a local minima, allowing the the algorithm to run for longer gives a higher likelihood of finding a quality solution. Tatar *et al.* allowed their NSGA-III to run for 1000 generations, which took 5 hours on a 50-core computer. The results of their method where competitive with an experienced human sound designer.

The synthesizer programming task in their problem was much more complex than the problem presented in this experiment. Initial experiments showed both GAs were able to perform well when allowed to run for 100 generations with a population of 300 individuals. For the NSGA this took about 20 minutes on a late-2013 MacBook Pro. Although this time is much better than 5hrs, this still limited the number of target examples that could be used for evaluation.

Twenty minutes is also a long time for application that is expected to be used in a music production context. To experiment with reducing the runtime of the GAs, a more severely constrained test was designed. Each GA was allowed to run for only 25 generations using a population of 100 individuals. These values were selected to significantly reduce the runtime and to validate whether competitive solutions could still be generated. With these constraints the runtime was approximately 75 seconds for the NSGA-III and 41 seconds for the basic GA when predicting parameters for a 1 second long target audio.

5.4.3 Mutation and Crossover

The remaining hyperparameters that control each GA are the rate of mutation and the rate of crossover. The rate of mutation sets the likelihood that a genotype is randomly modified during each generation. The rate of mutation was 30% for the basic GA and 50% for the NSGA-III. Rate of crossover sets the likelihood that an individual is combined with another individual to create a new "child" individual. The rate of crossover for both GAs was set to 50%.

5.4.4 Warm Start Genetic Algorithm

A novel hybrid method that combines the strengths of deep learning and genetic algorithms was developed for this experiment. Deep learning models front-load the computational complexity during model training, which can take several minutes to several hours, however are comparatively fast during parameter prediction (less than 100ms for the models included in this experiment). However, deep learning ap-

proaches have yet to achieve the same accuracy and consistency as genetic algorithms for parameter estimation.

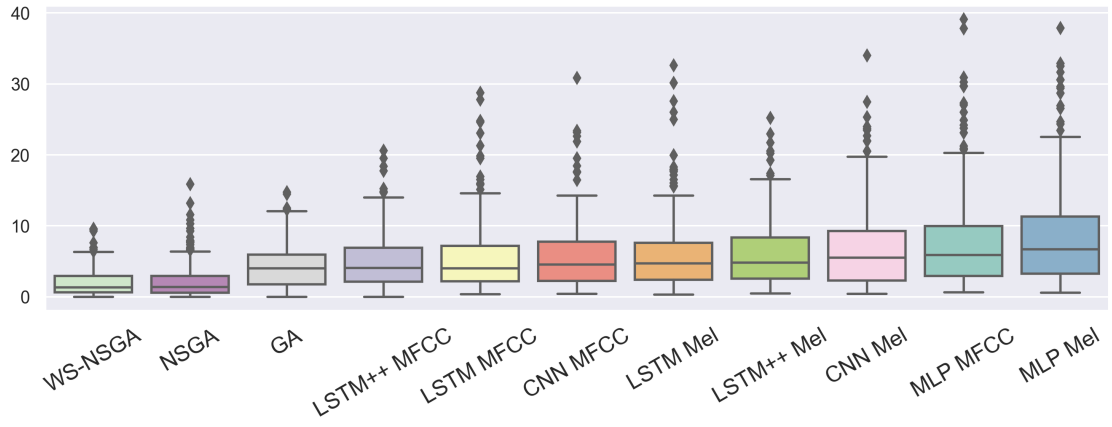
The method proposed here leverages the inference speed of deep learning models to support the generation of the initial population for a NSGA-III. This gives the the genetic algorithm a head start during optimization instead of starting with a population of purely random solutions. Because of this, the method is called *Warm-Start NSGA* or *WS-NSGA* for short.

To generate the initial population, parameters are first predicted using a pre-trained model. The MFCC-LSTM++ model was selected based on results presented in the following section. This produces a single individual for the population. Forty nine mutated versions of this individual are generated and added to the population. Finally, to add variation to the population another fifty random individuals are created. The algorithm is then run exactly the same as the regular NSGA, however now with only ten generations. This method takes approximately 30 seconds in total per prediction.

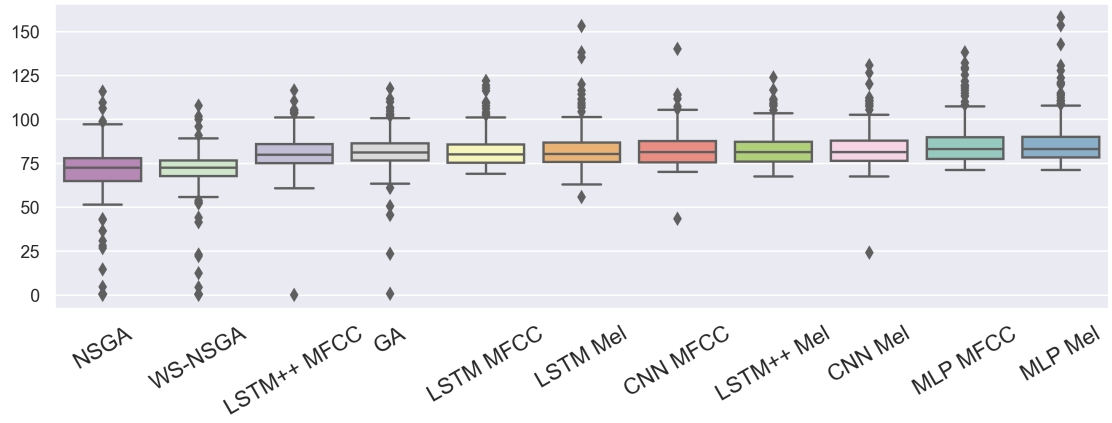
5.5 Evaluation

Evaluation was carried out by measuring the ability of each technique to accurately perform inverse synthesis on a set of 250 testing sounds. The same seven parameter subset of sounds was used for the test dataset, this means that an exact match was possible, which provided a known baseline for comparing each of the methods. Each of the 11 different techniques were run on each of the 250 test sounds and parameters estimated. The trained models for all the deep learning approaches were used for estimation and inference on a single target took $\approx$ 70ms on a MacBook Pro. The GAs were each run on each of the 250 target sounds, which took $\approx$ 5 hours using the NSGA-III and $\approx$ 3 hours with the basic GA, and $\approx$ 2 hours using the WS-NSGA.

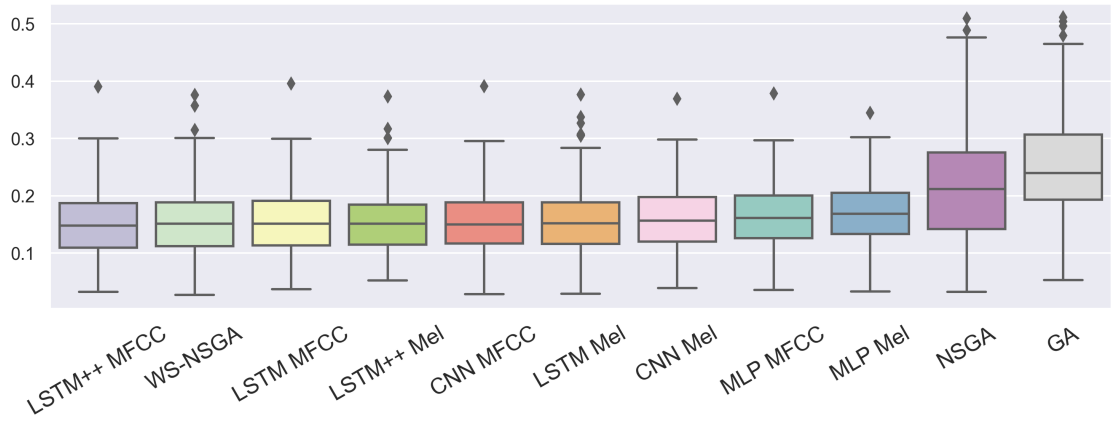
Audio was rendered using Dexed and the predicted parameters. The results were evaluated quantitatively using three different metrics: 1) MFCC MAE, 2) Log Spectral Distance (LSD), and 3) Parameter MAE. The first two metrics are calculated on the audio directly and the the third on the predicted parameters. Each metric is described in more detail below and all the results are summarized in table 5.2 and in box plots shown in figure 5.7.



(a) MFCC MAE



(b) Log Spectral Distance



(c) Parameter MSE

Figure 5.7: Box plots showing the objective measurements comparing each estimation method using a 250 sample evaluation dataset for sound matching.

	MFCC	LSD	Parameter
MFCC-MLP	5.9 ± 6.8	83.2 ± 12.0	0.1615 ± 0.0521
MFCC-LSTM	4.0 ± 4.9	80.1 ± 9.1	0.1515 ± 0.0579
MFCC-LSTM++	4.1 ± 3.9	79.9 ± 10.2	$\mathbf{0.1483 \pm 0.0552}$
MFCC-CNN	4.5 ± 4.7	81.5 ± 10.0	0.1501 ± 0.0547
Mel-MLP	6.7 ± 7.1	83.1 ± 13.2	0.1686 ± 0.0499
Mel-LSTM	4.7 ± 5.2	80.4 ± 11.3	0.1524 ± 0.0565
Mel-LSTM++	4.8 ± 4.8	81.4 ± 9.7	0.1513 ± 0.0536
Mel-CNN	5.5 ± 6.1	81.5 ± 10.9	0.1569 ± 0.0526
GA	4.0 ± 3.0	81.2 ± 11.3	0.2400 ± 0.0881
NSGA	1.4 ± 2.5	$\mathbf{72.5 \pm 27.3}$	0.2120 ± 0.0897
WS-NSGA	$\mathbf{1.3 \pm 1.8}$	72.6 ± 23.4	0.1511 ± 0.0604

Table 5.2: Summary of quantitative results for inverse synthesis evaluation. The values in bold are the scores with the lowest mean for that metric.

5.5.1 MFCC Error

MFCC error was calculated by computing the mean absolute error between MFCCs from a target audio and a prediction. Yee-King *et al.* [153] also used MFCCs to evaluate the quality of predicted sounds, however they used the euclidean distance between MFCCs as opposed to MAE. The MAE was used here as it showed results in a range that was more suitable for comparison. For this evaluation, MFCCs were calculated using a frame size of 2048 and a hop-size of 512. Figure 5.7a shows a box plot graph comparing all the techniques using this metric, ordered by the mean. There is quite high variance for all of the techniques and quite a few outliers with high error, even for the best techniques. However, based on the mean error, the GAs performed the best and the WS-NSGA was overall the best technique. Out of the deep learning models the LSTM based models using MFCCs for input performed the best.

5.5.2 Log Spectral Distance

The log spectral distance (LSD) is a metric that is calculated between two power spectrums. It was used by Masuda and Saito as a fitness objective in a genetic algorithm for inverse synthesis [90]. This metric is included to add more depth to the audio evaluation. Masuda and Saito identified a limitation with calculating error

between synthesized sounds using MFCCs, particularly, they highlighted the inability of MFCCs to capture specific frequency information as opposed to just the shape of the spectral envelope. LSD provides a more robust auditory evaluation of the results. LSD is calculated as:

$$LSD(P, \hat{P}) = \sqrt{\sum_{\omega} 10\log_{10}\left(\frac{P(\omega)}{\hat{P}(\omega)}\right)^2} \tag{5.5}$$

Where P is a target power spectrum, $\hat{P}$ is the power spectrum of a predicted sound, and ω is a frequency bin. Power spectrums were calculated using an STFT with an FFT size of 1024 samples and a hop-size of 512 samples. The average value of LSD over frames is used as the final metric in this evaluation. Lower values for LSD indicate a closer match. Figure 5.7b shows a box plot graph comparing the results. Again, the variance is quite high among all the techniques. Based on the mean LSD, the regular NSGA-III approach performed the best, followed closely by the WS-NSGA. The LSTM++ MFCC outperformed the basic GA based on this evaluation.

5.5.3 Parameter Error

Parameter error measures the mean absolute error (MAE) between the parameter values from a target and a prediction. In related work, Barkan *et al.* included this metric to evaluate the distance between target and estimated synthesizer parameters, it is included here for the same reason and to highlight the relationship between the auditory and parameter space. Figure 5.7c shows a box plot graph comparing the results of this evaluation. All the deep learning models outperformed the NSGA and GA in terms of mean parameter accuracy. Interestingly, the MFCC-LSTM++ performed the best out of all the methods, even the WS-NSGA, which used the MFCC-LSTM++ as a starting point. This means that the WS-NSGA in general found parameters that were more different from the target parameters, but resulted in more similar audio matches. These results highlight an important aspect regarding synthesizer programming: poor parameter reconstruction does not necessarily mean poor auditory results.

In addition to calculating the MAE for each target-prediction pair, the absolute error between individual parameters was calculated. This provides insight into how each technique handled the various parameters.

The frequency parameters affect the distribution of harmonics in the synthesized sound, whereas the envelope generator is related to the temporal aspect of the sound and how the frequencies evolve over time. Results for the five parameters related to the envelope generator are shown in table 5.3, and results for the two parameters related to the frequency of operator two are shown in table 5.4.

Results for the EG parameter evaluation show that the WS-NSGA performed the best in determining the rate for first and second stages of the EG, and at determining the level for the third stage. The MFCC-LSTM model was overall the best at estimating parameters for the EG, as well as for the rate of the third stage and the level of the second stage. All methods were most effective at determining the rate of the first stage, followed by the rate and level of the second stage, and worst at estimating the rate and level of the third stage. The duration of each of the first three stages range from 0 seconds to almost a minute. The length of the note is only one second long, so if the duration of the first stage is one second or longer, then the second and third stages will never be reached and will not be reflected in the audio result. This means that there are large portions of the synthesizer parameter space that are redundant for a note of one second long. This redundancy reflects a challenge for automatic synthesizer programming and one of the issues with determining loss based on parameter error.

Looking at the frequency parameters, the MFCC-LSTM++ outperformed all the other methods. This is an interesting result in light of the previously mentioned issues identified with MFCCs and their ability to capture precise frequency values. The Mel-LSTM++, which used Mel-Spectrograms, performed only slightly worse than the MFCC-LSTM++. Overall, the LSTM++ models performed the best at determining the frequency tuning and the coarse tuning was estimated more successfully than the fine tuning for all models.

5.6 Discussion

5.6.1 Deep Learning vs. Genetic Algorithms

RQ1 sought to explore the difference between deep learning approaches and genetic algorithms for inverse synthesis. Results show differences in terms of accuracy as well as computational complexity. The regular NSGA-III as well as the WS-NSGA methods outperformed all the other approaches in terms of the audio evaluation

	EG Rate			EG Level		
	1	2	3	2	3	Mean
MFCC-MLP	0.0279	0.1048	0.2443	0.1859	0.2072	0.1540
MFCC-LSTM	0.0129	0.0544	**0.2082**	**0.1643**	0.1920	**0.1264**
MFCC-LSTM++	0.0156	0.0533	0.2304	0.1670	0.1929	0.1318
MFCC-CNN	0.0200	0.0621	0.2202	0.1825	0.1911	0.1352
Mel-MLP	0.0379	0.1053	0.2618	0.2036	0.1982	0.1614
Mel-LSTM	0.0190	0.0626	0.2163	0.1811	0.1870	0.1332
Mel-LSTM++	0.0195	0.0607	0.2241	0.1848	0.1896	0.1357
Mel-CNN	0.0203	0.0735	0.2322	0.2023	0.1998	0.1456
GA	0.0326	0.1786	0.3196	0.2125	0.2812	0.2049
NSGA	**0.0095**	0.0834	0.2982	0.2464	0.2221	0.1719
WS-NSGA	**0.0095**	**0.0455**	0.2283	0.1712	**0.1850**	0.1279
Mean	0.0204	0.0804	0.2440	0.1911	0.2042	0.1480

Table 5.3: Absolute error on envelope generator parameters, averaged over all test items

metrics. Both these methods were constrained by the number of generations they were allowed to run for in order to reduce the running time; the WS-NSGA was limited to 10 generations and the NSGA-III was limited to 25. Despite these constraints, both methods produced high quality results. Figure 5.8 shows plots of the average minimum fitness values for each objective in relation to the generation. These plots show that the WS-NSGA consistently started with lower fitness values in the initial population, and was able to reach similar minimums to the NSGA-III after only ten generations. Although, the NSGA-III on average was able to find solutions with lower spectral error and FFT error.

To say with more certainty which approach produces the best results, a formal listening experiment would need to be conducted. However, the fact that the NSGA-III and WS-NSGA were consistently able to produce audio results that more closely resemble the target in terms of both MFCCs and LSD points to the quality of these approaches.

The GAs had an advantage because they were optimized directly on the audio signal from the specific target, whereas the deep learning models were optimized on the parameter values. Results from the objective metrics reflected these differences, the deep learning models reproduced the exact parameter values more accurately.

	Operator 2 Frequency		
	Coarse	Fine	Mean
MFCC-MLP	0.0680	0.2193	0.1437
MFCC-LSTM	0.0620	0.2061	0.1341
MFCC-LSTM++	**0.0599**	**0.1817**	**0.1208**
MFCC-CNN	0.0661	0.1945	0.1303
Mel-MLP	0.0747	0.2042	0.1394
Mel-LSTM	0.0605	0.2146	0.1376
Mel-LSTM++	0.0619	0.1835	0.1227
Mel-CNN	0.0678	0.2223	0.1451
GA	0.0971	0.2622	0.1796
NSGA	0.0688	0.2196	0.1442
WS-NSGA	0.0619	0.2014	0.1317
Mean	0.0681	0.2100	0.1390

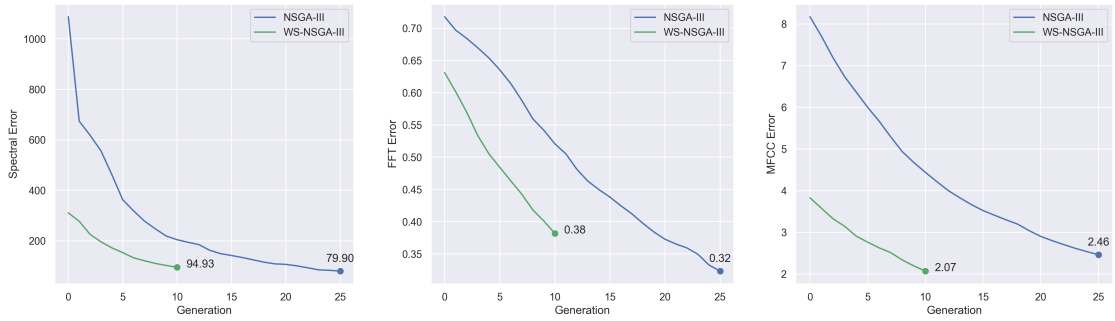
Table 5.4: Absolute error on operator two frequency parameters, averaged over all test items

Time Complexity

The trade-off between computational complexity and accuracy is displayed by the differences in the GA and deep learning approaches. In general the GAs produced results with higher match accuracy, but took longer to compute solutions. There is motivation for methods that are fast; in a music production context, automatic synthesizer programming techniques should reduce the the time required to program a synthesizer and reduce impediments to the creative workflow. The results obtained with the WS-NSGA are promising and show how GA and deep learning approaches can be combined to maximize the efficiency and accuracy of prediction.

5.6.2 Deep Learning Models

The second research question in this experiment, **RQ2**, sought to compare the different types of deep learning models used and the types of input that they used. Results show that the RNN based models outperformed the CNN using the same input representation. It is not surprising that the RNN models were more accurately able to model the temporal aspects of the input sound and achieved low error on the parameters associated with the envelope generator. RNNs are specifically designed



(a) Spectral Error (b) FFT Error (c) MFCC Error

Figure 5.8: Fitness values at each generation for the NSGA-III and WS-NSGA methods. Shows the minimum value for an objective amongst all individuals within the population at that generation, averaged across all 250 test samples used for evaluation.

to model relationships in sequential data, such as the framewise MFCCs.

An unexpected result was that the LSTM++ model performed the best at estimating the parameters associated with frequency using the MFCC input. This was unexpected based on the comments from Masuda *et al.* regarding the unsuitability of MFCCs for inverse synthesis [90]. One of the issues that was encountered when using the Mel-Spectrogram based models is that they were challenging to train. In early experiments many of the models began overfitting quite early when using the Mel-Spectrograms (see appendix C figures e-h). This was also an issue with the CNN models. Even introducing a learning rate schedule and attempting to optimize the learning rate still lead to difficulties with training and overfitting. Based on the successes that have been realized in other music and audio domains with Mel-Spectrograms and CNNs, the author was expecting to achieve better results with these approaches. Investigating alternative training methods and ways of using these approaches for inverse synthesis is an area for future work.

5.6.3 Parameters vs. Audio

The last research question, **RQ3**, sought to explore the relationship between parameter error and audio error for each method. The difference between the audio metrics and the parameter metrics shows that high error in terms of parameter values does not necessarily mean a high error in terms of the resulting audio. The NSGA-III method was one of the best methods in terms of auditory error, however, was the

second worst method in terms of parameter error – every deep learning approach outperformed it. The WS-NSGA model was in the top two for all metrics, both auditory and parameter. This further shows that achieving low parameter error is correlated with low auditory error, but is not a necessary condition for it.

5.7 Conclusion

This chapter presented an experiment that compared several different approaches to inverse synthesis using a VST emulation of the Yamaha DX7 synthesizer called Dexed. A subset of seven parameters from Dexed was used for this experiment, which turned Dexed from a complex 6-operator FM synthesizer into a relatively simple 2-operator FM synthesizer. This configuration was capable of producing a wide variety of different timbres and provided a benchmark inverse synthesis task for comparing approaches. Eleven different approaches, including eight deep learning models, two genetic algorithms, and a hybrid approach were evaluated on a test set of sounds.

Results showed that genetic algorithms perform best in terms of match accuracy, but suffer from long computation time. Deep learning models are fast during prediction, but are not as accurate at inverse synthesis as the multi-objective genetic algorithms. A hybrid approach was introduced in this chapter that used a pre-trained LSTM++ model during the generation of the initial population for a multi-objective genetic algorithm. The resulting method, called warm-start NSGA (WS-NSGA), was able to achieve similar results as the NSGA in terms of auditory accuracy in 40% of the computation time. These results show the potential for combining deep learning and genetic algorithms for inverse synthesis.

Amongst the deep learning methods, the recurrent neural networks outperformed the convolutional neural networks and fully-connected network. An unexpected result was that all the deep learning methods performed better with MFCCs compared to Mel-Spectrograms. This points to the need for future work on determining how the input audio representation affects network training in the context of synthesizer sounds.

Results of the this experiment also reveal a key aspect regarding the complexity of the synthesizer parameter space and its relationship to auditory results. Specifically, the results showed that there is redundancy in the parameter space that is conditional on the values of other parameters. In other words, there may be more than one, and potentially many, different parameter settings that produce the same or similar

auditory results. This points to the complexity of the mapping between synthesizer parameters and the associated auditory results – even in the case of the reduced seven parameter synthesizer used in this experiment. Developing a deeper insight into this relationship will be important for furthering research on automatic synthesizer programming.

Chapter 6

GPU Enabled Synthesis

In the preceding chapter, the author explored and compared various methods for inverse synthesis, one approach to automatic synthesizer programming. These experiments highlighted some of the challenges that researchers are faced with in pursuing this work. The most daunting challenge is related to the sheer complexity of the relationship between the parameter space and the perceptual (auditory) space; depending on the specifics of the synthesis engine and the number and type of parameters, there is significant variability within that relationship. In addition, there is no accepted best approach to computationally represent the perceptual space and define metrics within that space. Complexity exists within the open question: "How similar are these two sounds?" Can we quantify that similarity? Our ability to perform experiments that seek to solve these challenges is hindered by additional computational challenges associated with software synthesizers. Recent research has sought to automatically program commercially available VST synthesizers. This approach is well-founded considering these synthesizers represent tools that are currently being used by music producers and sound designers. However, these synthesizers are optimized for real-time use as opposed to use in the machine learning research where they introduce bottlenecks [90] and difficulties with learning in gradient descent methods.

In this chapter, the author presents a GPU-enabled modular synthesizer called *torchsynth* that was designed in response to the identified computational challenges. A large-scale dataset, generated with torchsynth, called synth1B1 is also presented in this chapter, along with two additional synthesized sound datasets generated using existing synthesizers. These datasets, and the torchsynth synthesizer, are made publicly available to support continued research in automatic synthesizer programming and synthesizer design.

6.1 Introduction

Machine learning progress has been driven by training regimes that leverage large corpora. The past decade has seen great progress in NLP and vision tasks using large-scale training. As early as 2007, Google [14] achieved state-of-the-art machine translation results using simple trillion-token n-gram language models. Recent work like GPT3 [15] suggests that it is preferable to do less than one epoch of training on a large corpus, rather than multiple epochs over the same examples. Even tasks with little training data can be attacked using self-supervised training on a larger, related corpus followed by a transfer-learning task-specific fine-tuning step.

Name	Type	#hours	Free	Multi-modal
Diva [41]	synth	12	yes	parameters
FSD50K [44]	broad	108	yes	tags
NSynth [40]	notes	333	yes	tags
LibriSpeech [101]	speech	1000	yes	text
DAMP-VPB [66]	songs	1796	no	lyrics
AudioSet [47]	broad	4971	no	video+tags
YFCC100M [135]	broad	8081	yes	video
MSD [11]	songs	72222	no	tags+metadata
Jukebox [36]	songs	86667	no	lyrics+metadata
synth1B1	synth	1111111	yes	parameters

Table 6.1: Large-scale and/or synthesizer audio corpora.

Unfortunately, audio ML progress has been hindered by a dearth of large-scale corpora. Audio ML involves multiple epoch training on comparably small corpora compared to vision or NLP. Table 6.1 summarizes various large-scale and/or synthesizer audio corpora. For example, using AudioSet [47] requires scraping 5000 hours of YouTube videos, many of which become unavailable over time (thus impeding experimental control). FSD50K [44], a free corpus, was recently released to mitigate these issues, but contains only 108 hours of audio. To the best knowledge of the authors, the largest audio set used in published research is Jukebox [36], which scraped 1.2M songs and their corresponding lyrics. Assuming average song length is 4:20, we estimate their corpus is ≈90K hours.

6.1.1 Background and Motivation

Pre-training and learned representations

Tobin *et al.* [136] argue that learning over synthesized data enables transfer learning to the real-world. Multi-modal training offers additional benefits. Images evince sizes of objects immediately, while object size is difficult to glean through pure linguistic analysis of large corpora [38]. Multi-model learning of audio – with video in [29] and semantic tags in [43] – has led to strong audio representations. Contrastive audio learning approaches like [116] can be used in multi-modal settings, for example by learning the correspondence between a synthesized sound and its underlying parameters. However, training such models is limited by small corpora and/or the relatively slow synthesis speed of traditional CPU-based synths (Niizumi, p.c.), [90].

6.2 Main Contributions

The synth1B1 corpus and torchsynth software provide a fast, open approach for researchers to do large-scale audio ML pre-training and develop a deeper understanding of the complex relationship between the synthesizer parameter space and resulting audio. A variety of existing research problems can use synth1B1, including:

- Perceptual research into audio, such as crafting auditory distance measures and inferring timbre dimensions. [140]
- Inverse synthesis, i.e. mapping from audio to underlying synthesis parameters. [153, 41]
- Inferring macro-parameters of synthesizers that are more perceptually relevant. [41, 132]
- Audio-to-MIDI. [54]
- Imitation of natural sounds with established synthesis architectures.

Researchers can also use the synth1B1 corpus to arbitrage innovations from adjacent ML fields, namely: large-scale multi-modal, self-supervised, and/or contrastive learning, and transfer-learning through fine-tuning on the downstream task of interest, particularly tasks with few labelled examples.

6.2.1 synth1B1

synth1B1 is a corpus consisting of one million hours of audio: one billion 4-second synthesized sounds. The corpus is multi-modal: each sound includes its corresponding synthesis parameters. We use deterministic random number generation to ensure replicability – even of noise oscillators. One tenth of the examples are designated as the test set. Researchers can denote subsamples of this corpus as synth1M1, synth10M1, *etc.*

Data augmentation has been used on small-scale corpora to increase the amount of labelled training data. As discussed in §6.1, large-scale one-epoch training is preferable, which is possible using synth1B1's million-audio-hours.

Besides sheer size, another benefit of synth1B1 is that it is multi-modal: instances consist of both audio *and* the underlying parameters used to generate this audio. The use of traditional synthesis paradigms allows researchers to explore the complex interaction between synthesizer parameter settings and the resulting audio in a thorough and comprehensive way. Large-scale contrastive learning typically requires data augmentation (*e.g.,* image or spectrogram deformations) to construct positive contrastive-pairs [20, 99]. However, this sort of faux-contrastive-pair creation is not necessary when the underlying latent parameters are known in a corresponding modality.

6.2.2 torchsynth

synth1B1 is generated *on the fly* by torchsynth 1.0. torchsynth is an open-source modular synthesizer and is GPU-enabled. torchsynth renders audio at 16200x real-time on a single V100 GPU. Audio rendered on the GPU can be used in downstream GPU learning tasks without the need for expensive CPU-to-GPU move operations, not to mention disk reads. It is faster to render synth1B1 *in-situ* than to download it. torchsynth includes a replicable script for generating synth1B1. To accommodate researchers with smaller GPUs, the default batchsize is 128, which requires between 1.9 and 2.4 GB of GPU memory, depending upon the GPU. If a train/test split is desired, 10% of the samples are marked as test. Because researchers with larger GPUs seek higher-throughput with batchsize 1024, $9 \cdot 1024$ samples are designated as train, the next 1024 samples as test, *etc.* The default sampling rate is 44.1kHz. However, sounds can be rendered at any desired sample rate. Detailed instructions are contained at the torchsynth URL for the precise protocol for replicably generating

synth1B1 and sub-samples thereof.

6.2.3 Questions in Synthesizer Design, and New Pitch and Timbre Datasets and Benchmarks

When generating synthesized datasets, one needs to sample the parameter space. Typically this is achieved through naïvely sampling parameters uniformly and rendering the resulting audio. Due to the complexity of the parameter space and interaction between parameters, this may lead to a large number of redundant, extreme frequency, and/or unnatural sounds. We propose several new open challenges in synthesizer design, specifically focusing on the task of designing parameters and sampling them, including:

- How do you measure the perceptual diversity of a synthesizer's sounds? How do you maximize it?
- How do you tune a synthesizer to imitate existing synthesizers? Is there a way to sample the parameters so the resulting audio sounds like a human-designed preset?

In §6.6 we demonstrate a principled approach to attacking these tasks. The main research barrier to solving these tasks is the lack of an automatic, perceptually-relevant auditory distance measures. To evaluate existing auditory distance measures, we devise two new evaluation methodologies and concurrently release timbre and pitch-datasets, each representing 22.5 and 3.4 hours of audio respectively, for the following open-source synthesizers: a DX7 clone and Surge, as DOI 10/f7dg and DOI 10/f652, respectively. These datasets represent "natural" synthesis sounds—i.e. presets designed by humans, not just a computer randomly flipping knobs—which we use in two ways: a) New benchmarks for evaluating audio representations. b) Evaluating the similarity of different sound corpora.

6.3 Design Methodology

6.3.1 Synth Modules

torchsynth's design is inspired by hardware modular synthesizers which contain individual units. Each module has a specific function and parameters, and they can

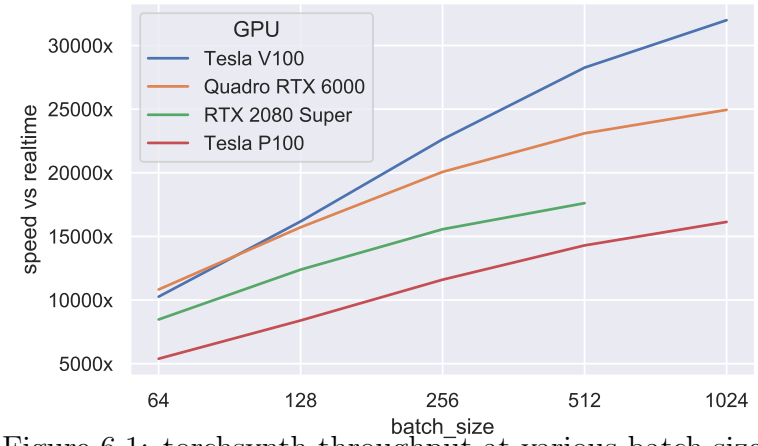


Figure 6.1: torchsynth throughput at various batch sizes.

be connected together in various configurations to construct a synthesizer. There are three types of modules in torchsynth: audio modules, control modules, and parameter modules. Audio modules operate at audio sampling rate (default 44.1kHz) and output audio signals. Examples include voltage-controlled oscillators (VCOs) and voltage-controlled amplifiers (VCAs). Control modules output control signals that modulate the parameters of another module. For speed, these modules operate at a reduced control rate (default 441Hz). Examples of control modules include ADSR envelope generators and low frequency oscillators (LFOs). Parameter modules simply output parameters. An example is the monophonic "keyboard" module that has no input, and outputs the note midi f0 value and duration.

To take advantage of the parallel processing power of a GPU, all modules render audio in batches. Larger batches enable higher throughput on GPUs. Figure 6.1 shows torchsynth's throughput at various batch sizes on a single GPU. GPU memory consumption $\approxeq 1216 + (8.19 \cdot \text{batch\_size})$ MB, including the torchsynth model. The default batch size 128 requires ≈2.3GB of GPU memory, and is 16200x faster than realtime on a single V100 GPU. A batch of 4 of randomly generated ADSR envelopes is shown in Figure 6.2.

6.3.2 Synth Architectures

The default configuration in torchsynth is the Voice, which is the architecture used in synth1B1. The Voice comprises the following modules: a Monophonic Keyboard, two LFOs, six ADSR envelopes (each LFO module includes two dedicated ADSRs: one for rate modulation and another for amplitude modulation), one Sine VCO, one

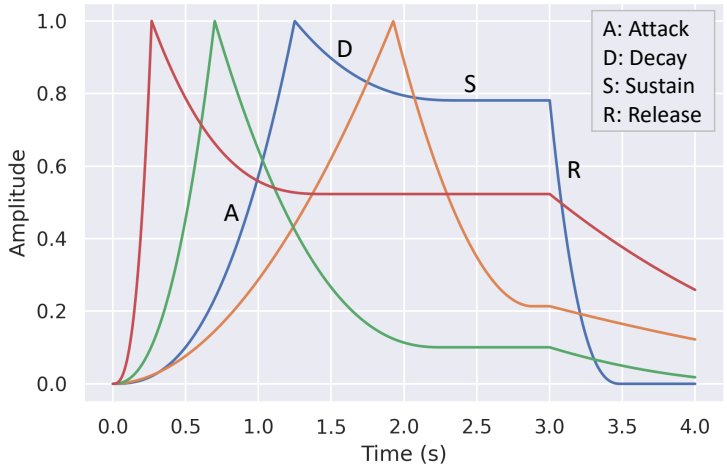


Figure 6.2: Batch of four randomly generated ADSR envelopes. Each section for one of the envelopes is labelled.

SquareSaw VCO, one Noise generator, VCAs, a Modulation Mixer and an Audio Mixer. Modulation signals generated from control modules (ADSR and LFO) are upsampled to the audio sample rate before being passed to audio rate modules. Figure 6.3 shows the configuration and routing of the modules comprised by Voice.

While the Voice is the default architecture of torchsynth 1.0, any number of synth architectures can be configured using the available modules. For example, a 4-operator frequency modulation (FM) [24] synthesizer inspired by Ableton Live's Operator instrument is currently in development.

6.3.3 Parameters

Module parameters can be expressed in human-readable form with predetermined min and max values, such as $0 \leq$ midi f0 ≤ 127. These human-intepretable values are used by the DSP algorithms of each module. Internally, parameters are stored in a corresponding normalized range $[0, 1]$. synth1B1 parameters are sampled uniformly from the normalized range. However, there is potentially a non-linear mapping between the internal range and the human-readable range. Besides the fixed min and max human-readable values, each parameter has hyperparameters "curve" and "symmetry" that determine how internal [0, 1] values are transformed to the human-readable values. The curve can specify a particular logarithmic, linear, or exponential sampling approach, *e.g.* to emphasize higher or lower values. Symmetric curves, which alternately emphasize the center or edges of the distribution, are used for parameters

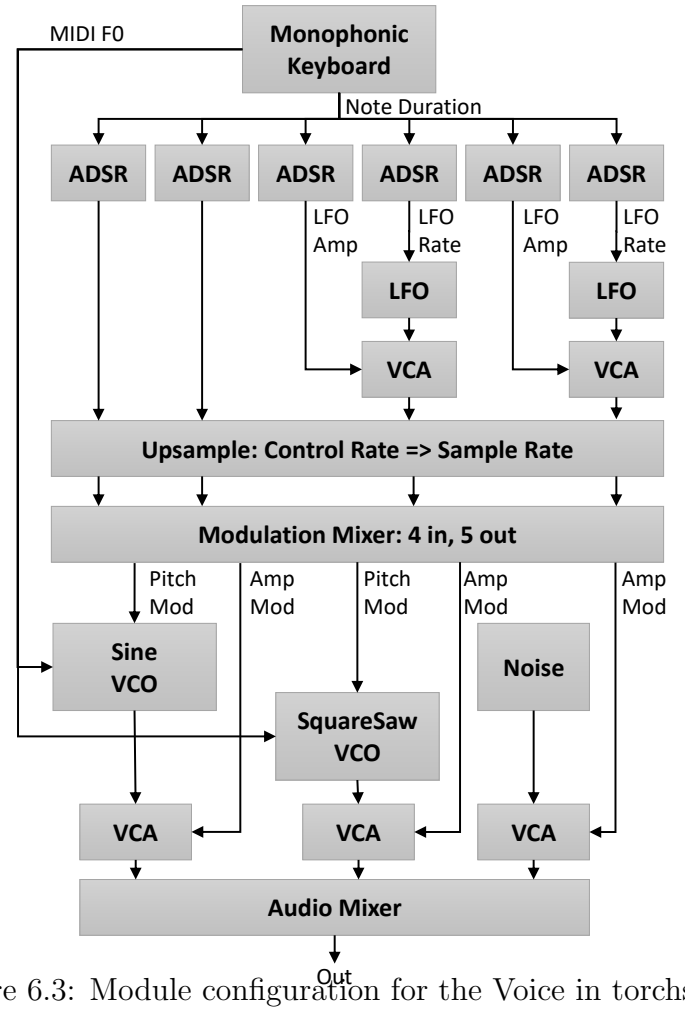


Figure 6.3: Module configuration for the Voice in torchsynth

such as oscillator tuning where the default value is 0 and can take on a range of both positive and negative values. An example set of non-linear curves is shown in Figure 6.4.

In the authors' nomenclature, a particular choice of hyperparameter settings, which correspond to random sample space of markedly different sonic character, are called *nebulae*. The initial Voice nebula was designed by the authors based upon intuition and prior experience with synthesizers. We experiment with tuning the hyperparameters of Voice to generate different nebulae in §6.6.

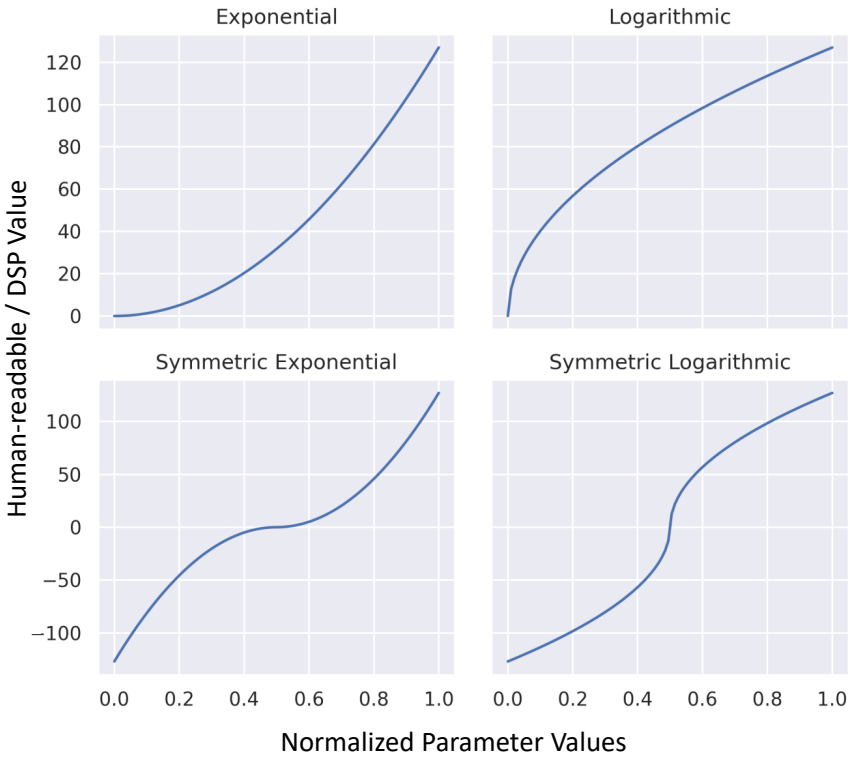


Figure 6.4: Examples of parameter curves used to convert to and from normalized parameter values and the human-readable values used in the DSP algorithms. The top two curves are non-symmetric curves, mapping to values in the range [0, 127]. The bottom two curves are symmetric, mapping to values in the range [-127, 127].

6.4 Evaluation of Auditory Distances

The authors seek to quantify the diversity of sounds that can be generated with torchsynth, given a particular nebula; or similarly, to quantify to what extent a certain nebula can capture the variability of sounds in another dataset. In order to do so, we first need a reliable measure of similarity or dissimilarity between pairs of sounds, also known as an auditory distance.

Auditory distances have many applications in audio ML. They provide the basis for quantitative evaluation and optimization criteria. In a sense, the auditory distance measure is the "ear" of a model. For example, auditory distances can be used to evaluate the similarity of two sounds that were generated by different synthesis engines. By extension, a well-tuned distance could estimate whether two sounds are perceptually indistinguishable to human listeners. For example, a 4-second sinusoid

at 30 kHz, when compared to 4-seconds of silence, should have an auditory distance of zero if the distance is properly tuned to the typical range of human hearing. Likewise, two random instances of white noise should have a perceptual distance of zero, or near zero, despite their entirely different waveforms.

Auditory distances typically involve computing some multidimensional representation of a sound, then computing a distance over the representation space [137]. To perform a controlled evaluation of auditory distances, the authors devised two experiments using two new datasets. Sounds in each dataset are RMS-level normalized using the normalize package.

		Spearman in preset			DCG across presets		
Representation	model choice	mean	Surge	DX7	mean	Surge	DX7
OpenL3 [28]	env, mel256, 6144	**0.821**	**0.746**	0.896	0.880	0.908	0.852
OpenL3 [28]	env, mel256, 6144, normed	**0.821**	**0.747**	0.895	0.809	0.883	0.735
OpenL3 [28]	music, mel256, 6144, normed	0.817	0.732	**0.903**	0.820	0.916	0.724
OpenL3 [28]	music, mel256, 6144	0.813	0.722	**0.903**	0.892	**0.942**	0.842
Coala [43]	dual_ae_c, normed	0.813	0.729	0.896	0.555	0.547	0.564
Coala [43]	dual_e_c, normed	0.811	0.737	0.884	0.569	0.576	0.563
Wavenet [40]	normed	0.810	0.717	**0.903**	0.582	0.591	0.573
OpenL3 [28]	music, linear, 6144	0.808	0.722	0.895	0.874	**0.943**	0.805
OpenL3 [28]	music, mel256, 512	0.804	0.710	0.899	**0.904**	**0.943**	**0.864**
OpenL3 [28]	music, mel256, 512, normed	0.801	0.705	0.897	0.585	0.606	0.564
Wavenet [40]		0.789	0.675	**0.903**	0.835	0.893	0.777
Coala [43]	dual_ae_c	0.776	0.658	0.893	0.748	0.756	0.740
Coala [43]	dual_e_c	0.750	0.630	0.871	0.681	0.710	0.652
MSS [39, 127]	linear+log, [4096 ... 64]	0.792	0.690	0.894	0.543	0.555	0.531
MSS [39, 127]	log, [4096 ... 64]	0.786	0.689	0.884	0.542	0.566	0.518
MSS [39, 127]	linear, [4096 ... 64]	0.658	0.410	**0.905**	0.447	0.343	0.551
Coala [43]	cnn, normed	0.555	0.303	0.806	0.485	0.433	0.537
Coala [43]	cnn	0.552	0.297	0.807	0.714	0.614	0.815

Table 6.2: Performance of representations on experiments defined in § 6.4.3 and 6.4.4. Best scores, and scores within 0.002 of the best, are bold-faced. ℓ_1 distance was used because it outperformed ℓ_2. We sort by mean spearman within a preset.

6.4.1 DX7 Timbre Dataset

Given 31K unique human-devised presets for the DX7[1], the authors generated 4-second samples on a fixed midi pitch (69 = A440) with a note-on duration of 3 seconds. For each preset, we varied only the velocity, from 1–127. This dataset is

[1]We used this clone: `github.com/bwhitman/learnfm`

built on the hypothesis—verified through informal listening tests—that velocity, while carrying no timbral information itself, effects a meaningful, monotonic variation in timbre when it is explicitly programmed into a DX7 patch. Not all DX7 patches are velocity sensitive, and some are more sensitive than others. Sounds that were completely identical—*i.e.* each sample matched with error 0—were removed from the dataset. 8K presets had only one unique sound. The median was 51 unique sound per preset, mean 41.9, stddev 27.4.

6.4.2 Surge Pitch Dataset

The open-source Surge synthesizer is a versatile subtractive synthesizer with a variety of oscillator algorithms: Classic, Sine, Wave-table, Window, FM2, FM3, S&H Noise and Audio Input. To explore another dimension of variability, in this case *pitch*, we used the Surge synthesizer Python API and the 2.1K standard Surge presets. Here we held the velocity constant at 64, and varied midi pitch values from 21–108, the range of a grand piano. Only a small percentage of presets (like drums and sound effects) had no meaningful pitch variation, and thus no perceptual ordering as pitch increases. However, the inclusion of noise in many of these sounds precluded the use of automatic filtering of perceptually indistinct sounds. Therefore, a small fraction of presets are unclassifiable, imposing a uniform upper bound in accuracy across the board for all auditory distances.

6.4.3 Distance Experiment 1: Timbral and Pitch Ordering Within a Preset

In this experiment, the authors measure the ability of an auditory distance to order sounds by timbre, or by pitch, in the DX7 and Surge datasets, respectively. In effect, the experiment is two evaluations in parallel, run on two separate datasets.

The authors sample a random preset with at least 3 unique sounds. For each sound s, we pick a random sound s_l from this preset with a lower rank (using the DX7 set, this would be a sound having the same pitch but a lower velocity; for the Surge dataset this is a sound having the same velocity but lower pitch); and a random sound s_h with higher rank.

For each of s, s_l and s_h, we compute the distance $d(\cdot, \hat{s})$ between this sound and all other sounds $\hat{s}$ in the dataset. While s is the sample of interest, distance measures are

strictly non-negative. Therefore, we seek a concurrent metric to determine whether the compared sound $\hat{s}$ is "above" or "below" s. If the sound $\hat{s}$ is closer to s_l, we determine the sign of the distance to s to be negative. If $\hat{s}$ is closer to s_h, we determine the sign of the distance to s be positive. As a result, we have a signed distance metric comparing the sound s to every other sound in the dataset.

This set of distances is then correlated to the ground-truth index of pitch, or velocity (depending on the dataset). The correlation, here a Spearman rank correlation, reflects the extent to which the signed distance can properly order the dataset by variability in pitch or velocity. One limitation of this methodology for inducing a forced ranking from simple distance is that if, say, $s = 80, s_l = 31, s_h = 81$, and $\hat{s} = 79$, we might judge $\hat{s}$ as closer to s_h and thus above s. We controlled for this by using the same choice for every auditory distance of s_h and s_l given s.

Formally, we estimate:

$$\mathop{\mathbb{E}}_{S\in P, s\in S, s_l, s_h \sim S, s_l < s < s_h} \left[\rho_{\hat{s}\in S} \Big(\text{rank}(\hat{s}), d(s,\hat{s}) \cdot \text{sgn}\big(d(s_h,\hat{s}) < d(s_l,\hat{s})\big) \Big) \right] \tag{6.1}$$

P is the set of presets, S sounds in that preset, and ρ is spearman.

6.4.4 Distance Experiment 2: Determine a Sound's Preset

A good distance measure should have low distance between sounds generated by the same preset. For each trial, we sample 200 different presets. We sample 2 unique sounds from each preset. For each sound, we compute its distance against the 399 other sounds, and then compute the discounted cumulative gain (DCG) [145] of the sound from the same preset, with binary relevance. The DCG is computed for all 400 sounds in the trial. We perform 600 trials.

In the Surge dataset, to control for the helical nature of pitch perception [120], the second sound was always an interval of six semitones (AKA a tritone, *diabolus in musica*) from the first note. This ensured that pitches were close, but avoided similar partials due to overlapping harmonics that could be easily matched.

6.4.5 Evaluation Results

To evaluate the perceptual similarity between two audio samples, we need a good representation to compute distances: a) The multi-scale spectrogram distance has been used in a variety of applications, particularly in speech synthesis [144, 148] but also in music [12, 39, 36]; b) NSynth Wavenet [40] is a Wavenet-architecture trained on NSynth musical notes; c) OpenL3 [28] was trained multi-modally on AudioSet audio and video, on two distinct subsets: music and environmental sounds; d) Coala [43] was trained multi-modally on Freesound audio and their corresponding tags.

The authors experimented with a variety of hyperparameter settings for the representations. The best results are in Table 6.2. ℓ_1 distance was used because it gave better results than ℓ_2 across the board. For Coala and NSynth Wavenet, normalizing improves the spearman scores, but harms the DCG across presets. Normalization had little effect on OpenL3. OpenL3 (music, mel256, 512) achieves the best score on DCG across presets, and its compactness makes it an appealing choice for the remaining experiments in the paper.

6.5 Similarity between Audio Datasets

To evaluate the similarity between two sets of audio samples X and Y, the authors use the maximum mean discrepancy (MMD) [50]. We use the following MMD formulation, assuming X and Y both have n elements:

$$\mathrm{MMD}(X, Y) = \frac{1}{nn} \sum_{i,j=0}^{n} 2 \cdot d(x_i, y_j) - d(x_i, x_j) - d(y_i, y_j) \tag{6.2}$$

MMD allows us to use our chosen distance measure—OpenL3 (music, mel256, 512) ℓ_1—as the core distance d.

For Surge and DX7, we selected sounds with midi pitch 69 and velocity 64. We also generated a set of 4-second samples of white-noise, and used excerpts from the FSD50K evaluation set [44], which is broad-domain audio, trimmed to 4 seconds. From each corpus, we randomly sampled 2000 sounds, to match the size of the smallest corpus (Surge). We performed 1000 MMD trials, each time comparing $n = 1000$ sounds from one corpus to $n = 1000$ sounds from another, randomly sampled each trial. To estimate the diversity within a particular corpus, we evaluated MMD over 1000 distinct 50/50 partitions of the corpus.

MMD	std	corpus 1	corpus 2
4.396	0.123	white	white
21.409	4.729	dx7	dx7
23.732	3.615	FSD50K	FSD50K
24.130	5.251	torchsynth	torchsynth
27.824	9.821	surge	surge
2751.519	80.955	torchsynth	surge
2884.843	67.264	surge	dx7
3001.857	71.888	torchsynth	FSD50K
3637.845	79.265	torchsynth	dx7
4756.952	112.705	surge	FSD50K
7413.105	111.897	dx7	FSD50K
13202.202	61.558	white	FSD50K
16985.319	92.992	white	torchsynth
18488.926	67.277	white	surge
20374.929	78.886	white	dx7

Table 6.3: MMD results comparing different audio sets, including the stddev of the MMD over the 1000 trials.

Table 6.3 shows the result of average MMD computations between different audio corpora. 0.0 would be perfectly identical. Some results are expected, whereas some are counter-intuitive and suggest pathologies in the OpenL3 distance measure. These results are sometimes perceptually incoherent, and suggest that existing auditory distance measures will impede progress in automatic synthesizer design, as we will illustrate in the following section.

- White-noise is the most similar to itself of all comparisons.
- FSD50K broad-domain sounds are, strangely, considered to have less within-corpus diversity than torchsynth or Surge sounds. However, the variance is high enough that it is hard to have statistical confidence in this unexpected result.
- More troubling are low-variance estimates that torchsynth is more similar to FSD50k than a dx7 synth. *A priori*, one would expect that synths would sound more similar to each other than broad domain audio.
- As expected, white noise is the least similar to DX7 synth sounds of all corpora, as the DX7 has no noise oscillator.

6.6 torchsynth Hyper-Parameter Tuning

Given its highly flexible architecture, how can we guarantee the maximum diversity of sounds within the default nebula? Similarly, to what extent can torchsynth adopt the characteristics of a given corpus of audio? Recall from §6.3 and Figure 6.4 that for each module parameter, the choice of scaling curve is a hyperparameter. Initial hyperparameters were chosen perceptually and based upon prior-knowledge of typical synth design.

In principle, we can use MMD (Equation 6.2) as an optimization criterion to tune these hyperparameters a) to maximize sonic diversity; or b) model the characteristics of another dataset. We use Optuna [1], initializing with 200 random grid-search trials, and subsequently using CMA-ES sampling for 800 trials. In each trial, we generate 256 random torchsynth sounds with the Optuna-chosen hyperparameters. Hyperparameter curves were sampled log-uniform in the range [0.1, 10]. The top 25 candidates were re-evaluated using 30 different MMD trials, to pick the best hyperparameters. However, MMD estimates are only as good as the underlying similarity metric (OpenL3-ℓ_1) that it uses.

For these experiments, the authors and non-author musicians conducted blinded listening experiments of the tuned nebula and our manually-chosen nebula, and listened to 64 random sounds. Only after independent qualitative evaluation did we unblind which nebula was which.

6.6.1 Restricting hyperparameters

Many torchsynth 1.0 Voice default nebula sounds have an eerie sci-fi feel to them. To find the drum nebula, we used Optuna to choose hyperparameters to *minimize* the OpenL3-ℓ_1-MMD against 10K one-shot percussive sounds [106]. All hyperparameters were allowed to be tuned. We had hoped to find that OpenL3-ℓ_1-MMD would find appropriate percussive curves.

Overall, the authors found the drum hyperparameters unpleasant to listen to. This negative result was surprising. Sounds did not resemble percussion. There was extreme use of high and low pitch. Low pitches were clicky and gurgly, high pitches were painful or often inaudible. There was some nice use of LFO, but little use of square shape, little noise, and low diversity. Perhaps wide modulation sweeps attempted to compensate for the broadband energy in the transients of drum sounds.

The authors were curious if this negative result was due to failure of the distance

measure, or instead a systemic limitation in the design of the torchsynth 1.0 Voice and its parameter sampling approach. We hand-tuned the hyperparameters to create a drum nebula, which is shared as part of our repository. While not all the sounds produced sound like drum hits, many have a quality akin to early drum machines—the distribution of sounds is overall much more percussion-like. We encourage the reader to listen to this nebula, which will be available on torchsynth site.

In addition to confirming the perceptual-deafness of our distance measure, hand-designing the drum nebula demonstrated one limitation in torchsynth 1.0: Synth parameters are all sampled independently. Thus, one cannot construct a nebula that samples only kicks and snares. Sampling occurs on the continuum between them. In future work, we are interested in investigating multivariate sampling techniques, which would allow more focused cross-parameter modal sound sampling.

6.6.2 Maximizing torchsynth diversity

The authors attempted to tune our hyperparameters to maximize torchsynth MMD, i.e. increase the perceptual diversity of sounds generated by torchsynth itself. As before, Optuna was used to choose hyperparameters that maximized the OpenL3-ℓ_1-MMD and thus increase the diversity of sounds. Nonetheless, the "optimized" nebula exhibited pathologies in pitch, favouring extremely low and high pitches. We hypothesize that OpenL3-ℓ_1 overestimates perceptually diversity in these frequency ranges. We performed numerous experiments restricting the hyperparameters Optuna could and could not modify, such as prohibiting changes to midi f0 and VCO tuning and mod depth. Consistently, listeners preferred our manually design nebula to automatically designed ones in blind tests. We consider this another important negative result. Open questions remain:

- For what hyperparameter choices is a particular auditory distance perceptually-inaccurate?
- How do we craft an auditory distance measure that can *perceptually* optimize synthesizer diversity, or similarity to an existing sound corpus?

6.7 Open Questions, Issues, and Future Work

Many experiments in automatic synthesizer design hinge on having a perceptually-relevant auditory distance measure. The distance measure is the artificial "ear" of the network. OpenL3 (music, mel256, 512) ℓ_1 performed well on our quantitative synthesizer experiments (Table 6.2), but exhibited many issues in qualitative listening tests, in particular its insensitivity to extreme pitch and inability to model percussion.

Learning a perceptually-relevant auditory distance measure is an open research question. Manocha *et al.* [89] use manually-annotated "just noticeable differences" (JND) trials generated using active learning to induce a perceptual distance measures. However, they only work with speech and do not include pitch variations, so their model was inappropriate for our task.

Unknown pathologies in auditory distance measures impede researchers from performing a variety of useful experiments. Most crucially, the lack of perceptually accurate auditory distance measure prevented us from precisely estimating how many perceptually different sounds are expressible by torchsynth, as well as other synthesizers like Surge and DX7. By contrast, a good artificial "ear" for music opens the door to many possible advances in synthesizer design, including:

- Estimating and maximizing the diversity of synthesizer.
- Mimicking existing synthesizers through automation.
- Inverse synthesis, transcription, and the other tasks described in §6.2.

Our negative results on automatic synthesizer design using auditory distances present valuable challenges for the community to investigate.

Nonetheless, the potential impact (§6.1 and §6.2) of the synth1B1 corpus is unaffected, because of its enormous size, speed, and corresponding multi-modal latent parameters.

6.8 Future Work

torchsynth 1.0 focuses on high throughput and creating a (subjectively) perceptually diverse synth1B1 dataset. There are a handful of improvements we want to add to torchsynth:

- Stress-tested differentiable modules.

- Subtractive filters.
- Additional architectures including an FM synthesizer.
- Multivariate parameter selection.
- High-throughput modules that resemble human speech.
- A standardized modular architecture for high-throughput audio *effect* research.

6.9 Conclusions

This chapter described synth1B1, a multi-modal corpus of synthesizer sounds with their corresponding latent parameters, generated on-the-fly 16200x faster than realtime on a single V100 GPU. This corpus is 100x bigger than any audio corpus present in the literature. Accompanying this dataset is the open-source modular GPU-optional torchsynth package. The authors hope that larger-scale multi-modal training will help audio ML accrete the benefits demonstrated by previous NLP and vision breakthroughs.

The authors freely release pitch and timbre datasets based upon "natural" synthesis sounds, and novel evaluation tasks on which we benchmark a handful of audio representations. This chapter also presented several novel research questions, including how to estimate and maximize the diversity of a synthesizer, as well as how to mimic existing synthesizers. Additionally, this chapter outlined issues and open research questions that currently impede this sort of experimental work, in particular demonstrating negative results of auditory distance measures.

Chapter 7

Designing an Exploratory Synthesizer Interface

In this chapter the author introduces Synth Explorer: a prototype of an automatic synthesizer programming interface built on top of the torchsynth synthesizer described in the previous chapter. Synth Explorer is an interactive visual browsing interface for torchsynth that was built with the goal of supporting both novice and expert users in the process of navigating and developing an understanding of the vast and complex sonic space expressed by a synthesizer. It builds upon previous research in the area of sound visualization, and the design is informed by concepts from the the fields of creativity support tools (CSTs) and music interaction [125, 60]. These concepts, along with the taxonomy of automatic synthesizer programming interaction approaches presented in chapter 3 grounds the the development of Synth Explorer and provides a framework for the development of future systems.

Synth Explorer is designed with music producers, audio practitioners, and synthesizer users in mind – all individuals who regularly require synthesized sounds for their creative projects. Currently, users must manually program a sound, find an existing preset, or find a pre-existing sound in a collection of samples. Challenges with manually programming sounds is addressed in detail in chapter 2. Searching for presets and pre-existing sounds typically involves browsing through items that are displayed in a list-based user-interface. Any searching is based on filenames or semantic tagging [77]. In conversations held by Kristena Andersen at the RedBull Music Academy [2], music producers expressed the challenges associated with navigating large collections of audio and their desires for improved methods for interaction. Particular emphasis

was placed on the potential impact of tools that aid the creative process. The field of creative music information retrieval has also had the focus of finding ways to address the challenges of searching for sounds [65] and visual browsing interfaces have been identified as beneficial for allowing users to navigate large collections of sounds more efficiently than traditional list based methods [139].

The interface implemented in Synth Explorer is intended to support the efficient navigation of synthesizer sounds, allowing users to develop a "sound palette" for their project. Users are able to "look under the hood" of the visual interface and directly interact with the underlying control interface for these sounds. They can make fine adjustments to the parameter settings to refine their selected sounds as well as to gain insight into the parameter settings that led to the resulting sound. In this way, users can interact with the synthesis engine and control interface at a level abstraction of that supports their position on the learning curve. The graphical interface is designed to support novice users. While the interface includes some technical audio terms, the drag and drop interface is modelled on an interaction modality that will be familiar to most computer users and will allow anyone to quickly start constructing visualizations and exploring sounds. The inclusion of the technical names will allow users to begin to build up an understanding of the different dimensions of sound and how they relate to the underlying synthesizer parameters.

7.1 Related Work

Exploration-based interfaces for synthesizers was reviewed in chapter 3 and visualization of sounds based on sound similarity has been studied in depth and is reviewed by Cooper *et al.* [27]. Within the field of automatic synthesizer programming, Synth Explorer is related to the data-driven approach used by in SynthAssist [16]; all the sounds used in Synth Explorer are pre-computed and audio features are stored in a database. Synth sounds are then displayed on the interface based on sound similarity. Looking beyond synthesizer sounds, work that is particularly relevant to Synth Explorer is DrumSpace, a 2D visualization interface developed by Turqois *et al.* for exploring large collections of drum sounds [139]. A component of their study was a subjective evaluation that compared user experiences in browsing for drum sounds using a traditional list-based layout of samples to 2D visual layouts. For the 2D layouts they compared one method that automatically sorted sounds based on sound similarity against one that organized sounds based on the filename. Users reported

having a significant preference for for the 2D based layout, but had no significant preference for the type of layout. Participants were able to explore a much larger number of audio examples in a shorter period of time when using the 2D interfaces. Turqois *et al.* also identified that the layout based on sound similarity caused some confusion to the users; the layout was based on a set of audio features which had been reduced to 2D using a dimensionality reduction technique called t-SNE [141], because algorithms like t-SNE create complex combinations of higher dimensional features into a lower dimensional embedding for visualization, the meaning of the resulting dimensions used for visualization are often hard to identify. Turqois *et al.* also suggest that colour could be a helpful addition to the interface. Building upon this work, the author of this thesis conducted further experiments on the perceptual relevance of different methods of dimensionality reduction techniques for visualizing drum samples in 2D and identified MDS as having the highest correlation with user similarity ranking [123]. Synth Explorer builds on these tools and leverages similar techniques for creating 2D visualizations of synthesizer sounds.

An interesting CST that is related to the audio synthesis problem is a tool developed by Andrews for exploring and generating graphics using a 2D interface [3]. Their work used an algorithmic method for generating 2D graphics, which a user is able to interact with to control the generation of new graphics. Because of the vastness of the space of potential images that could be generated by the algorithm, the user interface utilizes a spatial layout of images to allow the user to quickly assess a set of possibilities which they can then use to guide the generation of further images. While this interface was focused on the generation of images, the interaction ideology is similar to that of Synth Explorer, which is built on top of an algorithmic audio synthesis engine; the space of possible sounds that a synthesizer can produce is far too vast for a user to navigate all at once, so a small random subset of that space is initially presented and the user is then able add more sounds and hone in on a particular result.

7.2 Design Principles

This section presents an overview of design principles, informed by the HCI fields of music interaction and creativity support, that are intended to guide the development of automatic synthesizer programming interfaces. These principles provide a framework for designing interactions from the perspective that a synthesizer is a musical

tool that users *want* to engage with in the context of a creative pursuit – whether that is designing sounds for a film soundtrack, composing a piece of electronic music, or simply enjoying creating sounds and playing a synthesizer. Because of the breadth of what a creative pursuit can encompass, these principles are intended to be taken as suggestions to help situate the design process.

7.2.1 Music Interaction

Music and Human-Computer Interaction [60], or simply Music Interaction, is the field of research related to the use of interactive systems that involve computers for any kind of musical activity. Music interaction not only draws heavily from other areas of HCI research, but also responds to the needs and desires of the music community. There are unique considerations that make music interaction different from other fields of HCI. A musical instrument is not a utilitarian tool whose development should be ever-improved and made more efficient. Musical instruments are played; sometimes that is the only goal. Tanaka [131] identifies that imperfections and limitations of a musical instrument give an instrument character. This was reflected by music producers interviewed by Andersen who expressed the role of serendipity and "happy accidents" in their creative process [2], and identified this as an important consideration for designing new music production tools. McDermott [92] identifies the importance of engagement in musical interaction and the relation that bears to the concept of *flow*. Mihaly Csikszentmihalyi coined the term *flow* to describe a state of highly focused concentration that is related to experiencing an activity as being deeply satisfying and engaging [30]. The role that the learning curve plays is crucial to the level of engagement that a player experiences when playing a musical instrument, both in the short-term and the long-term. Holland [60] concludes, "In order to remain engaging, consuming and flow-like, activities that involve musical instruments must offer continued challenges at appropriate levels of difficulty: not too difficult, and not too easy."

Automatic synthesizer programming tools can be thought of as being extensions of a musical instrument. With this in mind, the goal should not necessarily be to provide a perfectly optimized experience that completely takes over the task of programming and using a synthesizer. For example, example-based inverse synthesis approaches may play an important role in an automatic synthesizer programming tool; however, they may only be one part of a larger interaction paradigm that pro-

vides other opportunities for expression. The inclusion of design features that allow for "happy accidents" to occur and unexpected use-cases to be realized can lead to more engaging and rewarding experiences. In fact, there is a rich history in music technology of musicians using devices in an *incorrect* way with great success. The Roland TB-303 is an excellent example of this: the synthesizer was a failure in terms of its initial goal of generating realistic bass sounds. However, its synthetic "squelchy" tones resulted in it leading a successful second life as an electronic dance music instrument [143]. It is impossible to design an interface with the unexpected use in mind; however, leaving enough room for features to be used in unexpected ways is a method to encourage long-term engagement. In addition, related to the concept of engagement is the consideration of the role that the learning curve plays while using a synthesizer. Building in interactions that support users as they progress along the learning curve will also encourage both short-term and long-term engagement.

7.2.2 Creativity Support

Related to music interaction is the study of creativity support tools. Design guidelines borne out of research in creativity support tools is relevant to the design of tools that support synthesizer users. Shneiderman [125] outlines a set of design principles for developing creativity support tools which include the following: support exploratory search; enable collaboration; provide rich history keeping; and design with low thresholds, high ceilings, and wide walls. Davis *et al.* focus on the role that CSTs play in supporting novices engaging in creative tasks and the relationship that the environment plays in creativity [32]. In their work, the authors identify two types of novice users: domain novices and tool novices. Domain novices are new to both the creative domain as well as using the creativity support tool. Tool novices have experience with the creative domain, but are novices at using a particular tool. To help evaluate and promote the development of creativity support tools for novices, they also propose a theory of creativity support based on three cognitive theories: embodied creativity, situated creativity, and distributed creativity.

Embodied creativity is based on the premise that creativity is intrinsically linked to the interaction that a user has with their environment. It is through interacting with their world that an individual is able to make creative ideas more concrete and express themselves. Chapter 3 of this thesis reviewed various approaches to automatic synthesizer programming, which included six different interaction paradigms:

evaluation interfaces, use of descriptive words, vocal imitations, exploration interfaces, example-based interfaces, and intuitive controls. These represent some of the ways in which a system might support a user in expressing and developing their creative ideas when using a synthesizer. An interaction may including one or more these paradigms, but is not limited to this specific set of interactions.

Situated creativity is related to the concept of flow. In the context of creativity support, situated creativity is linked to how much effort a user must apply when using a tool to carry out a task. As a user becomes more comfortable with a tool, it gradually starts to feel like a natural extension of their body and they are enabled to explore deeper expression of their creativity. This is related to the learning curve and level of engagement that a musical interaction is able to support, which is discussed in the previous section.

Distributed creativity is focused on the tasks that a human can offload to a particular tool during a creative task. By handing over a portion of a creative task to a support tool, a novice user may be able to arrive at rewarding results earlier in the process, thereby motivating them to continue to engage in the creative process and enhance their skills. A large portion of the previous work in automatic synthesizer programming has been dedicated to the development of algorithms that automate the task of programming a particular sound in a synthesizer. These methods provide opportunities for distributing the creative load, and may be especially helpful to novice users who are at early stages of the learning curve.

7.3 Synth Explorer Design

Synth Explorer is designed to support exploration and provide a low threshold of entry for users to begin working with synthesized sounds within a creative context. It is an exploration-based interface that overlays a synthesizer and provides a visual representation of sounds generated by that synthesizer. The visual interface is aimed at tool novices [32]: users who may have expertise in composing music and working with digital audio workstations, but may have limited experience in working with audio synthesizers. To support progression along the learning curve, users are able to explore and adjust the underlying synthesizer parameter settings for any sound to help them develop their understanding of programming.

Synth Explorer is expected to be an accessory tool used during the creative process of composing digital music in addition to a users' main digital audio workstation. For

example, a user might be composing a film soundtrack in their workstation of choice and find that they are desiring to add a new synthesized instrument to their score. In this example, the user would open up Synth Explorer, browse for a sound that fits their needs, and then load those sounds back into their workstation. While navigating between two different programs may seem like an impediment to the creative flow, it is hypothesized that the added benefits of the Synth Explorer interface will outweigh the inconvenience of switching between programs. Additionally, future iterations of the tool could be implemented as a VST plugins as well as host VST synthesizer plugins to support better integration with current music production processes.

The user interaction methodology was designed according to the theories of cognition presented by Davis *et al.* for supporting novice users working with CSTs: embodied creativity, situated creativity, and distributed creativity [32]. The design of Synth Explorer in relation to these three aspects is presented below.

Embodied Creativity

The browsing interface should support embodied creativity – the layout and interaction with the browsing interface should use multiple modes of sensory cognition to facilitate understanding. Spatialization of multimedia objects based on similarity has been used in previous related work [3] and is used here to provide deeper insight into the relationship between sounds. Based on the previous issues identified with 2D visualizations of sound, which caused confusion to users [139], the user here is given control over how the visualization is constructed to empower them to develop an understanding of the spatial relationship between sounds. Specifically, users are able to assign audio features to the x, y, and colour of each sound object on the 2D layout.

Situated Creativity

Synth Explorer should support a user in maintaining creative flow while working on their project. Additionally, the user interface should have a low threshold of entry to enable novices to quickly start engaging with sounds and working towards realizing their creative goal. The conscious effort required to use Synth Explorer should quickly dissipate into the subconscious, allowing the user to instead focus the entirety of their attention on the creative task at hand. At the same time, Synth Explorer should implement features that support longer-term engagement by challenging users as they

progress along the synthesizer programming learning curve. The visual exploration interface is intended to support short-term engagement aimed at novice users – while still remaining enjoyable for more experienced users – and to support more depth and longer-term engagement by allowing users to modify synthesizer parameters directly on a separate interface.

Distributed Creativity

Synth Explorer should offload the laborious task of organizing large collections of sounds and navigating the sonic space represented by a particular synthesizer. The system should also not take full control over the creative process and provide the user with room to express creative control. In a traditional synthesizer, a user would need to learn how to program sounds themselves, rely on experts to create and organize presets, or organize presets themselves. Applying distributed creativity to Synth Explorer, the user should offload the task of programming and organizing synth presets over to the system and shift their energy to the task of curating a selection of sounds that fit the needs of their creative vision.

7.4 Implementation

7.4.1 Sound Generation and 2D Mapping

The core of Synth Explorer is the synthesizer. The torchsynth synthesizer introduced in §6.2.2 was used for this prototype system. The default *Voice* architecture for torchsynth was used, which is a basic two oscillator + noise synthesizer with multiple modulation sources and a modulatable amplifier. Sounds are generated from a subset of the synth1B1 dataset (see §6.2.1) and the resulting four second audio clips and parameter settings are saved. Audio features are computed on each audio clip and the dimensionality reduced in order to visualize these sounds in two dimensions based on sound.

Audio features are designed to capture perceptually relevant aspects of audio in a compact format. In DrumSpace, the 2D visualization of samples was produced by computing a set of audio features and then performing dimensionality reduction to two-dimensions using t-SNE [139, 141]. In subsequent work, the UMAP dimensionality reduction algorithm [95] was found to be an effective alternative to t-SNE in terms of time-complexity and visualization quality [71]. Synth Explorer uses two

different UMAP embeddings as options for visualization: an embedding based on mel-frequency cepstral coefficients and an embedding based on spectral features [104]. In addition to these dimensionality reduced embeddings, a set of eight low-level audio features computed using librosa[1] are included. In addition to audio features, parameter values are also included as features that can be visualized. This supports users in beginning to build up an understanding of the complex relationship between the parameter space and auditory space. The parameter setting (preset) for each sound comprises 78 individual parameter values. UMAP embeddings are computed on the parameter settings to create two dimensional parameter features for visualization. Additionally, the keyboard pitch is included.

The database of audio files and associated features allows the synthesizer sound space to be automatically organized based on these features. This addresses the distributed creativity component of the design. The user offloads the task of generating synthesizer sounds and organizing them to the system. Due to the speed of the synthesis system and audio feature extraction process, nearly ten thousand unique sounds can be generated and analyzed in about 20 minutes. This corresponds to over ten hours of synthesized audio. Listening to and organizing all these sounds would be impractical – or at least extremely mundane – for any user, whether they are a novice or expert.

7.4.2 Browsing Interface

Using the guiding cognitive theories for developing creativity support tools, an interface was designed based on the aforementioned related work. A drag and drop interface, which is an interaction paradigm common to consumer computer user interfaces, is employed, that allows users to assign different features to the dimensions of the visualization. Instead of forcing a layout on the end user, the user has the ability to decide which feature / embedding they want to use to construct their visualization. This decision was made based on participant feedback that they were confused by the sound similarity visualization used in DrumSpace [139]. An attempt is made to address this issue by explicitly providing the user with a set of features ranging from concrete (keyboard pitch) to more abstract (spectral embedding using UMAP), and allowing them to decide which they want to use to create the visualization. Additionally, tooltips are provided for each feature which provides an opportunity for the

[1] `https://librosa.org/doc/latest/index.html`

user to learn more about the underlying dimensions and their relationship to audio perception.

The final interface is shown in figure 7.1. The workflow of the interface is divided into four different sections: adding sounds, constructing the visualization, exploring and saving sounds, and downloading saved sounds. An overview of each step in the workflow is provided in the following sections.

Figure 7.1: Synth Explorer User Interface

Adding Sounds

The space of possible sounds that can be produced by a synthesizer is vast. Ten thousand four second audio clips of patches from *torchsynth* represents only a subset of the possible sounds that torchsynth is capable of producing. That being said, presenting a user with ten thousand audio clips on a 2D layout would be overwhelming. The user is initially presented with a small random subset of possible outcomes. The user is given the option to control how many sounds are added to the visualization at once using a slider control that ranges between 1 and 250. They are then able to add that many sounds to the their visualization by dragging and dropping the UI object representing a particular synthesizer onto the visualization area. In this way, they can explore the sound space in an iterative fashion. This section of the user interface is shown in figure 7.2. Currently only one synthesizer is shown in the interface; however,

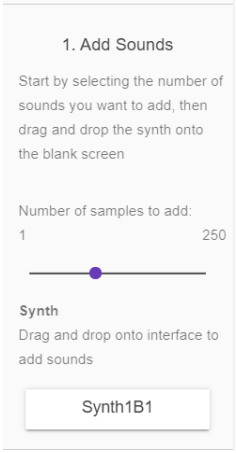


Figure 7.2: Synth Explorer Step 1

any number could be included. This would allow the user to compare and explore multiple synthesizers at any given time.

Once a user drops a synthesizer onto the visualization, the requested number of synthesizer sounds are randomly selected from the database and added to the interface. A single sound is represented as a single point on the 2D visualization.

Constructing the visualization

The next section of the user interface is focused on allowing the user to control their visualization. Specifically, this section allows users to decide which features are assigned to which dimension of the visualization. The available features are shown on the left side of the user interface in figure 7.3. Using the same drag and drop paradigm, the user is able to drag any of the features onto the visualization surface in order to modify the layout of audio samples. When the user initiates a drag and drop interaction, large drop areas representing the different dimensions of the visualization appear for the user to choose from. The available dimensions are the x-axis, y-axis, and the colour of the points. Once a feature is dropped onto one of the dimensions of the visualization, the points representing the sounds automatically shift into the new position to reflect the changes. Any feature can be associated with any dimension, allowing the user to explore the relationship between the features and develop a visualization that meets their needs.

A decision was made to use the technical names of the features. While this may be confusing to some users, there are unfortunately not many good alternatives to this problem. Tooltips are provided to give novices an approximate non-technical

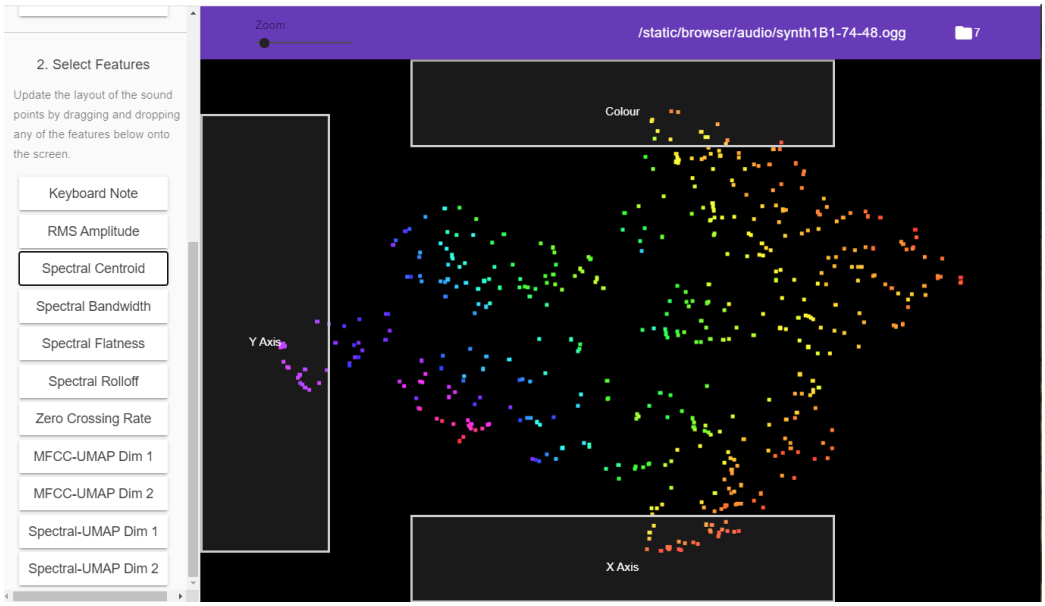


Figure 7.3: Synth Explorer UI: Updating features

description of the feature – the hope is that these users will still feel inspired to explore these different features and learn how they relate to the auditory dimensions of sounds.

Exploring and Saving Sounds

The majority of the space on the user interface is occupied by the 2D browsing space, which is shown in figure 2. This is the space where synthesizer sounds are represented as points on a scatter plot. To listen to a particular sound, the user simply hovers their cursor over the point representing a sound. They receive both auditory and visual feedback once they intersect with a point. The sound represented by the point plays and a small circle burst the same colour as the point is animated into view at the point of interaction. Users can also zoom into the visualization using their mouse scroll wheel and navigate the space using a click and drag operation. Each of these gestures is meant to reflect operations that would feel intuitive and natural to an experienced computer user.

This mode of interaction provides a fast way for the user to preview a large number of samples. The goal of the 2D layout and exploration method is to provide the user with an embodied method for exploring sounds. The 2D spatial layout using colours provides users with a visual representation of the sonic space that they can interact

with and create their own associations with. The interaction itself is easy to master as it simply requires dragging a mouse around the screen and listening to sounds. Modifying the layout using drag and drop interactions also requires little conscious effort. These intuitive interactions were implemented to support situated creativity and thereby aid creative flow.

When a user has listened to a sound that they like and want to save, they can press the "s" key on their keyboard to save that clip to a "sound palette". This is equivalent to adding an item to an online shopping cart. The user can then continue to explore sounds.

At this point the user is able to repeat any of the preceding steps and continue to explore the visualization: add more sounds, modify the dimensions with different features, explore the space, adjust parameters for a specific sound, and then save the resulting sounds.

Adjusting Parameters

At any point during exploration users can move to a secondary interface and work directly with the synthesizer parameters. This interface, shown in figure 7.4, is essentially a regular control interface for the torchsynth synthesizer that contains slider controls to update the values for all 78 of the parameters in the torchsynth Voice. These 78 parameters are grouped by the specific module that they control (see 6.3 for a diagram of all the modules in Voice). When navigating to this screen, the parameter settings for the sound that was selected from the visual exploration interface will be showing; this allows the user to gain insight into the specific parameter values for that particular sound, as well as to make modifications on the sound.

Downloading

Once the user has sufficiently explored the sonic space and saved a palette of sounds they would like to use, they can click on the download icon next to their saved sounds and download all the synthesizer sound files they have saved. Once they have done this they are free to use those sounds for whatever creative task they would like.

Figure 7.4: Interface for adjusting the parameter values for a synthesizer patch, allowing for fine-tuning of sounds found on the visual exploration interface

7.4.3 Technical Implementation Details

Synth Explorer is implemented as a web application. The Django framework[2] was used to implement the backend of the web app. Django is written in python and provides an elegant data-model system for interacting with a database such as MySQL[3]. The sound generation and analysis portion of the application is written as a Django management command – when this command is run, a set of sounds is rendered using *torchsynth* and analyzed using librosa and UMAP. Once the samples have been analyzed and audio files saved to disk, the audio features and patch settings for each sound is saved in a MySQL database.

The frontend of the application is written using HTML, CSS, and JavaScript. The foundation of the application was based on code developed by Leon Feddden [4]. This code was modified to function within the Django framework and the user interaction paradigm was modified to support dynamic adding of synthesizer samples and user construction of the layout using drag and drop interactions. The visualization and animation is rendered using three.js[5].

[2] `https://www.djangoproject.com/`
[3] https://www.mysql.com/
[4] `https://github.com/fedden/umap_tsne_embedding_visualiser`
[5] `https://threejs.org/`

7.5 Evaluation

7.5.1 Creativity Support Index

The creativity support index (CSI) [21] is a psychometric survey that was designed to quantify the ability of a tool to assist a user with a creative task. The CSI evaluates a creativity support tool based on six different criteria: Exploration, Expressiveness, Immersion, Enjoyment, Results Worth Effort, and Collaboration. The CSI is structured as a questionnaire with two different sections. The first sections contains 12 questions, shown in figure 7.5, which participants answer with a score from 1-10 ("Highly Disagree" to "Highly Agree"). The second section contains 15 questions which compares each of the evaluation criteria pairwise and asks the participant the evaluate which criteria was better supported. For example, each question begins with "When doing this task, it's most important that I'm able to..." followed by two statements, where one statement is related to one of the evaluation criteria and the other is related to another. Participants are asked to select only one of the statements and each evaluation criteria is ranked against all the others. Based on the scores for these questions a CST is given a score out of 100. Synth Explorer was evaluated informally on two separate tasks that were designed to replicate a typical synthesizer use case in the context of music production.

Task 1: Synthesizer browsing for an existing project

In this task, the user is working on an existing musical project in a separate workstation. They are composing a piece of music and then are asked to find a new synthesizer sound for an additional track to their composition. They must leave the workstation, open Synth Explorer, browse for a new sound, then leave Synth Explorer and open the new sound back in the workstation they were initially working within.

Task 2: Creating a sound palette for a new project

In this task the user is beginning a new project and is searching for a set of synthesizer sounds to create the sonic palette for the new composition. They may have a particular sound in mind, but they are asked to explore the interface in an open-minded way to look for new sounds and create a collection as inspiration to start the new project.

Collaboration 1. The system or tool allowed other people to work with me easily. 2. It was really easy to share ideas and designs with other people inside this system or tool.
Enjoyment 1. I would be happy to use this system or tool on a regular basis. 2. I enjoyed using the system or tool.
Exploration 1. It was easy for me to explore many different ideas, options, designs, or outcomes, using this system or tool. 2. The system or tool was helpful in allowing me to track different ideas, outcomes, or possibilities.
Expressiveness 1. I was able to be very creative while doing the activity inside this system or tool. 2. The system or tool allowed me to be very expressive.
Immersion 1. My attention was fully tuned to the activity, and I forgot about the system or tool that I was using. 2. I became so absorbed in the activity that I forgot about the system or tool that I was using.
Results Worth Effort 1. I was satisfied with what I got out of the system or tool. 2. What I was able to produce was worth the effort I had to exert to produce it.

Figure 7.5: Agreement questions for the creativity support index

7.5.2 CSI Results

Results of the CSI evaluation are shown in table 7.1. These results show that Synth Explorer supported task 1 to a greater degree than task 2. Task 1 received an overall score of 78.67 whereas task 2 received an overall score of 71.67. The results showed that collaboration was not supported and received a score of zero for both tasks. This makes sense considering the nature of the tasks and the tool itself. Collaboration was not a part of the evaluated tasks. However, the tool itself does not currently support collaboration in any meaningful way other than allowing two users to sit next to each other and browse for synth sounds simultaneously. The tool supported exploration in both tasks more than any of the other attributes evaluated. This result is positive considering that was one of the major goals for the tool. Another area that was well supported by the tool is enjoyment, while the results and expressiveness of the tool itself were not highly rated. The lack of expressiveness also makes sense for the tool; Synth Explorer itself does not necessarily allow expression – it allows a user to explore sounds with the goal of maintaining a creative flow in a large creative context, despite it being more challenging to be expressive with the tool.

The results section of the evaluation did not score highly either. This was due

to the fact that the 2D layout of sounds was still challenging to navigate and make sense of for users. This is in part due to the underlying method for generating sounds as well as the sound mapping techniques. Randomly sampling a synthesizer is not the best way to capture meaningful sounds from the synthesizer and a lot of times produces dramatic or unusable results. Additionally, capturing sound similarity in two dimensions is still an open question that requires further work. Despite this, the interface was effective at enabling rapid exploration of a large number of sounds and it was enjoyable getting to explore manipulating the visualization by dragging and dropping the different features onto the plot.

Area	Q1	Q2	T1	T2	T1 Total	T2 Total
Collaboration	1	1	0	0	0	0
Enjoyment	7	8	5	4	75	60
Exploration	10	7	5	5	85	85
Expressiveness	5	4	1	2	9	18
Immersion	8	7	3	2	45	30
Results	5	6	2	2	22	22
Total Score					**78.67**	**71.67**

Table 7.1: Results of creativity support index questionnaire.

7.6 Future Work and Conclusion

This chapter has introduced the Synth Explorer creativity support tool. The intention of this tool is to support users in the process of working with synthesizers and finding new sounds for creative projects. The development of Synth Explorer was based on a set of design principles informed by the fields of music interaction and creativity support tools. A further goal of the tool is to support novice users who might have experience in music production, but do not have experience working with synthesizers. Synth Explorer was designed to support novices using an approach to creativity support based on cognitive theory which emphasizes embodied, situated, and distributed creativity. The designed tool uses a 2D visualization of synthesizer sounds to arrange sounds spatially and uses colours to support an embodied approach to exploration. By using a simple drag and drop interface and browsing of sounds using a visual layout, users are able to quickly start exploring and remain engaged in their creative task. A secondary interface containing the individual parameter values allows users to engage in manual synthesizer programming using a sound from the

visual interface as a starting point. This encourages long-term engagement through the inclusion of a more challenging interaction paradigm that provides a user with more depth of control.

An evaluation of the tool using the creativity support index revealed insight into the strengths and weaknesses of the tool and helped to identify areas for potential further work. The most glaring limitation of the tool is that it lacks support for collaboration; however, collaboration support could be added relatively easily based on the implementation. Since Synth Explorer is built as a web application using the Django framework, a user login system could be easily added to allow users to save and share collections of sounds that they have curated. Different synthesizers and layouts could also be shared through a similar system. Another limitation of the current system is in the available options to filter and search for sounds. Currently, users are limited to adding random samples to the interface and exploring different configurations of that. In order to support users in arriving at more useful results and collections of sounds, future iterations could provide additional search options, such as a search by similarity, or additional tools for selecting and filtering the current selection. Integrating more advanced sound searching methods, such as an example-based interaction paradigm using one of the approaches discussed in chapter 5, would also benefit future development. Providing support for playing the sounds using a real-time keyboard interface could also encourage more engagement and help the tool feel more like an extension of a musical instrument.

Chapter 8

Conclusions

This thesis has explored the topic of automatic synthesizer programming and has presented work to support continued research in this field. The main research question that was asked at the beginning of this thesis was: "How can designers of synthesizer programming interfaces enable more people to be more creative more often?" Automatic synthesizer programming seeks to answer this question through research focused on providing more intuitive methods for users to communicate their ideas to their synthesizers.

Chapter 2 provided a background of synthesizers and discussed the specific challenges associated with synthesizer programming. The core of these challenges arises from the disconnect between the parameters used to control a synthesizer and the associated auditory result. The conceptual gap created by this disconnect is large and, accordingly, synthesizer users face a steep learning.

Automatic synthesizer programming research has explored a number of methods for aiding users to more easily translate their desired audio results into synthesizer parameters. Chapter 3 provided a survey of this research. Six different automatic synthesizer user interaction styles were identified and reviewed: 1) example-based interfaces, 2) evaluation interfaces, 3) using descriptive words, 4) vocal imitations, 5) intuitive controls, and 6) exploration interfaces.

Inverse synthesis, also referred to as sound matching, is an example-based automatic synthesizer programming method and has been a major focus of research in this field. The goal of inverse synthesis is to find a parameter setting for a particular synthesizer that matches a target sound as closely as possible. Chapter 3 reviewed some of the main approaches to inverse synthesis, including genetic algorithms and deep learning. Chapter 4 presented an open source software library that was devel-

oped as a part of this thesis to encourage shared evaluations and reproducible research [142]. This library, which was named SpiegeLib, contains all the code necessary to conduct inverse synthesizer experiments using VST software synthesizers and serves as a repository for sharing methods.

Chapter 5 discussed an inverse synthesizer experiment that was conducted using an open-source emulation of the Yamaha DX7 synthesizer. The goal of this experiment was two-fold: 1) to evaluate existing genetic algorithms (GAs) and deep learning approaches using a benchmark task, and 2) to introduce a novel hybrid approach for inverse synthesis. Results of the evaluation showed that amongst the existing methods, a multi-objective genetic algorithm (MOGA) [133] was most consistently able to produce high-quality results and outperformed the deep learning methods. Comparison within the deep learning approaches showed that a recurrent neural network architecture proposed by Yee-King *et al.* performed the best [153]. An unexpected result was that all the deep learning approaches performed better using an MFCC audio representation compared to a higher resolution Mel-Spectrogram representation.

While the MOGA based approach outperformed all the deep learning models, computing a single prediction was significantly more expensive in terms of compute time. A novel hybrid approach was introduced that leveraged the strengths of each method: this approach used deep learning for the initial population generation for a multi-objective genetic algorithm. This approach was able to generate solutions that were as good or better than the original MOGA in 40% of the computation time. These results show the potential for combining deep learning with other algorithmic approaches such as MOGAs for automatic synthesizer programming applications.

The results of these experiments also highlighted some of the challenges currently facing inverse synthesis research. The most daunting of these challenges is the complexity of the synthesizer parameter space. Developing a deeper understanding of the relationship between synthesizer parameters and the resulting audio will be critical in advancing research on inverse synthesis.

Motivated by the identified challenges facing automatic synthesizer programming research, a large-scale dataset (synth1B1) and an open-source GPU-enabled modular synthesizer (torchsynth) were developed as a part of this thesis work to support further research in this area. These are presented in chapter 6. Improving the efficacy of deep learning approaches for automatic synthesizer programming is a promising area for future development. The synth1B1 dataset and associated torchsynth synthesizer should allow researchers to more efficiently conduct training and pre-training of deep

learning models on synthesizers, and more readily explore the relationship between parameters and the associated auditory output.

The final chapter of this thesis, chapter 7, discussed a new automatic synthesizer programming application built on top of the torchsynth synthesizer. This application, called Synth Explorer, provides a visual representation of sounds that are arranged on a two-dimensional interface based on sound similarity. The arrangement and colour of the visualization is controlled by the user, allowing them to construct a visual representation of synthesizer sounds that suits their creative needs. It is an example of an exploration-based interface, and was designed to encourage novice users to explore the auditory space of a synthesizer and begin to learn the relationship between sounds and synthesizer parameters. This development was based on a proposed design framework derived from the fields of creativity support [125] and music interaction [60]. One of the benefits of exploration-based interfaces is that they engage the user in the process of searching for sounds within a synthesizer, as opposed to taking over the process as is the case with example-based (inverse synthesis) applications. Creating effective and engaging automatic synthesizer programming interfaces will likely benefit from the combination of multiple interaction styles. For example, adding an example-based search option within Synth Explorer, using the methods identified in chapter 5, would help support exploration and efficient searching of a synthesizer.

8.1 Future Work

The author hopes that the open-source software and datasets published as a component of this thesis will contribute to further research into automatic synthesizer programming. Clearly, additional work is required to continue to deepen our understanding of the relationship between synthesizer parameters and the resulting audio. The synth1B1 dataset and associated torchsynth synthesizer provide a tool to support research in this direction. However, synth1B1 currently only represents sounds generated using a subtractive synthesis paradigm. Developing more synthesizers in torchsynth using other synthesis methods, such as FM, will help support further research.

One of the limiting factors that was highlighted in chapter 6 is the lack of a suitable audio representation that captures the salient perceptual qualities of synthesizer sounds. The development of a perceptually relevant representation of audio, and especially timbre, will help researchers quantify the similarity of synthesizer sounds and

further support the development of automated methods of synthesizer programming. Further exploration of the role of audio representations in deep learning approaches will also help to support future development in this area.

A current limitation of deep learning approaches for inverse synthesis is the inability to calculate training loss on audio results. This is because there are currently no available methods for computing gradients over traditional software synthesizers, which eliminates them from being used within the training loop of a deep learning algorithm. Instead, researchers must calculate training loss using parameter error. This approach is limited due to the complex nature of the synthesis parameter space, as discussed in chapter 5. Developing methods that allow for synthesizers to be included in the training loop of a deep learning algorithm is a promising route for improving the performance of these methods for inverse synthesis. Ramírez *et al.* recently published work that used stochastic gradient estimation to include black-box audio effects within a deep learning model [107]. This approach could likely be adapted to work with synthesizers. Engel *et al.*'s recent work on differentiable digital signal processing (DDSP) provides another possible avenue for including synthesis algorithms within deep learning models [39]. The torchsynth synthesizer introduced in chapter 6 was inspired by this work and future work on that includes updating it to support the computation of gradients over synthesis modules.

Even if the technical hurdles in automatic synthesizer programming can be solved, it will still be necessary to develop more intuitive user interfaces to better assist synthesizer users in achieving their creative goals. The user study conducted by Kreković identified the desire amongst experienced synthesizer users for improved methods for working with their synthesizers [79]. The responses also suggested that intuitive control interfaces and example-based approaches would be helpful. Determining the best approaches to integrating artificial intelligence and machine learning based techniques with a user interface that supports creativity will require future work.

8.2 Final Remarks

A well designed sound on a synthesizer has the ability to enhance a musical composition and transport a listener to another world. Those who have mastered the art of synthesizer programming have contributed some of the most widely acclaimed musical recordings and film scores; from the best-selling classical compositions of Bach re-imagined by Wendy Carlos in *Switched on Bach* to the *Deep Note* (the "THX

sound") designed by Dr. Moorer that played at the beginning of the 1983 premiere of *Star Wars: Episode VI - Return of the Jedi*. With the rise of the personal computer, technology for music and audio production has become increasingly decentralized. Anyone with a laptop or a mobile phone has the power to make sounds with a synthesizer. However, anyone who has tried to program a new sound into a synthesizer will attest to the fact that the usability of these devices has yet to catch up to their availability. The author hopes that the work presented in this thesis well help to contribute to the goal of achieving the best possible mode of automatic synthesizer programming so as to enable users to express themselves more freely with synthesizers and to empower them to create the next great sound.

Bibliography

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *ACM SIGKDD*, 2019.

[2] Kristina Andersen and Peter Knees. Conversations with expert users in music retrieval and research challenges for creative mir. In *ISMIR*, pages 122–128, 2016.

[3] Christopher Andrews. An embodied approach to ai art collaboration. In *Proceedings of the 2019 on Creativity and Cognition*, page 156–162, New York, NY, USA, 2019. Association for Computing Machinery.

[4] American National Standards Institute (ANSI). American national psychoacoustical terminology, 1973.

[5] Richard D Ashley. A knowledge-based approach to assistance in timbral design. In *Proceedings of the 1986 International Computer Music Conference*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 1986.

[6] Oren Barkan and David Tsiris. Deep synthesizer parameter estimation. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3887–3891. IEEE, 2019.

[7] Oren Barkan, David Tsiris, Ori Katz, and Noam Koenigstein. Inversynth: Deep estimation of synthesizer parameter configurations from audio signals. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 27(12):2385–2396, 2019.

[8] James W Beauchamp. Synthesis by spectral amplitude and "brightness" matching of analyzed musical instrument tones. *Journal of the Audio Engineering Society*, 30(6):396–406, 1982.

[9] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[10] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[11] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The Million Song Dataset. In *ISMIR*, 2011.

[12] Adrien Bitton, Philippe Esling, and Tatsuya Harada. Neural granular sound synthesis. *CoRR*, abs/2008.01393, 2020.

[13] David Sanchez Blancas and Jordi Janer. Sound retrieval from voice imitation queries in collaborative databases. In *Audio Engineering Society Conference: 53rd International Conference: Semantic Audio*. Audio Engineering Society, 2014.

[14] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *EMNLP-CoNLL*, 2007.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, and Pranav Shyam *el al.* Language models are few-shot learners. In *NeuroIPS*, 2020.

[16] Mark Cartwright and Bryan Pardo. Synthassist: an audio synthesizer programmed with vocal imitation. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 741–742, 2014.

[17] Mark Cartwright and Bryan Pardo. Synthassist: Querying an audio synthesizer by vocal imitation. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 363–366, June 2014.

[18] Mark Cartwright and Bryan Pardo. Vocalsketch: Vocally imitating audio concepts. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 43–46. ACM, 2015.

[19] Mark Cartwright, Bryan Pardo, and Josh Reiss. Mixploration: Rethinking the audio mixer interface. In *The 19th international conference on Intelligent User Interfaces*, pages 365–370. ACM, February 2014.

[20] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *ICML*, PMLR, 2020.

[21] Erin Cherry and Celine Latulipe. Quantifying the creativity support of digital tools through the creativity support index. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(4):1–25, 2014.

[22] Michael Chinen and Naotoshi Osaka. Genesynth: noise band-based genetic algorithm analysis/synthesis framework. In *Proceedings of the International Computer Music Conference*, 2007.

[23] Keunwoo Choi, George Fazekas, and Mark Sandler. Automatic tagging using deep convolutional neural networks. *arXiv preprint arXiv:1606.00298*, 2016.

[24] John M Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 1973.

[25] Ross Clement. Automatic synthesiser programming. In *Proceedings of the 2011 International Computer Music Conference*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2011.

[26] Tamlin S Conner, Colin G DeYoung, and Paul J Silvia. Everyday creative activity as a path to flourishing. *The Journal of Positive Psychology*, 13(2):181–189, 2018.

[27] Matthew Cooper, Jonathan Foote, Elias Pampalk, and George Tzanetakis. Visualization in audio-based music information retrieval. *Computer Music Journal*, 30(2):42–62, 2006.

[28] J. Cramer, H-.H. Wu, J. Salamon, and J. P. Bello. Look, listen and learn more: Design choices for deep audio embeddings. In *ICASSP*, 2019.

[29] Jason Cramer, Ho-Hsiang Wu, Justin Salamon, and Juan Pablo Bello. Look, listen, and learn more: Design choices for deep audio embeddings. In *ICASSP*, 2019.

[30] Mihaly Csikszentmihalyi. *Flow: The psychology of optimal experience*, volume 1990. Harper & Row New York, 1990.

[31] Palle Dahlstedt. Creating and exploring huge parameter spaces: Interactive evolution as a tool for sound generation. In *ICMC*, 2001.

[32] Nicholas Davis, Holger Winnemöller, Mira Dontcheva, and Ellen Yi-Luen Do. Toward a cognitive theory of creativity support. In *Proceedings of the 9th ACM Conference on Creativity & Cognition*, pages 13–22, 2013.

[33] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

[34] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[35] N Delprat, Ph Guillemain, and R Kronland-Martinet. Parameter estimation for non-linear resynthesis methods with the help of a time-frequency analysis of natural sounds. In *Proceedings of the 1990 International Computer Music Conference*, pages 88–90, 1990.

[36] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A generative model for music. *arXiv preprint arXiv:2005.00341*, 2020.

[37] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. In *International Conference on Learning Representations*, 2018.

[38] Yanai Elazar, Abhijit Mahabal, Deepak Ramachandran, Tania Bedrax-Weiss, and Dan Roth. How large are lions? inducing distributions over quantitative attributes. In *ACL*, 2019.

[39] Jesse Engel, Lamtharn (Hanoi) Hantrakul, Chenjie Gu, and Adam Roberts. DDSP: Differentiable digital signal processing. In *ICLR*, 2020.

[40] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Mohammad Norouzi, Douglas Eck, and Karen Simonyan. Neural audio synthesis of musical notes with wavenet autoencoders. In *ICML*. PMLR, 2017.

[41] Philippe Esling, Naotake Masuda, Adrien Bardet, Romeo Despres, and Axel Chemla-Romeu-Santos. Flow synthesizer: Universal audio synthesizer control with normalizing flows. *Applied Sciences*, 10(1):302, 2020.

[42] Russ Ethington and Bill Punch. Seawave: A system for musical timbre description. *Computer Music Journal*, 18(1):30–39, 1994.

[43] X. Favory, K. Drossos, T. Virtanen, and X. Serra. Coala: Co-aligned autoencoders for learning semantically enriched audio representations. In *ICML Workshop on Self-supervision in Audio and Speech*, 2020.

[44] Eduardo Fonseca, Xavier Favory, Jordi Pons, Frederic Font, and Xavier Serra. FSD50K: an open dataset of human-labeled sound events. *CoRR*, abs/2010.00475, 2020.

[45] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[46] Ohad Fried, Zeyu Jin, Reid Oda, and Adam Finkelstein. Audioquilt: 2D arrangements of audio samples using metric learning and kernelized sorting. In *New Interfaces for Musical Expression*, pages 281–286, June 2014.

[47] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. Audio Set: An ontology and human-labeled dataset for audio events. In *ICASSP*, 2017.

[48] Darrell Gibson and Richard Polfreman. Analyzing journeys in sound: usability of graphical interpolators for sound design. *Personal and Ubiquitous Computing*, pages 1–14, 2020.

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

[50] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *JMLR*, 2012.

[51] John M Grey. Multidimensional perceptual scaling of musical timbres. *the Journal of the Acoustical Society of America*, 61(5):1270–1277, 1977.

[52] Brahim Hamadicharef and Emmanuel C Ifeachor. Intelligent and perceptual-based approach to musical instruments sound design. *Expert Systems with Applications*, 39(7):6476–6484, 2012.

[53] Alex Harden. Minimoog photograph cc by 2.0.

[54] Curtis Hawthorne, Andrew Stasyuk, Adam Roberts, Ian Simon, Anna Huang, Sander Dieleman, Erich Elsen, Jesse Engel, and Douglas Eck. Enabling factorized piano music modeling and generation with the maestro dataset. In *ICLR*, 2019.

[55] Ben Hayes and Charalampos Saitis. There's more to timbre than musical instruments: semantic dimensions of fm sounds. In *The Proceedings of the 2nd International Conference on Timbre (Timbre 2020)*, 2020.

[56] Sebastian Heise, Michael Hlatky, and Jörn Loviscach. Automatic cloning of recorded sounds by software synthesizers. In *Audio Engineering Society Convention 127*. Audio Engineering Society, 2009.

[57] Romain Hennequin, Anis Khlif, Felix Voituret, and Manuel Moussallam. Spleeter: A fast and state-of-the art music source separation tool with pre-trained models. Late-Breaking/Demo ISMIR 2019, November 2019. Deezer Research.

[58] Shawn Hershey, Sourish Chaudhuri, Daniel PW Ellis, Jort F Gemmeke, Aren Jansen, R Channing Moore, Manoj Plakal, Devin Platt, Rif A Saurous, Bryan Seybold, et al. Cnn architectures for large-scale audio classification. In *2017 ieee international conference on acoustics, speech and signal processing (icassp)*, pages 131–135. IEEE, 2017.

[59] Elizabeth Hinkle-Turner. *Women Composers and Music Technology in the United States: Crossing the Line*. Ashgate Publishing, Ltd., 2006.

[60] Simon Holland, Katie Wilkie, Paul Mulholland, and Allan Seago. Music interaction: understanding music and human-computer interaction. In *Music and human-computer interaction*, pages 1–28. Springer, 2013.

[61] Andrew Horner. Envelope matching with genetic algorithms. *Journal of New Music Research*, 24(4):318–341, 1995.

[62] Andrew Horner. Wavetable matching synthesis of dynamic instruments with genetic algorithms. *Journal of the Audio Engineering Society*, 43(11):916–931, 1995.

[63] Andrew Horner and James Beauchamp. Piecewise-linear approximation of additive synthesis envelopes: a comparison of various methods. *Computer Music Journal*, 20(2):72–95, 1996.

[64] Andrew Horner, James Beauchamp, and Lippold Haken. Machine tongues xvi: Genetic algorithms and their application to fm matching synthesis. *Computer Music Journal*, 17(4):17–29, 1993.

[65] Eric J Humphrey, Douglas Turnbull, and Tom Collins. A brief review of creative mir. *ISMIR Late-Breaking News and Demos*, 2013.

[66] Smule Inc. DAMP-VPB: Digital Archive of Mobile Performances - Smule Vocal Performances Balanced, 2017.

[67] Intellijel. Eurorack 101, Aug 2019. Available at `https://intellijel.com/support/eurorack-101/`.

[68] Katsutoshi Itoyama and Hiroshi G Okuno. Parameter estimation of virtual musical instrument synthesizers. In *ICMC*, 2014.

[69] David A Jaffe and Julius O Smith. Extensions of the karplus-strong plucked-string algorithm. *Computer Music Journal*, 7(2):56–69, 1983.

[70] Mark Jenkins. *Analog synthesizers: understanding, performing, buying: from the legacy of Moog to software synthesis*. Routledge, 2019.

[71] Yang Jiale and Zhang Ying. Visualization method of sound effect retrieval based on umap. In *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 1, pages 2216–2220. IEEE, 2020.

[72] Colin G Johnson. Exploring the sound-space of synthesis algorithms using interactive genetic algorithms. In *Proceedings of the AISB'99 Symposium on*

Musical Creativity, pages 20–27. Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1999.

[73] Colin G Johnson and Alex Gounaropoulos. Timbre interfaces using adjectives and adverbs. In *Proceedings of the 2006 conference on New interfaces for musical expression*, pages 101–102. IRCAM, 2006.

[74] James Justice. Analytic signal processing in music computation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(6):670–684, 1979.

[75] Jong Wook Kim, Justin Salamon, Peter Li, and Juan Pablo Bello. Crepe: A convolutional representation for pitch estimation. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 161–165. IEEE, 2018.

[76] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[77] Peter Knees and Kristina Andersen. Searching for audio by sketching mental images of sound: A brave new idea for audio retrieval in creative music production. In *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval*, pages 95–102, 2016.

[78] Sebastian Kraft and Udo Zölzer. Beaqlejs: Html5 and javascript based framework for the subjective evaluation of audio quality. In *Linux Audio Conference, Karlsruhe, DE*, 2014.

[79] Gordan Kreković. Insights in habits and attitudes regarding programming sound synthesizers: A quantitative study. In *Proceedings of the 16th Sound and Music Computing Conference*, 2019.

[80] Gordan Kreković, Antonio Pošćić, and Davor Petrinović. An algorithm for controlling arbitrary sound synthesizers using adjectives. *Journal of New Music Research*, 45(4):375–390, 2016.

[81] Carol L Krumhansl. Why is musical timbre so hard to understand. *Structure and perception of electroacoustic sound and music*, 9:43–53, 1989.

[82] Gwendal Le Vaillant, Thierry Dutoit, and Sébastien Dekeyser. Improving synthesizer programming from variational autoencoders latent space. In

Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx20in21), 2021.

[83] Gwendal Le Vaillant, Thierry Dutoit, and Rudi Giot. Analytic vs. holistic approaches for the live search of sound presets using graphical interpolation. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 227–232, 2020.

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[85] Guillaume Lemaitre and Davide Rocchesso. On the effectiveness of vocal imitations and verbal descriptions of sounds. *The Journal of the Acoustical Society of America*, 135(2):862–873, 2014.

[86] Martin Lindh. Beyond a skeuomorphic representation of subtractive synthesis. In *IUI Workshops*, 2018.

[87] Sean Luke. Stochastic synthesizer patch exploration in edisyn. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 188–200. Springer, 2019.

[88] Matthieu Macret and Philippe Pasquier. Automatic design of sound synthesizers as pure data patches using coevolutionary mixed-typed cartesian genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 309–316. ACM, 2014.

[89] Pranay Manocha, Adam Finkelstein, Richard Zhang, Nicholas J. Bryan, Gautham J. Mysore, and Zeyu Jin. A differentiable perceptual audio metric learned from just noticeable differences. In *Interspeech*, 2020.

[90] Naotake Masuda and Daisuke Saito. Quality diversity for synthesizer sound matching. In *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx20in21)*, 2021.

[91] Stephen McAdams. *The Perceptual Representation of Timbre*, pages 23–57. Springer International Publishing, Cham, 2019.

[92] James McDermott, Toby Gifford, Anders Bouwer, and Mark Wagy. Should music interaction be easy? In *Music and human-computer interaction*, pages 29–47. Springer, 2013.

[93] Brian McFee, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, volume 8, 2015.

[94] Sam McGuire and Nathan Van der Rest. *The musical art of synthesis*. CRC Press, 2015.

[95] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020.

[96] Daniel Mintz. Toward timbral synthesis: a new method for synthesizing sound based on timbre description schemes. Master's thesis, Citeseer, 2007.

[97] Thomas Mitchell and Charlie Sullivan. Frequency modulation tone matching using a fuzzy clustering evolution strategy. In *Audio Engineering Society Convention 118*, May 2005.

[98] Christopher Mitcheltree and Hideki Koike. Serumrnn: Step by step audio vst effect programming. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 218–234. Springer, 2021.

[99] Daisuke Niizumi, Daiki Takeuchi, Yasunori Ohishi, Noboru Harada, and Kunio Kashino. BYOL for audio: Self-supervised learning for general-purpose audio representation. *CoRR*, abs/2103.06695, 2021.

[100] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[101] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *ICASSP*, 2015.

[102] Bryan Pardo, Mark Cartwright, Prem Seetharaman, and Bongjun Kim. Learning to build natural audio production interfaces. *Arts*, 8(3):110, 2019.

[103] Russell G Payne. A microcomputer-based analysis/resynthesis scheme for processing sampled sounds using fm. In *Proceedings of the 1987 International Computer Music Conference*, 1987.

[104] Geoffroy Peeters. A large set of audio features for sound description (similarity and classification) in the cuidado project. *CUIDADO Ist Project Report*, 54(0):1–25, 2004.

[105] Popolon. Vcv rack screenshot cc by-sa 4.0, November 2018.

[106] Antonio Ramires, Pritish Chandna, Xavier Favory, Emilia Gómez, and Xavier Serra. Neural percussive synthesis parametrerised by high-level timbral features. In *ICASSP*, 2020.

[107] Marco A Martínez Ramírez, Oliver Wang, Paris Smaragdis, and Nicholas J Bryan. Differentiable signal processing with black-box audio effects. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 66–70. IEEE, 2021.

[108] Curtis Rasmussen. Evaluating the usability of software synthesizers: An analysis and first approach. Master's thesis, University of Guelph, 2018.

[109] Jean-Claude Risset and David L Wessel. Exploration of timbre by analysis and synthesis. In *The psychology of music*, pages 113–169. Elsevier, 1999.

[110] Curtis Roads and Max Mathews. Interview with max mathews. *Computer Music Journal*, 4(4):15–22, 1980.

[111] Curtis Roads, John Strawn, et al. *The computer music tutorial*. MIT press, 1996.

[112] Fanny Roche, Thomas Hueber, Maëva Garnier, Samuel Limier, and Laurent Girin. Make that sound more metallic: Towards a perceptually relevant control of the timbre of synthesizer sounds using a variational autoencoder. *Transactions of the International Society for Music Information Retrieval (TISMIR)*, 4:52–66, 2021.

[113] Martin Roth and Matthew Yee-King. A comparison of parametric optimization techniques for musical instrument tone matching. In *Audio Engineering Society Convention 130*. Audio Engineering Society, 2011.

[114] Joseph Rothstein. *MIDI: A comprehensive introduction*, volume 7. AR Editions, Inc., 1992.

[115] Martin Russ. *Sound synthesis and sampling*. Routledge, 2012.

[116] Aaqib Saeed, David Grangier, and Neil Zeghidour. Contrastive learning of general-purpose audio representations. In *ICASSP*, 2021.

[117] Hugo Scurto, Bavo Van Kerrebroeck, Baptiste Caramiaux, and Frédéric Bevilacqua. Designing deep reinforcement learning for human parameter exploration. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 28(1):1–35, 2021.

[118] Allan Seago. A new interaction strategy for musical timbre design. In *Music and human-computer interaction*, pages 153–169. Springer, 2013.

[119] Allan Seago, Simon Holland, and Paul Mulholland. A critical analysis of synthesizer user interfaces for timbre. In *Proceedings of the XVIII British HCI Group Annual Conference*. HCI, Research Press International, 2004.

[120] Roger N Shepard. Geometrical approximations to the structure of musical pitch. *Psychological review*, 1982.

[121] Jordie Shier, Kirk McNally, and George Tzanetakis. Analysis of drum machine kick and snare sounds. In *Audio Engineering Society Convention 143*. Audio Engineering Society, 2017.

[122] Jordie Shier, Kirk McNally, and George Tzanetakis. Sieve: A plugin for the automatic classification and intelligent browsing of kick and snare samples. In *3rd Workshop on Intelligent Music Production*. WIMP, September 2017.

[123] Jordie Shier, Kirk McNally, George Tzanetakis, and Ky Grace Brooks. Manifold learning methods for visualization and browsing of drum machine samples. *Journal of the Audio Engineering Society*, 69(1/2):40–53, 2021.

[124] Jordie Shier, George Tzanetakis, and Kirk McNally. Spiegelib: An automatic synthesizer programming library. In *Audio Engineering Society Convention 148*. Audio Engineering Society, 2020.

[125] Ben Shneiderman. Creativity support tools: Accelerating discovery and innovation. *Communications of the ACM*, 50(12):20–32, 2007.

[126] Benjamin D Smith. Play it again: Evolved audio effects and synthesizer programming. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, pages 275–288. Springer, 2017.

[127] Christian J. Steinmetz and Joshua D. Reiss. auraloss: Audio focused loss functions in PyTorch. In *DMRN+15*, 2020.

[128] Dan Stowell. *Making music through real-time voice timbre analysis: machine learning and timbral control*. PhD thesis, Queen Mary University of London, 2010.

[129] Andy Kelleher Stuhl. Reactions to analog fetishism in sound-recording cultures. *The Velvet Light Trap*, (74):42–53, 2014.

[130] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[131] Atau Tanaka. Musical performance practice on sensor-based instruments. *Trends in Gestural Control of Music*, 13(389-405):284, 2000.

[132] Kıvanç Tatar, Daniel Bisig, and Philippe Pasquier. Latent timbre synthesis. *Neural Computing and Applications*, 33(1):67–84, 2021.

[133] Kıvanç Tatar, Matthieu Macret, and Philippe Pasquier. Automatic synthesizer preset generation with presetgen. *Journal of New Music Research*, 45(2):124–144, 2016.

[134] Art Tavana. Democracy of sound: Is garageband good for music?, 2015.

[135] Bart Thomee, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. YFCC100M: The new data in multimedia research. *Communications of the ACM*, 2016.

[136] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017.

[137] Joseph Turian and Max Henry. I'm sorry for your loss: Spectrally-based audio distances are bad at pitch. In *ICBINB@NeurIPS*, 2020.

[138] Joseph Turian, Jordie Shier, George Tzanetakis, Kirk McNally, and Max Henry. One billion audio sounds from gpu-enabled modular synthesis. *arXiv preprint arXiv:2104.12922*, 2021.

[139] Chloé Turquois, Martin Hermant, Daniel Gómez-Marín, and Sergi Jordà. Exploring the benefits of 2d visualizations for drum samples retrieval. In *Proceedings of the 2016 acm on conference on human information interaction and retrieval*, pages 329–332, 2016.

[140] Cyrus Vahidi, George Fazekas, Charalampos Saitis, and Alessandro Palladini. Timbre space representation of a subtractive synthesizer. In *Timbre 2020*, 2020.

[141] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[142] Patrick Vandewalle, Jelena Kovacevic, and Martin Vetterli. Reproducible research in signal processing. *IEEE Signal Processing Magazine*, 26(3):37–47, 2009.

[143] Richard Vine. Tadao kikumoto invents the roland tb-303. The Guardian, June 2011.

[144] Xin Wang, Shinji Takaki, and Junichi Yamagishi. Neural source-filter waveform models for statistical parametric speech synthesis. *IEEE ACM Trans. Audio Speech Lang. Process.*, 2020.

[145] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. A theoretical analysis of NDCG type ranking measures. In *COLT*. PMLR, 2013.

[146] David L Wessel. Timbre space as a musical control structure. *Computer music journal*, pages 45–52, 1979.

[147] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[148] Ryuichi Yamamoto, Eunwoo Song, and Jae-Min Kim. Parallel wavegan: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram. In *ICASSP*, 2020.

[149] Matthew Yee-King and Martin Roth. Synthbot: an unsupervised software synthesizer programmer. In *Proceedings of the International Computer Music Conference*, 2008.

[150] Matthew John Yee-King. The evolving drum machine. In *Music-AL workshop, ECAL conference*, volume 2007, 2007.

[151] Matthew John Yee-King. *Automatic sound synthesizer programming: techniques and applications*. PhD thesis, University of Sussex, 2011.

[152] Matthew John Yee-King. The use of interactive genetic algorithms in sound design: a comparison study. *ACM Comput. Entertainment*, 14(3), 2016.

[153] Matthew John Yee-King, Leon Fedden, and Mark d'Inverno. Automatic programming of vst sound synthesizers using deep networks and other techniques. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):150–159, 2018.

[154] Asterios Zacharakis, Konstantinos Pastiadis, and Joshua D Reiss. An interlanguage study of musical timbre semantic dimensions and their acoustic correlates. *Music Perception: An Interdisciplinary Journal*, 31(4):339–358, 2012.

[155] Yichi Zhang and Zhiyao Duan. Iminet: Convolutional semi-siamese networks for sound search by vocal imitation. In *2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 304–308. IEEE, 2017.

[156] Yichi Zhang and Zhiyao Duan. Visualization and interpretation of siamese style convolutional neural networks for sound search by vocal imitation. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2406–2410. IEEE, 2018.

[157] Yichi Zhang, Junbo Hu, Yiting Zhang, Bryan Pardo, and Zhiyao Duan. Vroom! a search engine for sounds by vocal imitation queries. In *Proceedings of the 2020 Conference on Human Information Interaction and Retrieval*, pages 23–32, 2020.

Appendix A

Model Architectures

The model architectures provided below were used in the experiments conducted in chapter 5. Each model estimated a subset of nine parameters from *Dexed* to match a target audio. n is the batch size:

Table A.1: MFCC MLP

Layer Type	Output Shape	Num Parameters
Input	(n, x)	
Dense	(n, 256)	146688
ReLU	(n, 256)	
Dense	(n, 128)	32896
ReLU	(n, 128)	
Dense	(n, 64)	8256
ReLU	(n, 64)	
Dropout (0.3)	(n, 64)	
Dense	(n, 7)	455
Total		188,295

Table A.2: Mel-Spectrogram MLP

Layer Type	Output Shape	Num Parameters
Input	(n, x)	
Dense	(n, 256)	721152
ReLU	(n, 256)	
Dense	(n, 128)	32896
ReLU	(n, 128)	
Dense	(n, 64)	8256
ReLU	(n, 64)	
Dropout (0.4)	(n, 64)	
Dense	(n, 7)	455
Total		762,759

Table A.3: MFCC LSTM

Layer Type	Output Shape	Num Parameters
Input	(n, x)	
LSTM	(n, 44, 64)	19968
LSTM	(n, 44, 64)	33024
LSTM	(n, 64)	33024
Dropout (0.3)	(n, 64)	
Dense	(n, 7)	455
Total		86,471

Table A.4: Mel-Spectrogram LSTM

Layer Type	Output Shape	Num Parameters
Input	(n, x)	
LSTM	(n, 64, 128)	88576
LSTM	(n, 64, 128)	131584
LSTM	(n, 128)	131584
Dropout (0.2)	(n, 128)	
Dense	(n, 7)	903
Total		352,647

Table A.5: MFCC LSTM++

Layer Type	Output Shape	Num Parameters
Input	(n, x)	
Bidirectional LSTM	(n, 256)	145408
Dropout (0.3)	(n, 256)	
Dense	(n, 128)	32896
ELU	(n, 128)	
Highway Layer 1-7	(n, 128)	33024
Dense	(n, 7)	903
Total		410,375

Table A.6: Mel-Spectrogram LSTM++

Layer Type	Output Shape	Num Parameters
Input	(n, x)	
Bidirectional LSTM	(n, 256)	177152
Dropout (0.4)	(n, 256)	
Dense	(n, 128)	32896
ELU	(n, 128)	
Highway Layer 1-7	(n, 128)	33024
Dense	(n, 7)	903
Total		442,119

Table A.7: MFCC CNN

Layer Type	Kernel Size	Output Shape	Num Parameters
Input		(n, x)	
Conv2D (Stride=2,2)	(3,3)	(n, 22, 7, 16)	160
ReLU		(n, 22, 7, 16)	
Batch Normalization		(n, 22, 7, 16)	64
Conv2D (Stride=2,2)	(3,3)	(n, 11, 4, 32)	4640
ReLU		(n, 11, 4, 32)	
Batch Normalization		(n, 22, 7, 16)	128
Conv2D (Stride=2,2)	(3,3)	(n, 6, 2, 32)	9248
ReLU		(n, 6, 2, 32)	
Batch Normalization		(n, 22, 7, 16)	128
Conv2D (Stride=2,2)	(3,3)	(n, 3, 1, 64)	18496
ReLU		(n, 3, 1, 64)	
Batch Normalization		(n, 22, 7, 16)	256
Conv2D (Stride=2,2)	(3,3)	(n, 2, 1, 64)	36928
ReLU		(n, 2, 1, 64)	
Batch Normalization		(n, 22, 7, 16)	256
Dropout (0.3)		(n, 2, 1, 64)	
Flatten		(n, 128)	
Dense		(n, 512)	66048
Dropout (0.3)		(n, 512)	
Dense		(n, 512)	262656
Dropout (0.3)		(n, 512)	
Dense		(n, 7)	3591
Total			402,599

Table A.8: Mel-Spectrogram CNN

Layer Type	Kernel Size	Output Shape	Num Parameters
Input		(n, x)	
Conv2D (Stride=2,2)	(3,3)	(n, 32, 22, 16)	160
ReLU		(n, 32, 22, 16)	
Conv2D (Stride=2,2)	(3,3)	(n, 16, 11, 32)	4640
ReLU		(n, 16, 11, 32)	
Conv2D (Stride=2,2)	(3,3)	(n, 8, 6, 32)	9248
ReLU		(n, 8, 6, 32)	
Conv2D (Stride=2,2)	(3,3)	(n, 4, 3, 64)	18496
ReLU		(n, 4, 3, 64)	
Conv2D (Stride=2,2)	(3,3)	(n, 2, 2, 64)	36928
ReLU		(n, 2, 2, 64)	
Dropout (0.3)		(n, 2, 2, 64)	
Flatten		(n, 256)	
Dense		(n, 128)	32896
Dropout (0.3)		(n, 128)	
Dense		(n, 128)	16512
Dropout (0.3)		(n, 128)	
Dense		(n, 128)	16512
Dropout (0.3)		(n, 128)	
Dense		(n, 7)	903
Total			136,295

Appendix B

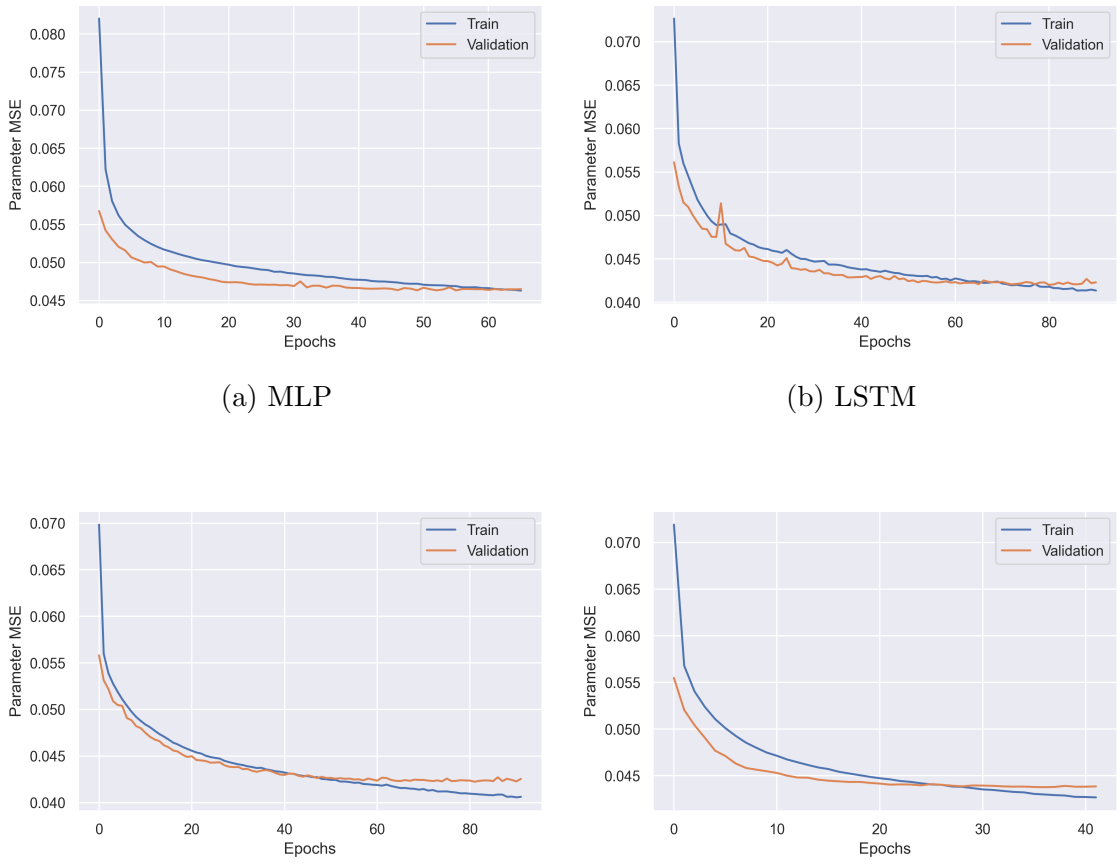
Model Hyperparameters

Table B.1: Training Hyperparameters

Model	Batch Size	η	η Decay Rate
MFCC-MLP	64	0.0005	0
MFCC-LSTM	128	0.001	0
MFCC-LSTM++	64	0.001	0
MFCC-CNN	32	0.001	3
Mel-MLP	32	0.00005	0
Mel-LSTM	32	0.0005	0
Mel-LSTM++	32	0.001	0
Mel-CNN	64	0.001	4

Appendix C

Model Training Loss



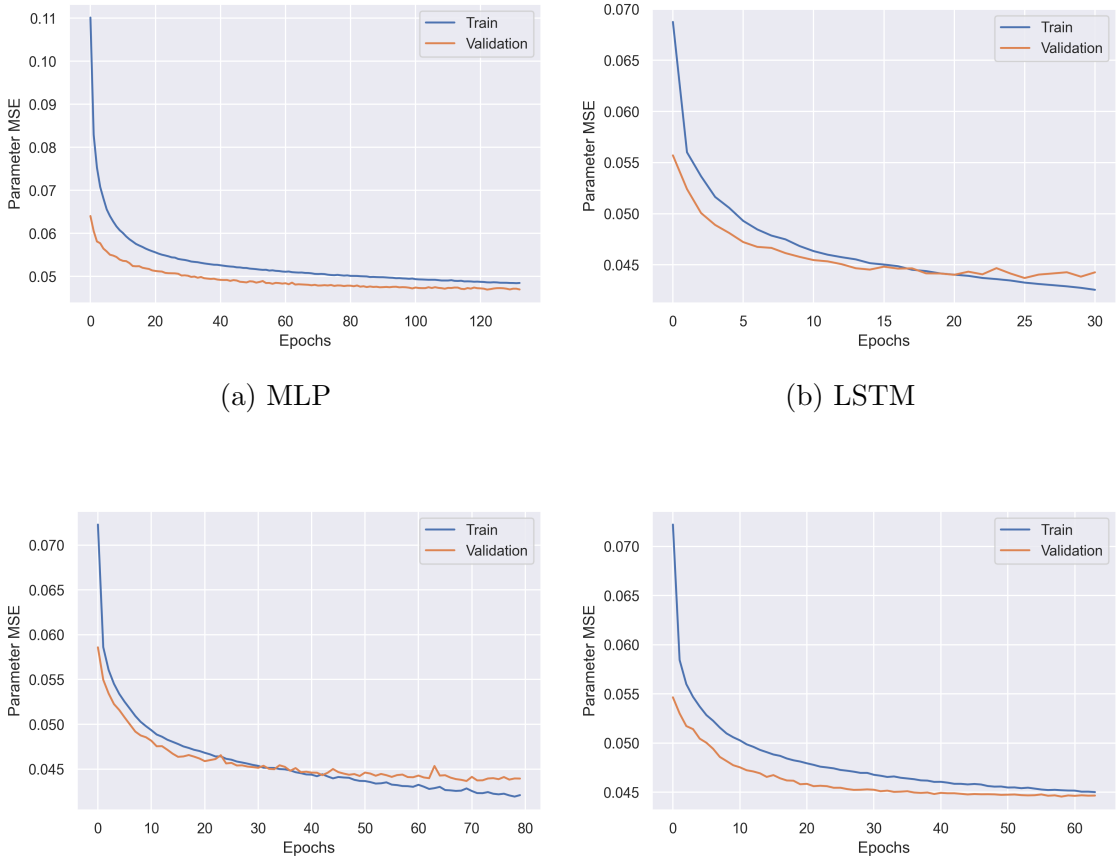
(a) MLP

(b) LSTM

(c) LSTM++

(d) CNN

Figure C.1: MFCC Model Training and Validation Loss Plots



(a) MLP

(b) LSTM

(c) LSTM++

(d) CNN

Figure C.2: Mel-Spectrogram Model Training and Validation Loss Plots

Appendix D

JAES Publication Summary

A brief summary/abstract is provided here of the author's paper entitled "Manifold Learning Methods for Visualization and Browsing of Drum Machine Samples", which was published in 2021 in the Journal of the Audio Engineering Society (JAES) [123] and co-authored by Kirk McNally, George Tzanetakis, and Ky Grace Brooks.

The use of electronic drum samples is widespread in contemporary music productions, with music producers having an unprecedented number of samples available to them. The task of organizing and selecting from these large collections can be challenging and time-consuming, which points to the need for improved methods for user interaction. This paper presents a system that computationally characterizes and organizes drum machine samples in two-dimensions based on sound similarity. The goal of the work is to support the development of intuitive drum sample browsing systems. The methodology presented explores time segmentation, which isolates temporal subsets from the input signal prior to audio feature extraction, as a technique for improving similarity calculations. Manifold learning techniques are compared and evaluated for dimensionality reduction tasks, and used to organize and visualize audio collections in two-dimensions. This methodology is evaluated using a combination of objective and subjective methods including audio classification tasks and a user listening study. Results show that using time-segmentation lead to overall higher correlations with subjective rankings and that using MDS dimensionality reduction with time-segmentation lead to higher correlations with subjective rankings for similarities in two-dimensions. Finally, we present an open-source audio plug-in developed using the JUCE software framework that incorporates the findings from this study into an application that can be used in the context of a music production environment.