The Cooper Union
Albert Nerken School of Engineering
Electrical Engineering

Autoencoding Neural Networks as Musical Audio Synthesizers

Joseph T. Colonel

November 2018

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Engineering

Advised by Professor Sam Keene

Acknowledgements

I would like to thank Sam Keene for his continuing guidance, faith in my efforts, and determination to see me pursue studies in machine learning and audio.

I would like to thank Christopher Curro for seeing this work through its infancy.

I would like to thank the Cooper Union's Dean of Engineering's office, Career Development Services, and Electrical Engineering department for funding my trip to DAFx 2018 to present my thesis work.

Special thanks to Yonatan Katzelnik, Benjamin Sterling, Ella de Buck, Richard Yi, George Ho, Jonathan Tronolone, Alex Hu, and Ingrid Burrington for stewarding this work and stretching it beyond what I thought possible.

Finally I would like to thank my family and friends for their nonstop love and support. I would be nowhere without you all.

Abstract

Methodology for designing and training neural network autoencoders for applications involving musical audio is proposed. Two topologies are presented: an autoencoding sound effect that transforms the spectral properties of an input signal (named ANNe); and an autoencoding synthesizer that generates audio based on activations of the autoencoder's latent space (named CANNe). In each case the autoencoder is trained to compress and reconstruct magnitude short-time Fourier transform frames. When an autoencoder is trained in such a manner it constructs a latent space that contains higher-order representations of musical audio that a musician can manipulate.

With ANNe, a seven layer deep autoencoder is trained on a corpus of improvisations on a MicroKORG synthesizer. The musician selects an input sound to be transformed. The spectrogram of this input sound is mapped to the autoencoder's latent space, where the musician can alter it with multiplicative gain constants. The newly transformed latent representation is passed to the decoder, and an inverse Short-Time Fourier Transform is taken using the original signal's phase response to produce audio.

With CANNe, a seventeen layer deep autoencoder is trained on a corpus of C Major scales played on a MicroKORG synthesizer. The autoencoder produces a spectrogram by activating its smallest hidden layer, and a phase response is calculated using phase gradient heap integration. Taking an inverse short-time Fourier transform produces the audio signal.

Both algorithms are lightweight compared to current state-of-the-art audio-producing machine learning algorithms. Metrics related to the autoencoders' performance are measured using various corpora of audio recorded from a MicroKORG synthesizer. Python implementations of both autoencoders are presented.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

2 Background **3**

2.1 Audio Signal Processing 3

2.1.1 Analogue to Digital Conversion 3

2.1.2 Short-Time Fourier Transform 4

2.2 Musical Audio 10

2.2.1 Western Music Theory 11

2.2.2 Timbre 12

2.2.3 Traditional Methods of Musical Audio Synthesis 12

2.3 Machine Learning 13

2.3.1 Definition 13

2.3.2 Autoencoding Neural Networks 14

2.3.3 Latent Spaces 16

2.3.4 Machine Learning Approaches to Musical Audio Synthesis 17

3 ANNe Sound Effect **19**

3.1 Architecture 19

3.2 Corpus and Training Regime 20

3.3 Discussion of Methods 21

3.3.1 Training Methods 21

3.3.2 Regularization 22

3.3.3 Activation Functions 23

3.3.4 Additive Bias 24

3.3.5 Corpus 24

3.4 Optimization Method Improvements 24

3.5 ANNe GUI 28

3.5.1 Interface 28

3.5.2 Functionality 29

3.5.3 Guiding Design Principles 30

3.5.4 Usage 30

3.5.5 “Unlearned” Audio Representations 32

4 CANNe Synthesizer **33**

4.1 Architecture 33

4.2 Corpus 34

4.3 Cost Function 36

4.4 Feature Engineering 38

4.5 Task Performance and Evaluation 38

4.6 Spectrogram Generation 40

4.7 Phase Generation with PGHI 41

4.8 CANNE GUI 41

5 Conclusions and Future Work **43**

5.1 Conclusions 43

5.2 Future Work 44

Bibliography **45**

A Python Code **50**

A.1 ANNe Backend 50

A.2 CANNe Backend 54

A.3 CANNe GUI 65

List of Tables

3.1 Single Layer Autoencoder Topology MSEs 25

3.2 Three Layer Autoencoder Topology MSEs 25

3.3 Deep Topology MSEs and Train Times 28

4.1 5 Octave Dataset Autoencoder validation set SC loss and Training Time . . 39

4.2 1 Octave Dataset Autoencoder validation set SC loss and Training Time . . 39

List of Figures

2.1 Gaussian Window and its Frequency Response *https://docs.scipy.org/doc/scipy-1.0.0/reference/generated/scipy.signal.gaussian.html* 5

2.2 Hann Window and its Frequency Response *https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.signal.hanning.html* 5

2.3 Spectrogram of a speaker saying "Free as Air and Water" 7

2.4 Linear frequency scale vs Mel scale 10

2.5 Triangular Mel frequency bank 10

2.6 Comparison of a time signal with its spectrogram and cepstrogram 11

2.7 Multi-layer Autoencoder Topology 15

2.8 MNIST Example: Interpolation between 5 and 9 in the pixel space 16

2.9 MNIST Example: Interpolation between 5 and 9 in the latent space 16

3.1 The topology described in Table 3.1. x represents the varied width of the hidden layer. 26

3.2 The topology described in Table 3.2. y represents the varied width of the deepest hidden layer. 26

3.3 The “deep” autoencoder topology described in Table 3.3. 26

3.4 Plots demonstrating how autoencoder reconstruction improves when the width of the hidden layer is increased. 27

3.5 The ANNe interface 28

3.6 Block diagram demonstrating ANNe’s signal flow 29

4.1 Final CANNe Autoencoder Topology 34

4.2 Autoencoder Reconstructions without L2 penalty 37

4.3 Sample input and recosntruction using three different cost functions 40

4.4 Sample input and recosntruction using three different cost functions 40

4.5 Mock-up GUI for CANNe. 42

4.6 Signal flow for CANNe. 42

Chapter 1

Introduction

The first instance of digital music can be traced back to 1950, with the completion of the CSIRAC computer in Australia [5]. Engineers designing the computer equipped it with a speaker that would emit a tone after reading a specific piece of code. Programmers were able to modify this debugging mechanism to produce tones at given intervals. Since then, the complexity of digital music and sound synthesis has scaled with the improvements of computing machines. For example, early digital samplers such as the Computer Musician Melodian and Fairlight CMI cost thousands of dollars upon release in the late 1970s and could hold fewer than two seconds of recorded audio. Now, portable recorders can record hours of professional quality audio for less than one hundred dollars.

With the advancement of computing hardware, so too have methods of digital music synthesis advanced. Many analogue methods of sound synthesis, including additive and subtractive synthesis, have be modeled in software synthesizers and digital audio workstations (DAWs). DAWs have also come to provide an array of tools to mimic musical instruments, from guitars to tubas. As such, musicians looking to push the boundaries of music and sound design have turned to more advanced algorithms to create new sonic palettes.

Recently, musicians have turned to machine learning and artificial intelligence to augment

music production. Advancements in the field of artificial neural networks have produced impressive results in computer vision and categorization tasks. Several attempts have been made to bring these advancements to digital music and sound synthesis. Google's Wavenet uses a convolutional neural network to synthesize digital audio sample-by-sample, and has been used to create complex piano performances [25]. Google's NSynth uses an autoencoding neural network topology to encode familiar sounds, such as a trumpet or a violin, and output sounds that interpolate a timbre between the two [7].

While implementations such as these are impressive both in technical scale and imagination, they are often too large and computationally intensive to be used by musicians. Algorithms that have reached market, on the other hand, frequently are handicapped and restricted in order to reduce computational load. There remains an opportunity for a neural network approach to music synthesis that is light enough to run on typical workstations while providing the musician with meaningful control over the network's parameters.

Two neural network topologies are presented: an autoencoding sound effect,(named ANNe [3]) , that transforms the spectral properties of an input signal based on the work presented in [20]; and an autoencoding synthesizer that generates audio based on activations of the autoencoder's latent space (named CANNe [4]). These models are straightforward to train on user-generated corpora and lightweight compared to state-of-the-art algorithms. Furthermore, both generate in real time at inference. Chapter 2 covers background information regarding audio processing and analysis, Western music, and machine learning. Chapter 3 outlines our experiment design, setup, performance metrics, and Python implementation for the ANNe sound effect. Chapter 4 outlines our experiment design, setup, performance metrics, and Python implementation of CANNe synthesizer. Chapter 5 concludes the thesis with observations and suggestions for future work.

Chapter 2

Background

2.1 Audio Signal Processing

A brief overview of the basics of audio signal processing follows. Because the model presented in this thesis deals with digital audio exclusively, this discussion will primarily focus on digital signals.

2.1.1 Analogue to Digital Conversion

Human hearing occurs when the brain processes excitations of the organs in the ear caused by vibrations in air. Human beings can hear frequencies from approximately 20Hz to 20,000Hz, though this range shrinks at the high end due to aging and overexposure to loud sounds. As such, when converting sound from analogue to digital, it is necessary to bandlimit the signal from DC to 20,000Hz [9].

Let $x_c(t)$ be a bandlimited continuous-time signal with no spectral power above Ω_N Hz. The Nyquist-Shannon sampling theorem states that $x_c(t)$ is uniquely determined by its samples $x[n] = x_c(nT)$, $n = 0, \pm 1, \pm 2, \ldots$ if

$$\Omega_s = \frac{2\pi}{T} \geq 2\Omega_N \tag{2.1}$$

Ω_s is referred to as the sampling frequency and Ω_N is referred to as the Nyquist frequency [17]. To ensure that all audible portions of a signal are maintained when converted from analogue to digital, the sampling frequency must be at least 40,000 Hz. To adhere to CD quality audio standards, the sampling rate used throughout this work is 44,100 Hz.

2.1.2 Short-Time Fourier Transform

Analysis of musical audio signals often involves discssuion of pitch and frequency. Thus it is useful to obtain a representation of a finite discrete signal in terms of its frequency content. One useful representation is the Short-Time Fourier Transform (STFT). The STFT of a signal $x[n]$ is

$$X(n,\omega) = \sum_{m=-\infty}^{\infty} x[n+m]w[m]e^{-j\omega m} \tag{2.2}$$

where $w[n]$ is a window sequence and ω is the frequency in radians. Note that the STFT is periodic in ω with period 2π. Thus we need only consider values of ω for $-\pi \leq \omega \leq \pi$.

Two window sequences were considered for use in this thesis. The first, and the one ultimately chosen, is the Hann window, defined as

$$w_{Hann}[n] = \begin{cases} 0.5 - 0.5cos(2\pi n/M) & 0 \leq n \leq M \\ 0 & otherwise \end{cases}$$

The second window sequence considered is the Gaussian window, defined as

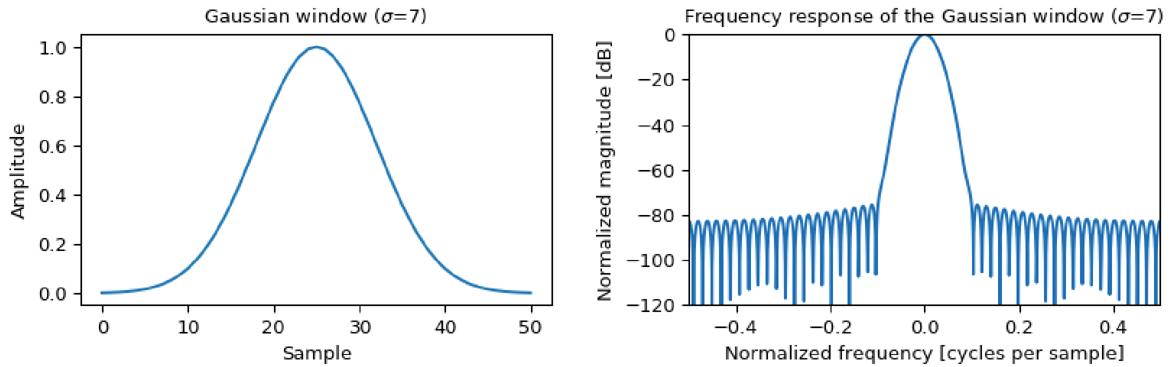


Figure 2.1: Gaussian Window and its Frequency Response *https://docs.scipy.org/doc/scipy-1.0.0/reference/generated/scipy.signal.gaussian.html*

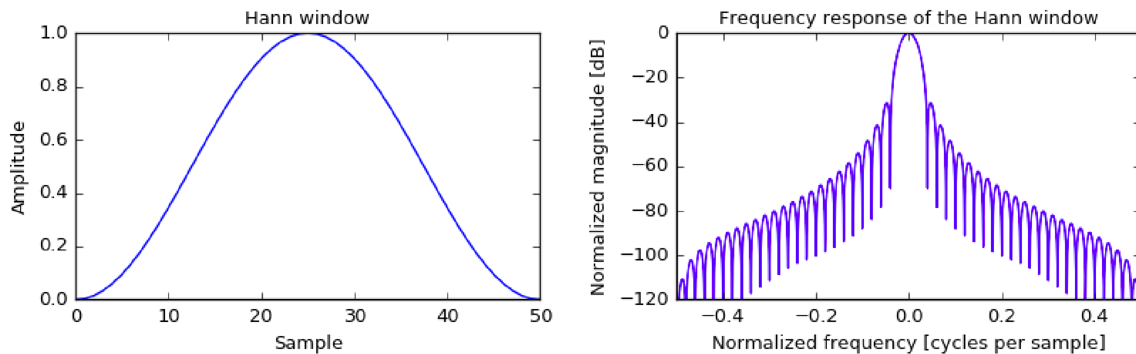


Figure 2.2: Hann Window and its Frequency Response *https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.signal.hanning.html*

$$w_{Gaussian}[n] = \begin{cases} e^{-\frac{1}{2}\left(\frac{n-(M-1)/2}{\sigma(M-1)/2}\right)^2} & 0 \leq n \leq M \\ 0 & otherwise \end{cases}$$

When choosing a window function for the STFT, one must be cognizant of spectral smearing and resolution issues [17].

Refer to Figures 2.1 and 2.2 for plots of the Gaussian and Hann windows. Spectral resolution refers to the STFT's ability to distinguish two sinusoidal components of a signal with fundamental frequencies close to one another. Spectral resolution is influenced primarily

by the width of the main lobe of the window's frequency response. Spectral leakage, on the other hand, refers to a window's tendency to smear a sinusoid's fundamental frequency into neighboring frequency bins. Spectral leakage is influenced primarily by the relative amplitude of the main lobe to the side lobes.

The Gaussian window possesses optimal time-frequency characteristics, i.e. it achieves minimum time-frequency spread [18]. However, it is not often found native to signal processing libraries. The Hann window provides low leakage but slightly decreased frequency resolution when compared to the Gaussian window. However, the Hann window is implemented natively in many signal processing libraries and thus is chosen for use in this work. These quick native implementations allow for the software implementations presented in this work to run in real time, which is essential for ease of use by musicians.

A signal's STFT can be inverted using an overlap-add procedure [21]. First, a windowed output frame is obtained via:

$$\hat{x}'_m(n) = \frac{1}{N} \sum_{k=-N/2}^{N/2-1} \hat{X}'_m(e^{j\omega_k}) e^{j\omega_k n} \tag{2.3}$$

Then, the final output is reconstructed by overlapping and adding the windowed output frames:

$$\hat{x}(n) = \sum_{m} \hat{x}'_m(n - mR) \tag{2.4}$$

where R is the hop size, or how many samples are skipped between frames. This analysis and resynthesis becomes an identity operation if the analysis windows sum to unity, i.e.

$$A_w(n) \triangleq \sum_{m=-\infty}^{\infty} w(n - mR) = 1 \tag{2.5}$$

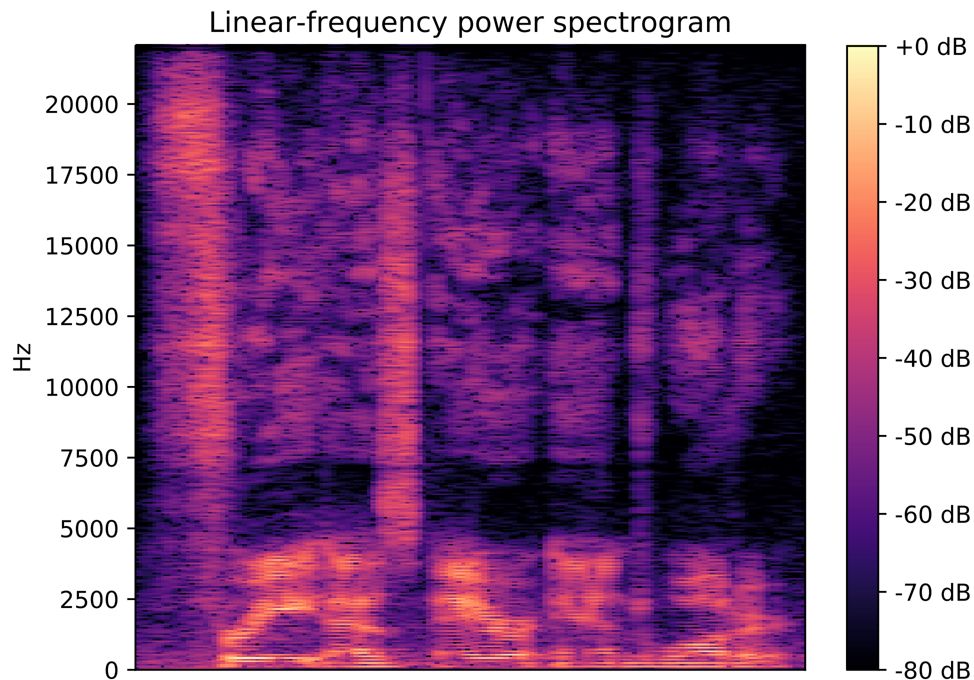


Figure 2.3: Spectrogram of a speaker saying "Free as Air and Water"

The Spectrogram

The STFT is a complex valued signal, which means that each $X(n_i, \omega_j)$ has a magnitude and phase component. We refer to a three dimensional representation of the magnitude of the STFT $\|X(n, \omega)\|_2$ as "the spectrogram." The spectrogram is used throughout audio analysis because it succinctly describes a signal's spectral power distribution, or how much energy is present in different frequency bands, over time. Refer to Figure 2.3 for an example spectrogram.

Phase Construction

When both a signal's STFT magnitude and phase response are available, the spectrogram can be inverted into a time signal. However, should no phase information be present at all, inversion becomes difficult. Algorithms that produce a spectrogram based on input parameters, such as the synthesizer presented in Section 4, often do not produce a corresponding

phase response and thus necessitate a phase construction technique in order to produce audio.

A non-iterative algorithm for the construction of a phase response can be derived based on the direct relationship between the partial derivatives of the phase and logarithm of the magnitude response with respect to the Gaussian window [19]. A fudge factor is applied to modify this derivation for use with a Hann window.

Let us denote the logarithm of the magnitude of a given spectrogram $\|X(n,\omega)\|_2$ as $s_{log,n}(m)$, where n denotes the n-th time-frame of the STFT and m represents the m-th frequency channel. The estimate of the scaled phase derivative in the frequency direction $\tilde{\phi}_{(w,n)}(m)$ and in the time direction $\tilde{\phi}_{(t,n)}(m)$ expressed solely using the magnitude can be written as

$$\tilde{\phi}_{\omega,n}(m) = -\frac{\gamma}{2aM}(s_{log,n+1}(m) - s_{log,n-1}(m)) \tag{2.6}$$

$$\tilde{\phi}_{t,n}(m) = \frac{aM}{2\gamma}(s_{log,n}(m+1) - s_{log,n}(m-1)) + 2\pi am/M \tag{2.7}$$

where M denotes the total number of frequency bins and $\tilde{\phi}_{t,n}(0,n) = \tilde{\phi}_{t,n}(M/2,n) = 0$. These equations come from the Cauchy-Riemann equations, which outline the necessary and sufficient conditions for a function of complex variables to be differentiable. Because a Hann window of length 4098 is used to generate the STFT frames in this work, $\gamma = 0.25645 \times 4098^2$.

Given the phase estimate $\tilde{\phi}_{n-1}(m)$, the phase $\tilde{\phi}_n(m)$ for a particular m is computed using one of the following equations:

$$\tilde{\phi}_n(m) \leftarrow \tilde{\phi}_{n-1}(m) + \frac{1}{2}(\tilde{\phi}_{t,n-1}(m) + \tilde{\phi}_{t,n}(m)) \tag{2.8}$$

$$\tilde{\phi}_n(m) \leftarrow \tilde{\phi}_n(m-1) + \frac{1}{2}(\tilde{\phi}_{\omega,n}(m-1) + \tilde{\phi}_{\omega,n}(m)) \tag{2.9}$$

$$\tilde{\phi}_n(m) \leftarrow \tilde{\phi}_n(m+1) - \frac{1}{2}(\tilde{\phi}_{\omega,n}(m+1) + \tilde{\phi}_{\omega,n}(m)) \tag{2.10}$$

Mathematically speaking, these equations perform numerical differentiation using finite difference estimation on consecutive phase approximations. In this work, equations 2.7 and 2.8 are used to construct a phase response with $\tilde{\phi}_0(0)$ initialized to 0. These equations were chosen because they are causal, i.e. they do not rely on future frame calculations. These equations only rely on frames $n-1$ and n.

Cepstral Analysis

In certain applications, such as automatic speech detection, it is helpful to think of a signal's frequency response as the product of an excitation signal and a voicing signal:

$$\|X(n,\omega)\|_2 = \|E(n,\omega)\|_2 \|V(n,\omega)\|_2 \tag{2.11}$$

For tasks such as speaker detection, the voicing signal is sufficient to identify a speaker. The following procedure is used to separate the two signals [17]. First, $X(n,\omega)$ is run through a mel scale filter bank (Figure 2.5), which is a series of triangular filters centered on mel scale frequencies. The Mel scale (Figure 2.4) is a scale of pitches perceived by listeners to be equal in distance from one another and is thought to more accurately model human hearing than a linear frequency scale [6].

The excitation and voicing components can then be separated using a logarithm

$$log(\|X(n,\omega)\|_2) = log(\|E(n,\omega)\|_2) + log(\|V(n,\omega)\|_2) \tag{2.12}$$

Finally a Discrete Fourier Transform is taken, though this reduces to a Discrete Cosine Transform because $log(\|X(n,\omega)\|_2)$ is an even signal. The resulting signal is referred to as Mel Frequency Cepstral Coefficients, and are used as features throughout machine listening algorithms. As with the STFT, a signal produces a series of MFCCs over time, and a three

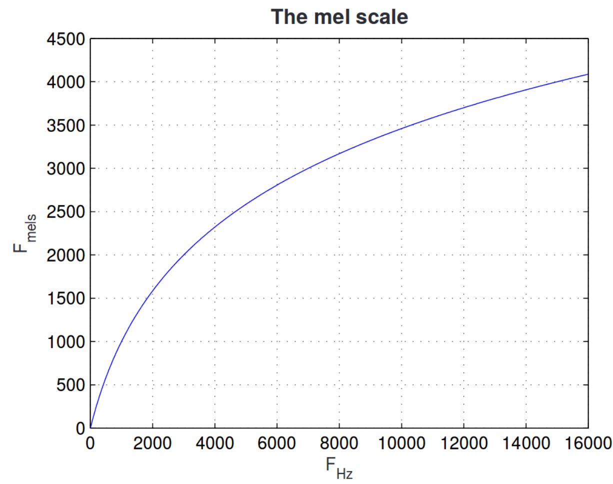


Figure 2.4: Linear frequency scale vs Mel scale

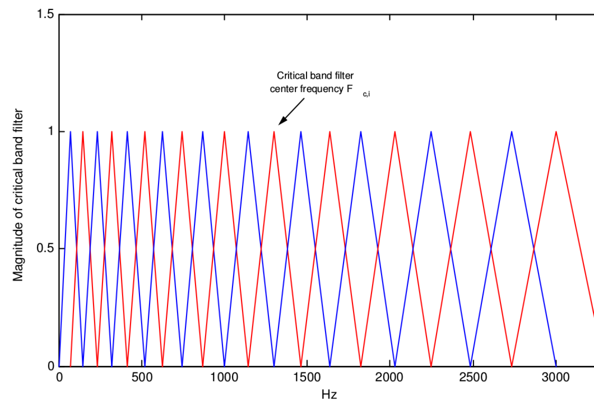


Figure 2.5: Triangular Mel frequency bank

dimensional representation of these coefficients changing over time is called a cepstrogram. Refer to Figure 2.6 for a comparison of a time signal's spectrogram and cepstrogram. The excitation signal will be present in the lower cepstrum bands, and the voicing signal will be present in the higher cepstral bands.

2.2 Musical Audio

As this thesis deals with musical audio, it is necessary to outline what is meant by "music." Though a definition of music would seem obvious to most readers, care must be taken to

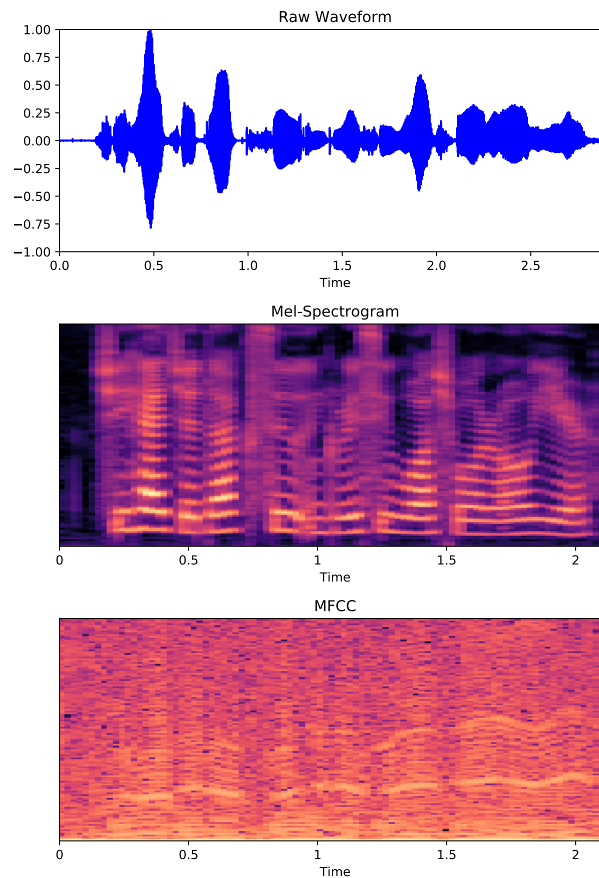


Figure 2.6: Comparison of a time signal with its spectrogram and cepstrogram

distinguish between popular music heard on the radio in the West and the mathematical abstractions of music. Traditional western music has strict technical formulations, which are presented below. A quick definition of timbre follows. Finally, as the corpora used in this work are generated from a MicroKORG synthesizer, the last subsection defines the two types of electronic sound synthesis the synthesizer uses: additive synthesis and subtractive synthesis.

2.2.1 Western Music Theory

Western music uses what is known as equal temperment to divide an octave, or a range of frequencies starting at f_0 and ending at $2f_0$, into twelve equally spaced notes. The frequency

of the nth note in an octave with root note f_0 can be expressed as $\sqrt[12]{2}^n * f_0$. The distance between two notes f_0 and $\sqrt[12]{2} * f_0$ is called a half step. Two half steps make a whole step.

Furthermore, Western music creates a scale using eight notes across an octave. The most common scale heard in popular music is the major scale. This scale is constructed by choosing a root note and the following steps: whole-whole-half-whole-whole-whole-half.

The letters A-G are used to label notes. The C major scale, equivalent to playing the white keys on a piano, runs C-D-E-F-G-A-B. When tuning instruments, the concert A note (A above middle C) is tuned to be 440 Hz. This places middle C at about 262 Hz.

2.2.2 Timbre

Timbre describes the quality of a sound or tone that distinguishes it from another with the same pitch and intensity. For example, a violin playing concert A sounds different from a piano playing concert A. Timbre primarily relies on the spectral characteristics of a sound, including the spectral power in its various harmonics, though the temporal aspects such as envelope and decay also influence its perception.

2.2.3 Traditional Methods of Musical Audio Synthesis

Two traditional methods of electronic music synthesis are additive and subtractive synthesis. In a standard electronic synthesizer, these methods are used to generate waveforms with different timbres, which are stored as "patches". The musician can toggle which patch he or she wants to use and play notes using a keyboard.

In additive synthesis, waveforms such as sine, triangle, and sawtooth waves are generated and added to one another to create a sound. The parameters of each waveform in the sum are controlled by the musician, and the fundamental frequency is chosen by striking a key.

In subtractive synthesis, a waveform such as a square wave or sawtooth wave is generated and then filtered to subtract and alter harmonics. In this case, the parameters of the filter and input waveform are controlled by the musician, and the fundamental frequency is chosen by striking a key.

2.3 Machine Learning

The following section offers a broad definition of machine learning, presents a formulation of the autoencoding neural network, explains how the latent space of an autoencoder can be used for creative purposes, and discusses current machine learning approaches to audio synthesis.

2.3.1 Definition

A broad definition of a computer program that can learn was suggested by Mitchell in 1997 [15]:

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, and measured by P, improves with experience E.

Increasingly complex classes of machine learning algorithms have been designed as computing storage and power have improved.

Machine learning tasks are usually described in terms of how the machine learning system should process an example, or collection of quantitative features, expressed as a vector $x \in \mathbb{R}^d$. Common tasks for a computer program to learn include classification, which produces a function $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$ that maps an input to a numerically identified

category, and regression, which produces a function $f : \mathbb{R}^n \to \mathbb{R}$ that maps an input to a continuous output.

Machine learning algorithms experience an entire dataset, or "corpus". These experiences E can be broadly categorized as supervised or unsupervised. In the supervised case each data point x_i in the corpus is associated with a label y_i, and the algorithm is trained to map $x_i \to \hat{y}_i \approx y_i$. In the unsupervised case, no data point is labelled. Instead, the algorithm used to learn useful properties of the structure of the dataset.

The performance measure P is specific to the task T being carried out by the system and measures the accuracy of the model. In the supervised case, P is calculated by evaluating a cost function $f : (y, \hat{y}) \to \mathbb{R}$ that measures how close the prediction $\hat{y}$ is to y. This cost function is measured on a portion of the dataset that was **not** used to train the algorithm.

2.3.2 Autoencoding Neural Networks

An autoencoding neural network (i.e. autoencoder) is a machine learning algorithm that is typically used for unsupervised learning of an encoding scheme for a given input domain, and is comprised of an encoder and a decoder [26]. For the purposes of this work, the encoder is forced to shrink the dimension of an input into a latent space using a discrete number of values, or "neurons." The decoder then expands the dimension of the latent space to that of the input, in a manner that reconstructs the original input.

In a single layer model, the encoder maps an input vector $x \in \mathbb{R}^d$ to the hidden layer $y \in \mathbb{R}^e$, where $d > e$. Then, the decoder maps y to $\hat{x} \in \mathbb{R}^d$. In this formulation, the encoder maps $x \to y$ via

$$y = f(Wx + b) \tag{2.13}$$

where $W \in \mathbb{R}^{(e \times d)}$, $b \in \mathbb{R}^e$, and $f(\cdot)$ is an activation function that imposes a non-linearity

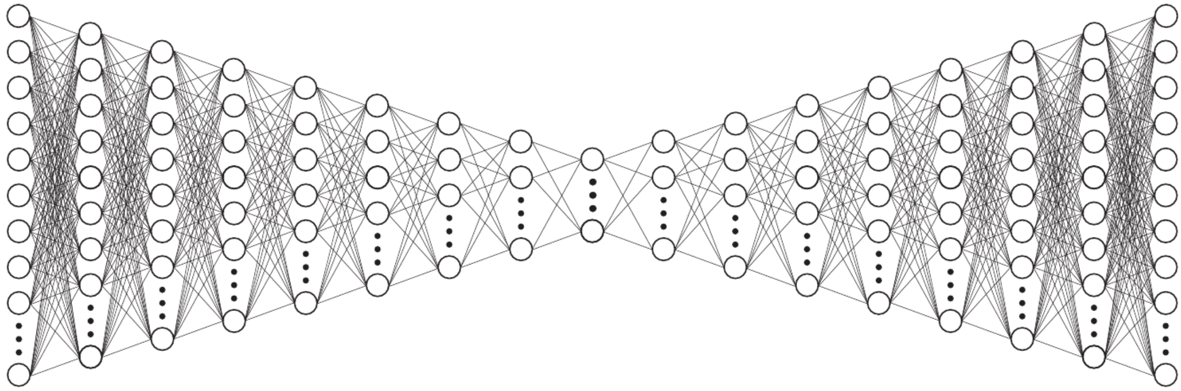


Figure 2.7: Multi-layer Autoencoder Topology

in the neural network. The decoder has a similar formulation:

$$\hat{x} = f(W_{\text{out}} y + b_{\text{out}}) \tag{2.14}$$

with $W_{\text{out}} \in \mathbb{R}^{(d \times e)}$, $b_{out} \in \mathbb{R}^d$.

A multi-layer autoencoder acts in much the same way as a single-layer autoencoder. The encoder contains $n > 1$ layers and the decoder contains $m > 1$ layers. Using equation 2.13 for each mapping, the encoder maps $x \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$. Treating x_n as y in equation 2.14, the decoder maps $x_n \rightarrow x_{n+1} \rightarrow \ldots \rightarrow x_{n+m} = \hat{x}$.

The autoencoder trains the weights of the W's and b's to minimize some cost function. This cost function should minimize the distance between input and output values.The choice of activation functions $f(\cdot)$ and cost functions depends on the domain of a given task.

Figure 2.7 depicts a generic typical multi-layer autoencoder. The mapping function can be read from left to right. The input x is represented by the leftmost column of neurons, and the output $\hat{x}$ is represented by the rightmost column. The smallest middle column is the "latent space" or "hiddenmost layer."

The autoencoders used in this work are tasked with encoding and reconstructing STFT

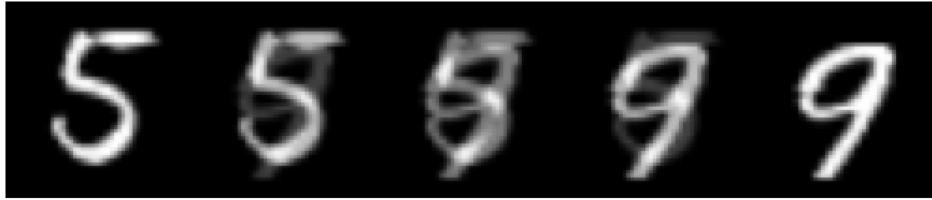


Figure 2.8: MNIST Example: Interpolation between 5 and 9 in the pixel space

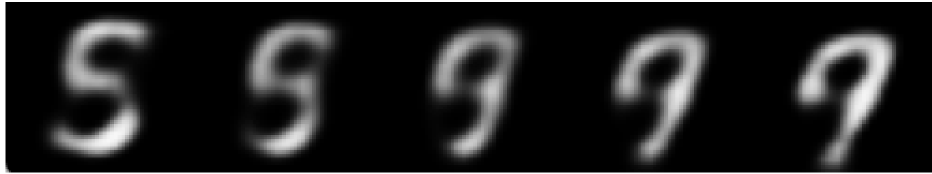


Figure 2.9: MNIST Example: Interpolation between 5 and 9 in the latent space

magnitude frames, with the aim of having the autoencoder produce a latent space that contains high level descriptors of musical audio.

2.3.3 Latent Spaces

The latent space that an autoencoder constructs can be viewed as a compact space that contains high level descriptions of the corpus used for training. In order for the autoencoder to perform well at reconstruction tasks, this latent space is forced to represent the most relevant descriptors of the corpus.

Once constructed the latent space can become a powerful creative tool, allowing for semantically meaningful interpolation between two inputs. Consider the following visual example.

The MNIST dataset consists of thousands of images of handwritten digits [13]. All images in the dataset are greyscale and are 28x28 pixels. Two images in the dataset are shown in Figure 2.8, as well as an interpolation between the two images in “pixel space.” The interpolation of these two images in the pixel space produces a cross-fading effect, whereby the 5 and 9 are proportionally summed and averaged. As can be seen, the middle interpolation makes little to no sense semantically (i.e. as a handwritten digit) and instead creates a jumble of pixels from both the 5 and 9. Fortunately, an autoencoder can be used to interpolate more

meaningful outputs between the 5 and 9.

An autoencoder can be trained to encode and decode these images, thereby constructing a latent space containing high level representations of handwritten digits. By interpolating images from the latent space, rather than the pixel space, semantically meaningful images are produced. The interpolation shown in Figure 2.9 shows a handwritten digit transforming from a 5 to a 9 rather than crossfading from one to the other.

This ability to interpolate meaningful and novel examples from a latent space is why the autoencoder is used in this work. Any DAW can crossfade two sounds for a musician, but few if any can interpolate between two input sounds in a semantically consistent manner. This thesis aims to present a novel tool that allows musicians to harness the capabilities of an autoencoder's latent space to generate novel audio.

2.3.4 Machine Learning Approaches to Musical Audio Synthesis

Recently, machine learning techniques have been applied to musical audio sythesis. One version of Google's Wavenet architecture uses convolutional neural networks (CNNs) trained on piano performance recordings to prooduce raw audio one sample at a time [25]. The outputs of this neural network have been described as sounding like a professional piano player striking random notes. Another topology, presented by Dadabots, uses recurrent neural networks (RNNs) trained to reproduce a given piece of music [1]. These RNNs are given a random initialization and then left to produce music in batches of raw audio samples. Another Google project, NSynth [7], uses autoencoders to interpolate audio between the timbres of different instruments. While all notable in scope and ability, these models require immense computing power to train. These requirements are often prohibitively expensive for an end user and thus do not allow for musicians to have any control over the design of the algorithm's architecture. For example, the architecture presented by Dadabots requires

24 hours to train on a GPU, and takes five minutes to generate 10 seconds of audio. In other words these barriers prevent musicians from having a meaningful dialogue with the tools they are given, which is a missed opportunity for creative innovation.

Another approach, proposed by Andy Sarroff, uses a small autoencoding neural network (autoencoder) [20]. Compared to the work of [25] and [7], this architecture has the advantage of being easy to train by new users. Furthermore, this lightweight architecture allows for real time tuning and audio generation.

In [20]'s implementation, the autoencoder's encoder compresses an input magnitude short-time Fourier transform (STFT) frame to a latent representation, and its decoder uses the latent representation to reconstruct the original input. The phase information bypasses the encoder/decoder entirely. By modifying the latent representation of the input, the decoder generates a new magnitude STFT frame. However, [20]'s proposed architecture suffers from poor performance, measured by mean squared error (MSE) of a given input magnitude STFT frame and its reconstruction. This poor MSE performance suggests a lack of robust encodings for input magnitude STFT frames and thus a poor range of musical applications. The work presented in Section 3 builds on these initial results and improves the designed autoencoder through modern techniques and frameworks. These improvements reduce MSE for each of [20]'s proposed topologies, thus improving the representational capacity of the neural network's latent space and widening the scope of the autoencoder's musical applications.

The work presented in Section 4 expands and improves on the work in Section 3. Section 4 introduces a new corpus more suited to designing a synthesizer as well as a larger and more powerful autoencoder architecture than that of Section 3. Encodings are improved by introducing a new cost function to the autoencoder's training regime. Finally a phase construction technique is implemented that can invert spectrograms generated by the autoencoder rather than relying on an input phase response.

Chapter 3

ANNe Sound Effect

This section outlines the experimental procedure done to reproduce and improve on the work presented in [20]. This includes exploring different neural network training methods, investigating different activation functions to be used in the neural network's hidden layers, training several different architectures, using regularization techniques, and weighing the pros and cons of the additive bias terms.

When implemented in code, this autoencoder is considered a "sound effect" rather than a "synthesizer." This is because the autoencoder generates no phase information and thus cannot invert the STFT on its own. Instead, the output STFT is inverted by using phase information passed from the input.

3.1 Architecture

Several different network topologies were used, varying the depth of the autoencoder, width of hidden layers, and choice of activation function. [20]'s topology is reproduced, using the Adam training method instead of using stochastic gradient descent with momentum 0.5.

Afterwards a seven layer deep autoencoder is designed that can be used for unique audio effect generation and audio synthesis. For both the single layer and three layer deep models, no additive bias term b was used, and all activations were the sigmoid (or logistic) function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.1}$$

The seven layer deep model uses both the sigmoid activation function and the rectified linear unit (ReLU) [16]. The ReLU is formulated as

$$f(x) = \begin{cases} 0 & , x < 0 \\ x & , x \geq 0 \end{cases} \tag{3.2}$$

As outlined in [20], the autoencoding neural network takes 1025 points from a 2048 point magnitude STFT frame as its input, i.e. $x \in [0, 1]^{1025}$. These 1025 points represent the DC and positive frequency values of a given frame's STFT.

A more in depth look at the neural network design choices made is in the **3.3 Discussion of Methods** section.

3.2 Corpus and Training Regime

All topologies presented in this section were trained using $50,000$ magnitude STFT frames, with an additional $10,000$ frames held out for testing and another $10,000$ for validation. Since [20]'s original corpus was not available, the audio used to generate these frames was an hour's worth of improvisation played on a MicroKORG synthesizer/vocoder. To ensure the integrity of the testing and validation sets, the dataset was split on the "clip" level. This means that the frames in each three sets were generated from distinct passages in the improvisation, which ensures that duplicate or nearly duplicate frames are not found across

the three sets. The MicroKORG has a maximum of four note polyphony for a given patch, thus the autoencoder must learn to encode and decode mixtures of at most four complex harmonic tones. These tones often have time variant timbres and effects, such as echo and overdrive.

The neural network framework was handled using TensorFlow [2]. All training used the Adam method for stochastic gradient descent with mini-batch size of 100 [11]. Across the different autoencoder topologies explored, learning rates used for training varied from 10^{-3} to 10^{-4}. For the single layer deep (Figure 3.1) and three layer deep (Figure 3.2) autoencoders, the learning rate was set to 10^{-3} for the duration of training, which was 300 epochs. For the deep autoencoder (Figure 3.3) the learning rate was set to 10^{-4} for the duration of training, which was 500 epochs. This is a departure from [20]'s proposed methodology, which used stochastic gradient descent with learning rate 5×10^{-3} and momentum 0.5 [24].

The encoder and decoder are trained on these magnitude STFT frames to minimize the MSE of the original and reconstructed frames.

3.3 Discussion of Methods

There are several distinctions between the architecture originally proposed by [20] and the architectures used in this work: the choice of the autoencoder's stochastic training method, the regularization techniques used to create a robust latent space, the activation functions chosen, the use of additive bias terms b, and the corpus used for training.

3.3.1 Training Methods

The improved MSEs in Table 3.1 and Table 3.2 demonstrate the ability of the Adam method to train autoencoders in this context better than the momentum method [20] used. The

momentum method produced MSEs orders of magnitude higher than Adam, suggesting that the momentum method found a poor local minimum and did not explore further. The result of the 8-neuron hidden layer in Figure 4.1 demonstrates the poor reconstructions produced by an autoencoder with MSE on the order of 10^{-3}. [20]'s performance suggests that the momentum method produced similar results. While these reconstructions are interesting to listen to, they do not accurately reconstruct an input magnitude STFT frame. The adaptive properties of the Adam technique ensure that the autoencoder searches the weight space in order to find robust minima.

3.3.2 Regularization

[20] suggested using denoising techniques to improve the robustness of autoencoder topologies. During the course of the work presented here it was found that denoising was not necessary to create robust one and three layer deep autoencoders. However, issues were encountered when training the seven layer deep autoencoder topology.

Two regularization techniques were explored: dropout and an $l2$ penalty [22] [12]. Dropout involves multiplying a Bernoulli random vector $z \in \{0, 1\}^{t_i}$ to each layer in the autoencoder, with t_i equal the dimension of the ith layer. Dropout encourages robustness in an autoencoder's encoder and decoder, and the autoencoder's quantitative performance did reflect this. However, it was found that the dropout regularizer hampered the expressiveness of the autoencoder when generating audio because it ignored slight changes to the latent space.

The second technique, $l2$ regularization, proved to perform the best in qualitative listening comparisons. This technique imposes the following addition to the cost function:

$$C(\theta_n) = \frac{1}{N_{obs}} \sum_{k=1}^{N_{obs}} (\hat{x} - x)^2 + \lambda_{l2} \|\theta_n\|_2 \tag{3.3}$$

where λ_{l2} is a tuneable hyperparameter and $\|\theta_n\|_2$ is the Euclidean norm of the autoencoder's weights. This normalization technique encourages the autoencoder to use smaller weights in training, which was found to improve convergence.

3.3.3 Activation Functions

It was found that when using sigmoids as activation functions throughout the seven layer deep model, the autoencoder did not converge. This is potentially due to the vanishing gradient problem inherent to deep training [10]. To fix this, sigmoid activations were used on only the deepest hidden layer and the output layer. Rectified linear units (ReLUs) were used for the remaining the layers. This activation function has the benefit of having a gradient of either zero or one, thus avoiding the vanishing gradient problem.

The choice of sigmoid activation for the deepest hidden layer of the deep model was motivated by the use of multiplicative gains for audio modulation. Because the range of the sigmoid is strictly greater than zero, multiplicative gains were guaranteed to have an effect on the latent representation, whereas a ReLU activation may be zero, thus invalidating multiplicative gain.

The choice of sigmoid activation for the output layer of the deep model was twofold. First, the normalized magnitude STFT frames used here have a minimum of zero and maximum of one, which neatly maps to the range of the sigmoid function. Second, it was found that while the ReLU activation on the output would produce acceptable MSEs, the sound of the reconstructed signal was often tinny. The properties of the sigmoid activation lend themselves to fuller sounding reconstructions.

3.3.4 Additive Bias

Finally, it was found that using additive bias terms b in Equation 2.13 created a noise floor on output STFT frames. With the bias term present, using gain constants in the hidden layer produced noisy results. Though additive bias terms did improve the convergence of the deep autoencoder, they were ultimately left out in the interest of musical applications.

3.3.5 Corpus

An issue with [20] is the use of several genres of music to generate a dataset. As different genres have different frequency profiles, the neural network's performance drops. For example, a rock song's frequency profile can be broken down as the sum of spiky low frequency content created by drums, tonal components from guitar and bass, complex vocal profiles, and high frequency activity from cymbals. Including several genres of music in a corpus trains an autoencoder to be a jack of all trades, but master of none. By focusing the corpus on tonal sounds, this work encourages the neural network to master representations of those sounds. Thus when it comes time for modifying an input, the autoencoder's latent space contains representations of similar yet distinct synthesizer frequency profiles.

3.4 Optimization Method Improvements

Table 3.1 and Table 3.2 compare the MSEs of the network topologies as implemented by [20] (Sarroff) and as implemented here (ANNe).

The first column of Table 3.1 describes the autoencoder's topology, with the first integer representing the neuron width of the hidden layer. The first column of Table 3.2 describes the autoencoder's topology, with the first integer representing the neuron width of the first

Table 3.1: Single Layer Autoencoder Topology MSEs

Hidden Layer Width	Sarroff MSE	ANNe MSE
8	4.40×10^{-2}	5.30×10^{-3}
16	4.14×10^{-2}	5.28×10^{-3}
64	2.76×10^{-2}	7.10×10^{-4}
256	1.87×10^{-2}	1.64×10^{-4}
512	1.98×10^{-2}	9.62×10^{-4}
1024	3.52×10^{-2}	7.13×10^{-5}

Table 3.2: Three Layer Autoencoder Topology MSEs

Hidden Layer Widths	Sarroff MSE	ANNe MSE
256-8-256	1.84×10^{-2}	1.91×10^{-3}
256-16-256	1.84×10^{-2}	1.19×10^{-3}
256-32-256	1.84×10^{-2}	7.30×10^{-4}

layer, the second integer representing the neuron width of the second layer, and the third integer representing the neuron width of the third layer. Table 3.3 shows the MSEs of a seven layer deep autoencoder, with hidden layer widths $512 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 128 \rightarrow 256 \rightarrow 512$. Table 3.3 also shows the MSEs of three different topologies that were chosen for the deep autoencoder: one with sigmoid activations throughout, one with ReLU activations throughout, and a hybrid model. This hybrid model used a sigmoid on the innermost hidden layer and on the output layer, with all other layers using a ReLU activation. This hybrid topology performed best in minimizing MSE. All train times were measured using TensorFlow 1.2.1 running on an Intel® Core™ i5-6300HQ CPU @ 2.30GHz.

Figure 4.1 shows graphs of a single input magnitude STFT frame (top) and corresponding reconstructions (bottom), with magnitude on the y axis and frequency bin on the x axis. Contrary to [20]'s work, the signal reconstruction improves both qualitatively and quantitatively as the depth of the hidden layer is increased.

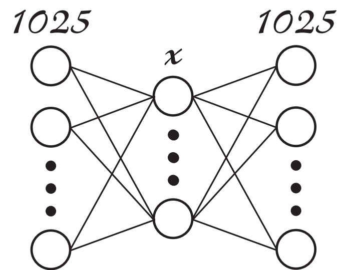


Figure 3.1: The topology described in Table 3.1. x represents the varied width of the hidden layer.

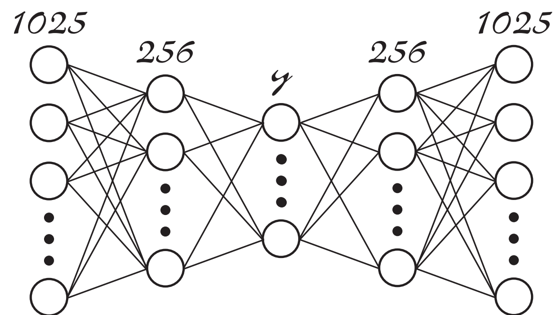


Figure 3.2: The topology described in Table 3.2. y represents the varied width of the deepest hidden layer.

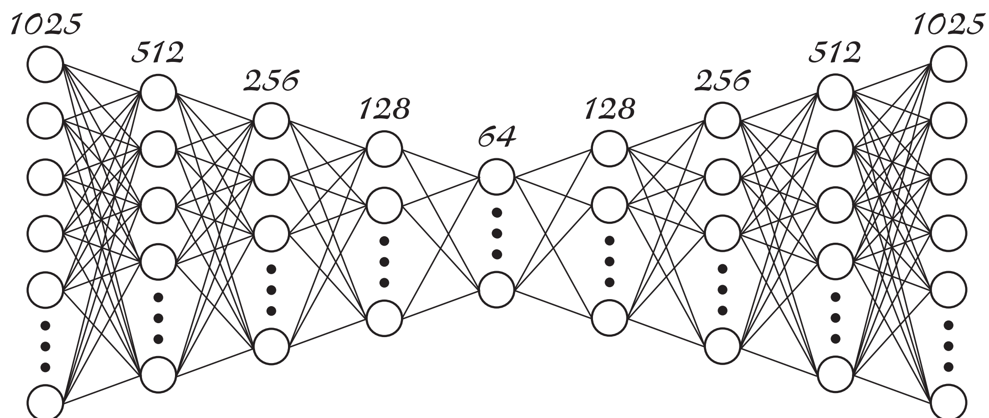


Figure 3.3: The "deep" autoencoder topology described in Table 3.3.

Figure 3.4: Plots demonstrating how autoencoder reconstruction improves when the width of the hidden layer is increased.

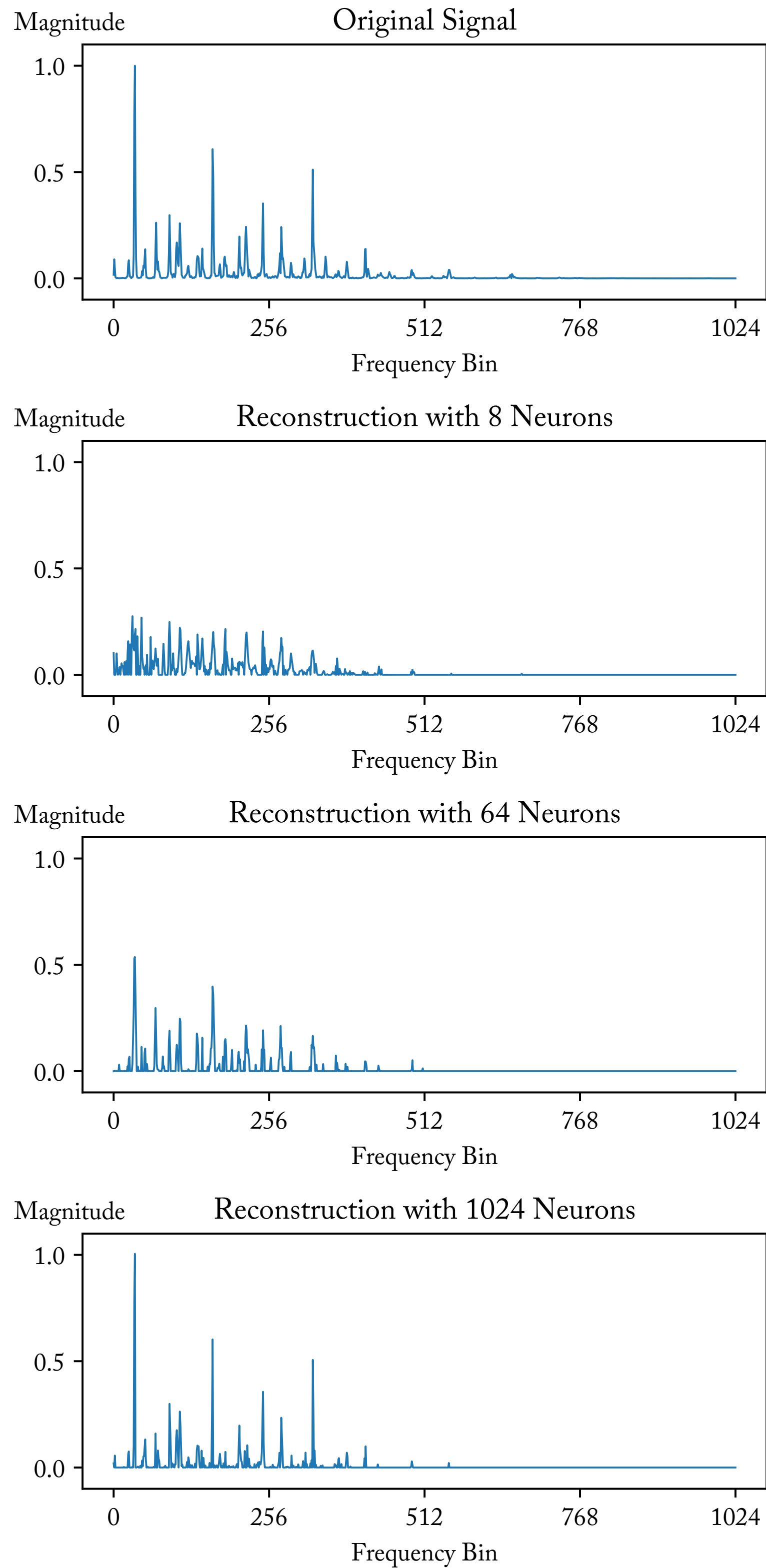
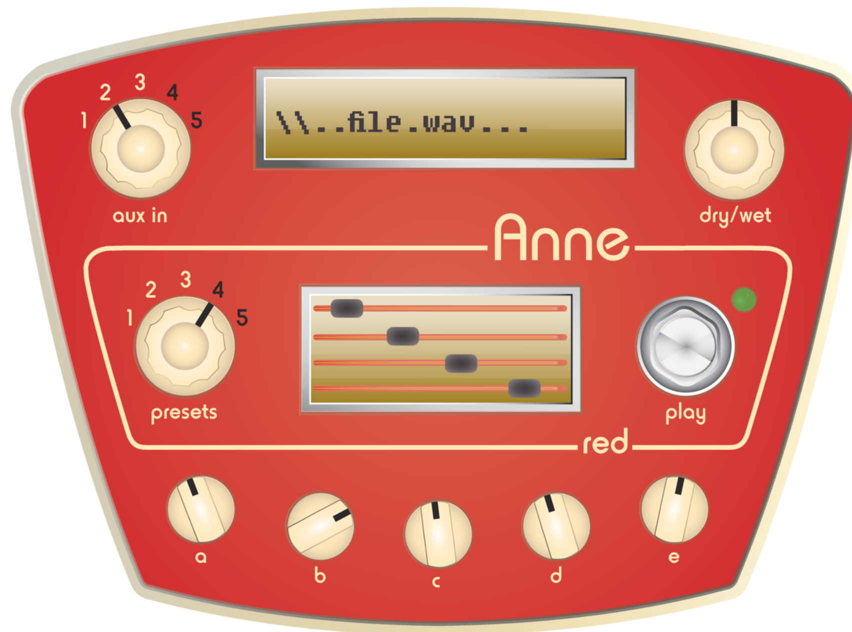


Table 3.3: Deep Topology MSEs and Train Times

Activations	MSE	Time to Train
All Sigmoid	1.72×10^{-3}	20 minutes
All ReLU	8.00×10^{-2}	60 minutes
Hybrid	4.91×10^{-4}	25 minutes

Figure 3.5: The ANNe interface



3.5 ANNe GUI

3.5.1 Interface

The following section present a graphical user interface (GUI) 'ANNe' that allows users to modify audio signals with the neural network presented in this section. First, a user loads an audio file that they want to modify. Then, they adjust values that modify the autoencoding neural network. Finally, the program processes the input file through the neural network which outputs a new audio file. The GUI's front end was coded in C++ using Qt Creator and interacts with a backend coded in Python3.

All audio processing was handled by the librosa Python library [14]. In this application,

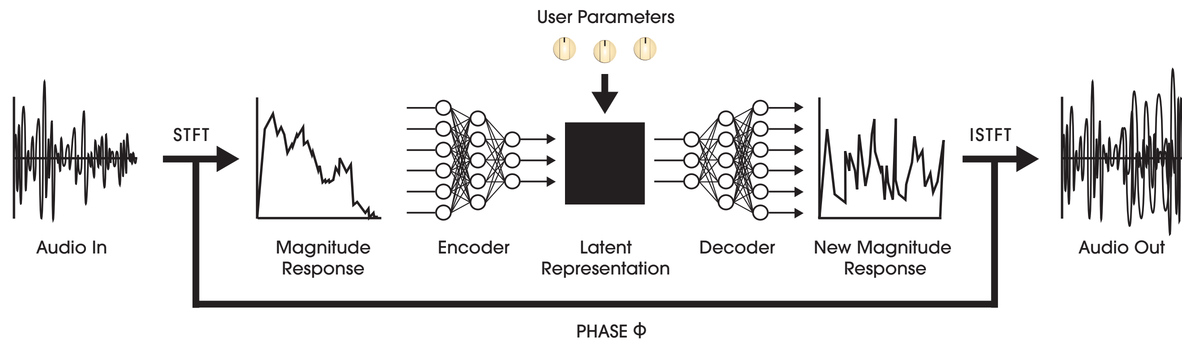


Figure 3.6: Block diagram demonstrating ANNe's signal flow

librosa was used to read *.wav* files, sample them at 22.05kHz, perform STFTs of length 2048 with centered Hann window, hop length 512 (75% overlap), and write *.wav* files with sampling frequency 22.05kHz from reconstructed magnitude STFT frames. The phase of each magnitude STFT frame was passed directly from input to output, circumventing the autoencoder. ANNe is available on github at `https://github.com/JTColonel/ANNe` and has been tested on Ubuntu 16.04 LTS and Arch Linux distributions.

3.5.2 Functionality

On startup ANNe initializes the Python backend, which loads the neural network topology and saved network weights.

ANNe begins processing the input audio signal by performing a STFT. On a frame-by-frame basis the inputs phase is saved untouched in memory, but the magnitude response is normalized to $[0, 1]$. The normalizing factor is saved and stored in memory, and the normalized magnitude response is passed to the neural network. The neural network first encodes the normalized magnitude response into a 64 dimensional latent space, and then subsequently decodes the latent representation into a new normalized magnitude response. ANNe then reapplies the frames normalizing factor, and performs an ISTFT with the original phase.

3.5.3 Guiding Design Principles

In total, ANNe contains nearly 4000 neurons and 1.4 million weight constants that could be modulated to create an audio effect. In practice, however, it is unreasonable to present an entry level user with so many tunable parameters and expect them to use a program. The challenge was to design an interface that would be immediately recognizable to musicians and sound designers and allow users to access the full potential of the neural network. As such, guitar pedals were used for inspiration.

Despite having innumerable design parameters, guitar pedals expose only a handful of them to a user. Furthermore, by presenting knobs and dials as actuators, guitar pedal designers simultaneously presents users with an intuitive interface while tacitly limiting the parameters altering the pedals circuitry.

With these principles in mind, ANNe allows users to modify the 64 neurons in the hidden layer with knobs and sliders.

3.5.4 Usage

ANNe does not process audio in real time. Instead, ANNe processes an entire audio file in one go. Thus in order to begin using ANNe, a user first must load a file that they want to modify. A user chooses the file they would like to load by turning the *aux in* dial. Settings 1-3 are prepackaged *.wav* files (a piano striking middle C, a violin section tuning, and a snare drum hit respectively), and settings 4-5 allow a user to select their own file.

The choice of preset sounds are intentional - it was chosen to give the user a sense of the range of sound domains we found worked well with ANNe. First, all the preset sounds are no longer than two seconds. While it is possible to load a track of any length into the program, more interesting results were found when working with small clips of audio. Second, the

presets encompass a wide range of harmonic and timbral complexity. In testing, ANNe could scramble the piano notes harmonic profile to sound like a bass kick, and the snares to sound like an 808 cowbell.

After choosing a sound to modify the user then selects a *preset* and tunes the knobs a-e. These are what allow a user to modify the 64 neuron hidden layer. Knobs a through e specify multiplicative gain constants that get sent to the hidden layer, and the *preset* specifies how those gains get mapped to the hidden layer.

Each preset 1-3 divides the hidden layer vector H into five adjacent, non-overlapping sub-vectors $H = \{h_a, h_b, ..., h_e\}$

When it comes time to process an input file, each neuron in a sub-vector is multiplied by the gain constant specified by its corresponding knob *after the sigmoid activation*. For example, each neuron contained in h_b is multiplied by the gain set by knob b. The *presets* determine how long each sub-vector is: preset 1 has each sub-vector approximately equal in length, preset 2 increases their size logarithmically from “a” to “e”, and preset 3 decreases their size logarithmically from “a” to “e”. Options 4 and 5 allow for a user to specify their own sub-vector lengths by moving the sliders and saving their positions for later use.

The GUI places constraints so that each sub-vector contains at least one neuron and so that the gain constants can take a value in $[0.5, 5.5]$.

Finally, the user tunes the dry/wet knob and hits play to process and listen to the output file.The dry/wet knob adds the original and output file in proportion. When the knob is set to a value of x the output file becomes

$$y_{out} = \frac{(1-x) * y_{\text{pre-processed}} + x * y_{\text{post-processed}}}{\|(1-x) * y_{\text{pre-processed}} + x * y_{\text{post-processed}}\|} \tag{3.4}$$

This allows a user to mix the original audio clip with the audio output by the neural network.

3.5.5 “Unlearned” Audio Representations

During training, the sigmoid activation at the hidden layer forces the autoencoder to map input magnitude responses inside a 64 dimensional unit hypercube within a 64 dimensional latent space. This hypercube contains “learned” representations of the corpus. ANNe’s multiplicative gain constants allows a user to alter how an input signal is mapped to this latent space, or to put it another way, ANNe allows a user to morph an input signals latent representation in 64 dimensional space.

If the gain constants a-e are all set less than or equal to 1, it is guaranteed that the signal’s latent representation is moving within the “learned” space, or unit hypercube. However, when values are greater than 1, that condition is no longer guaranteed. If, say, the gain constant a is set to 4, and some neurons in h_a are greater than 0.25, then ANNe has pushed the signals latent representation out of the “learned” space and into what we call an “unlearned” space.

After testing the model, it was found that setting gain constants greater than two frequently pushed a signal’s latent representation into the unlearned space. With gain constants greater than 2 the output signals were found to be harmonically distinct from the input signal, whereas applying very large gains on the order of 10^2 before the sigmoid activation (thus pushing the hidden layer’s neurons towards a value of zero or one, i.e. the boundary of the learned space) generated output signals harmonically similar to the input. By allowing users to access audio from this unlearned space, ANNe allows users to explore completely novel sound.

Chapter 4

CANNe Synthesizer

This section expands on the work presented in Section 3 and presents CANNe, an autoencoding neural network synthesizer. CANNe acts in much the same way as ANNe, but includes a phase construction technique. This allows CANNe to generate audio directly from activating its hiddenmost layer, rather than relying on the phase response of an input signal.

By using some of the design principles that led to ANNe, the autoencoder was made much deeper (from seven to seventeen layers). The choice of overall architecture, corpus, cost function, and feature engineering are presented and justified.

A GUI is also presented at the end of the section.

4.1 Architecture

A fully-connected, feed-forward neural network acts as the autoencoder. Refer to Figure 4.1 for an explicit diagram of the network architecture. The number above each column of neurons represents the width of that hidden layer. Design decisions regarding activation

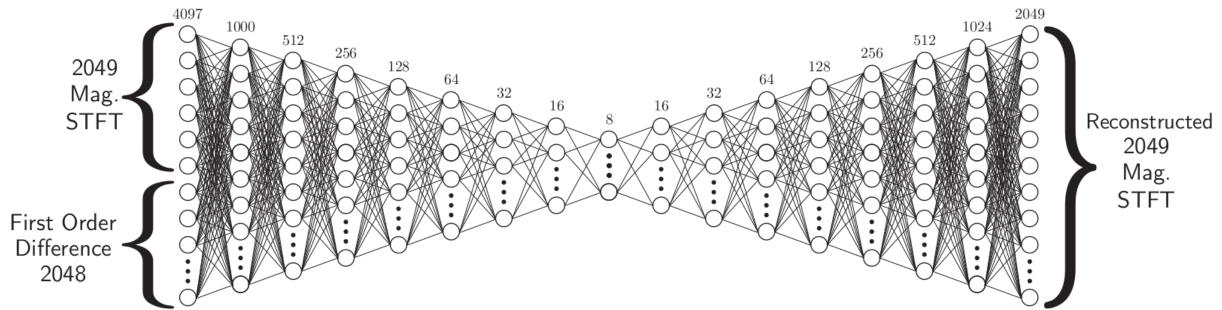


Figure 4.1: Final CANNe Autoencoder Topology

functions, input augmentation, and additive biases are discussed below.

In order for training to converge, the ReLU was chosen as the activation function for all layers of the autoencoder. Though a hybrid model was shown to have the best performance in the previous section, the sigmoid activations in the hidden layer and output layer prevented the autoencoder from converging during training. It is suspected that the vanishing gradient problem is to blame [10].

4.2 Corpus

In this work a multi-layer neural network autoencoder is trained to learn representations of musical audio. The aim is to train an autoencoder to contain high level, descriptive audio features in a low dimensional latent space that can be reasonably handled by a musician. As in the formulation above, dimension reduction is imposed at each layer of the encoder until the desired dimensionality is reached.

The autoencoding neural network used here takes 2049 points from a 4096-point magnitude STFT $s_n(m)$ as its target, where n denotes the frame index of the STFT and m denotes the frequency index. Each frame is normalized to $[0, 1]$. This normalization allows the autoencoder to focus solely on encoding the timbre of an input observation and ignore its

loudness relative to other observations in the corpus.

Two corpora were used to train the autoencoder in two separate experiments. The first corpus is comprised of approximately $79,000$ magnitude STFT frames, with an additional $6,000$ frames held out for testing and another $6,000$ for validation. This makes the corpus $91,000$ frames in total. The audio used to generate these frames is composed of five octave C Major scales recorded from a MicroKORG synthesizer/vocoder across 80 patches.

The second corpus is a subset of the first. It is comprised of one octave C Major scales starting from concert C. Approximately $17,000$ frames make up the training set, with an additional $1,000$ frames held out for testing and another $1,000$ for validation.

In both cases, 70 patches make up the training set, 5 patches make up the testing set, and 5 patches make up the validation set. These patches ensured that different timbres are present in the corpus. To ensure the integrity of the testing and validation sets, the dataset is split on the "clip" level. This means that the frames in each of the three sets are generated from distinct passages in the recording, which prevents duplicate or nearly duplicate frames from appearing across the three sets.

By restricting the corpus to single notes played on a MicroKORG, the autoencoder needs only to learn higher level features of harmonic synthesizer content. These tones often have time variant timbres and effects, such as echo and overdrive. Thus the autoencoder is also tasked with learning high level representations of these effects.

4.3 Cost Function

Three cost functions were considered for use in this work:

Spectral Convergence (SC) [23]

$$C(\theta_n) = \sqrt{\frac{\sum_{m=0}^{M-1}(s_n(m) - \hat{s}_n(m))^2}{\sum_{m=0}^{M-1}(s_n(m))^2}} \tag{4.1}$$

where θ_n is the autoencoder's trainable weight variables,$s_n(m)$ is the original magnitude STFT frame, $\hat{s}_n(m)$ is the reconstructed magnitude STFT frame, and M is the total number of frequency bins in the STFT

Mean Squared Error (MSE)

$$C(\theta_n) = \frac{1}{M}\sum_{m=0}^{M-1}(s_n(m) - \hat{s}_n(m))^2 \tag{4.2}$$

and Mean Absolute Error (MAE)

$$C(\theta_n) = \frac{1}{M}\sum_{m=0}^{M-1}|s_n(m) - \hat{s}_n(m)| \tag{4.3}$$

Ultimately, SC (Eqn. 4.1) was chosen as the cost function for this autoencoder instead of mean squared error (MSE) or mean absolute error (MAE).

The decision to use SC is twofold. First, its numerator penalizes the autoencoder in much the same way mean squared error (MSE) does. Reconstructed frames dissimilar from their input are penalized on a sample-by-sample basis, and the squared sum of these deviations dictates magnitude of the cost. This ensures that SC is a valid cost function because perfect

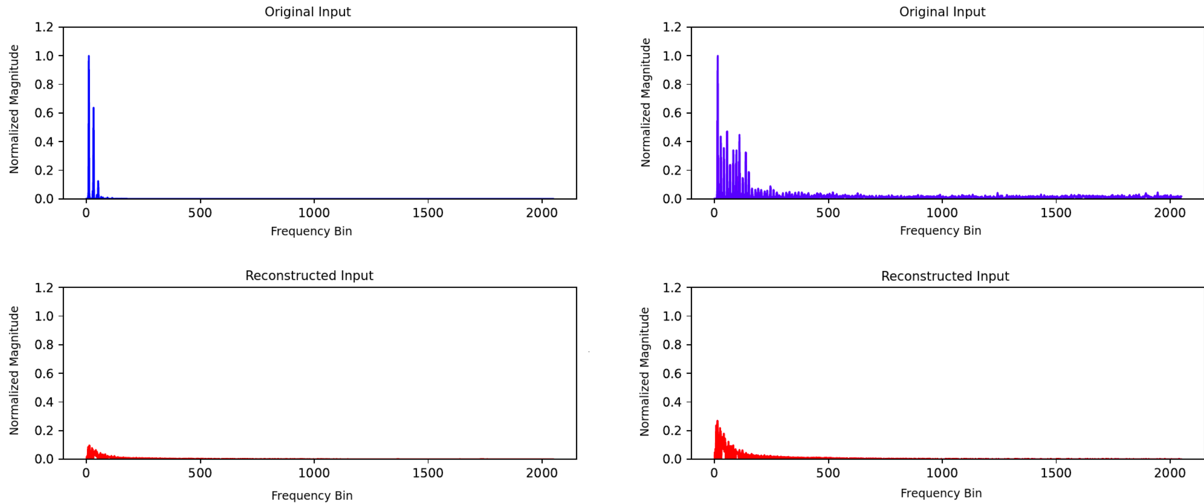


Figure 4.2: Autoencoder Reconstructions without L2 penalty

reconstructions have a cost of 0 and similar reconstructions have a lower cost than dissimilar reconstructions.

The second reason, and the primary reason SC was chosen over MSE, is that its denominator penalizes the autoencoder in proportion to the total spectral power of the input signal. Because the training corpus used here is comprised of "simple" harmonic content (i.e. not chords, vocals, percussion, etc.), much of a given input's frequency bins will have zero or close to zero amplitude. SC's normalizing factor gives the autoencoder less leeway in reconstructing harmonically simple inputs than MSE or MAE. Refer to Figure 4.3 for diagrams demonstrating the reconstructive capabilities each cost function produces.

As mentioned in [3], the autoencoder does not always converge when using SC by itself as the cost function. See Figure 4.2 for plotted examples. Thus, an L2 penalty is added to the cost function

$$C(\theta_n) = \sqrt{\frac{\sum_{m=0}^{M-1}(s_n(m) - \hat{s}_n(m))^2}{\sum_{m=0}^{M-1}(s_n(m))^2}} + \lambda_{l2}\|\theta_n\|_2 \tag{4.4}$$

where λ_{l2} is a tuneable hyperparameter and $\|\theta_n\|_2$ is the Euclidean norm of the autoencoder's weights [12]. This normalization technique encourages the autoencoder to use smaller

weights in training, which was found to improve convergence. For this work λ_{l2} is set to 10^{-10}. This value of λ_{l2} is large enough to prevent runaway weights while still allowing the SC term to dominate in the loss evaluation.

4.4 Feature Engineering

To help the autoencoder enrich its encodings, its input is augmented with higher-order information. Augmentations with different permutations of the input magnitude spectrum's first-order difference,

$$x_1[n] = x[n+1] - x[n] \tag{4.5}$$

second-order difference,

$$x_2[n] = x_1[n+1] - x_1[n] \tag{4.6}$$

and Mel-Frequency Cepstral Coefficients (MFCCs) were used.

MFCCs have seen widespread use in automatic speech recognition, and can be thought of as the "spectrum of the spectrum." In this application, a 100 band mel-scaled log-transform of $s_n(m)$ is taken. Then, a 50-point discrete-cosine transform is performed. The resulting aplitudes of this signal are the MFCCs. Typically the first few cepstral coefficients are orders of magnitude larger than the rest, which can impede training. Thus before appending the MFCCs to the input, the first five cepstral values are thrown out and the rest are normalized to [-1,1].

4.5 Task Performance and Evaluation

Tables 4.1 and 4.2 show the SC loss on the validation set after training. For reference, an autoencoder that estimates all zeros for any given input has a SC loss of 1.0. All train

Input Append	Validation SC	Training Time
No Append	0.257	25 minutes
1st Order Diff	0.217	51 minutes
2nd Order Diff	0.245	46 minutes
1st and 2nd Order Diff	0.242	69 minutes
MFCCs	0.236	29 minutes

Table 4.1: 5 Octave Dataset Autoencoder validation set SC loss and Training Time

Input Append	Validation SC	Training Time
No Append	0.212	5 minutes
1st Order Diff	0.172	6 minutes
2nd Order Diff	0.178	6 minutes
1st and 2nd Order Diff	0.188	7 minutes
MFCCs	0.208	6 minutes

Table 4.2: 1 Octave Dataset Autoencoder validation set SC loss and Training Time

times presented were measured by training the autoencoder for 300 epochs, using the Adam method with mini-batch size 200, on an Nvidia Titan V GPU.

As demonstrated, the appended inputs to the autoencoder improve over the model with no appendings. Results show that while autoencoders are capable of constructing high level features from data unsupervised, providing the autoencoder with common-knowledge descriptive features of an input signal can improve its performance.

The model trained by augmenting the input with the signal's 1st order difference (1st-order-appended model) outperformed every other model. Compared to the 1st-order-appended model, the MFCC trained model often inferred overtonal activity not present in the original signal (Figure 4.4). While it performs worse on the task than the 1st-order-append model, the MFCC trained model presents a different sound palette that a musician may find interesting.

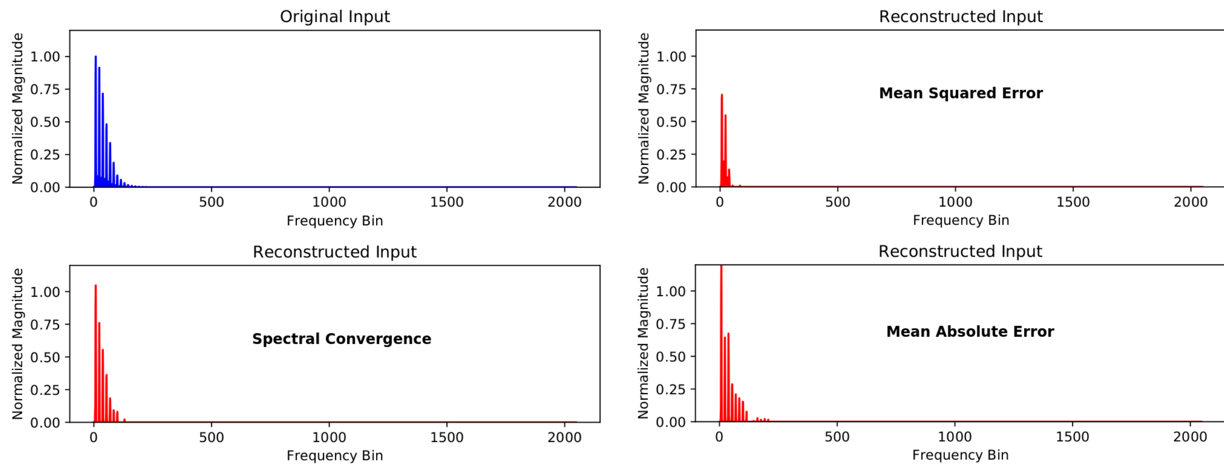


Figure 4.3: Sample input and recosntruction using three different cost functions

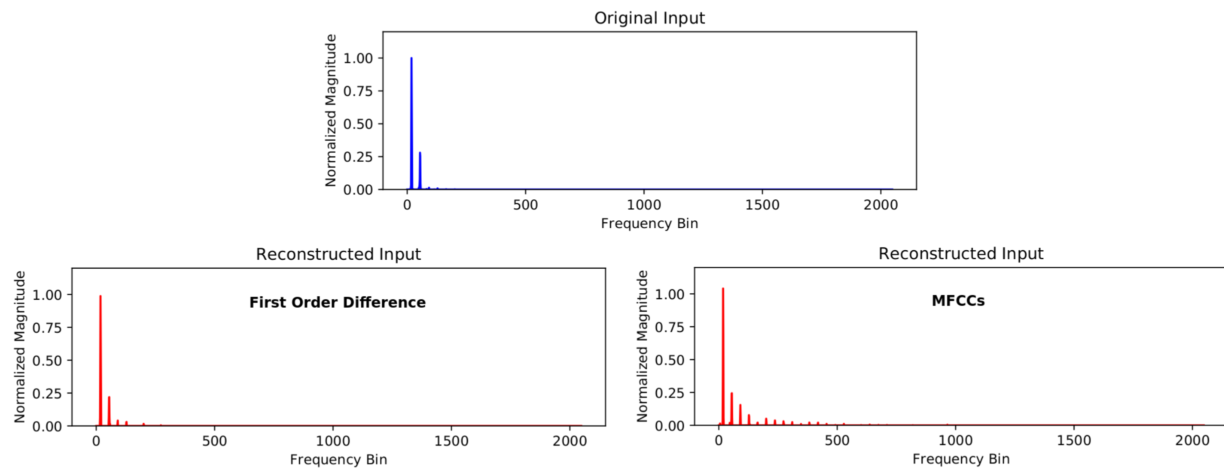


Figure 4.4: Sample input and recosntruction using three different cost functions

4.6 Spectrogram Generation

The training scheme outline above forces the autoencoder to construct a latent space contained in $\mathbb{R}^8$ that contains representations of synthesizer-based musical audio. Thus a musician can use the autoencoder to generate spectrograms by removing the encoder and directly activating the 8 neuron hidden layer. However, these spectrograms come with no phase information. Thus to obtain a time signal, phase information must be generated as well.

4.7 Phase Generation with PGHI

Phase Gradient Heap Integration (PGHI) [19] is used to generate the phase for the spectrogram.

An issue arises when using PGHI with this autoencoder architecture. A spectrogram generated from a constant activation of the hidden layer contains constant magnitudes for each frequency value. This leads to the phase gradient not updating properly due to the 0 derivative between frames. To avoid this, uniform random noise drawn from $[0.999, 1.001]$ is multiplied to each magnitude value in each frame. By multiplying this noise rather than adding it, we avoid adding spectral power to empty frequency bins and creating a noise floor in the signal.

4.8 CANNE GUI

We realized a software implementation of our autoencoder synthesizer, "CANNe (Cooper's Autoencoding Neural Network)" in Python using TensorFlow, librosa, pygame, soundfile, and Qt 4.0. Tensorflow handles the neural network infrastructure, librosa and soundfile handle audio processing, pygame allows for audio playback in Python, and Qt handles the GUI. Figure 4.6 depicts CANNe's signal flow.

Figure 4.5 shows a mock-up of the CANNe GUI, and Figure 4.6 shows CANNe's signal flow. A musician controls the eight Latent Control values to generate a tone. The Frequency Shift control performs a circular shift on the generated magnitude spectrum, thus effectively acting as a pitch shift. It is possible, though, for very high frequency content to roll into the lowest frequency values, and vice-versa.

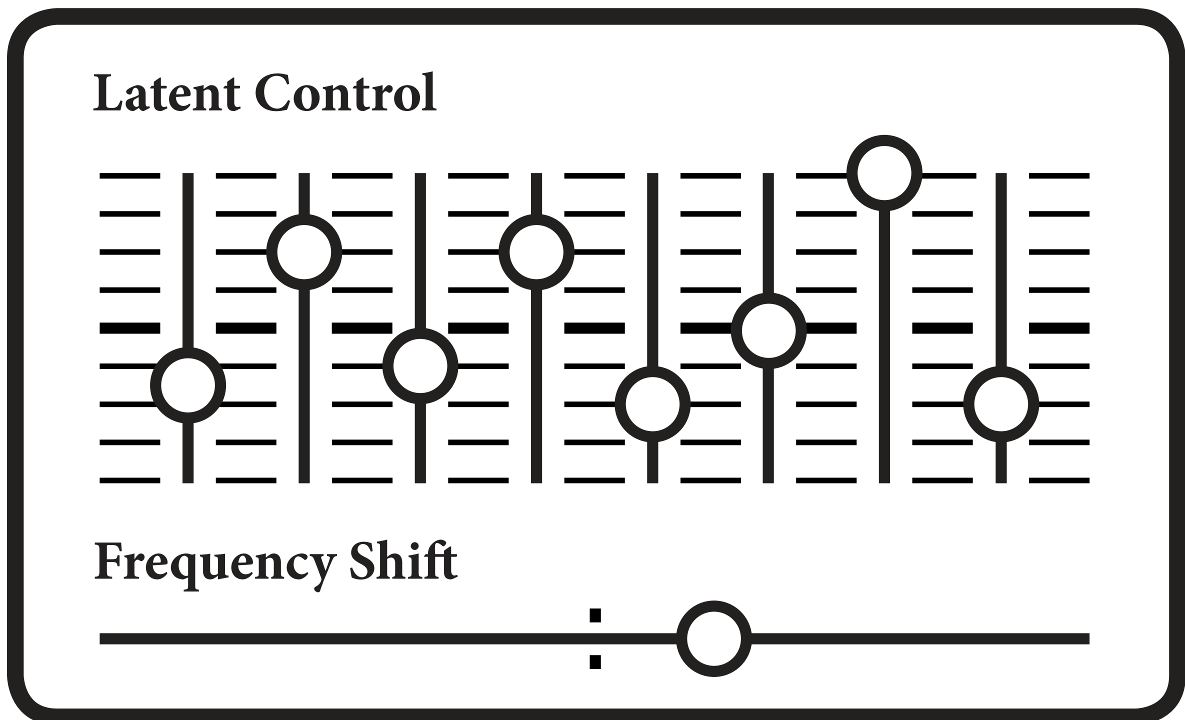


Figure 4.5: Mock-up GUI for CANNe.

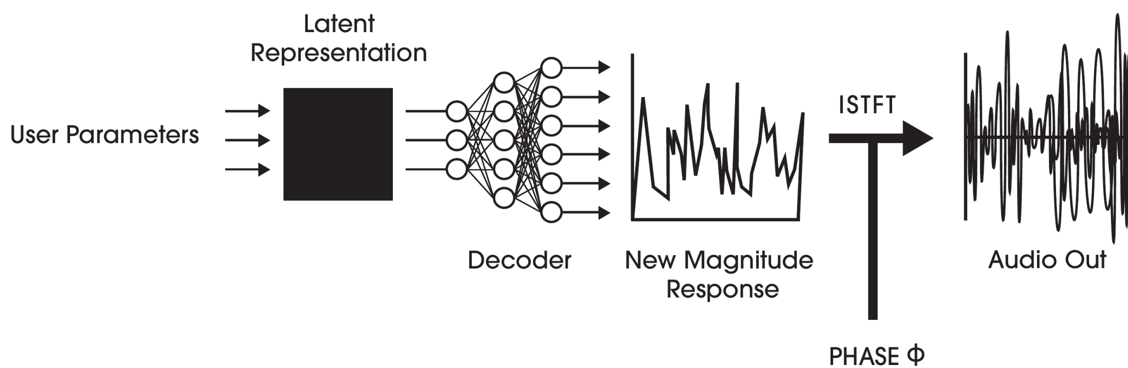


Figure 4.6: Signal flow for CANNe.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Two methods of training and implementing autoencoding neural networks for musical audio applications are presented. Both autoencoders are trained to compress and reconstruct Short-Time Fourier Transform magnitude frames of musical audio.

The first autoencoder, ANNe, acts as a sound effect. An input sound's spectrogram is mapped to the autoencoder's latent space, where it can be modified by a user. The newly modified latent representation is then sent through the decoder, and an altered spectrogram is produced. This spectrogram is inverted using the original signal's phase information to generate new audio.

The second autoencoder, CANNe, acts as a musical audio synthesizer. An autoencoder is trained on recordings of C major scales played on a MicroKORG synthesizer, thereby constructing a latent space that contains high level representations of individual notes. A user can then send activations to the latent space, which when fed through the decoder produce a spectrogram on the output. Phase gradient heap integration is used to construct

a phase response from the spectrogram, which is then used to invert the spectrogram into a time signal.

Each autoencoder has been implemented with a GUI that was designed with the musician in mind. Both topologies are lightweight when compared to state-of-the-art neural methods for audio synthesis, which allows the musician to have a meaningful dialogue with the network architecture and training corpora.

5.2 Future Work

The author sees three main directions in which future work can take for this thesis.

First, it would be worthwhile to explore using variational autoencoders (VAE) rather than standard autoencoders [8]. VAEs target a Gaussian distribution at the latent space rather than a deterministic latent space. Cost function parameters can be placed on these latent distributions to encourage them to decouple. Experiments with VAEs have demonstrated that this decoupling can increase the ease-of-use for creative tools.

Second, alternative representations of audio can be used to train the autoencoder. Representations such as the constant-Q transform offer more compact representations than the STFT [8]. Though more dense than the STFT, these transforms offer more resolution in the lower frequency bands of a signal, where human beings can distinguish more detail in audio, than the higher frequency bands. Though perfect reconstruction is not guaranteed with such transforms, machine learning techniques have been applied to minimize reconstruction error.

Finally, a more robust dataset that tags note information along with the raw STFT may allow the autoencoders to perform better on reconstruction tasks [8]. Conditioning inputs to the autoencoder based on its note may help musicians design sounds with a target root

harmonic.

Bibliography

[1] Generating black metal and math rock: Beyond bach, beethoven, and beatles. http://dadabots.com/nips2017/generating-black-metal-and-math-rock.pdf, Zack Zukowski and Cj Carr 2017.

[2] M. Abadi. Tensorflow: Learning functions at scale. *ICFP*, 2016.

[3] J. Colonel, C. Curro, and S. Keene. Improving neural net auto encoders for music synthesis. In *Audio Engineering Society Convention 143*, Oct 2017.

[4] J. Colonel, C. Curro, and S. Keene. Neural network autoencoders as musical audio synthesizers. *Proceedings of the 21st International Conference on Digital Audio Effects (DAFx-18). Aveiro, Portugal*, 2018.

[5] P. Doornbusch. Computer sound synthesis in 1951: The music of csirac. *Computer Music Journal*, 28(1):1025, 2004.

[6] P. Doornbusch. Computer sound synthesis in 1951: The music of csirac. *Computer Music Journal*, 28(1):1025, 2004.

[7] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, and M. Norouzi. Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders. *ArXiv e-prints*, Apr. 2017.

[8] P. Esling, A. Bitton, et al. Generative timbre spaces with variational audio synthesis. *Proceedings of the 21st International Conference on Digital Audio Effects (DAFx-18). Aveiro, Portugal*, 2018.

[9] B. Gold, N. Morgan, and D. Ellis. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music*. Wiley, 2011.

[10] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[12] A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In *NIPS*, volume 4, pages 950–957, 1991.

[13] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[14] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, pages 18–25, 2015.

[15] T. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[16] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[17] A. Oppenheim and R. Schafer. *Discrete-Time Signal Processing*. Pearson Education, 2011.

[18] Z. Pruša, P. Balazs, and P. L. Søndergaard. A noniterative method for reconstruction of phase from stft magnitude. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 25(5):1154–1164, 2017.

[19] Z. Pruša and P. L. Søndergaard. Real-time spectrogram inversion using phase gradient heap integration. In *Proc. Int. Conf. Digital Audio Effects (DAFx-16)*, pages 17–21, 2016.

[20] A. Sarroff. Musical audio synthesis using autoencoding neural nets, Dec 2015.

[21] J. Smith, X. Serra, S. U. C. for Computer Research in Music, Acoustics, and C. System Development Foundation (Palo Alto. *PARSHL: an analysis/synthesis program for non-harmonic sounds based on a sinusoidal representation.* Number no. 43 in Report. CCRMA, Dept. of Music, Stanford University, 1987.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[23] N. Sturmel and L. Daudet. Signal reconstruction from stft magnitude: A state of the art.

[24] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[25] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[26] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

Appendix A

Python Code

A.1 ANNe Backend

```
1 import tensorflow as tf
2 import tensorflow.contrib.slim as slim
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import os
6 import librosa
7 import sys
8 #
   |
9
10 sess = tf.Session()
11
12 #Creates weight variables for the ANN and groups them in a collection for use
     in L2 regularization
13 def weight_variable(shape_, name_):
14  initial = tf.truncated_normal(shape_, name=name_, stddev=0.15) #Initialized
```

```
        with a truncated normal random variable
15   tf.add_to_collection('l2', tf.reduce_sum(tf.pow(initial,2))) #Adding to L2
        collection, summing squares
16   return tf.Variable(initial)
17
18 #Takes a file path to a .wav file and outputs said file processed using the
        trained ANN
19 #Audio file must be in the same folder as this script
20 #Dependent on the LIBROSA python library and os library
21 def write_audio(filename,outname, net_):
22   data_path = filename #Finds file path to script and appends filename
23   y, sr = librosa.load(data_path) #Loads audio into python with samples 'y'
        and sampling rate 'sr' - 22.05kHz by default
24   D = librosa.stft(y) #STFT of input audio saved as D
25   mag = D
26   mag = np.abs(mag) #Magnitude response of the STFT
27   remember = mag.max(axis=0)+0.000000001 #Used for normalizing STFT frames (
        with addition to avoid division by zero)
28   mag = mag / remember #Normalizing
29   phase = np.angle(D) #Phase response of STFT
30   mag = np.transpose(mag) #Because dimensions were giving problems
31   new_mag = np.asarray(sess.run(net_, feed_dict={x_:mag})) #Process magnitude
        STFT frames through the ANN
32   new_mag = np.transpose(new_mag) #Again dimensions were giving problems
33   new_mag *= remember #Un-normalize the STFT frames
34   E = new_mag*np.exp(1j*phase) #Use magnitude and phase to produce complex-
        valued STFT
35   out = librosa.istft(E) #Inverse STFT
36   out = out/(np.max(out)+0.2)
37   filename = filename[:-4]+'_'
38   librosa.output.write_wav(filename+str(outname),out,sr) #Write output
39
40 def tune_knobs(filename_,tag_,knobs_):
```

```
41    ckpt = tf.train.latest_checkpoint('checkpoints')
42    saver.restore(sess, ckpt)
43    length = len(knobs_)
44    a = np.ones((1,fc4)) #Pre-allocation
45    its = fc4 // length
46    for w in range(length):
47     a[:,(w*its):] = knobs_[w] #Vector of knob values
48    a = np.float32(a) #Type matching to latent representation
49    layer1 = tf.nn.relu(tf.matmul(x_, W_fc1))
50    layer2 = tf.nn.relu(tf.matmul(layer1, W_fc2))
51    layer3 = tf.nn.relu(tf.matmul(layer2, W_fc3))
52    layer4 = tf.nn.sigmoid(tf.matmul(layer3, W_fc4))
53    layer4 = tf.multiply(layer4,a)
54    layer5 = tf.nn.relu(tf.matmul(layer4, W_fc5))
55    layer6 = tf.nn.relu(tf.matmul(layer5, W_fc6))
56    layer7 = tf.nn.relu(tf.matmul(layer6, W_fc7))
57    output_ = tf.nn.sigmoid(tf.matmul(layer7, W_fc8))
58    filename = filename_
59    write_audio(filename,tag_, output_)
60
61  #Generating weights for the fully connected layers
62  #fc- refers to the -th fully connected layer's neuron width
63  input_size = 1025
64  fc1 = 512
65  fc2 = 256
66  fc3 = 128
67  fc4 = 64
68  fc5 = fc3
69  fc6 = fc2
70  fc7 = fc1
71  output_size = input_size
72  W_fc1 = weight_variable([input_size, fc1],'W_fc1')
73  W_fc2 = weight_variable([fc1, fc2],'W_fc2')
```

```
74 W_fc3 = weight_variable([fc2, fc3],'W_fc3')
75 W_fc4 = weight_variable([fc3, fc4],'W_fc4')
76 W_fc5 = weight_variable([fc4, fc5],'W_fc5')
77 W_fc6 = weight_variable([fc5, fc6],'W_fc6')
78 W_fc7 = weight_variable([fc6, fc7],'W_fc7')
79 W_fc8 = weight_variable([fc7, input_size],'W_fc8')
80 print('Fully Connected Layer Params Defined')
81
82 #Generating placeholders for the input and label data
83 x_ = tf.placeholder(tf.float32, shape=[None, 1025])
84 y_ = tf.placeholder(tf.float32, shape=[None, 1025])
85
86 #Multiplying fully connected layers with ReLU/Sigmoid activations
87 layer1 = tf.nn.relu(tf.matmul(x_, W_fc1))
88 layer2 = tf.nn.relu(tf.matmul(layer1, W_fc2))
89 layer3 = tf.nn.relu(tf.matmul(layer2, W_fc3))
90 layer4 = tf.nn.sigmoid(tf.matmul(layer3, W_fc4))
91 layer5 = tf.nn.relu(tf.matmul(layer4, W_fc5))
92 layer6 = tf.nn.relu(tf.matmul(layer5, W_fc6))
93 layer7 = tf.nn.relu(tf.matmul(layer6, W_fc7))
94 output_ = tf.nn.sigmoid(tf.matmul(layer7, W_fc8))
95
96
97 saver = tf.train.Saver()
98
99 restore = True
100
101 if restore:
102  ckpt = tf.train.latest_checkpoint('checkpoints')
103  saver.restore(sess, ckpt)
104
105 print('Everything Loaded \n')
106 while True:
```

```
107     s = input()
108     s = s.split()
109     filename = s[0]
110     knobs = s[1].split(',')
111     nameout = s[2]
112     print(filename)
113     tune_knobs(filename, nameout, knobs)
114     if s == 'break':
115         break
116 
117 
118 # filename = sys.argv[1]
119 # knobs = sys.argv[2].split(',')
120 # nameout = sys.argv[3]
121 # tune_knobs(filename, nameout, knobs)
122 
123 print('Everything Done')
```

A.2 CANNe Backend

```
1  import tensorflow as tf
2  import numpy as np
3  import matplotlib
4  matplotlib.use('Agg')
5  import matplotlib.pyplot as plt
6  from matplotlib import animation
7  import os
8  import librosa
9  import sys
10 import scipy as sci
11 import soundfile as sf
12 from time import time
```

```
13 from tqdm import tqdm
14
15 def do_rtpghi_gaussian_window(mag,len_window,hop_length_):
16   threshold = 1e-3
17   pie = np.pi
18   relative_height = 0.01
19   width_ = (len_window/2)/np.sqrt(-2*np.log(relative_height))
20   gaussian_window = sci.signal.get_window(('gaussian',width_),len_window)
21   mag = np.clip(mag, threshold, None)
22   log_mag = np.log(mag)
23   qwe = np.shape(log_mag)
24   recon_phase_der = np.zeros(qwe)
25   recon_phase_output =  np.zeros(qwe)# np.random.uniform(low=0,high=2*pie,size
       =qwe)
26   M_freqs = qwe[0]
27   N_frames = qwe[1]
28   freq_time_ratio = -1*(pie/4)*(np.power(len_window,2)/np.log(relative_height)
       )
29   scale_constant_6 = (hop_length_*M_freqs)/(-2*freq_time_ratio)
30
31   #This is Equation 6 from the paper, which requires no look-ahead frames
32   for ii in range(1,M_freqs-1):
33     recon_phase_der[ii,:] = scale_constant_6*(log_mag[ii+1,:]-log_mag[ii-1,:])
         +(pie*hop_length_*ii/(M_freqs))
34   for jj in range(1,N_frames-1):
35     bins_to_randomize = mag[:,jj]==threshold
36     recon_phase_output[:,jj] = recon_phase_output[:,jj-1]+0.5*(recon_phase_der
         [:,jj-1]+recon_phase_der[:,jj])
37     #recon_phase_output[bins_to_randomize,jj] = np.random.uniform(low=0,high=2*
         pie,size=np.shape(log_mag[mag[:,jj]==threshold,jj]))
38   E = mag*np.exp(1j*recon_phase_output)
39   return librosa.istft(E, hop_length=hop_length_)
40
```

```
41 #Topology AutoEncoder:
42 #Generating weights for the fully connected layers fc- refers to the -th
       fully connected layer's neuron width
43 class Topology:
44  def __init__(self,input_size):
45   ##Calculated Below
46   self.fc = np.zeros((15)).astype(int)
47   self.b = {}
48   self.W_fc = {}
49   self.output_size = 2049
50   self.input_size = input_size
51 
52   ##Constant Values belonging to topology:
53   self.chkpt_name = 'checkpoints'
54   self.min_HL = 8
55   self.epochs = 300 #Number of epochs the ANN is trained for - 300 should be
        sufficient
56   self.learning_rate_adam = 1e-3 #ADAM learning rate - 1e-3 was found to
        produce robust ANNs
57   self.l2_lamduh = 1e-10 #Lamda value for L1 Regularization
58   self.batch_size = 100 #Typical batch size for ADAM useage
59   self.fc = [1000,512,256,128,64,32,16,8,16,32,64,128,256,512,1024]
60 
61   for i in range(15):
62    self.b[i] =self.getBiasVariable(self.fc[i],'b_' + str(i))
63   self.b[15] = self.getBiasVariable(self.output_size,'b_13')
64 
65   #Making weight variables
66   self.W_fc[0] = self.getWeightVariable([self.input_size, self.fc[0]],'W_fc1'
        )
67   for i in range(1,15):
68    self.W_fc[i] = self.getWeightVariable([self.fc[i - 1],self.fc[i]],'W_fc' +
          str(i + 1))
```

```
69        self.W_fc[15] = self.getWeightVariable([self.fc[14], self.output_size],'
            W_fc14')
70
71      def getBiasVariable(self,shape_, name_):
72        initial = tf.truncated_normal([shape_], name=name_, stddev=0.15) #
            Initialized with a truncated normal random variable
73        return tf.Variable(initial)
74
75      #Creates weight variables for the ANN and groups them in a collection for
          use in L2 regularization
76      def getWeightVariable(self,shape_, name_):
77        initial = tf.truncated_normal(shape_, name=name_, stddev=0.15) #Initialized
             with a truncated normal random variable
78        tf.add_to_collection('l2', tf.reduce_sum(tf.pow(initial,2))) #Adding to L2
            collection, summing squares
79        return tf.Variable(initial)
80
81    class OperationMode:
82      def __init__(self,train=False,new_init=False,validation=False,control=False,
          bias=False):
83        self.train = train
84        self.new_init = new_init
85        self.validation = validation
86        self.control = control
87        self.bias = bias
88
89    class ANNeSynth:
90      def __init__(self,operationMode):
91        self._operationMode = operationMode
92        self._sess = tf.Session()
93
94        #Load the stft so we have an input_size (from the topology)
95        self.loadDataSet()
```

```
96
97      ##Generating placeholders for the input and label data
98      self.x_ = tf.placeholder(tf.float32, shape=[None, self.topology.input_size
            ])
99      self.y_ = tf.placeholder(tf.float32, shape=[None, self.topology.output_size
            ])
100     self.controller = tf.placeholder(tf.float32, shape=[None, self.topology.
            min_HL])
101     ##
102     self.makeTensorFlowLayers()
103
104
105   def loadDataSet(self):
106     #Loading 95,443 Magnitude STFT frames saved as .npy (Loading in data)
107     filename = 'oo_all_frames.npy' #5 Octave Static Data used for training net
108     # filename = 'oo_all_frames.npy' #One Octave set
109     data_path = os.path.join(os.getcwd(),filename)
110     self.frames = np.load(data_path)
111     self.frames = np.asarray(self.frames)
112     n_mels_ = 512
113     n_mfccs_ = 256
114     mel_append = librosa.feature.melspectrogram(S=np.transpose(self.frames),
            n_mels = n_mels_)
115     mfcc_append = np.transpose(librosa.feature.mfcc(S=librosa.core.power_to_db(
            mel_append), n_mfcc = n_mfccs_))
116     mfcc_append = mfcc_append[:,26:]
117     mel_append = np.transpose(mel_append)
118     first_diff = np.diff(self.frames)
119     second_diff = np.diff(self.frames, n=2)
120     self.frames = np.hstack((self.frames,first_diff))
121     self.frames = np.hstack((self.frames, second_diff))
122     # self.frames = np.hstack((self.frames,mfcc_append))
123     #self.frames = np.hstack((self.frames,mel_append))
```

```
124     print(np.shape(self.frames))
125
126     #Five Octave Dataset
127     # self.validate = self.frames[84712:,:]
128     #One Octave Dataset
129     self.validate = self.frames[17998:,:]
130
131     self.topology = Topology(np.shape(self.frames)[1])
132
133   def recurseThroughLayer(self,layer,i,desired_stop):
134     Product = tf.matmul(layer,self.topology.W_fc[i])
135
136     if(self._operationMode.bias):
137       new_layer = tf.nn.relu(tf.add(Product,self.topology.b[i]))
138     else:
139       new_layer = tf.nn.relu(tf.add(Product,0))
140
141     if(i == desired_stop):
142       return new_layer
143     else:
144       return self.recurseThroughLayer(new_layer,i + 1,desired_stop)
145
146   def makeTensorFlowLayers(self):
147     ##Making the tensorflow layers from bias and weight variables
148     initialLayer = tf.nn.relu(tf.add(tf.matmul(self.x_, self.topology.W_fc[0]),
          self.topology.b[0]))
149     initialLayer2 = tf.nn.relu(tf.add(tf.matmul(self.controller, self.topology.
          W_fc[8]),self.topology.b[8]))
150     self.modulators = tf.placeholder(tf.float32, shape=[None, self.topology.fc
          [7]])
151     self.outputLayer = self.recurseThroughLayer(initialLayer,1,15)
152     self.outputLayer2 = self.recurseThroughLayer(initialLayer2,9,15)
153     self.initfilter = tf.multiply(self.recurseThroughLayer(initialLayer,1,7),
```

```
        self.modulators)
154     self.outputLayer4 = self.recurseThroughLayer(self.initfilter,8,15)
155 
156   def trainNeuralNetwork(self):
157     #Splitting self.frames into different buffers
158     #Five Octave Dataset
159     # train = self.frames[:78991,:]
160     # test = self.frames[78991:84712,:]
161     # validate = self.frames[84712:,:]
162     #One Octave Dataset
163     train = self.frames[:16685,:]
164     test = self.frames[16685:17998,:]
165     validate = self.frames[17998:,:]
166 
167     #Generating Parameters for the Neural Network and Initializing the Net
168     total_batches = int(len(train)/self.topology.batch_size) #Number of batches
          per epoch
169     l2 = tf.reduce_sum(tf.get_collection('l2'))
170     # loss2 = tf.reduce_mean(tf.pow(y_ - output_, 2)) # MSE error
171 
172     subt = self.y_ - self.outputLayer
173     arg1 = tf.pow(subt, 2)
174     arg2 = tf.reduce_mean(tf.pow(self.y_,2))
175     self.loss2 = tf.divide(tf.reduce_mean(arg1),arg2) #Spectral Convergence
        calculation for input and output magnitude STFT frames
176     self.loss3 = tf.reduce_mean(arg1)
177     self.loss4 = tf.reduce_mean(tf.abs(subt))
178     loss = self.loss2+self.topology.l2_lamduh*l2 #Imposing L2 penalty
179     train_step = tf.train.AdamOptimizer(self.topology.learning_rate_adam).
        minimize(loss)
180 
181     ###Loads the trained neural network into memory
182     if self._operationMode.new_init:
```

```
183    self._sess.run(tf.global_variables_initializer())
184   else:
185    ckpt = tf.train.latest_checkpoint(self.topology.chkpt_name)
186    self.saver.restore(self._sess, ckpt)
187   #Trains the neural net for the number of epochs specified above
188   #Prints test accuracy every 10th epoch
189   text_file = open("metrics.txt", "a")
190   starting_time = time()
191   for i in tqdm(range(self.topology.epochs)):
192    frames = np.random.permutation(train) #permuting the training data between
         epochs improves ADAM's performance
193    for _ in range(total_batches):
194     batch = frames[_*self.topology.batch_size:_*self.topology.batch_size+self
         .topology.batch_size] #Generates batch of size batch_size for training
195     self._sess.run(train_step, feed_dict={self.x_:batch, self.y_:batch[:,0:
         self.topology.output_size]})
196    tes = np.reshape(test[:,:],[-1,self.topology.input_size]) #Reshaping test
        array to fit with TF
197    if i%10 == 1:
198     self.saver.save(self._sess, self.topology.chkpt_name+'/my-model',
         global_step=i)
199     temp_value = self._sess.run(self.loss2, feed_dict={self.x_:tes, self.y_:
         test[:,0:self.topology.output_size]})
200     text_file.write('\n%g'% i)
201     text_file.write('\ntest accuracy %g'% temp_value)
202   #print('test accuracy %g'% self._sess.run(self.loss2, feed_dict={self.x_:
       tes, self.y_:test[:,0:self.topology.output_size]}))
203   end_time = time()-starting_time
204   print('Training Complete \n Evaluating Model')
205   text_file.write('\n%g'% i)
206   val = np.reshape(validate[:,:],[-1,self.topology.input_size])
207   temp_value = self._sess.run(self.loss2, feed_dict={self.x_:val, self.y_:
       validate[:,0:self.topology.output_size]})
```

```
208     text_file.write('\nvalidation accuracy %g'% temp_value)
209     text_file.write('\ntook %g seconds'% end_time)
210     text_file.close()
211     self.plotTrainingFigures()
212
213   def plotTrainingFigures(self):
214     #Plots 5 examples of the ANN's output given a magnitude STFT frame as input
          as 5 separate pdfs
215     #Dependent on the matplotlib library
216     test = np.asarray(self.validate) #This is not a good move DON'T KNOW WHY IT
         'S HERE
217     for disp in range(10):
218       x_axis = np.arange(self.topology.output_size) #X-axis for magnitude
          response
219       orig = np.reshape(test[disp*200+200,:],[-1,self.topology.input_size]) #
          Pulling frames from the 'test' batch for plotting
220       orig_hat = np.reshape(self._sess.run(self.outputLayer,feed_dict={self.x_:
          orig}),[self.topology.output_size,-1]) #Processing frame using ANN
221       plt.figure(1)
222       plt.subplot(211)
223       plt.plot(x_axis,np.transpose(orig[:,0:self.topology.output_size]),color='b
          ') #Plots the original magnitude STFT frame
224       plt.ylim([0,1.2])
225       plt.subplot(212)
226       plt.plot(x_axis,orig_hat,color='r') #Plots the output magnitude STFT frame
227       plt.tight_layout()
228       plt.ylim([0,1.2])
229       plotname = 'HL'+str(self.topology.fc[0])+'-'+str(self.topology.fc[1])+'-'+
          str(self.topology.fc[2])+'-'+str(disp)+'.pdf'
230       plt.savefig(plotname, format = 'pdf', bbox_inches='tight')
231       plt.clf()
232     print('Plotting Finished')
233
```

```
234  def execute(self, values, filename='long'):
235   self.saver = tf.train.Saver()
236   if not self._operationMode.train:
237    ckpt = tf.train.latest_checkpoint(self.topology.chkpt_name)
238    self.saver.restore(self._sess, ckpt)
239   else:
240    self.saver = tf.train.Saver()
241    print('hi')
242    self.trainNeuralNetwork()
243
244   #Prints validation accuracy of the trained ANN
245   if self._operationMode.validation:
246    print('validation accuracy %g'% self._sess.run(self.loss2, feed_dict={
247     self.x_:self.validate, self.y_:self.validate[:,0:self.topology.
         output_size]}))
248
249   if self._operationMode.control:
250    len_window = 4096 #Specified length of analysis window
251    hop_length = 1024 #Specified percentage hop length between windows
252    t = time()
253    n_frames = 750
254    mag_buffer = np.zeros((self.topology.output_size,1))
255    activations = values[:,0:8]
256    print(values)
257    for ii in range(n_frames):
258     orig_hat = np.reshape(self._sess.run(self.outputLayer2,feed_dict={self.
         controller:activations}),[self.topology.output_size,-1])
259     mag_buffer = np.hstack((mag_buffer,orig_hat))
260    mag_buffer = 50*mag_buffer#*np.random.uniform(low=0.999, high=1.001, size=
        np.shape(mag_buffer))#+np.random.uniform(low=1,high=20,size=np.shape(
        mag_buffer))
261    bass_boost = (np.exp(np.linspace(0.95,-0.95,self.topology.output_size)))
262    for ii in range(n_frames):
```

```
263       mag_buffer[:,ii] = np.roll(mag_buffer[:,ii],int(values[:,8]))*bass_boost
264     T = do_rtpghi_gaussian_window(mag_buffer, len_window,hop_length) #
          Initializes phase
265     T = 0.8*T/np.max(np.abs(T))
266     crossfade_time = 0.35
267     crossfade_time = int(crossfade_time*44100)
268     fade_in = np.log(np.linspace(1,2.71,crossfade_time))
269     fade_out = np.log(np.linspace(2.71,1,crossfade_time))
270     T[:crossfade_time] = fade_in*T[:crossfade_time]+fade_out*T[len(T)-
          crossfade_time:]
271     U = T[:len(T)-crossfade_time]
272     sf.write(filename+'.wav', U, 44100, subtype='PCM_16')  #Must be 16bit PCM
          to work with pygame
273     elapsed = time() - t
274     print('Method took '+str(elapsed)+' seconds to process the whole file')
275     print('The whole file is '+str(len(U)/44100)+' seconds long')
276
277   def load_weights_into_memory(self):
278    self.saver = tf.train.Saver()
279    ckpt = tf.train.latest_checkpoint(self.topology.chkpt_name)
280    self.saver.restore(self._sess, ckpt)
281
282   def play_synth(self,values):
283    len_window = 4096 #Specified length of analysis window
284    hop_length = 1024 #Specified percentage hop length between windows
285    n_frames = 200
286    mag_buffer = np.zeros((self.topology.output_size,1))
287    activations = values[:,0:8]
288    for ii in range(n_frames):
289     orig_hat = np.reshape(self._sess.run(self.outputLayer2,feed_dict={self.
          controller:activations}),[self.topology.output_size,-1])
290     mag_buffer = np.hstack((mag_buffer,orig_hat))
291    mag_buffer = 50*mag_buffer#*np.random.uniform(low=0.999, high=1.001, size=
```

```
        np.shape(mag_buffer))#+np.random.uniform(low=1,high=20,size=np.shape(
        mag_buffer))
292     bass_boost = (np.exp(np.linspace(0.95,-0.95,self.topology.output_size)))
293     for ii in range(n_frames):
294       mag_buffer[:,ii] = np.roll(mag_buffer[:,ii],int(values[:,8]))*bass_boost
295     T = do_rtpghi_gaussian_window(mag_buffer, len_window,hop_length) #
        Initializes phase
296     T = 0.8*T/np.max(np.abs(T))
297     crossfade_time = 0.4
298     crossfade_time = int(crossfade_time*44100)
299     fade_in = np.log(np.linspace(1,2.71,crossfade_time))
300     fade_out = np.log(np.linspace(2.71,1,crossfade_time))
301     T[:crossfade_time] = fade_in*T[:crossfade_time]+fade_out*T[len(T)-
        crossfade_time:]
302     U = T[:len(T)-crossfade_time]
303     sf.write('loop.wav', U, 44100, subtype='PCM_16') #Must be 16bit PCM to
        work with pygame
```

A.3 CANNe GUI

```
1  import sys
2  from PyQt4.QtGui import *
3  from PyQt4.QtCore import *
4  from PyQt4 import QtGui
5  from canne import *
6  import os
7  import pygame
8
9  mode = OperationMode(train=False,new_init=False,control=True)
10 synth = ANNeSynth(mode)
11
12 class sliderGui(QWidget):
```

```
13  def __init__(self, parent = None):
14   super(sliderGui, self).__init__(parent)
15   layout = QVBoxLayout()
16   layout2 = QHBoxLayout()
17
18   self.generateButton = QtGui.QPushButton('Save', self)
19   self.generateButton.clicked.connect(self.generate)
20
21   self.playButton = QtGui.QPushButton('Pause',self)
22   self.playButton.clicked.connect(self.pause)
23
24   layout = QtGui.QVBoxLayout(self)
25   layout.addWidget(self.playButton)
26   layout.addWidget(self.generateButton)
27   layout.addLayout(layout2)
28
29   self.s1 = QSlider(Qt.Vertical)
30   self.s2 = QSlider(Qt.Vertical)
31   self.s3 = QSlider(Qt.Vertical)
32   self.s4 = QSlider(Qt.Vertical)
33   self.s5 = QSlider(Qt.Vertical)
34   self.s6 = QSlider(Qt.Vertical)
35   self.s7 = QSlider(Qt.Vertical)
36   self.s8 = QSlider(Qt.Vertical)
37   self.s9 = QSlider(Qt.Horizontal)
38
39   self.addSlider(self.s1,layout2)
40   self.addSlider(self.s2,layout2)
41   self.addSlider(self.s3,layout2)
42   self.addSlider(self.s4,layout2)
43   self.addSlider(self.s5,layout2)
44   self.addSlider(self.s6,layout2)
45   self.addSlider(self.s7,layout2)
```

```
46     self.addSlider(self.s8,layout2)
47     self.addSlider(self.s9,layout2)
48     self.s9.setMinimum(-30)
49     self.s9.setMaximum(30)
50     self.s9.setValue(0)
51     self.s9.setTickInterval(3)
52
53     self.setLayout(layout)
54     self.setWindowTitle("CANNe")
55
56   def addSlider(self,slider,layout):
57     slider.setMinimum(0)
58     slider.setMaximum(40)
59     slider.setValue(10)
60     slider.setTickPosition(QSlider.TicksBelow)
61     slider.setTickInterval(2)
62     layout.addWidget(slider)
63     slider.sliderReleased.connect(self.valuechange)
64
65   def valuechange(self):
66     tmp = np.zeros((1,9))
67     tmp[0,0] = self.s1.value()
68     tmp[0,1] = self.s2.value()
69     tmp[0,2] = self.s3.value()
70     tmp[0,3] = self.s4.value()
71     tmp[0,4] = self.s5.value()
72     tmp[0,5] = self.s6.value()
73     tmp[0,6] = self.s7.value()
74     tmp[0,7] = self.s8.value()
75     tmp /= 10.
76     tmp[0,8] = 2*self.s9.value()
77     synth.play_synth(tmp)
78     pygame.mixer.music.load('loop.wav')
```

```
79       pygame.mixer.music.play(-1)
80
81
82     def generate(self):
83       tmp = np.zeros((1,9))
84       tmp[0,0] = self.s1.value()
85       tmp[0,1] = self.s2.value()
86       tmp[0,2] = self.s3.value()
87       tmp[0,3] = self.s4.value()
88       tmp[0,4] = self.s5.value()
89       tmp[0,5] = self.s6.value()
90       tmp[0,6] = self.s7.value()
91       tmp[0,7] = self.s8.value()
92       tmp /= 10.
93       tmp[0,8] = self.s9.value()
94       text, ok = QInputDialog.getText(self, 'Save File', 'Enter filename:')
95       if ok:
96         filename_=str(text)
97         synth.execute(tmp,filename_)
98
99
100
101    def pause(self):
102      pygame.mixer.music.stop()
103
104
105  def main():
106    synth.load_weights_into_memory()
107    pygame.init()
108    pygame.mixer.init(channels=1)
109    app = QApplication(sys.argv)
110    ex = sliderGui()
111    ex.show()
```

```
112     sys.exit(app.exec_())
113
114 if __name__ == '__main__':
115     main()
```