Audio Generation from a Single Sample using Deep Convolutional Generative Adversarial Networks

by

Levi Pfantz

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

Master of Science

Computer Science

At

The University of Wisconsin–Whitewater

December, 2021

Graduate Studies

The members of the Committee approve the thesis of
Levi Pfantz presented on (December 3rd, 2021)

Dr. Athula Gunawardena, Chair

Dr. Lopamudra Mukherjee

Dr. Jiazhen Zhou

Audio Generation from a Single Sample using Deep Convolutional Generative Adversarial Networks

By

Levi Pfantz

The University of Wisconsin-Whitewater, Year 2021

Under the Supervision of Dr. Athula Gunawardena

Training neural networks require sizeable datasets for meaningful output. It is difficult to acquire large datasets for many types of data. This is especially challenging for individuals and small organizations. We have taken SinGAN [1], a model that works to address those issues in the image domain, and extended it to work in the audio domain. Our new model, called AudioSinGAN, uses deep convolutional generative adversarial networks (DCGAN) trained on a single audio sample to generate new, unique, audio samples. Like SinGAN, AudioSinGAN uses a pyramid of unique GANs, each responsible for learning and generating different levels of detail. Our system is capable of generating unique audio with clear features from the single input audio clip. We explore and discuss the realities of converting and tuning a generative adversarial network (GAN) built for images into one built for audio and our results. We also present a database of audio clips generated by AudioSinGAN and use Singular Value Decomposition to analyze the dataset and confirm that our model successfully generates audio belonging to unique classes. We also learn that a challenge facing our system is audio that contains multiple audio sources overlapping each other. Finally, we discuss methods to address this issue including splitting audio into frequency band before processing.

ACKNOWLEDGMENTS

Firstly, I would like to thank my supervisor Dr. Athula Gunawardena. Your patience and guidance have been invaluable through every stage from inception to polish. I truly could not have completed this without you. Your willingness to share knowledge, experience, and time is what allowed me to take this from an idea in my head to completion.

I would also like to thank my committee members Dr. Jiazhen Zhou and Dr. Lopamudra Murjehkee. Your feedback allowed me to refine my thesis in ways that were sorely needed.

I also owe a debt of gratitude to all my other professors at both the University of Wisconsin-Whitewater and the University of Wisconsin-Green Bay, but I must especially thank Dr. Ankur Chattopadhyay. I would not have attended graduate school and performed research without your mentorship and encouragement during my years at UW–Green Bay.

My family's support has been crucial to me being able to complete this thesis. I am incredibly grateful for my wife Tiegan Pfantz and my parents Mary and Daryl Pfantz. My mother checked my work for grammar and spelling errors, my wife helped me rank audio samples, and my father shared his encouragement as someone who has undergone writing a thesis. All three have been there for me, encouraged me, and supported me in a myriad of ways.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background and Motivation

One of the greatest hurdles of building an effective machine learning system is acquiring data in sufficient quantity and quality. For example, AlexNet [2] used about 1.2 million training images to achieve state-of-the-art results at the time. Modern generative adversarial networks (GANs) often use 50,000 plus images or data points in the training process [3] [4] [5] [6].

The requirement for large datasets means that it is difficult to create a machine learning system that works with a specific kind of data unless there is a preexisting dataset. For example, there are publicly available datasets for often studied subjects such as images of faces, images of handwritten digits, and data used in diagnosing medical problems such as dimensions of possibly cancerous masses. However, for many types of data, there isn't an accessible dataset. This means that if an entity would like to use that type of data with a machine learning system they would need to either gather a dataset from the real world, synthesize an artificial dataset, or some combination of the two. This may be costly or difficult for an organization and is often out of reach for an individual. Finding ways to reduce the data required for machine learning systems would benefit organizations but more importantly it would benefit individual researchers and creators. Although this is an issue throughout the machine learning world this thesis will specifically focus on the issue in regard to Generative Adversarial Networks.

The work in this thesis is based on a GAN called SinGAN [1]. SinGAN is a system that works with images but in this work, we convert it to work with audio instead. Because we focus on converting SinGAN to audio we call our resulting system AudioSinGAN[1] [2]. This system will be used for generation of new audio based off of a single audio input.

The output from our current system is inferior to machine learning systems such as WaveGAN [7] or WaveNet [8] that train on many samples. In its current form the system does not produce audio samples of high enough quality to be useful for real-world situations. Future work on the system, however, could allow for use of the system for the generation of audio in the domains of art, sound design, and accessibility. Examples from past machine learning systems include speech generation for speech impaired individuals [8] [9] and audio generation for artistic purposes [10]. Finally, there is potential for AudioSinGAN to be extended in similar ways to SinGAN for purposes such as audio editing, resampling, and style transfer/harmonization.

Our goals for this project are to convert SinGAN to work with audio, to test the limits of our converted AudioSinGAN, to produce and analyze a datset of results and to setup well for future work on AudioSinGAN. Our contributions are as follows:

1. We created a working version of AudioSinGAN.
2. We explored the necessary adjustments to convert SinGAN to AudioSinGAN.
   (a) Model structural changes.
   (b) Hyper-parameter tuning.
   (c) Helper code rewrites.
3. We provide an in-depth discussion of the process and results.
4. We have created a dataset of audio clips from Audio SinGAN and analyzed it using Singular Value Decomposition.
5. We discuss AudioSinGAN's potential for future research.

[1] Project GitHub: `https://github.com/LPfantz/AudioSinGAN`

[2] Audio Samples: `https://sites.google.com/view/audiosingan`

1.2 Related Work

One strategy for solving the requirement of large datasets is to augment the architecture to achieve equal or better results with fewer data points. For example, Nvidia's StyleGAN2-ada [11] is a powerful GAN capable of rivaling its predecessor's StyleGAN [5] and StyleGAN-2 [6] with a training set of only a few thousand images by using an adaptive discriminator where images are augmented as they are exposed to the discriminator.

Audio generation with neural networks is a relatively well-studied field but it is commonly done with autoregressive models. One of the better-known examples is WaveNet [8] which powers Google Assistant. Although when it comes to GANs audio has been less studied than images, there has been notable work in the area. WaveGan [7] in particular has been an inspiration for network architecture while working on this thesis. A number of other interesting GAN's have also been created for the purpose of generating audio using one method or another [9] [12] [13].

When it comes to GANs trained on a single data point, research has largely been based off of SinGAN [1]. Work has already been done to improve SinGAN within the image domain [14] [15] [16] as well as in the video domain [17] [18].

One work similar to ours in many ways is MP3Net [19] which is an audio-based GAN that uses a similar pyramid of different resolution audio scales but still trains on a dataset of many images.

As far as we know our work is the only research into using GANs to generate audio from only one training image.

1.3 Process

The process followed for the work on this thesis started with SinGAN [1]. SinGAN is short for single GAN because it takes in a single image and trains on multiple versions of the image at different resolutions. For this thesis, we took the SinGAN system and adopted it to work with audio instead. The process involves fixing bugs in the original system, transforming the

architecture to work with audio instead of images, experimenting to discover which parameters work well with audio in the changed system, discovering what type of audio work well with the system through experimentation and induction from the original visual system, and finally experimenting with various techniques not used in the original system to try and improve our new, transformed system.

1.4 Outline

Chapter 2 lays out the theoretical background for neural networks, deep convolutional neural networks, generative adversarial networks, deep convolutional generative adversarial networks and goes into detail about the working of SinGAN. Chapter 3 deals with the process of converting SinGAN to work with audio and tuning it for optimal results. Chapter 4 discusses the results of that process and output of AudioSinGAN. Chapter 5 is our conclusion and some thoughts on future research.

Chapter 2

From Neural Networks to SinGAN

2.1 Basics of Neural Networks

Neural networks are a machine learning paradigm based on biological nervous systems. Their conception was motivated by the fact that the nervous systems of animals and especially humans have capabilities that far exceed those of computers in many ways. The ability to replicate such capabilities in computers would be (and attempts to achieve this have already resulted in) large strides forward in our technological capabilities [20]. Many excellent papers explain the basics of neural networks [21] [22] [23].

Neural networks are mathematical structures built out of nodes called neurons. Neurons are connected in a variety of different configurations. The classic, fully connected neural network consists of one or more layers of neurons. The input is fed into the first layer, processed, and then fed to the next layer ad infinitum until there are no more layers. Then an activation function or threshold function is applied to calculate the output of the neural network.

Neurons are connected to other neurons in the preceding and the following layer as applicable. Each of these connections has a weight. The output of a neuron is calculated using all its incoming connections, the corresponding weights, and a bias or threshold belonging to the receiving neuron. It can be described with Equation 2.1.

$$\text{netinput}_i = \sum_j w_{ij} * x_j + \mu_i \tag{2.1}$$

In Equation 2.1 [20] i is the neuron which is receiving input and j represents each neuron with connections going from it to i. w_{ij} is the weight for the connection from j to i and x_j is

the output from j. Finally the μ is the bias for neuron i. $Netinput_i$ is the final output of neuron i.

An activation function is how the final output of a neuron is calculated. Different activation functions can be used for different situations. An example of one common activation function which is used in our work is Leaky ReLU (Rectified Linear Units). Leaky ReLU is defined by Equation 2.2:

$$f(x) = \begin{cases} ax, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.2}$$

This means that when the netinput is calculated it is unaltered by ReLU and if it is negative then it is multiplied by a value. This value may be something like 0.1, 0.01, or 0.2 as in SinGAN [1]. Similar to ReLU, the activation function that leaky ReLU is based on, results in the sparser firing of neurons compared to other activation functions. Sparser firing comes with several advantages [24]. Figure 2.1 shows the graph of the Leaky ReLU function [25].

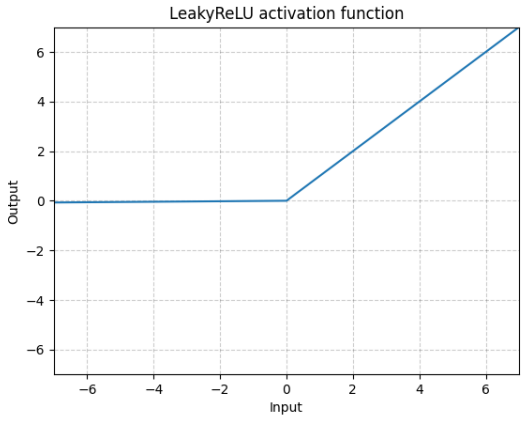

Figure 2.1 Leaky ReLU Graph.

A single neuron with five incoming connections using the leaky ReLU activation function can be visualized as Figure 2.2 which shows a neuron with five incoming connections.

The neuron calculates its net input from the connections and then applies its activation function and that is the neuron's output.

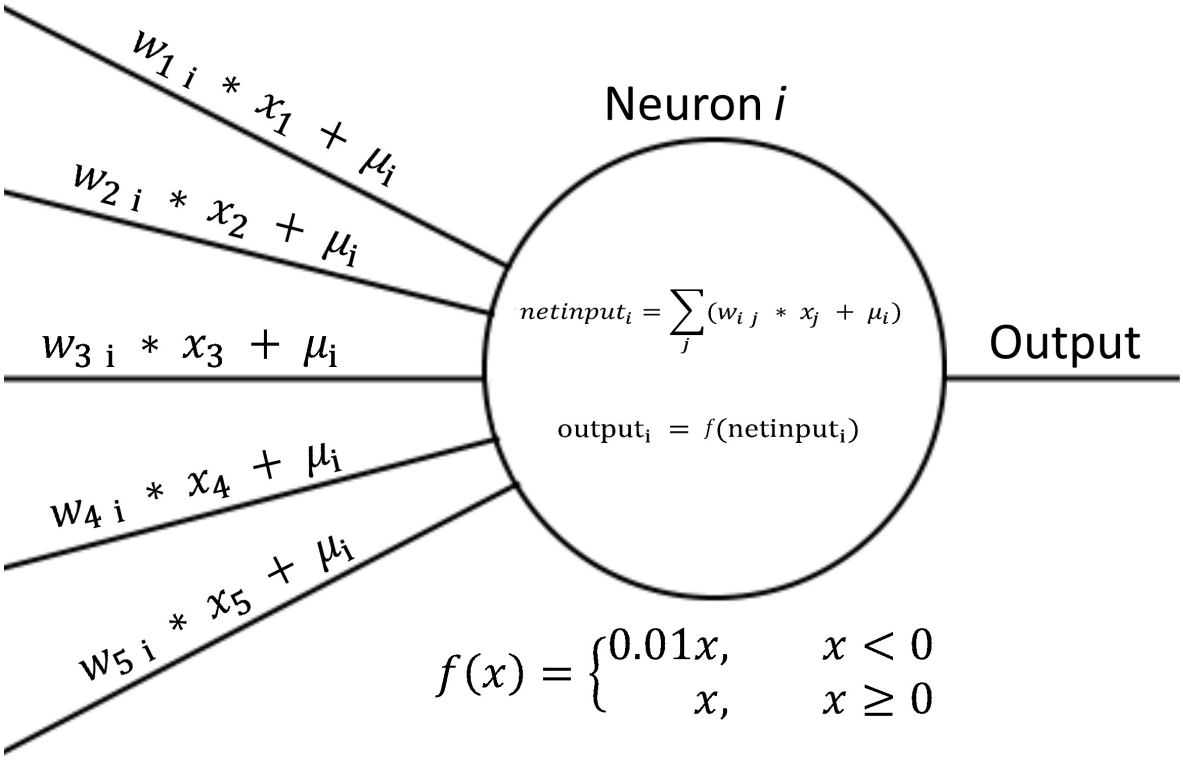

Figure 2.2 An example of a neuron with five inputs using the Leaky ReLU activation function.

When neural networks are trained an input is fed into the network and the output is calculated. These weights are adjusted to get different outputs from the same input.

Neural networks are trained through a process called backpropagation. In this process, the output of a neural network is compared to what would be ideal for the given input and the difference is calculated and called loss. The weights of connections and biases of neurons are then altered through gradient descent (or a similar method) and backpropagation to minimize loss.

2.1.1 Perceptrons

It's relatively simple to go from a single neuron to a simple model that can be used in the real world. The classic example of this is a perceptron as visualized in Figure 2.1.1.

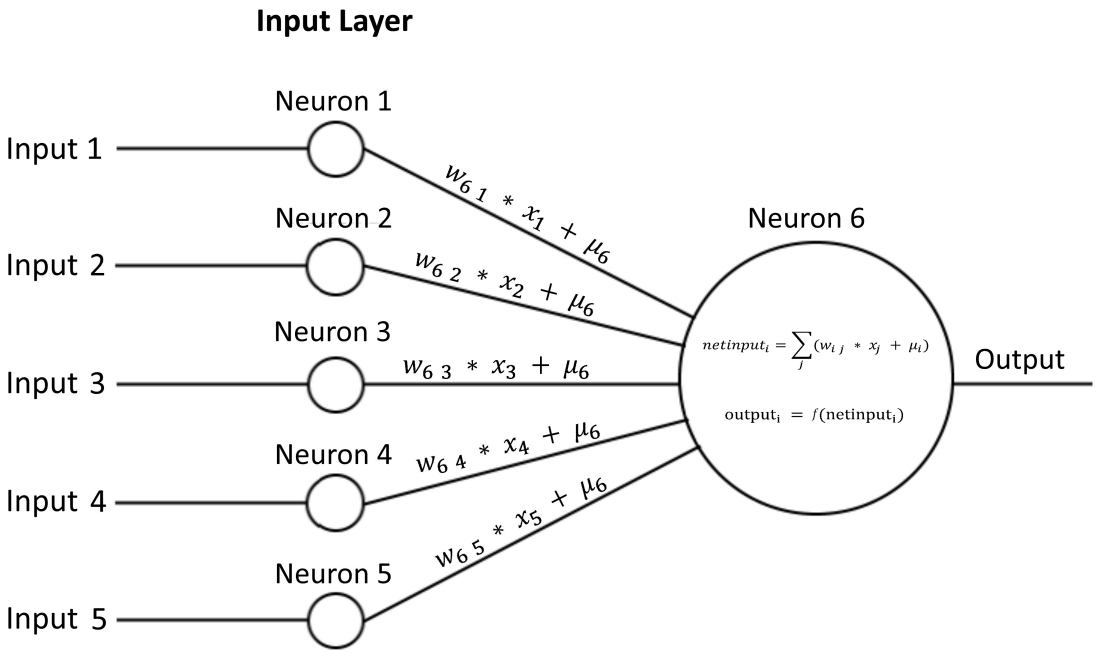

Figure 2.3 An example of a perceptron with five inputs.

A perceptron is effectively just a single neuron set up to take in inputs and give outputs. Note that in the image the neurons in the input layer aren't true neurons in that they don't apply weights, biases, or an activation function. They simply form connections from the input to the single true neuron. They could be visualized as just inputs without neurons. However, visualizing them as neurons in an input layer makes discussing and visualizing both the perceptron and the multilayer perceptron easier.

The classic perceptron wouldn't use an activation function but instead would use a threshold function defined as Equation 2.3:

$$f(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{2.3}$$

The extension of a perceptron is the multilayer perceptron (MLP) as seen in Figure 2.4.

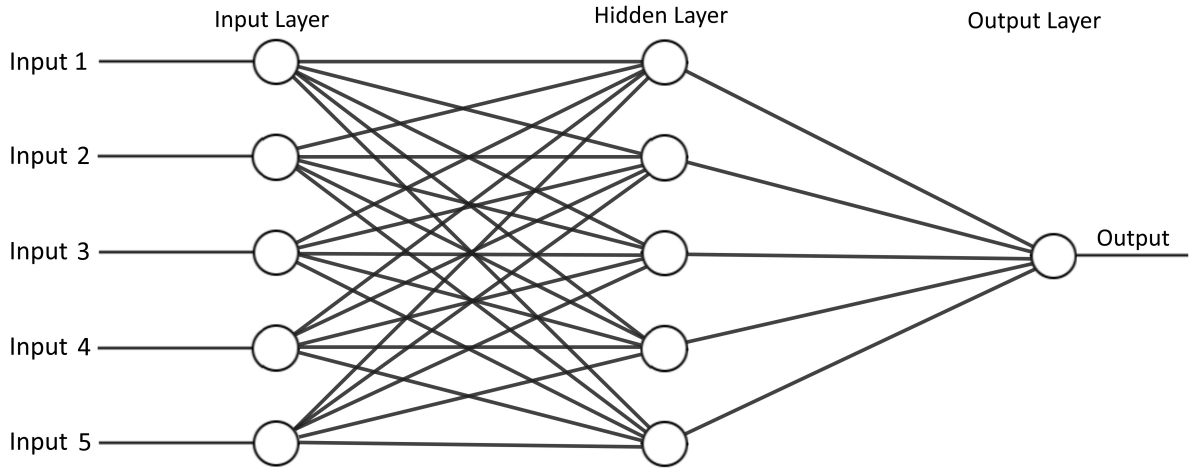

Figure 2.4 An example of a multilayer perceptron with one hidden layer.

This multilayer perceptron has an input layer similar to the perceptron with neurons that simply pass input through them. It also has a hidden layer that consists of true neurons. Each input is connected to each neuron in the input layer. In the same vein, each neuron in the hidden layer is connected to the neuron in the output layer. An MLP may have an arbitrary number of hidden layers and an arbitrary number of neurons in the hidden layers.

2.1.2 Gradient Descent and Back propagation

A neural network's output is determined by both its inputs and the values used for its weights and biases. Training a neural network then involves optimizing its weights and biases

until the network achieves the desired output. This is done with an algorithm called backpropagation (backwards propagation of errors) which uses gradient descent.

Gradient descent starts with calculating a gradient vector defined as Equation 2.4.

$$\nabla f(x_1, x_2, ...x_n) = \begin{bmatrix} \frac{df}{dx_1} \\ \frac{df}{dx_2} \\ . \\ . \\ . \\ \frac{df}{dx_3} \end{bmatrix} \tag{2.4}$$

This gradient vector provides the direction to maximize the output value of the function. Its magnitude represents the local change in intensity. In practice, the parameters in any function (including but not limited to a neural network) can be optimized using gradient descent.

The first step in this process is to calculate the gradient (represented by the ∇ symbol). If we are trying to minimize the output (as we would be with a loss function), we take the negative of the gradient vector. Secondly, a scalar value is chosen as a learning rate and multiplied by the gradient vector. Recall that the magnitude represents the local change in intensity. If it is not decreased using the learning rate, it may overshoot the goal of optimizing the output. The learning rate is a value less than 1 such as 0.1 or 0.01 and is multiplied by the magnitude.

Once the gradient vector is calculated and adjusted by the learning rate then the value of each parameter of the function (in the case of neural networks this means weights and biases) are adjusted by the corresponding value in the reduced vector gradient.

Unless we had too high of a learning rate, over-adjusted our parameters, and overshot our minimum value, the loss is now less than it was and the process can be repeated as many times as necessary to keep optimizing the function.

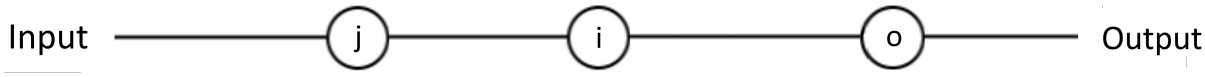

Figure 2.5 A Three Layer Neural Network.

Neural networks can be thought of as a function composed of many other functions. To accomplish the task of calculating the gradient of a neural network the backpropagation algorithm was created. We'll take a toy example with an input layer, a single hidden layer, and an output layer. Each Layer will have a single neuron. For a visualization see Figure 2.1.2

The cost function can then be described as the result of several functions. In Equation 2.5, $_o$ refers to the single output neuron in the output layer.

$$c = c(a(\sum_{j=0}^{l_2} \mu_{oj} + w_{oj} * a(\mu_{jk} + w_{jk} * x_k))) \tag{2.5}$$

The functions used above are defined below.

$$\text{Activation Function: } a(z) \tag{2.6}$$

$$\text{Cost function: } c(y') \tag{2.7}$$

$$\text{Weighted Input: } z_x(x) = xw + \mu \tag{2.8}$$

$$\text{Weighted Input: } z_w(w) = xw + \mu \tag{2.9}$$

$$\text{Weighted Input: } z_\mu(\mu) = xw + \mu \tag{2.10}$$

Note that from here on we can treat z as a single variable function that takes in either x, w, or μ as an input. This is because we will only ever be taking the derivative of c concerning one value in x. This means that the other two terms are treated as constants. We will denote which we are using as our parameter with a subscript for z. The cost function uses some method to measure the difference between the output of the neural network and the target output. The activation function will take in z and perform some operation on it. We will leave those two undefined for now as they can be set to many different functions. In practice, the output (or

result of the activation function) for the final layer, assuming there is only one node in the output layer, is the output of the neural network. See Equation 2.11.

$$a = \text{y'} = \text{Output of MLP} \tag{2.11}$$

If we are trying to calculate the derivative relative to a specific parameter, most of the time the total equation isn't necessary. Instead, we only need to use the portion of the equation that is affected by changing the variable. Every other part of the equation is independent of the parameter and can be treated as a constant. Therefore when we take the derivative it can be ignored.

This means that if we want to find the derivative of c with relation to w_{oj} we can use Equation 2.12 with function z written out completely:

$$\frac{dc}{d}w_{oj}c(a(z_w(w_{oj}))) = \frac{dc}{da} * \frac{da}{dz} * \frac{dz_w}{dw_{oj}} \tag{2.12}$$

Backpropagation works backward through the neural network and its first step is to calculate the derivatives of c in relation to the parameters for the connections between the output layer and the layer that precedes. In other words, it uses the previous formula to calculate $\frac{dc}{dw_{oj}}$ for every value of j.

When taking the derivative of c in relation to a preceding parameter in the function (i.e. closer to the input layer) we need to use a longer version of our output function with more variables. For example, if we want to calculate $\frac{dc}{dw_{jk}}$ we can use the (still reduced version) of our original loss function expressed in Equation 2.13.

$$\frac{dc}{dw_{jk}}c(a(z_x(a(z_w(w_{jk}))))) = \frac{dc}{da} * \frac{da}{dz_x} * \frac{dz_x}{da} * \frac{da}{dz_w} * \frac{z_w}{dw_{jk}}$$

(2.13)

Notice that in all these equations we exclude terms that are independent of the variables that we are taking derivatives relative to. In this way, backpropagation works backward through the neural network calculating the derivatives of the cost function relative to different parameters.

The algorithm saves the results as it works through the neural network to reuse past results as equations and parts of equations repeat.

Once it has worked back through the network it will have calculated the derivative of the cost function relative to every parameter in the network. It can now use these derivatives to perform gradient descent to optimize the neural network and minimize the cost function.

2.2 Convolutional Neural Networks

With traditional neural networks, every input is connected to every neuron in the first hidden layer. In a color image, this means that there are *width x height x channels* connections that need to be considered in the neuron's activation function and that need to have weights adjusted during propagation. If more layers are added then every neuron in a layer is connected to every neuron in the next layer. This results in two problems. First, in a large color image, this means that there is a lot of computational power required when using the net or training it. Second, this opens the door for overfitting. Overfitting is a common issue in machine learning where an algorithm specializes too much in a training set and fails to generalize to the larger data pool that the training set is drawn from.

One solution to these problems is to use a convolutional neural network [26]. Instead of connecting a neuron in a given layer to every other neuron in the following layer, we will connect it to a subset of neurons in the following layer. The grouping of neurons in the following layer connected to the original neuron is called a kernel. In a two dimensional input such as an image, the kernel can be visualized as a square. With one-dimensional input such as audio, it can be visualized as a line. Each neuron in a layer is connected to a kernel of neurons in the next layer. The kernel can be visualized as a square that moves across the layer and at each location connects to a different neuron in the preceding layer. How far the kernel moves is called the stride. With a two-dimensional layer and a stride of one when the kernel is moved a

single column or row is left behind and a new column or row is added. If the stride is two then two columns or rows are left behind and two columns or rows are added.

Because a group of neurons in one layer connects to s a single neuron in the next layer the system is usually set up using variables such as kernel size and stride so that each layer is smaller than the previous layer. If the goal is to have the network perform some transformation on the data then the output is often smaller than the input due to the convolutional layers downscaling it. One commonly used solution to this is to add padding around the input. More data, often numerical 0s (black pixels in an image), are added around the input to increase the size of the output. For further reading on convolutional neural networks see [27] [28] [29] [30].

2.3 Generative Adversarial Neural Networks

2.3.1 The Basics

The generative adversarial network (GAN) structure, proposed in 2014 by Goodfellow et al. [31], is an unsupervised machine learning system used for data generation. This type of system involves two competing neural networks playing an adversarial game. The analogy given in the originating paper is of money forgers contesting with police. The two adversaries compete and as the forgers learn and get better at producing fake money the police also learn and get better at discerning fake money from real. This competition continues until the forgers can make money that is identical to real currency and the best guess of the police is only as good as a random guess.

The classic GAN presented by Goodfellow et al. [31] names the two neural networks as the generator (the forger) and the discriminator (the police). Training data is put into the system (non-forged money in the analogy) and the generator tries to copy it while the discriminator tries to distinguish real from fake. The discriminator's loss is calculated based on how well it can classify both real data and generated data. Equation 2.14, as presented by the authors in the classic GAN paper, is used for ascending the discriminator's stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right] \tag{2.14}$$

The generator's loss is calculated based on how well the discriminator can classify the generated data. The classic GAN paper presents the Equation 2.15 for descending the generator's stochastic gradient.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right] \tag{2.15}$$

Theoretically, the ascension/descent of gradients to optimize the discriminator and generator continues until the generative neural network can produce output indistinguishable from the trading data.

Note that the GAN loss function is effectively minimizing the Jensen-Shannon distance. Jensen-Shannon is built on top of the Kullback-Leibler divergence which is shown in Equation 2.16.

$$KL(P_r \| P_g) = \sum_{x \in \mathcal{X}} P_r \log\left(\frac{P_r(x)}{P_g(x)}\right) \tag{2.16}$$

This divergence has a problem where $P_g(x)$ is 0 causing division by 0. Jensen-Shannon was introduced as an improvement and is defined by Equation 2.17.

$$JS\left(P_r, P_g\right) = KL\left(P_r \| P_m\right) + KL\left(P_g \| P_m\right) \tag{2.17}$$

In this equation $P_m = (P_r + P_g)/2$ or the average of the two distributions.

While Jensen-Shannon is defined everywhere, it does have its issues. When either $P_g(x)$or$P_r(x)$ is 0 the result is log 2 no matter how close or far apart the distributions truly are. This decreases the effectiveness of the loss function as there isn't a change in loss as the distance between the two distributions varies. We can say that both the Kullback-Leibler divergence and the

Jensen-Shannon divergence aren't continuous. Under certain circumstances, their slopes can't be descended by gradient descent to reach a minimum.

GANs may be constructed out of fully connected neural networks. However, in modern research, they are more commonly built as Deep Convolutional Generative Adversarial Networks (DCGANs) [32]. These are GANs built out of deep convolutional networks instead of fully connected networks. They have many of the advantages that convolutional networks bring and as demonstrated in their originating paper are excellent when working with images.

2.3.2 WGAN-GP

Due to the problems inherent in the Kullback-Leibler and Shannon-Jensen divergences, the Wasserstein GANs [33] were introduced. These GANs relabel the discriminator as the critic and derive their loss functions from the Wasserstein or Earthmover distance. This distance can be considered to be the cost to transform one distribution into another distribution in the cheapest way.

$$\mathrm{EMD}(P, Q) = \inf_{\gamma \in \Pi(P,Q)} E_{(x,y)\sim\gamma}[\|x - y\|] \tag{2.18}$$

Equation 2.18 is not useful as a loss function of a GAN as it is intractable. However, using the Kantorovich-Rubinstein [34] duality a more useful function can be derived.

$$\min_G \max_{D\in\mathcal{D}} \underset{\boldsymbol{x}\sim P_r}{E}[D(\boldsymbol{x})] - \underset{\tilde{\boldsymbol{x}}\sim P_g}{E}[D(\tilde{\boldsymbol{x}}))\Big] \tag{2.19}$$

In the Kantorovich-Rubinstein duality (Equation 2.19) $\boldsymbol{x}$ is a real data point and $D(\boldsymbol{x})$ is the critic's realness score for it. Similarly, $\tilde{\boldsymbol{x}}$ is a data point created by the generator, and $D(\tilde{\boldsymbol{x}})$ is the realness score calculated by the critic for that data point. D (the critic) must be a 1-Lipschitz function. The set of 1-Lipschitz functions is $\mathcal{D}$. Using a Lipschitz function keeps the critic as a differential function that gradient descent can work with.

To ensure that the critic stays a 1-Lipschitz function the parameters of the neural network are clipped to stay within set values. While this technique works it is, by the authors of the original WGAN paper's [33] own admission, a poor technique. Wasserstein GAN - Gradient

Figure 2.6 Examples Results from BigGAN. Four success and one mistake.

Penalty (WGAN-GP) [35] was introduced to address this problem. WGAN-GP uses a soft regulation strategy based on the fact that a 1-Libschitz function has gradients with a norm of at most 1. WGAN-GP then penalizes by the square difference from one of the gradient's norms.

2.3.3 GANs and Images

One of the most common usages of GANs is image synthesis. By training a GAN on a dataset of images it can be optimized to produce realistic images. One example of this is BigGAN [3]. BigGAN is a GAN that works with images and specifically focuses on synthesizing images from complex data sets at high resolutions.

Another interesting research area with GANs is in the picture-to-picture transformations. An example of this is taking in a sketch (or edges) of an image and outputting a complete, nearly, photo-realistic image [36] [4] [37]. Another would be producing an image from a semantic segmentation mask [36] [38]. Although style transfer has often been done with methods other than GANs [39] it can also be done with GANs as it is with SinGAN [1].

PatchGAN [36] is an interesting example and also relevant as SinGAN draws from it. PatchGAN specializes in image-to-image translation. Examples of domains it's been used in include semantic mappings to realistic images such as houses, black and white images to color, and day images to night images. PatchGAN is named so because it was designed so that the discriminator only analyzes both the fake and the ground truth images in patches. These patches are taken in a convolutional pattern so that the entire image is represented. The results

are averaged and that is used for the output of the discriminator. Working with smaller samples of images allows the system to create more detailed results.

2.3.4 GANs and Audio

Due to their success with images, there have been attempts to train GANs to produce audio. When working with images the input images are read as floating-point numbers and the neural network outputs more floating-point numbers which are then transformed into an image. Things aren't always so simple with audio. One common approach [12] to working with audio in machine learning is to transform audio into a spectrogram. This is an image representation of a sound that allows a neural network to process audio as if it were an image. The GAN can then produce an image that can be transformed back into raw audio with either a neural network [40] or a linear transformation. Both of these may lose quality, however, often neural network solutions perform better than linear transformations [40]. This has the advantage that it can be processed as an image which is where most GAN research has happened. The other approach is to use audio directly in the neural network as with WaveGAN [7] and GAN TTS [9]. Outside of these two approaches, there are a few others such as using midi notes [13]

2.3.5 SinGAN

SinGAN [35] is an innovative deep convolutional generative adversarial neural network that uses the Wasserstein GAN loss function. SinGAN can generate realistic images based on a single input image using creative and interesting techniques. We will explain these techniques in this section and in later sections discuss their uses for generating audio with AudioSinGAN.

The system consists of a pyramid of generative adversarial networks. The individual GANs are based on PatchGAN [36]. They also use WGAN-GP [35] for a loss function.

To train the system the input image is downsampled by a certain amount for each level of the pyramid. The lowest level (let it be n) has the smallest version of the input image and each ascending level has a larger version. At the lowest level of the pyramid, the generator takes in a noise map and is taught to map it to an image that can fool the discriminator. The

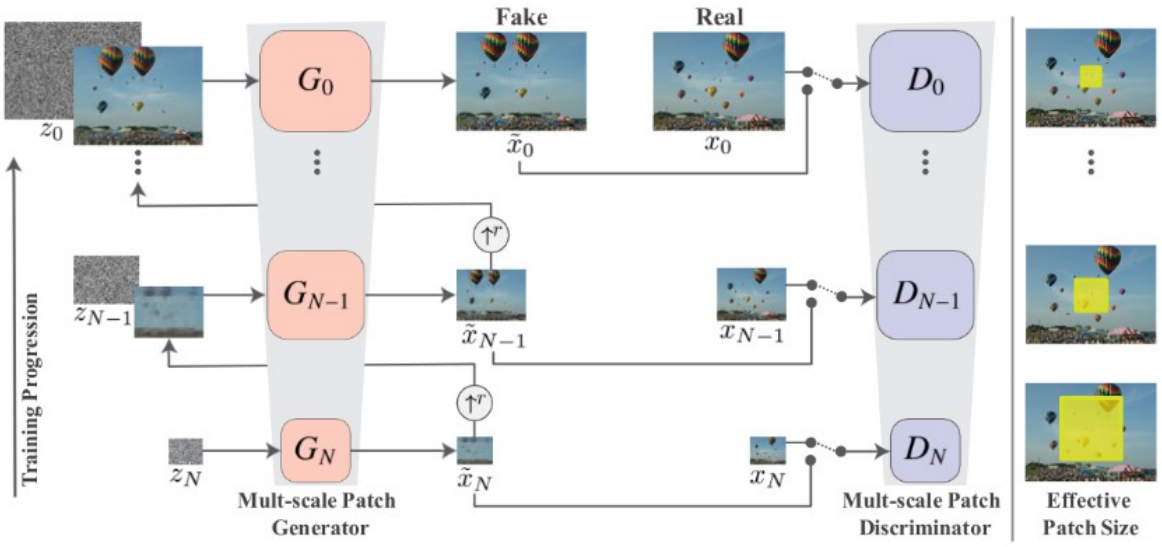

Figure 2.7 SinGAN's [1] multilevel pyramid / pipeline.

discriminator, being based on PatchGAN, has a limited receptive field and so can analyze both the fake images from the generator and the real downsampled image from the discriminator in fixed chunks of pixels.

The generator at the bottom layer is trained for a set number of epochs. The final, resulting fake image (let it be z) is upscaled (with bilinear interpolation) and fed into the next level of the pyramid (let it be n+1.) Here z is combined with a noise map which is fed into the generator and produces an image which is again combined with z. The generator is therefore performing an image-to-image task. The GAN is once again trained for the set number of epochs.

At this level n+1, the discriminator, is using a real image that is a downsampled version of the image input by the user but larger than the version used at level n. The discriminator analyzes the image in patches (the network's receptive field) which is the same size at all levels of the pyramid. This process is repeated for every layer of the pyramid.

The individual GANs in SinGAN are based on PatchGAN [36] and have the signature feature of the discriminator looking at the image in patches. Because the receptive field (or patch) is the same at each level even though it is training on different sized versions of the same image it learns to recreate large structures and similar details on a very downsampled

version of the image at the low levels of the pyramid. At the high levels, it learns to recreate textures and fine details as that is all the limited receptive field can handle at one time at that level of the pyramid.

SinGAN [1] works well on default settings when an image's superstructures allow for some variation. For example, in the image presented above of birds flying the random variation in the structure does not impair how realistic the image is. When there is little room for variance in superstructures, however, SinGAN struggles. An example of this phenomenon occurs with faces. On default settings, variations on images of faces come out looking mangled and like nothing human. However, this can be fixed by training the system on an image of a face normally but starting from a higher level of the pyramid for sample generation. This results in superstructures that are the same as the original but textures that have randomness in them. An example would be a face that is the same but has different patterns of freckles.

SinGAN can also be used for other functions such as harmonization, paint to image, editing, super-resolution, and single image animation. These are performed by using a trained GAN pyramid in different ways. Harmonization and paint to image use techniques such as inputting an image at a higher level of the image pyramid to alter textures while leaving super structures the same. Editing is performed similarly by taking in an edited version of the training image and inputting it higher in the pyramid than the bottom level. This way only textures are changed but the superstructures are left alone. Super-resolution is performed by upscaling an image, injecting it into the highest level of the pyramid, and then repeating with the output multiple times. For animation from a single image several random variations are taken and stitched together as the frames of a video.

Chapter 3

Adapting SinGAN to Audio

While presenting WaveGAN [7], its creators also discuss the steps they took to bootstrap a two-dimensional GAN meant for images and convert it for use with audio. They also suggest their work could be used as a guide for altering image generation models for use with audio. Not all of their suggestions were relevant to this project due to some differences in the GAN they started with compared to SinGAN but some of their suggestions were helpful and acted as a starting point for adapting SinGAN. Firstly, they point out the necessity for flattening the kernel and stride from two dimensions to one. They also advised removing batch normalization which we found to be helpful. Finally, t was necessary to alter the number of input channels from three (ideal for color images) to one (ideal for mono audio).

There was also helper code (see Appendix A) that needed to be changed from loading, processing, re-sampling, and saving images to doing the same with audio. We also had to set up tools to run and monitor our system as it ran. Once these tasks were accomplished, we were free to focus on altering numerous parameters that define the neural networks in our system.

3.1 Platforms / Tools

3.1.1 Torchaudio and bug fixing

SinGAN [1] is built on PyTorch [41] a popular, open-source machine learning framework. When working with audio we incorporated Torchaudio, a library that is part of the PyTorch project specifically built for working with audio and machine learning. This library includes helpful tools for loading, saving, and processing audio. However, because Torchaudio is a

recent addition to PyTorch we needed to use a more updated version of PyTorch than SinGAN was built with. Unfortunately, using newer versions of PyTorch caused errors in the system.

Eventually, we discovered the cause of the error and solved it. The portion of the SinGAN [1] code that handles training the models and updating their optimizers is based on the Deep Convolutional Generative Adversarial Network (DCGAN) example available from the PyTorch [41] documentation. However, as an optimization (as mentioned in the SinGAN supplementary material [42]) SinGAN performs three discriminator steps followed by three generator steps compared to the DCGAN example which alternates between one step of each.

PyTorch [41] checks for and disallows in-place updates for variables used for gradient computation. Old versions of PyTorch did not check for in-place operations performed by optimizers correctly. This allowed the SinGAN [1] code to run as it was. However, in PyTorch 1.5 this was patched, breaking SinGAN. The specific problem occurred because the optimizer for the generator took a step and then almost immediately backpropagated the error. It is not clear which specific variable caused the problem but changing the system to alternate the steps (like the DCGAN example from the PyTorch documentation) involved resetting many variables and successfully fixed the issue.

3.1.2 Julius

The resampling function built into Torchaudio proved to be a substantial bottleneck to training the system. To solve this problem we incorporated the library Julius [43]. Julius is an open-source implementation of the sinc resample algorithm by Julius O. Smith [44]. The implementation is based on PyTorch and can be run on the GPU. Using Julius and following the documentation's guidelines for maximizing speed allowed for much faster training. It is worth noting that after the code for AudioSinGAN was written Julius was integrated into pyrtoch [41].

3.1.3 Google Colab / Drive

We ran all our experiments on Google Colab. Google Colab is a platform built for research that allows running Python code on Google's servers with the usage of a GPU for free. Most of our experiments still took several hours to run and we ran over 300 experiments throughout our research.

3.1.4 WandB

The creators of SinGAN left behind code to graph loss curves using the Matplotlib library. However, we elected to use an online tool called Weights and Biases [45] (WandB) due to its superior ease of organization and tools for analyzing data.

3.2 Tuning Hyper-Parameters / System Variables

Tuning Hyper-parameters was more difficult than initially expected. Each AudioSinGAN model has three loss functions (generator, critic, and means-square error) for each level of the pyramid. Often these different loss functions seemed to disagree with each other, and the loss graphs were also often unhelpful with large sudden jumps or large static sections. There was no single, clear, consistent loss function to base our search on. We settled on evaluating results by evaluating generated samples by ear and subjectively determining if the result was an improvement or not.

The search for hyper-parameters had a baseline in SinGAN's [1] default hyper-parameters and recommendations found in other literature. The tuning process was done using an audio clip of a violin playing a scale and started by trying extreme values both larger and smaller than the baseline. Then we used progressively less and less extreme values centering around the recommended values. In every case the results were better around the baseline values. Once that was established, we used a brute force approach by trying all values near the baseline values. In some cases the baseline values were optimal and in others values near the baseline values were

superior. Further value tuning would be helped by developing and using an objective measurement of improvement in training that could be used for (ideally automated) hyper-parameter tuning.

Table 3.1 shows SinGAN's original hyper-parameters compared to what we selected for AudioSinGAN. The following sections go into more detail regarding the various hyper-parameters.

Table 3.1 SinGAN vs AudioSinGAN Hyper-Parameters.

Hyper-Parameter	SinGAN	AudioSinGAN
Layers	5	3
Epochs	2000	10000
Kernel	3x3	23
Stride	1	1
Dilation	1	12
Receptive Field	11x11	793
Learning Rate - Generator	0.0005	0.0001
Learning Rate - Discriminator	0.0005	0.0004
Loss Function	WGAN-GP + MSE Reconstruction	WGAN-GP + MSE Reconstruction
Reconstruction Weight	10	4
Batch Norm	Yes	No

3.2.1 Resolution levels on SinGAN's Pyramid

SinGAN uses a pyramid of images at different resolutions and corresponding GANs. When working with audio we chose to use a pyramid with audio at different sample rates. This is how many samples of audio are present in each second. A commonly used sample rate for good quality music is 44.1 kHz or 44,100 samples per second. However, low-quality audio at lower sample rates is used in some real-world situations. For example, audio sampled at 8kHz is sometimes used for sound effects [46]. The contrast between Figure 3.1 and Figure 3.2 is a

visual example of the difference. These illustrate the difference between 44.1 kHz and 8kHz. Every individual dot is a sample.

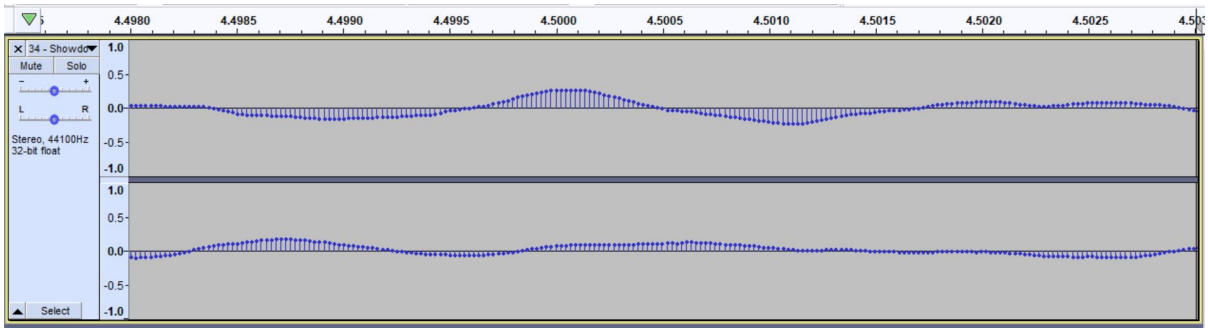

Figure 3.1 0.005 seconds of audio at 44.1 kHz.

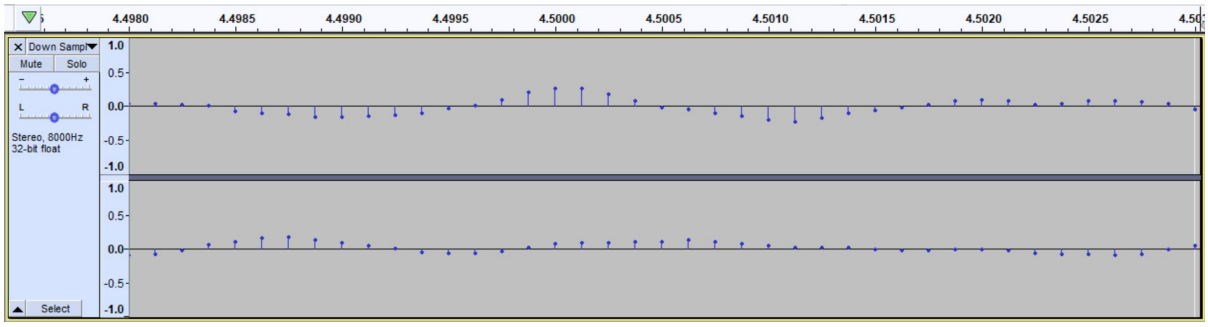

Figure 3.2 0.005 seconds of audio at 8 kHz.

At the lower resolution, the large structures of the sound wave are maintained but the small details are lost. This key detail is the same as it is in images of different resolutions where the superstructures in an image are maintained at low resolution but the details are lost. One different area is that audio at different sample rates is the same length while images at different resolutions are considered different sizes.

3.2.2 Sample Rate at Different Scales

SinGAN [1] has a difference of approximately 0.75 between the resolution of each scale of the pyramid. We roughly followed this example for the different sample rates of audio. While SinGAN dynamically calculates how many levels to use and the appropriate resolution

at each level, AudioSinGAN uses set sample rates at each level to optimize re-sampling time. We calculated the sample rate for each level of the pyramid with a difference of around 0.75 (following Julius' guidelines for fast re-sampling caused it to not be exact) between scales from 800 Hz to 16000 kHz. Throughout our experimentation, we tried many different variations such as using more scales or fewer scales. We did not notice too much of a difference between experiments as long as the scales range from small enough to large enough and as long as there are enough of scales present (4 or 5 at the minimum). Our final system converts audio input into mono audio at 16 kHz and uses 9 scales ranging from 800 Hz to 16 kHz.

3.2.3 Learning Rate, Layers, Epochs, and Channels

One of the most important and volatile hyperparameters is the learning rate. Initially, we suspected that our discriminators had too high of a learning rate and experimented with decreasing it. We also tested only updating the discriminator's weights every few steps. Neither that technique nor the various learning rates we tried were initially successful. Eventually, on the recommendation of a blog post on training GANs [47], we tried learning rates of 0.0001 for the generator and 0.0004 for the discriminator. These values for learning rates proved to be effective and are our final values after tuning.

Two other very important hyperparameters are how many layers are in the neural network and how many epochs the system is trained for. SinGAN uses 5 layers in both the discriminator and the generator and trains for 2000 epochs for each level of the pyramid. We found that we got better results with roughly similar running time by decreasing the number of layers and increasing the number of epochs. We found that 3 layers and about 10000 epochs worked better for AudioSinGAN.

We also experimented with how many channels to use between layers. SinGAN starts with 32 channels and then increases it as the image goes up the pyramid and gets larger. We didn't find that having more channels than 32 between each level of the pyramid made any noticeable difference.

3.2.4 Reconstruction Weight

SinGAN [1] uses both the WGAN-GP [35] loss function and a weighted mean squared error reconstruction loss function. We experimented with various weights for the reconstruction loss function and found that about 4 seemed to work well, though it isn't a particularly delicate hyperparameter, especially compared to the learning rate.

3.2.5 Receptive Field Size

One of the key points of SinGAN [1] is the GAN's receptive field. SinGAN uses the PatchGAN [36] method and so intentionally has a smaller receptive field. Because the receptive field is static at each level of the pyramid but the resolution increases, the discriminator sees smaller details at higher levels of the pyramid and larger structures at lower levels. As an image is generated it works its way up the pyramid with finer details being added at higher levels.

The WaveGAN [7] paper points out that the receptive field for audio systems must be much larger than for image systems. For example, an A4 note at 16kHz takes 36 samples for a single cycle. SinGANs naively has an 11 x 11 receptive field according to its creators. For a point of contrast, WaveGAN has a receptive field of 32,761 and after much experimentation, our converted system has a receptive field of 793 [48]. One of the variables that control the size of the receptive field in a convolutional neural network is the kernel size. We experimented with many kernel sizes to control the receptive field size before settling on 23.

We eventually realized we needed to expand the receptive field size but expanding the kernel didn't expand the receptive field nearly enough for how much extra computational power is required. Using dilation [49] of 12 increases the receptive field dramatically without increasing the processing power required.

3.2.6 Various experiments

We also tried many more varied experiments. We found that SinGANs use of schedulers to decrease the learning rate as the system trains weren't helpful to our version of the system and disabled it. We experimented with using mean absolute error loss instead of mean squared error

loss for the reconstruction loss but found that the results were inferior. We also experimented with keeping our sample rate the same at all levels of the pyramid and varying our kernel size (and therefore receptive field) instead.

3.2.7 Other Experiments

We attempted other experiments that failed to improve our output. These include normalizing generator output, updating weights on different numbers of steps, updating discriminator weights only when the generator's loss is decreasing, using different loss functions, using batch normalization, and various changes to the pyramid setup.

3.2.8 Audio Amplitude and Length

To achieve optimal results, it is important to consider the input audio's amplitude. Amplitude is the height of the crests of the sound wave. Higher amplitudes result in more distinct waveforms and produce better results. We found it helpful to increase the amplitude of audio clips as much as we could before feeding them into AudioSinGAN.

The length of the audio clip is also worth paying attention to. If the input audio clip is too long, then the system will require too many resources to run in a reasonable time frame. If the system is too short it makes it difficult for there to be clear, distinct structures. We found that audio clips of about 5 - 10 seconds worked well.

3.2.9 AudioSinGAN Pipeline

Figure 3.3 is an example of AudioSinGAN's pipeline. It can be contrasted to SinGAN's original pipeline in Figure 2.7. Like the original SinGAN model, training starts at the bottom level of the pyramid. The system learns to produce realistic super-structures in the audio and then sends the results up the pyramid. At each level the effective patch (or kernel) size gets smaller and the system fills in finer and finer details.

SinGAN dynamically determines the number of levels in the pyramid and the resolutions for each level. As explained in section 3.2.2 AudioSinGAN re-samples any input audio to 16000 hz and then uses set levels and sample rates.

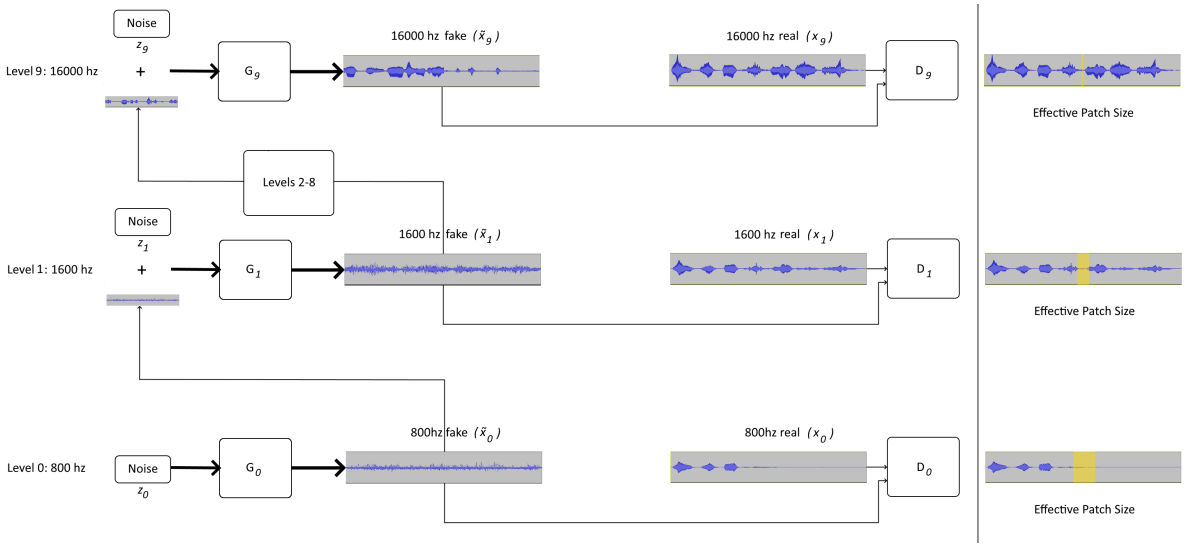

Figure 3.3 A diagram of AudioSinGAN's Pipeline.

3.2.10 AudioSinGAN Code

The codebase for AudioSinGAN is a heavily modified version of SinGAN [1]. Although some parts remain unchanged much of it required significant alterations. The altered code can be found in appendix A. Figure 3.4 is a simple top-down diagram of the code can be found below. It describes which scripts call functions from each other in a top down approach. There are three possible starting scripts each with their own main function. The file main_train.py starts training the audio on an input file, resume_at.py resumes stopped training, and random_samples.py generated samples from a trained model. The files resume_at.py and AudioSample.py are created by (with resume_at.py being heavily based on main_train) while the others are altered versions of SinGAN's code.

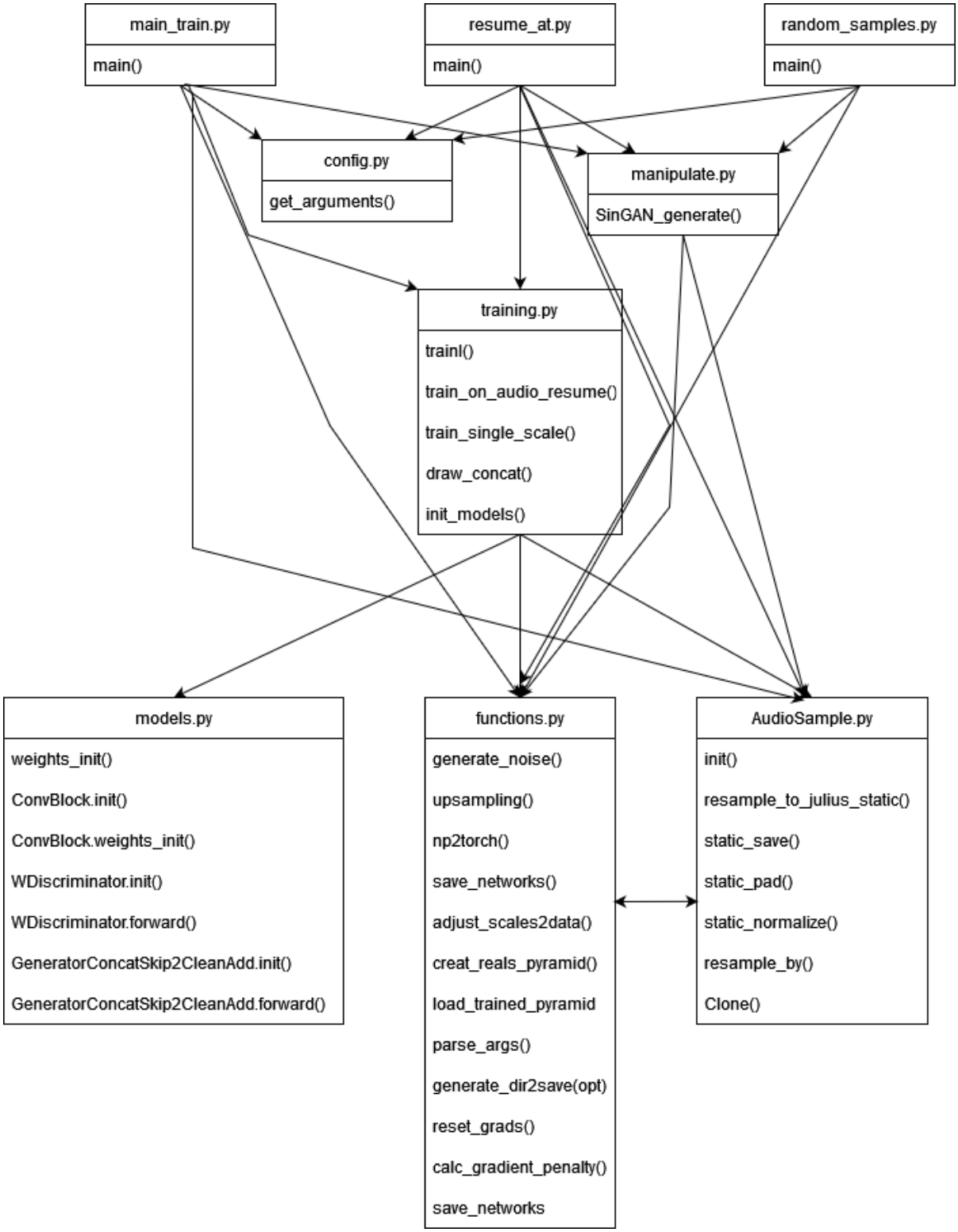

Figure 3.4 A top-down diagram of AudioSinGAN's Code.

3.3 Singular Value Decomposition

More details of singular value decomposition and its applications can be found in [50]. Let A be an $m \times n$ matrix, with $m \geq n$. A can be factorized to

$$A = U \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} V^T \tag{3.1}$$

where $U \in R^{m \times m}$ and $V \in R^{n \times n}$ are orthogonal, and $\Sigma \in R^{n \times n}$ is diagonal. If $rank(A) = r < m, n$ the above factorization of A can be written as,

$$A = [U_1 U_2] \begin{pmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix} = U_1 \Sigma_1 V_1^T \tag{3.2}$$

where $U_1 \in R^{m \times r}$, $U_2 \in R^{m \times (m-r)}$, $\Sigma_1 \in R^{r \times r}$, $V_1^T \in R^{r \times n}$, and $V_2^T \in R^{(n-r) \times n}$.

If $rank(A) = r < m, n$ the above factorization of A can be written as,

$$A = [U_1 U_2] \begin{pmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix} = U_1 \Sigma_1 V_1^T \tag{3.3}$$

where $U_1 \in R^{m \times r}$, $U_2 \in R^{m \times (m-r)}$, $\Sigma_1 \in R^{r \times r}$, $V_1^T \in R^{r \times n}$, and $V_2^T \in R^{(n-r) \times n}$, and the diagonal entries of Σ_1, $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_r > 0$.

$$\Sigma_1 = \begin{pmatrix} \sigma_1 & 0 & . & . & 0 \\ 0 & \sigma_2 & 0 & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & . & . & . & \sigma_r \end{pmatrix} \tag{3.4}$$

The simga values $\sigma_1, \sigma_2, \ldots, \sigma_r$ are called the singular values of A. The columns of V $(v_1, v_2, \ldots, v_r)$ are the right singular vectors of A and the columns of U $(u_1, u_2, \ldots, u_r)$ are the left singular vectors of A. Although, Σ is unique, this factorization is not unique (i.e., can flip signs of U, V).

In our application, we construct a matrix A for each audio class such that the columns of A are the training audio vectors of the selected class. For a given test vector z, we need to find a

projection on to the column space of A (i.e., the least square solution). The least square error for $Ax = z$ is same as the least square error for $Uy = z$ (i.e., $y = \Sigma_1 V_1^T x$. The least square solution for $Uy = z$ is $y = U^T z$ and hence the residual error vector is $z - UU^T z = (I - UU^T)z$. We call the columns of U as singular images.

3.3.1 SVD Algorithm:

a Training:

For the training set of each audio class i (i,e., violin, male voice etc), compute the SVD of A_i containing each set of vectors of one kind as columns.

b Classification:

For a given test audio vector, compute its relative residual (i.e., $e_i = ||(I - U_i U_i^T)z||_2$, least square error) in all audio classes. If one residual is lower than all the others, classify as that.

Chapter 4

Results

Although objective evaluation was out of our means we provide some visual examples of waveforms to demonstrate the output of AudioSinGAN.

4.1 Dataset

We created an initial version of a dataset from the variations created by SinGAN on seven samples. This dataset is available as both audio files and as a csv with audio vectorized[1].

Table 4.1 Dataset Statistics

Sample Name	Variations	Variation Length
allegro	50	7.2
female voice	50	8.8
flute	50	3.6
male voice	50	7
melodic hardcore	50	7.9
ocarina multiple	50	9.5
violin	50	9

The vectorized dataset has three versions. The full version has the total vectors of all samples. The limited version has the first 50,000 points of each vector for all samples. Finally,

[1]DataSet: `https://drive.google.com/drive/folders/1oHtNeX1whK9kJaOBwYOm0F2dRmDRxLK5?usp=sharing`

the small version has the first 50,000 points of the vectors of only the violin and male voice variations. For each class there are 51 examples including the input audio and 50 variations. Table 4.2 describes the layout of the csv and Table 4.3 describes class labels.

Table 4.2 Dataset Columns

C1 (int)	C2 (int)	C3 (binary)	C4 (binary)	C5+ (float)
Unique ID	Class	Variation (0) or Original (1)	Rejected(0) or Accepted (1)	Vectorized Data

Table 4.3 Dataset Classes

Class 0:	violin
Class 1:	male voice
Class 2:	female voice
Class 3:	allegro
Class 4:	flute
Class 5:	melodic hardcore
Class 6:	ocarina multiple

4.2 Successes and Failures

AudioSinGAN is sensitive to qualities in the input audio and performs better on some clips than others. The system performs best on waveforms with multiple, distinct clear structures which represent an independent sound. The waveform in Figure 4.1 is an example of a waveform with clear, distinct waveforms. Each crest is a distinct note played on a violin.

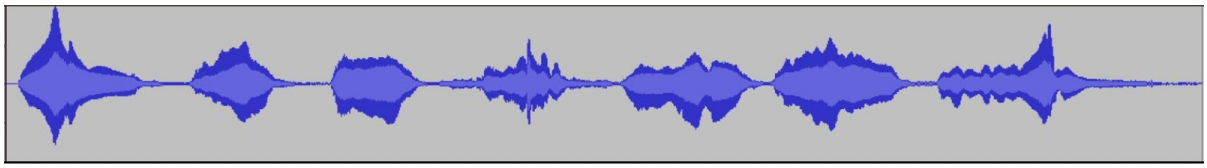

Figure 4.1 Input Violin Waveform

And below are several examples of output. The images of the waveforms in Figure 4.2 show our success at creating a system that can train on a single audio clip and generate variations on

it. The output audio clips lack audio quality. Despite this, however, they exhibit varied violin sounds distinct from what was present in the training clip.

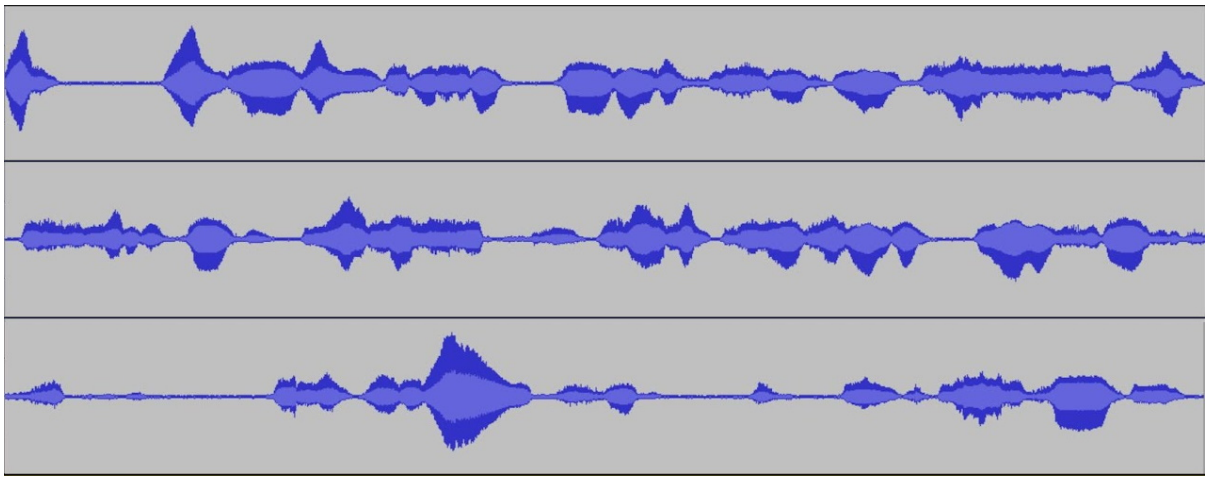

Figure 4.2 Violin Output Audio

To contrast, Figure 4.3 is a waveform with multiple instruments playing together. Our system doesn't work well with it because it has many different notes that blend and aren't distinct.

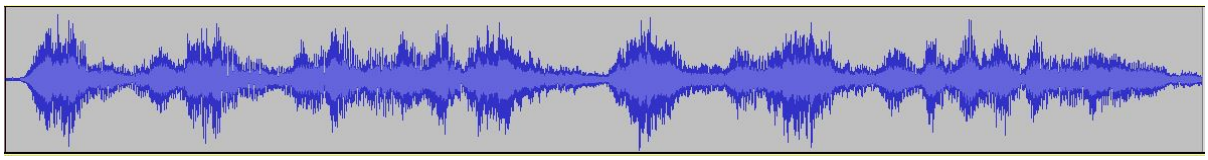

Figure 4.3 Allegro Waveform

The waveforms of the results in Figure 4.4 look similar to the original and the audio has some vague resemblance, but in comparison to the results from the violin the audio quality is far inferior and there is significantly less interesting audible variation.

More examples and audio samples are available online[2].

[2]Audio Samples: `https://sites.google.com/view/audiosingan`

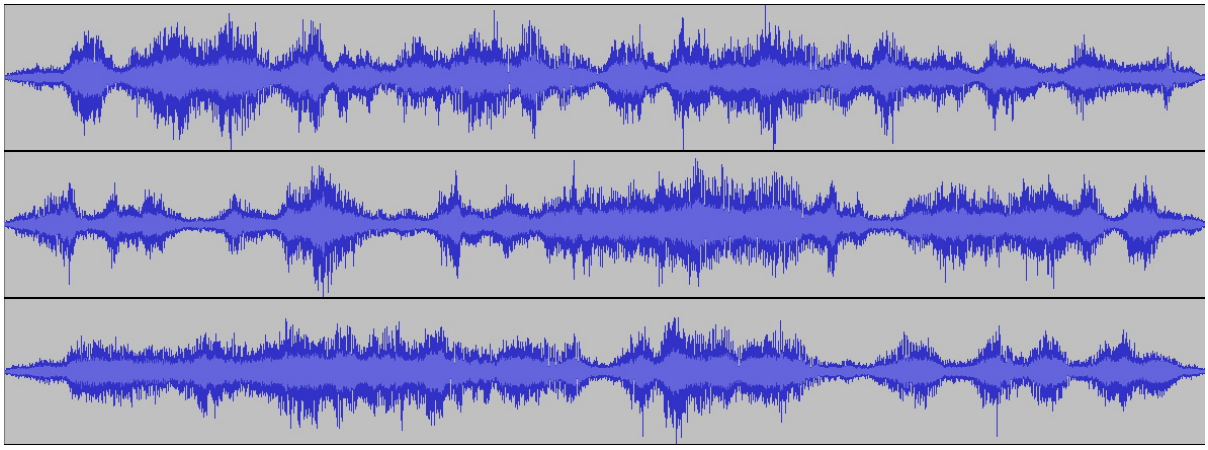

Figure 4.4 Allegro Output

4.3 Violin Audio Analysis

Although we are not experts in the domain of music there are several interesting details about the systems greatest success: violin. Firstly, it's worth noting that the violin samples are generated from the input clip that the system hyper-parameters were tuned to. They system was effectively optimized to be most successful on audio that meets some criteria pertaining to our violin clip. We have speculated that this criterion is the frequency of the music. This could explain why the system performs better on violin compared to flute (the second most successful case) despite both being scales with a similar structure to the sound wave. For comparison the violin clip has frequencies ranging from 292 Hz - 2330 Hz while the flute clip has frequencies ranging from 700 Hz - 5600 Hz.

Another interesting phenomenon from the violin input clip is that the D notes in the scale come through the clearest in the variations. Both a lower D note and a higher D note are present in the input clip. The repetition of two versions of the same note could partially explain why the system reconstructs them most clearly. Another contributing reason could be that the lower D note has the largest structure in the input sound wave.

4.4 Analysis using Singular Value Decomposition

With the help of a more knowledgeable musician we went through the violin samples and labeled them as accepted and rejected. We seek to understand what features define good output data initially through visual examination but more meaningfully through singular value decomposition as well.

A visual examination of Figures 4.5, 4.6, and 4.7 reveals that the original and accepted waveforms have clearer distinction between portions of the waveforms while the rejected sample has a large portion on the left that isn't distinct.

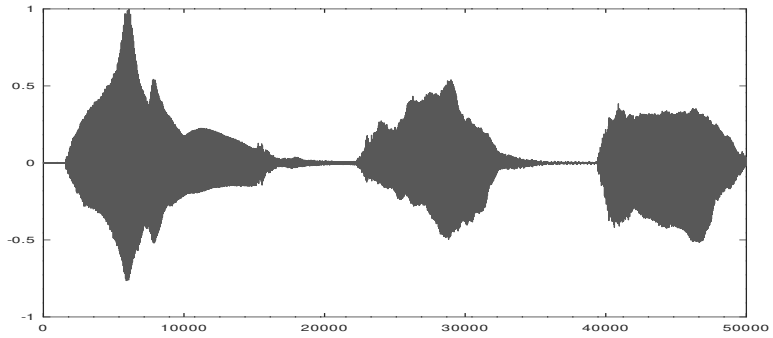

Figure 4.5 Original Violin Sample.

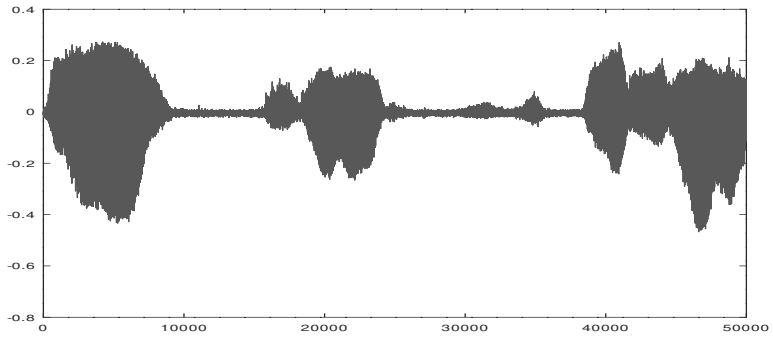

Figure 4.6 An Accepted AudioSinGan Generated Violin Variation Using the Original Sample Given in Figure 4.5.

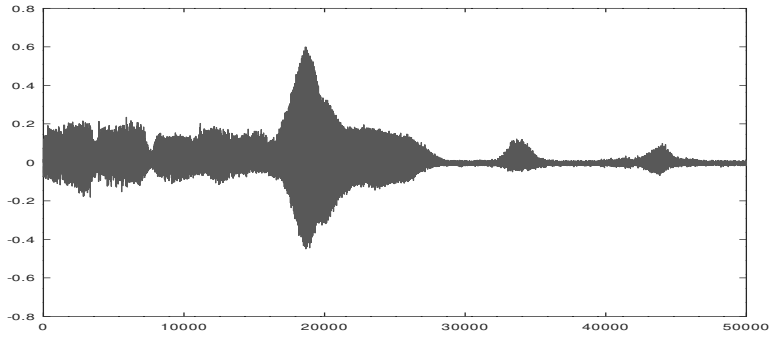

Figure 4.7 A Rejected AudioSinGan Generated Violin Variation Using the Original Sample Given in Figure 4.5.

We performed SVD to attempt to train a model to distinguish between rejected and accepted violin samples and had poor results. More labeled data, better quality data, and more experimentation with using data at different scales could help to improve the model. Although distinguishing rejected from accepted violin samples was not successful, we trained a model to distinguish between violin samples and human male voice samples and, even on a small training set, the model was 100% successful. See Figures 4.8 and 4.9 for examples of human voice waveforms.

See Section 3.3 for an in-depth discussion of training and classification methods. Interestingly, we find that both the violin class and the male voice class have large initial singular values followed by significantly smaller singular values. See Figures 4.10 and 4.11 for graphs of the singular values and Figures 4.12 - 4.15 for the waveforms that correspond to the first two singular values for both classes.

We use SVD (as laid out in section 3.3) to analyze violin and male voice and try and classify samples from our test set. We use the least square error for each sample to classify them into the appropriate class. Our model has 100 % success at distinguishing between the two but the test points are often very close to the boundary. This is especially true with male voice test samples. Figure 4.16 shows the difference between errors graphed. Table 4.4 and Table 4.5 numerically show the errors and the difference between errors for the samples.

We also analyze the success rate of the violin-voice classification compared to the rank of the SVD approximation. We find that the model has a success rate of 95% at rank five and at rank 18 the model has a success rate of 100%. See Table 4.6 for numerical data and Figure 4.17 for a graph of the data.

We find that the models 100% classification success rate and the high success rate at low ranks to be indicative of AudioSinGAN producing data belonging to distinct classes. The closeness of test samples error to class boundaries does show there is still more work to be done.

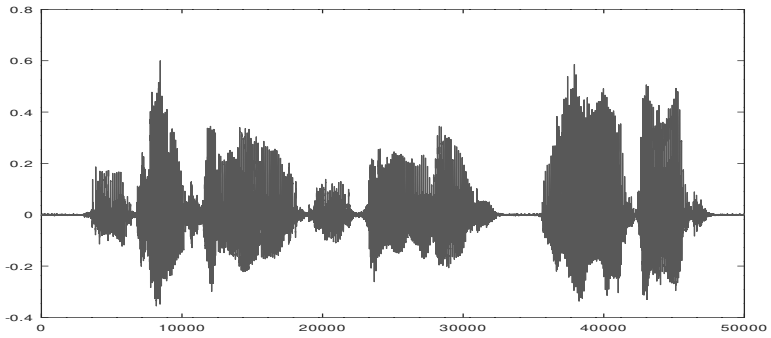

Figure 4.8 Original Male Voice Sample.

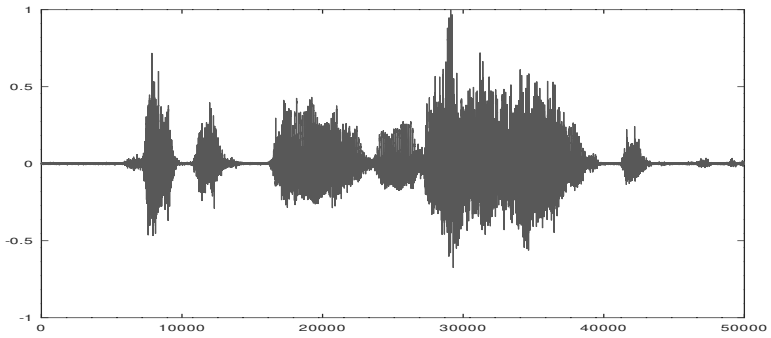

Figure 4.9 A Variation of Male Voice Sample

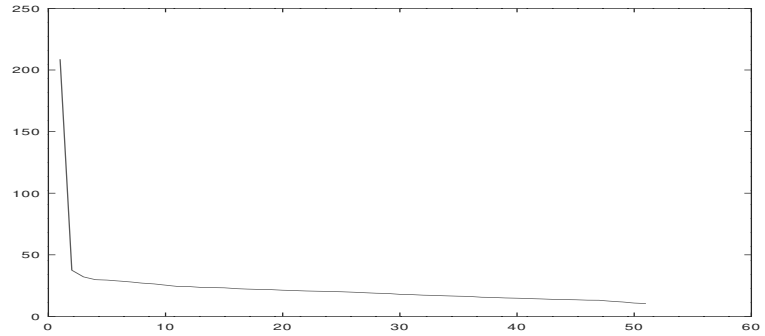

Figure 4.10 Singular Values for the Violin Class, Range = {208.7, 37.5, ..., 10.4}

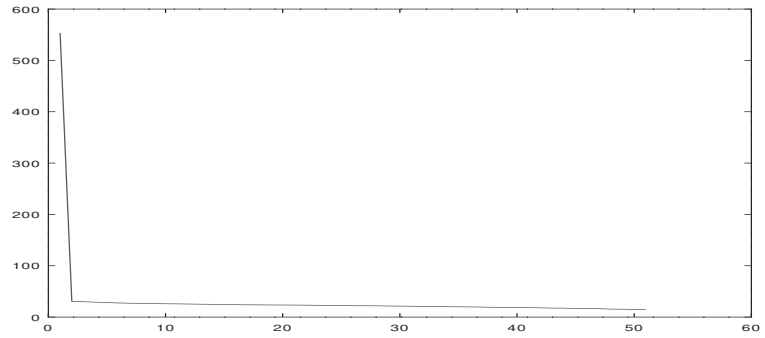

Figure 4.11 Singular Values for the Male Voice Class, Range = {553.4, 30.4, ..., 14.6}

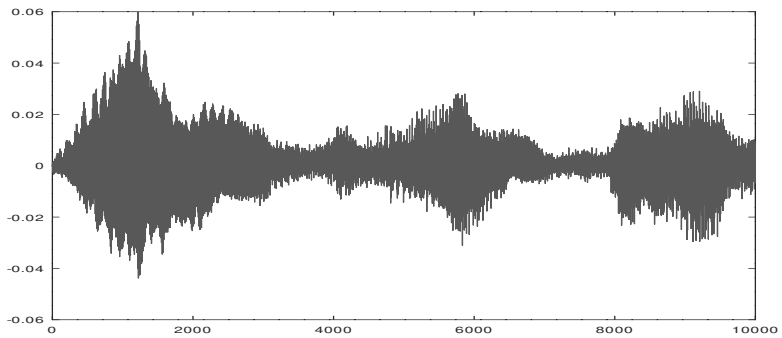

Figure 4.12 The First Violin Singular Image corresponding to Singular Value = 208.7

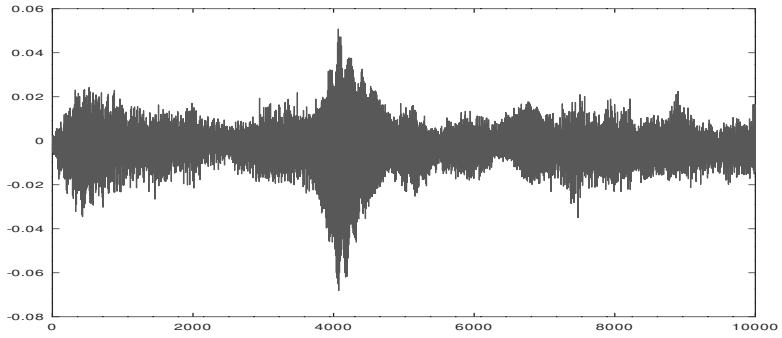

Figure 4.13 The Second Violin Singular Image corresponding to Singular Value = 37.5

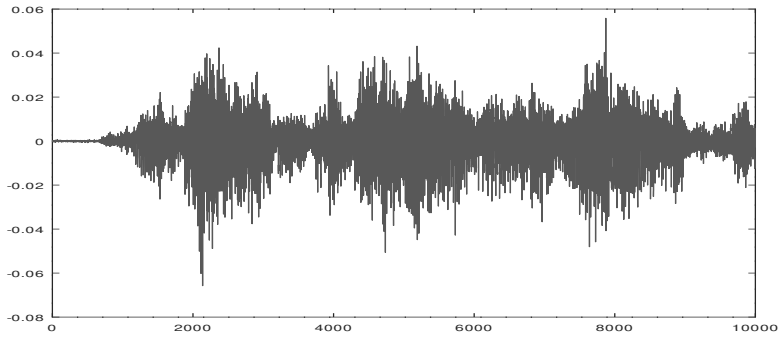

Figure 4.14 The First Male Voice Singular Image corresponding to Singular Value = 553.4

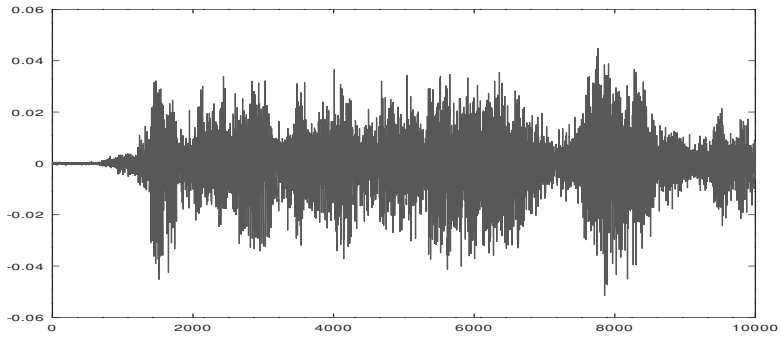

Figure 4.15 The Second Male Voice Singular Image corresponding to Singular Value = 30.4

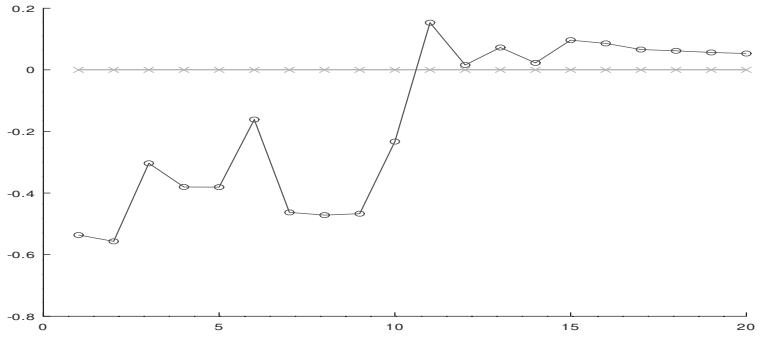

Figure 4.16 The Difference of Least Square Errors (e1-e2) for 20 Test Samples; (1-10) violin samples, (11-20) - voice samples, success rate = 100%

Table 4.4 Least Squared Errors: Violin Test Samples Rounded to two Decimal Points.

Error Violin (e1)	Error Male Voice (e2)	Difference (e1 - e2)
7.96	8.50	-0.54
9.40	9.95	-0.55
8.84	9.14	-0.30
11.49	11.87	-0.38
9.44	9.82	-0.38
10.25	10.42	-0.17
7.21	7.67	-0.46
9.68	10.15	-0.47
9.72	10.19	-0.47
7.63	7.86	-0.23

Table 4.5 Least Squared Errors: Male Voice Test Samples Rounded to two Decimal Points.

Error Violin (e1)	Error Male Voice (e2)	Difference (e1 - e2)
9.08	8.93	0.15
6.54	6.53	0.01
12.15	12.08	0.07
13.25	13.22	0.03
7.11	7.02	0.09
10.32	10.23	0.09
10.21	10.14	0.07
9.82	9.76	0.06
10.24	10.19	0.05
9.32	9.26	0.06

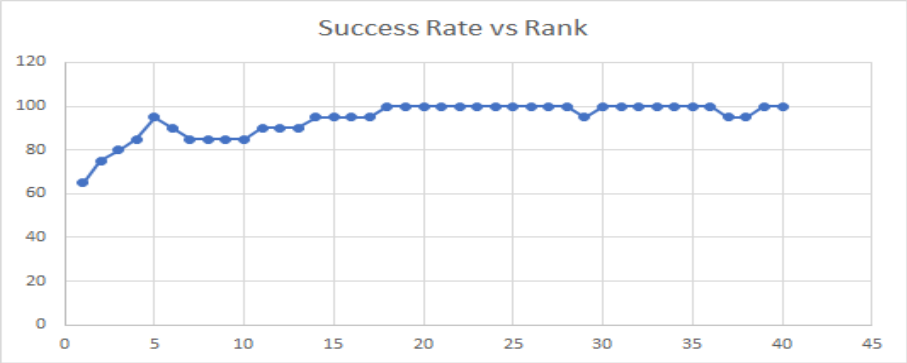

Figure 4.17 Success Rate of Violin-Voice Classification vs Rank of the SVD approximations

Table 4.6 Violin-Male Voice Rank and Success Rank

Rank	Success Rate %	Rank cont.	Success Rate % cont.
1	65	21	100
2	75	22	100
3	80	23	100
4	85	24	100
5	95	25	100
6	90	26	100
7	85	27	100
8	85	28	100
9	85	29	95
10	85	30	100
11	90	31	100
12	90	32	100
13	90	33	100
14	95	34	100
15	95	35	100
16	95	36	100
17	95	37	95
18	100	38	95
19	100	39	100
20	100	40	100

Chapter 5

Future Directions and Conclusion

SinGAN was extended by its original authors to perform several advanced functions. One of these were style transfer / harmonization where a hybrid image was fed into the system and the style of one part of the image is given to another. Another was re-scaling images to higher resolutions. These were performed by taking advantage of SinGAN's unique structure and feeding input into high levels of the pyramid to affect texture and not super-structures. Both of these have potential for being ported to audio. Harmonizing two separate audio clips to blend together easier would be useful to sound designers. Up-sampling audio with neural networks is an area already being studied [51] and AudioSinGAN's pyramid structure has potential for a unique contribution.

Multiple researchers have sought to improve SinGAN using a variety of techniques with potential for use in AudioSinGAN. Among these are employing an attention mechanism in the GAN [14], using alternative activation functions[15], and using "pixel shuffling" in the input at each level of the pyramid [16].

One problem we faced in this work was a lack of objective measurement of audio realness / results. We started the process of using SVD as a potential solution to this problem. In our work with SVD we created a model that can tell the violin class apart from the male voice class with 100% success. Improving our SVD model would allow for better tuning of hyper-parameters, creation of automatic feedback functions, superior understanding of the model, and a quick way to showcase the best results of AudioSinGAN.

A challenge faced in this work is the difficulty of classifying audio. This is especially difficult in cases where there may be multiple instruments or audio sources playing together.

This issue of overlapping audio is one faced by other researchers as well [52]. One way to address this would be to handle frequency bands in audio separately.

AudioSinGAN primarily alters an audio clip's phase and amplitude while generaly not affecting frequency. However, frequency is a key attribute of audio that would be interesting to explore in relation to AudioSinGAN. One possible explanation for the system working best on certain audio clips is that they happen to be within frequency bands that the system was tuned for. For example, our current system was optimized for violin and performs best on our violin input clip. Future work should include experimenting with tuning AudioSinGAN for a variety of types of music with varying frequencies. This could allow for splitting an input signal into multiple frequency bands and training them on variations of AudioSinGAN tuned for those frequencies. The pieces could then be combined into the final output. This would require a digital input audio clip to be converted to analog for splitting into frequencies, then the frequencies would be converted back to digital for processing, back to analog for mixing and finally back to digital. See Figure 5.1 on the next page.

A somewhat similar idea has been explored [53] as an attempt for faster and better results by training multiple levels of SinGAN's[1] pyramid in parallel with different hyper-parameters. It would also be interesting to study the individual GANs in such a system by analyzing their frequency and impulse responses.

We include an initial database of variations for these results. An important future direction would be to expand and improve that database. An improved database (including an objective measure of audio quality) along with more advanced usage of signal processing techniques and SVD would allow for future research and a better understanding of our model and the dataset.

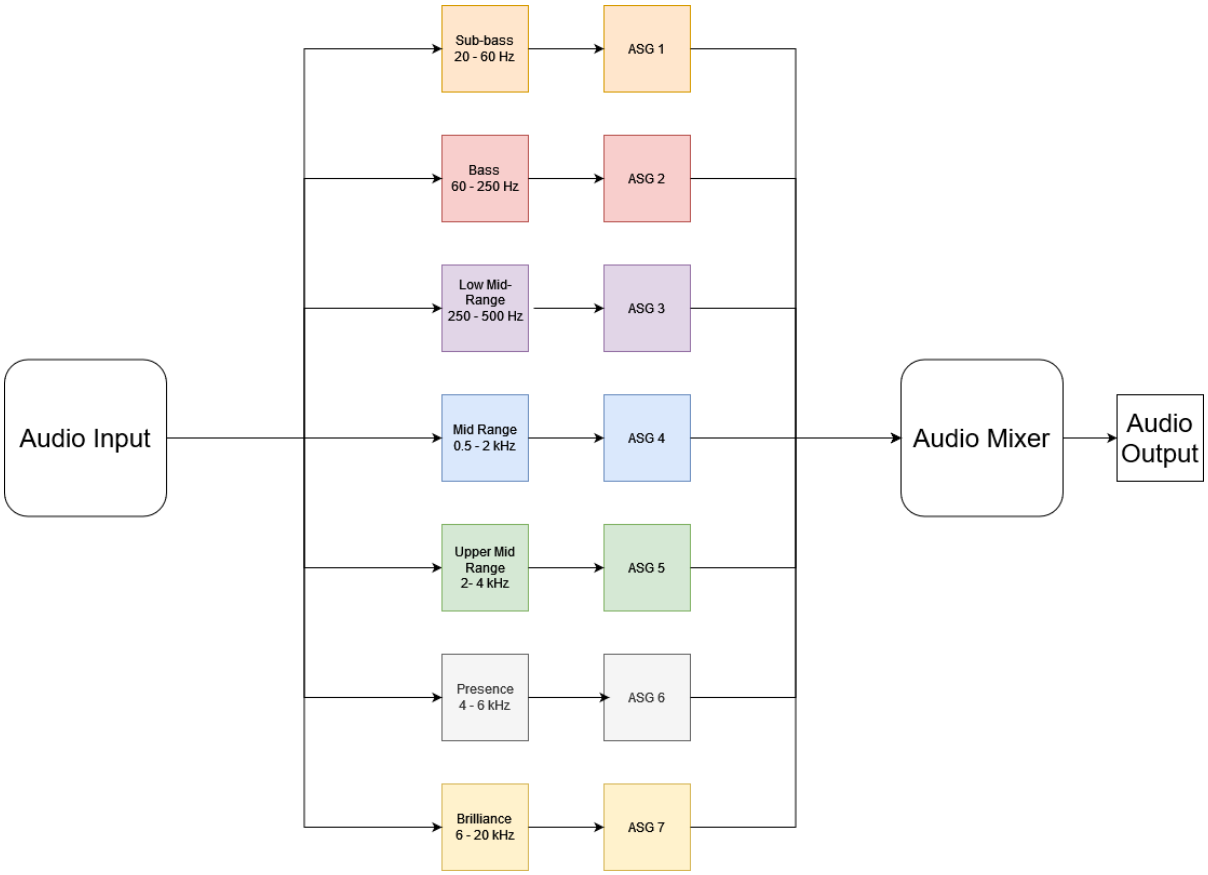

Figure 5.1 AudioSinGAN splitting audio into frequency bands and recombining it.

LIST OF REFERENCES

[1] T. R. Shaham, T. Dekel, and T. Michaeli, "Singan: Learning a generative model from a single natural image," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4570–4580, 2019.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[3] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *arXiv preprint arXiv:1809.11096*, 2018.

[4] W. Chen and J. Hays, "Sketchygan: Towards diverse and realistic sketch to image synthesis," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9416–9425, 2018.

[5] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4401–4410, 2019.

[6] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8110–8119, 2020.

[7] C. Donahue, J. McAuley, and M. Puckette, "Adversarial audio synthesis," *arXiv preprint arXiv:1802.04208*, 2018.

[8] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[9] M. Bińkowski, J. Donahue, S. Dieleman, A. Clark, E. Elsen, N. Casagrande, L. C. Cobo, and K. Simonyan, "High fidelity speech synthesis with adversarial networks," *arXiv preprint arXiv:1909.11646*, 2019.

[10] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, and I. Sutskever, "Jukebox: A generative model for music," *arXiv preprint arXiv:2005.00341*, 2020.

[11] T. Karras, M. Aittala, J. Hellsten, S. Laine, J. Lehtinen, and T. Aila, "Training generative adversarial networks with limited data," *arXiv preprint arXiv:2006.06676*, 2020.

[12] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "Gansynth: Adversarial neural audio synthesis," *arXiv preprint arXiv:1902.08710*, 2019.

[13] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang, "Midinet: A convolutional generative adversarial network for symbolic-domain music generation," *arXiv preprint arXiv:1703.10847*, 2017.

[14] S. Gu, R. Zhang, H. Luo, M. Li, H. Feng, and X. Tang, "Improved singan integrated with an attentional mechanism for remote sensing image classification," *Remote Sensing*, vol. 13, no. 9, p. 1713, 2021.

[15] R. Segawa and H. Hayashi, "Improved singan's performance by changing the activation function," *IEEJ Transactions on Electrical and Electronic Engineering*, 2021.

[16] M. Zheng, P. Zhang, Y. Gao, and H. Zou, "Shuffling-singan: Improvement on generative model from a single image," in *Journal of Physics: Conference Series*, vol. 2024, p. 012011, IOP Publishing, 2021.

[17] S. Gur, S. Benaim, and L. Wolf, "Hierarchical patch vae-gan: Generating diverse videos from a single sample," *arXiv preprint arXiv:2006.12226*, 2020.

[18] V. Sushko, J. Gall, and A. Khoreva, "One-shot gan: Learning to generate samples from single images and videos," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2596–2600, 2021.

[19] K. v. d. Broek, "Mp3net: coherent, minute-long music generation from raw audio with a simple convolutional gan," *arXiv preprint arXiv:2101.04785*, 2021.

[20] B. Warner and M. Misra, "Understanding neural networks as statistical tools," *The american statistician*, vol. 50, no. 4, pp. 284–293, 1996.

[21] P. D. Wasserman and T. Schwartz, "Neural networks. ii. what are they and why is everybody so interested in them now?," *IEEE expert*, vol. 3, no. 1, pp. 10–15, 1988.

[22] B. Macukow, "Neural networks–state of art, brief history, basic models and architecture," in *IFIP international conference on computer information systems and industrial management*, pp. 3–14, Springer, 2016.

[23] S. Kohli, S. Miglani, and R. Rapariya, "Basics of artificial neural network," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 9, pp. 745–751, 2014.

[24] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, JMLR Workshop and Conference Proceedings, 2011.

[25] "Leakyrelu - pytorch 1.10.0 documentation."

[26] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[28] Z. J. Wang, R. Turko, O. Shaikh, H. Park, N. Das, F. Hohman, M. Kahng, and D. H. Chau, "Cnn explainer: Learning convolutional neural networks with interactive visualization," *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[29] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, Ieee, 2017.

[30] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[31] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[32] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[33] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*, pp. 214–223, PMLR, 2017.

[34] C. Villani, *Optimal transport: old and new*, vol. 338. Springer, 2009.

[35] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in neural information processing systems*, pp. 5767–5777, 2017.

[36] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1125–1134, 2017.

[37] P. Sangkloy, J. Lu, C. Fang, F. Yu, and J. Hays, "Scribbler: Controlling deep image synthesis with sketch and color," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5400–5409, 2017.

[38] T. Park, M.-Y. Liu, T.-C. Wang, and J.-Y. Zhu, "Semantic image synthesis with spatially-adaptive normalization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2337–2346, 2019.

[39] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2414–2423, 2016.

[40] K. Kumar, R. Kumar, T. de Boissiere, L. Gestin, W. Z. Teoh, J. Sotelo, A. de Brébisson, Y. Bengio, and A. Courville, "Melgan: Generative adversarial networks for conditional waveform synthesis," *arXiv preprint arXiv:1910.06711*, 2019.

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

[42] T. R. Shaham, T. Dekel, and T. Michaeli, "Singan: Learning a generative model from a single natural image supplementary material," 2019.

[43] A. Défossez and I. Jordal, "Julius: Fast pytorch based dsp for audio and 1d signals," Oct 2021.

[44] J. O. Smith, *Digital Audio Resampling Home Page*. January 28, 2002.

[45] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com.

[46] "Sample rates - audacity development manual." https://alphamanual.audacityteam.org/man/Sample_Rates, Jun 2018.

[47] M. Pasini, "10 lessons I learnedt raining gans for a year." https://towardsdatascience.comlessons-i-/10-learned-training-generative-adversarial-networks-gans-for-a-year-c9071159628, Jul 2019..

[48] "Receptive field calculator - fomoro ai," https://web.archive.org/web/20200102073735/https://fomoro.com/research/article/receptive-field-calculator, May 2019.

[49] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[50] L. Elden, *Matrix Methods in Data Mining and Pattern Recognition*. Philadelphia, PA, USA: SIAM-Society for Industrial and Applied Mathematics, 2nd ed., 2019.

[51] R. Kumar, K. Kumar, V. Anand, Y. Bengio, and A. Courville, "Nu-gan: High resolution neural upsampling with gan," *arXiv preprint arXiv:2010.11362*, 2020.

[52] "Cornell bird calli dentification." https://www.kaggle.com/c/birdsong-recognition, 2020.

[53] T. Hinz, M. Fisher, O. Wang, and S. Wermter, "Improved techniques for training single-image gans," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 1300–1309, 2021.

Appendix A: SinGAN vs AudioSinGAN Code

The following code samples are an overview of the changes made to convert SinGAN to AudioSinGAN. In some cases they are an entire python file and in other cases they are samples of relevant code. Red highlighted code is SinGAN code we removed and green highlighted code is code we added. Code that isn't highlighted is code that there was no need to chage.

Listing A.1 main_train.py

```
from SinGAN.manipulate import *
from SinGAN.training import *
import SinGAN.functions as functions
import torchaudio
import os
import sys
from datetime import datetime
from SinGAN.AudioSample import AudioSample


class my_Logger(object):
    def __init__(self):
        self.console_out = sys.stdout
        self.file = open("logFile.txt", "a")
        self.encoding = "UTF-8"

    def write(self, input):
        self.console_out.write(input)
        self.file.write(input)

    def close(self):
        self.file.close()

    def flush(self):
        pass

if __name__ == '__main__':

    sys.stdout=my_Logger()

    print(str(sys.argv))

    parser = get_arguments()
    parser.add_argument('--input_dir', help='input image dir', default='Input/Images')
    parser.add_argument('--input_dir', help='input image dir',
                        default='Input/Audio') # Changed by Levi Pfantz 10/14/2020
    parser.add_argument('--input_name', help='input_image_name', required=True)
    parser.add_argument('--mode', help='task_to_be_done', default='train')


    opt = parser.parse_args()
```

```
42      opt = functions.post_config(opt)
43      opt.batch_norm=bool(opt.batch_norm)
44      opt.SR_pyr=[800, 1600, 2150, 2850, 3825, 5100, 6750, 9000, 12000, 16000]
45
46      opt.ker_size_pyr = [1205, 3015, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 301, 201]
47
48      if opt.single_level ¿0:
49          opt.SR_pyr=[opt.single_level]
50
51
52
53      Gs = []
54      Zs = []
55      reals = []
56      NoiseAmp = []
57      dir2save = functions.generate_dir2save(opt)
58
59      if (os.path.exists(dir2save)):
60          print('trained model already exist')
61      else:
62          try:
63              os.makedirs(dir2save)
64          except OSError:
65              pass
66          real = functions.read_image(opt)
67          functions.adjust_scales2image(real, opt)
68          train(opt, Gs, Zs, reals, NoiseAmp)
69          SinGAN_generate(Gs,Zs,reals,NoiseAmp,opt)
70
71      if not opt.not_cuda:
72          torch.set_default_tensor_type(torch.cuda.FloatTensor)
73
74      if int(opt.wandb) ¿ 0:
75          import wandb
76          wandb.init(project=opt.wandb_proj)
77
78      torchaudio.set_audio_backend(opt.audio_backend)
79      inputpath = opt.input_dir + "/" + opt.input_name
80
81
82
83  + real = AudioSample(opt, inputpath, sr=16000)
84      opt.stride=int(opt.stride)
85      opt.nfc=int(opt.nfc)
86      opt.min_nfc=int(opt.min_nfc)
87      print(real.data.shape)
88      functions.adjust_scales2data(real, opt)
89      opt.stop_scale=len(opt.SR_pyr)-1
90      print(real.data.shape)
91      train(opt, real, Gs, Zs, reals, NoiseAmp)
92      SinGAN_generate(Gs, Zs, reals, NoiseAmp, opt)
93      my_Logger.close()
```

Listing A.2 resume_at.py

```
# I added this file. When I made it I started with main_train as a base
# so they have a lot of overlap.

from config import get_arguments
from SinGAN.manipulate import *
from SinGAN.training import *
import SinGAN.functions as functions
import torchaudio
import sys
from datetime import datetime
from SinGAN.AudioSample import AudioSample

class my_Logger(object):
    def __init__(self):
        self.console_out = sys.stdout
        self.file = open("logFile.txt", "a")
        self.encoding = "UTF-8"

    def write(self, input):
        self.console_out.write(input)
        self.file.write(input)

    def close(self):
        self.file.close()

    def flush(self):
        pass

if __name__ == '__main__':

    sys.stdout=my_Logger()

    print(str(sys.argv))

    parser = get_arguments()
    parser.add_argument('--input_dir', help='input_image_dir',default='Input/Audio')
    parser.add_argument('--input_name', help='input_image_name', required=True)
    parser.add_argument('--mode', help='task_to_be_done', default='train')

    opt = parser.parse_args()
    opt = functions.post_config(opt)
    opt.SR_pyr=[800, 1600, 2150, 2850, 3825, 5100, 6750, 9000, 12000, 16000]

    if opt.single_level >0:
        opt.SR_pyr=[opt.single_level]


    Gs = []
    Zs = []
    reals = []
    NoiseAmp = []
    dir2save = functions.generate_dir2save(opt)


    if not opt.not_cuda:
        torch.set_default_tensor_type(torch.cuda.FloatTensor)
```

```
57
58      if int(opt.wandb) > 0:
59          import wandb
60          wandb.init(project=opt.wandb_proj)
61
62
63      torchaudio.set_audio_backend(opt.audio_backend)
64      inputpath = opt.input_dir + "/" + opt.input_name
65
66
67      real = AudioSample(opt, inputpath, sr=16000)
68      opt.stride=int(opt.stride)
69      opt.nfc=int(opt.nfc)
70      opt.min_nfc=int(opt.min_nfc)
71      print(real.data.shape)
72      functions.adjust_scales2data(real, opt)
73      opt.stop_scale=len(opt.SR_pyr)-1
74      print(real.data.shape)
75      Gs, Zs, reals_trash, NoiseAmp = functions.load_trained_pyramid(opt)
76      del reals_trash
77      reals=[]
78      reals = functions.creat_reals_pyramid(real, reals, opt, verbose=False)
79
80      opt.mode = 'train'
81      Gs=Gs[0:opt.level_to_resume_at]
82      Zs=Zs[0:opt.level_to_resume_at]
83      NoiseAmp=NoiseAmp[0:opt.level_to_resume_at]
84      in_s = torch.full([1, reals[0].shape[0], reals[0].shape[2]], 0, dtype=torch.float32, device=opt.device)
85      if opt.level_to_resume_at < len(opt.SR_pyr):
86          train_on_audio_resume(opt, real, Gs, Zs, reals, NoiseAmp, in_s)
87      SinGAN_generate(Gs, Zs, reals, NoiseAmp, opt)
88      my_Logger.close()
```

Listing A.3 config.py

```
1       parser.add_argument('--netG', default='', help="path_to_netG_(to_continue_training)")
2       parser.add_argument('--netD', default='', help="path_to_netD_(to_continue_training)")
3       parser.add_argument('--manualSeed', type=int, help='manual_seed')
4       parser.add_argument('-nc_z',type=int,help='noise # channels',default=3)
5       parser.add_argument('-nc_im',type=int,help='image # channels',default=3)
6       #Audio noise channles will be the same as audio channels
7       parser.add_argument('-nc_z',type=int,help='noise # channels',default=1)
8       #At least initally we'll be working with 1 channel audio
9       parser.add_argument('-nc_aud',type=int,help='image # channels',default=1)
10      parser.add_argument('--out',help='output_folder',default='Output')
11
12      \#networks hyper parameters:
13      parser.add\argument('--nfc', type=int, default=32)
14      #parser.add_argument('-batch_norm', action='store_true', help='use batch normalization (not yet implemented)', default=0)
15      parser.add_argument('-batch_norm', type=int, help='use batch normalization (not yet implemented)', default=0)
16      parser.add\_argument('--min\_nfc', type=int, default=32)
17      parser.add_argument('-ker_size',type=int,help='kernel size',default=3)
18      # SinGAN's 3x3 Kernel is flattened to a 9 kernel
19      parser.add_argument('-ker_size',type=int,help='kernel size',default=9)
20      parser.add_argument('--num_layer',type=int,help='number_of_layers',default=5)
21      parser.add_argument('--stride',help='stride',default=1)
```

```
22  parser.add_argument('--padd_size',type=int,help='net pad size',default=0) #math.floor(opt.ker_size/2)
23
24  parser.add_argument('–dilation', type=int, help='dilation at each layer', default=1)
25  parser.add_argument('–RELU_in_gen', type=int, help='Use RELU instead of leaky RELU in the generator', default=0)
26
27
28  #pyramid parameters:
29  parser.add_argument('--scale_factor',type=float,help='pyramid scale factor',default=0.75) #pow(0.5,1/6))
30  parser.add_argument('--noise_amp',type=float,help='addative noise cont weight',default=0.1)
31  \#optimization hyper parameters:
32  parser.add_argument('--niter', type=int, default=2000, help='number of epochs to train per scale')
33  parser.add_argument('--gamma',type=float,help='scheduler gamma',default=0.1)
34  parser.add_argument('–lr_g', type=float, default=0.0005, help='learning rate, default=0.0005')
35  parser.add_argument('–lr_d', type=float, default=0.0005, help='learning rate, default=0.0005')
36  parser.add_argument('–lr_g', type=float, default=0.0001, help='learning rate, default=0.0005')
37  parser.add_argument('–lr_d', type=float, default=0.0004, help='learning rate, default=0.0005')
38  parser.add_argument('--beta1', type=float, default=0.5, help='beta1 for adam. default=0.5')
39  parser.add_argument('–Gsteps',type=int, help='Generator inner steps',default=3)
40  parser.add_argument('–Dsteps',type=int, help='Discriminator inner steps',default=3)
41  parser.add_argument('–steps',type=int, help='Generator inner steps',default=3)
42  parser.add_argument('--lambda_grad',type=float, help='gradient penelty weight',default=0.1)
43  parser.add_argument('–alpha',type=float, help='reconstruction loss weight',default=10)
44  parser.add_argument('–alpha',type=float, help='reconstruction loss weight',default=4)
45  parser.add_argument('–dropout', type=float, help='dropout for discriminator', default=0)
46
47  #added by Levi
48  #use soundfile for windows and sox io for linux / google colab
49  parser.add_argument('–audio_backend', help='normalize audio input?', default='sox_io')
50  parser.add_argument('–norm', help='normalize audio input?', default=1)
51  parser.add_argument('–wandb', help='log stuff with wandb, you need to init before hand', default=0)
52  parser.add_argument('–wandb_proj', help='wandb Project name', default='AudioSinGAN')
53  parser.add_argument('–update_every_x', help='only update discriminator every x steps', default=0)
54  parser.add_argument('–steps_to_update', help='how many steps to update on', default=10)
55  parser.add_argument('–normalize_generator_output', type = int, help='In this experiment I normalize the output of the '
56                                                                      'generator (before loss is calculated) ', default=0)
57  parser.add_argument('–normalize_before_saving', type = int, help='normalize audio output before saving as a file', default=1)
58  parser.add_argument('–use_MAE', type = int, help='use MAE instead of LSE', default=0)
59  parser.add_argument('–update_only_with_lower_Gloss', type = int, help='update discriminator only when GLoss is '
60                                                                        'decreasing', default=0)
61  parser.add_argument('–use_schedulers', type = int, help='use DScheduler to decrease lr after 1600 epochs', default=0)
62  parser.add_argument('–make_input_tensor_even', type=int, help='make input tensor even', default=1)
63  parser.add_argument('–adjust_upsampled', type=int, help='adjust generated tensor size as they are up-sampled', default=1)
64  parser.add_argument('–change_channel_count', type=int, help='at higher levels of GAN pyramid use more channels', default=0)
65  parser.add_argument('–adjust_after_levels', type=int, help='adjust lr d after level', default=0)
66  parser.add_argument('–level_to_resume_at', type=int, help='level to resume training at', default=1)
67  parser.add_argument('–alt_pyramid_exp', type=int, help='instead of resampling audio, change kernel size at each '
68                                                         'layer of the pyramid', default=0)
69  parser.add_argument('–pad_with_noise', type=int, help='pad with noise instead of 0s', default=0)
70  parser.add_argument('–single_level', type=int, help='use only a single layer of size x', default=-1)
71  parser.add_argument('–save_fake_progression', type=int, help='use this to save a copy of the fake output every 500 epochs', default=-1)
72  parser.add_argument('–smooth_real_labels', type=int, help='smooth the real labels of the discriminator', default=-1)
73  parser.add_argument('–smooth_fake_labels', type=int, help='smooth the fake labels of the discriminator', default=-1)
74
```

```
75      return parser
```

Listing A.4 training.py

```
1   import torch.utils.data
2   import math
3   import wandb
4   import matplotlib.pyplot as plt
5   from SinGAN.AudioSample import AudioSample
6   from SinGAN.imresize import imresize
7
8   def train(opt,Gs,Zs,reals,NoiseAmp):
9       real_ = functions.read_image(opt)
10
11  def train(opt, real, Gs, Zs, reals, NoiseAmp):
12      in_s = 0
13      scale_num = 0
14      real = imresize(real_,opt.scale1,opt)
15      reals = functions.creat_reals_pyramid(real,reals,opt)
16
17      # real.resample_by(opt.scale1)
18      real.resample_to_julius(opt.SR_pyr[-1])
19      print("new shape", real.data.shape)
20
21      # reals, sr_list = functions.creat_reals_pyramid(real, reals, opt, verbose=False)
22      reals = functions.creat_reals_pyramid(real, reals, opt, verbose=False)
23      sr_list = opt.SR_pyr
24      nfc_prev = 0
25
26      while scale_num¡opt.stop_scale+1:
27          opt.nfc = min(opt.nfc_init * pow(2, math.floor(scale_num / 4)), 128)
28          opt.min_nfc = min(opt.min_nfc_init * pow(2, math.floor(scale_num / 4)), 128)
29      for x in reals:
30          print(x.device)
31
32      while scale_num ¡ opt.stop_scale + 1:
33          if opt.change_channel_count ¿ 0:
34              opt.nfc = min(opt.nfc_init * pow(2, math.floor(scale_num / 4)), 128)
35              opt.min_nfc = min(opt.min_nfc_init * pow(2, math.floor(scale_num / 4)), 128)
36
37          opt.out_ = functions.generate_dir2save(opt)
38          opt.outf = '%s/%d' (opt.out_,scale_num)}
39
40          try:
41              os.makedirs(opt.outf)
42          except OSError:
43                  pass
44              pass
45
46          # plt.imsave('%s/in.png' % (opt.out_), functions.convert_image_np(real), vmin=0, vmax=1)
47          # plt.imsave('%s/original.png' % (opt.out_), functions.convert_image_np(real_), vmin=0, vmax=1)
48          # plt.imsave('%s/real_scale.png' % (opt.outf), functions.convert_image_np(reals[scale_num]), vmin=0, vmax=1)
49          if opt.alt_pyramid_exp ¿ 0:
50              level=len(Gs)
```

```
51                  opt.ker_size=opt.ker_size_pyr[level]
52
53              D_curr, G_curr = init_models(opt)
54              if (nfc_prev == opt.nfc and opt.alt_pyramid_exp == 0):
55                  G_curr.load_state_dict(torch.load('%s/%d/netG.pth' % (opt.out_, scale_num - 1)))
56                  D_curr.load_state_dict(torch.load('%s/%d/netD.pth' % (opt.out_, scale_num - 1)))
57
58              z_curr, in_s, G_curr = train_single_scale(D_curr, G_curr, reals, sr_list, Gs, Zs, in_s, NoiseAmp, opt)
59
60              G_curr = functions.reset_grads(G_curr, False)
61              G_curr.eval()
62              D_curr = functions.reset_grads(D_curr, False)
63              D_curr.eval()
64
65              Gs.append(G_curr)
66              Zs.append(z_curr)
67              NoiseAmp.append(opt.noise_amp)
68
69              torch.save(Zs, '%s/Zs.pth' % (opt.out_))
70              torch.save(Gs, '%s/Gs.pth' % (opt.out_))
71              torch.save(reals, '%s/reals.pth' % (opt.out_))
72              torch.save(NoiseAmp, '%s/NoiseAmp.pth' % (opt.out_))
73
74              scale_num += 1
75              nfc_prev = opt.nfc
76              del D_curr, G_curr
77          return
78
79      def train_on_audio_resume(opt, real, Gs, Zs, reals, NoiseAmp, in_s):
80          scale_num = opt.level_to_resume_at
81
82          # real.resample_by(opt.scale1)
83          #real.resample_to_julius(opt.SR_pyr[-1])
84          #print("new shape", real.data.shape)
85
86          # reals, sr_list = functions.creat_reals_pyramid(real, reals, opt, verbose=False)
87          #reals = functions.creat_reals_pyramid(real, reals, opt, verbose=False)
88          sr_list = opt.SR_pyr
89          nfc_prev = 0
90
91          for x in reals:
92              print(x.device)
93
94          while scale_num ¡ opt.stop_scale + 1:
95              if opt.change_channel_count ¿ 0:
96                  opt.nfc = min(opt.nfc_init * pow(2, math.floor(scale_num / 4)), 128)
97                  opt.min_nfc = min(opt.min_nfc_init * pow(2, math.floor(scale_num / 4)), 128)
98
99              opt.out_ = functions.generate_dir2save(opt)
100             opt.outf = '%s/%d' % (opt.out_, scale_num)
101
102             #plt.imsave('%s/in.png' % (opt.out_), functions.convert_image_np(real), vmin=0, vmax=1)
103             #plt.imsave('%s/original.png' % (opt.out_), functions.convert_image_np(real_), vmin=0, vmax=1)
```

```
104             plt.imsave('%s/real_scale.png' % (opt.outf), functions.convert_image_np(reals[scale_num]), vmin=0, vmax=1)
105             try:
106                 os.makedirs(opt.outf)
107             except OSError:
108                 pass
109
110             D_curr,G_curr = init_models(opt)
111             if (nfc_prev==opt.nfc):
112                 G_curr.load_state_dict(torch.load('%s/%d/netG.pth' % (opt.out_,scale_num-1)))
113                 D_curr.load_state_dict(torch.load('%s/%d/netD.pth' % (opt.out_,scale_num-1)))
114             # plt.imsave('%s/in.png' % (opt.out_), functions.convert_image_np(real), vmin=0, vmax=1)
115             # plt.imsave('%s/original.png' % (opt.out_), functions.convert_image_np(real_), vmin=0, vmax=1)
116             # plt.imsave('%s/real_scale.png' % (opt.outf), functions.convert_image_np(reals[scale_num]), vmin=0, vmax=1)
117
118             z_curr,in_s,G_curr = train_single_scale(D_curr,G_curr,reals,Gs,Zs,in_s,NoiseAmp,opt)
119             D_curr, G_curr = init_models(opt)
120             #TODO: This next line doesn't work for the first scale on resume if change_channel_count is ¿ 0
121             if (nfc_prev == opt.nfc) or opt.change_channel_count==0:
122                 G_curr.load_state_dict(torch.load('%s/%d/netG.pth' % (opt.out_, scale_num - 1)))
123                 D_curr.load_state_dict(torch.load('%s/%d/netD.pth' % (opt.out_, scale_num - 1)))
124
125             G_curr = functions.reset_grads(G_curr,False)
126             z_curr, in_s, G_curr = train_single_scale(D_curr, G_curr, reals, sr_list, Gs, Zs, in_s, NoiseAmp, opt)
127
128             G_curr = functions.reset_grads(G_curr, False)
129             G_curr.eval()
130             D_curr = functions.reset_grads(D_curr,False)
131             D_curr = functions.reset_grads(D_curr, False)
132             D_curr.eval()
133
134             Gs.append(G_curr)
135             Zs.append(z_curr)
136             NoiseAmp.append(opt.noise_amp)
137             torch.save(Zs, '%s/Zs.pth' % (opt.out_))
138             torch.save(Gs, '%s/Gs.pth' % (opt.out_))
139             torch.save(reals, '%s/reals.pth' % (opt.out_))
140             torch.save(NoiseAmp, '%s/NoiseAmp.pth' % (opt.out_))
141
142             scale_num+=1
143             scale_num += 1
144             nfc_prev = opt.nfc
145             del D_curr,G_curr
146             del D_curr, G_curr
147         return
148
149     # On Laptop 100 epochs at first scale takes about 0:49 on debugger. Second scale: On 100 epochs at about 4:45
150     def train_single_scale(netD, netG, reals, sr_list, Gs, Zs, in_s, NoiseAmp, opt, centers=None):
151         curr_scale = len(Gs)
152         if curr_scale == 0:
153             print("It's 0!")
154         if curr_scale == 1:
155             print("It's 1!")
156         if curr_scale == 2:
157             print("It's 2!")
```

```
158
159         update_count = 1
160
161         curr_sr = sr_list[curr_scale]
162         real = reals[curr_scale]
163         opt.nzx = real.shape[2] # +(opt.ker_size-1)*(opt.num_layer)
164
165         opt.nz = real.shape[0]
166         opt.receptive_field = int(opt.ker_size) + ((int(opt.ker_size) - 1) * (int(opt.num_layer) - 1)) * int(opt.stride)
167
168         # We aren't going to worry about padding. It's easier to just make it 0 (i.e. do nothing) then
169         # completely remove it
170         # pad_noise = 0 #int(((opt.ker_size - 1) * opt.num_layer) / 2)
171         # pad_image = 0 #int(((opt.ker_size - 1) * opt.num_layer) / 2)
172
173
174 def train_single_scale(netD,netG,reals,Gs,Zs,in_s,NoiseAmp,opt,centers=None):
175         pad_num = int(((opt.ker_size - 1) * opt.dilation * opt.num_layer * (1/2))
176
177         real = reals[len(Gs)]
178         opt.nzx = real.shape[2]#+(opt.ker_size-1)*(opt.num_layer)
179         opt.nzy = real.shape[3]#+(opt.ker_size-1)*(opt.num_layer)
180         opt.receptive_field = opt.ker_size + ((opt.ker_size-1)*(opt.num_layer-1))*opt.stride
181         pad_noise = int(((opt.ker_size - 1) * opt.num_layer) / 2)
182         pad_image = int(((opt.ker_size - 1) * opt.num_layer) / 2)
183         if opt.mode == 'animation_train':
184             opt.nzx = real.shape[2]+(opt.ker_size-1)*(opt.num_layer)
185             opt.nzy = real.shape[3]+(opt.ker_size-1)*(opt.num_layer)
186             pad_noise = 0
187         m_noise = nn.ZeroPad2d(int(pad_noise))
188         m_image = nn.ZeroPad2d(int(pad_image))
189
190
191
192 + # Was an if then for animation train here I removed
193
194         alpha = opt.alpha
195
196         fixed_noise = functions.generate_noise([opt.nc_z,opt.nzx,opt.nzy],device=opt.device)
197         z_opt = torch.full(fixed_noise.shape, 0, device=opt.device)
198         z_opt = m_noise(z_opt)
199         fixed_noise = functions.generate_noise([opt.nc_z, opt.nzx], device=opt.device)
200         # This next line updated to bring it inline with later versions of pytorch by
201         # Levi Pfantz on 10/20/2020
202         z_opt = torch.full(fixed_noise.shape, 0, dtype=torch.float32, device=opt.device)
203         z_opt = AudioSample.static_pad(z_opt, pad_num, opt)
204
205         # setup optimizer
206         if len(Gs) == 3 and opt.adjust_after_levels ¿ 0:
207             opt.lr_d = 0.0005
208         if len(Gs) == 5 and opt.adjust_after_levels ¿ 0:
209             opt.lr_d = 0.0004
210         if len(Gs) == 7 and opt.adjust_after_levels ¿ 0:
211             opt.lr_d = 0.0003
```

```
212     if len(Gs) == 9 and opt.adjust_after_levels ¿ 0:
213         opt.lr_d = 0.0002
214     if len(Gs) == 11 and opt.adjust_after_levels ¿ 0:
215         opt.lr_d = 0.0001
216
217     optimizerD = optim.Adam(netD.parameters(), lr=opt.lr_d, betas=(opt.beta1, 0.999))
218     optimizerG = optim.Adam(netG.parameters(), lr=opt.lr_g, betas=(opt.beta1, 0.999))
219     schedulerD = torch.optim.lr_scheduler.MultiStepLR(optimizer=optimizerD,milestones=[1600],gamma=opt.gamma)
220     schedulerG = torch.optim.lr_scheduler.MultiStepLR(optimizer=optimizerG,milestones=[1600],gamma=opt.gamma)
221     schedulerD = torch.optim.lr_scheduler.MultiStepLR(optimizer=optimizerD, milestones=[1600], gamma=opt.gamma)
222     schedulerG = torch.optim.lr_scheduler.MultiStepLR(optimizer=optimizerG, milestones=[1600], gamma=opt.gamma)
223
224     errD2plot = []
225     errG2plot = []
226     D\_real2plot = []
227     D\_fake2plot = []
228     z\_opt2plot = []
229
230
231     final_GLoss_last = 100
232     final_GLoss = 0
233
234     # On Desktop cpu 100 epochs takes about 2:17
235     # on 300 epochs in about 14:45
236     for epoch in range(opt.niter):
237         # If this is the first epoch of the first level then z_opt needs to be defined?
238         if (Gs == []) & (opt.mode != 'SR_train'):
239             z_opt = functions.generate_noise([1,opt.nzx,opt.nzy], device=opt.device)
240             z_opt = m_noise(z_opt.expand(1,3,opt.nzx,opt.nzy))
241             noise_ = functions.generate_noise([1,opt.nzx,opt.nzy], device=opt.device)
242             noise_ = m_noise(noise_.expand(1,3,opt.nzx,opt.nzy))
243             z_opt = functions.generate_noise([1, opt.nzx], device=opt.device)
244             z_opt = z_opt.expand(1, 1, opt.nzx)
245             z_opt = AudioSample.static_pad(z_opt, pad_num, opt)
246             noise_ = functions.generate_noise([1, opt.nzx], device=opt.device)
247             noise_ = noise_.expand(1, 1, opt.nzx)
248             noise_ = AudioSample.static_pad(noise_, pad_num, opt)
249         else:
250             noise_ = functions.generate_noise([opt.nc_z,opt.nzx,opt.nzy], device=opt.device)
251             noise_ = m_noise(noise_)
252             noise_ = functions.generate_noise([opt.nc_z, opt.nzx], device=opt.device)
253             noise_ = AudioSample.static_pad(noise_, pad_num, opt)
254
255             ############################
256             # (2) Update G network: maximize D(G(z))
257             ###########################
258         for j in range(opt.Dsteps):
259         for j in range(opt.steps):
260             # train with real
261             netD.zero_grad()
262
263             # Very First time on Laptop takes:
264             output = netD(real).to(opt.device)
265             #D_real_map = output.detach()
```

```
266                 errD_real = -output.mean()#-a
267                 # D_real_map = output.detach()
268                 errD_real = -output.mean() # -a
269                 if opt.smooth_real_labels ¿ 0:
270                     print("smooth")
271                     real_mod=((torch.rand(1)*0.5)+0.7)
272                     errD_real = errD_real*real_mod
273             errD_real.backward(retain_graph=True)
274             D_x = -errD_real.item()
275
276             # train with fake
277             if (j==0) (epoch == 0):
278             # If this is the first step of the first epoch
279             if (j == 0) (epoch == 0):
280                 # If this is the first level
281                 if (Gs == []) & (opt.mode != 'SR_train'):
282                     prev = torch.full([1,opt.nc_z,opt.nzx,opt.nzy], 0, device=opt.device)
283                     # We need to setup a previous of random noise since there wasn't a previous level do derive from
284                     prev = torch.full([1, opt.nc_z, opt.nzx], 0, dtype=torch.float32, device=opt.device)
285                     in_s = prev
286                     prev = m_image(prev)
287                     z_prev = torch.full([1,opt.nc_z,opt.nzx,opt.nzy], 0, device=opt.device)
288                     z_prev = m_noise(z_prev)
289                     prev = AudioSample.static_pad(prev, pad_num, opt)
290                     # This next line updated to bring it inline with later versions of pytorch by
291                     # Levi Pfantz on 10/20/2020
292                     z_prev = torch.full([1, opt.nc_z, opt.nzx], 0, dtype=torch.float32, device=opt.device)
293                     z_prev = AudioSample.static_pad(z_prev, pad_num, opt)
294                     opt.noise_amp = 1
295                 elif opt.mode == 'SR_train':
296                     z_prev = in_s
297                     criterion = nn.MSELoss()
298                     RMSE = torch.sqrt(criterion(real, z_prev))
299                     opt.noise_amp = opt.noise_amp_init * RMSE
300                     z_prev = m_image(z_prev)
301                     prev = z_prev
302
303                 # removed an if else for SR
304
305                 else:
306                     prev = draw_concat(Gs,Zs,reals,NoiseAmp,in_s,'rand',m_noise,m_image,opt)
307                     prev = m_image(prev)
308                     z_prev = draw_concat(Gs,Zs,reals,NoiseAmp,in_s,'rec',m_noise,m_image,opt)
309                     # Because SinGAN calculates two different losses (adversarial term and reconstruction)
310                     # We will generate two new images with the past scale's generator and concat them with noise
311                     # The result is prev for adversarial and z_prev for reconstruction
312                     prev = draw_concat(Gs, Zs, reals, sr_list, NoiseAmp, in_s, 'rand',
313                                        opt) # Possible issue in draw concat audio
314                     prev = AudioSample.static_pad(prev, pad_num, opt)
315                     z_prev = draw_concat(Gs, Zs, reals, sr_list, NoiseAmp, in_s, 'rec', opt)
316
317                 criterion = nn.MSELoss()
318                 # RMSE = Root mean squared error
```

```
319                     # Calculate RMSE. We'll use it when we calculate the reconstruction loss
320                     RMSE = torch.sqrt(criterion(real, z_prev))
321                     opt.noise_amp = opt.noise_amp_init*RMSE
322                     z_prev = m_image(z_prev)
323                     opt.noise_amp = opt.noise_amp_init * RMSE
324                     z_prev = AudioSample.static_pad(z_prev, pad_num, opt)
325                 else:
326                     prev = draw_concat(Gs,Zs,reals,NoiseAmp,in_s,'rand',m_noise,m_image,opt)
327                     prev = m_image(prev)
328                     prev = draw_concat(Gs, Zs, reals, sr_list, NoiseAmp, in_s, 'rand', opt)
329                     prev = AudioSample.static_pad(prev, pad_num, opt)
330
331                 if opt.mode == 'paint_train':
332                     prev = functions.quant2centers(prev,centers)
333                     plt.imsave('%s/prev.png' % (opt.outf), functions.convert_image_np(prev), vmin=0, vmax=1)
334                 # removed an if for paint train
335
336                 if (Gs == []) & (opt.mode != 'SR_train'):
337                     noise = noise_
338                 else:
339                     noise = opt.noise_amp*noise_+prev
340                     noise = opt.noise_amp * noise_ + prev
341
342                 fake = netG(noise.detach(),prev)
343                 fake = netG(noise.detach(), prev)
344                 if opt.normalize_generator_output ¿ 0:
345                     prenorm_max=torch.max(fake)
346                     prenorm_min=torch.min(fake)
347                     fake = AudioSample.static_normalize(fake)
348                 output = netD(fake.detach())
349                 errD_fake = output.mean()
350                 if opt.smooth_fake_labels ¿ 0:
351                     fake_mod=((torch.rand(1)*0.5)+0.7)
352                     errD_fake=errD_fake*fake_mod
353                 errD\fake.backward(retain_graph=True)
354                 D_G_z = output.mean().item()
355
356                 gradient_penalty = functions.calc_gradient_penalty(netD, real, fake, opt.lambda_grad, opt.device)
357                 gradient_penalty.backward()
358
359                 errD = errD_real + errD_fake + gradient_penalty
360                 optimizerD.step()
361                 temp = errD
362
363             errD2plot.append(errD.detach())
364                 if opt.update_only_with_lower_Gloss == 0 or final_GLoss_last ¿= final_GLoss:
365                     if update_count % int(opt.steps_to_update) == 0 or int(opt.update_every_x) == 0:
366                         optimizerD.step()
367                     update_count += 1
368
369             ############################
370             # (2) Update G network: maximize D(G(z))
371             ###########################
372                 ###########################
```

```
373             # (2) Update G network: maximize D(G(z))
374             ###########################
375
376         for j in range(opt.Gsteps):
377             netG.zero_grad()
378
379             output = netD(fake)
380             #D_fake_map = output.detach()
381             # D_fake_map = output.detach()
382             errG = -output.mean()
383             errG.backward(retain_graph=True)
384             if alpha!=0:
385                 loss = nn.MSELoss()
386                 if opt.mode == 'paint_train':
387                     z_prev = functions.quant2centers(z_prev, centers)
388                     plt.imsave('%s/z_prev.png' % (opt.outf), functions.convert_image_np(z_prev), vmin=0, vmax=1)
389                 Z_opt = opt.noise_amp*z_opt+z_prev
390                 rec_loss = alpha*loss(netG(Z_opt.detach(),z_prev),real)
391             if alpha != 0:
392                 if opt.use_MAE ¿ 0:
393                     loss = nn.L1Loss()
394                 else:
395                     loss = nn.MSELoss()
396
397                 # if then for paint_train removed
398
399                 Z_opt = opt.noise_amp * z_opt + z_prev
400                 rec_loss = alpha * loss(netG(Z_opt.detach(), z_prev), real)
401                 rec_loss.backward(retain_graph=True)
402                 rec_loss = rec_loss.detach()
403             else:
404                 Z_opt = z_opt
405                 rec_loss = 0
406
407             if opt.update_only_with_lower_Gloss ¿ 0:
408                 if update_count ¿ 2:
409                     final_GLoss_last=final_GLoss
410                 final_GLoss=errG.detach() + rec_loss
411
412
413             optimizerG.step()
414
415         errG2plot.append(errG.detach()+rec_loss)
416         errD2plot.append(errD.detach())
417         errG2plot.append(errG.detach() + rec_loss)
418         D_real2plot.append(D_x)
419         D_fake2plot.append(D_G_z)
420         z_opt2plot.append(rec_loss)
421
422         if epoch % 25 == 0 or epoch == (opt.niter-1):
423         if int(opt.wandb) ¿ 0:
424             if opt.normalize_generator_output¿0:
425                 wandb.log(("prenom max at scale: " + str(len(Gs))): prenorm_max)
426                 wandb.log(("prenom min at scale: " + str(len(Gs))): prenorm_min)
```

```
427                 wandb.log("errD at scale: " + str(len(Gs))): errD.detach())
428                 wandb.log("errG at scale: " + str(len(Gs))): errG.detach() + rec_loss)
429                 wandb.log("D_real2plot at scale: " + str(len(Gs))): D_x)
430                 wandb.log("D_fake2plot at scale: " + str(len(Gs))): D_G_z)
431                 wandb.log("z_opt2pot at scale: " + str(len(Gs))): rec_loss)
432
433             if epoch % 25 == 0 or epoch == (opt.niter - 1):
434                 print('scale %d:[%d%d]' % (len(Gs), epoch, opt.niter))
435
436             if epoch % 500 == 0 or epoch == (opt.niter-1):
437                 plt.imsave('%s/fake_sample.png' % (opt.outf), functions.convert_image_np(fake.detach()), vmin=0, vmax=1)
438                 plt.imsave('%s/G(z_opt).png' % (opt.outf), functions.convert_image_np(netG(Z_opt.detach(), z_prev).detach()), vmin=0, vmax=1)
439                 #plt.imsave('%s/D_fake.png' % (opt.outf), functions.convert_image_np(D_fake_map))
440                 #plt.imsave('%s/D_real.png' % (opt.outf), functions.convert_image_np(D_real_map))
441                 #plt.imsave('%s/z_opt.png' % (opt.outf), functions.convert_image_np(z_opt.detach()), vmin=0, vmax=1)
442                 #plt.imsave('%s/prev.png' % (opt.outf), functions.convert_image_np(prev), vmin=0, vmax=1)
443                 #plt.imsave('%s/noise.png' % (opt.outf), functions.convert_image_np(noise), vmin=0, vmax=1)
444                 #plt.imsave('%s/z_prev.png' % (opt.outf), functions.convert_image_np(z_prev), vmin=0, vmax=1)
445
446
447             if epoch % 500 == 0 or epoch == (opt.niter - 1):
448             if epoch % 25 == 0 or epoch == (opt.niter - 1):
449                 # plt.imsave('%s/fake_sample.png' % (opt.outf), functions.convert_image_np(fake.detach()), vmin=0, vmax=1)
450                 z_opt_to_save=netG(Z_opt.detach, z_prev).detach
451
452                 if opt.normalize_before_saving ¿ 1:
453                     fake=AudioSample.static_normalize(fake)
454                     z_opt_to_save = AudioSample.static_normalize(z_opt_to_save)
455
456                 if opt.save_fake_progression ¿ 0:
457                     if epoch==0 and not os.path.exists('%s/savedProgression/' % (opt.outf)):
458                         os.makedirs('%s/savedProgression/' % (opt.outf))
459                     string=opt.outf+'/savedProgression/'+str(epoch)+'.wav'
460                     AudioSample.static_save(fake.detach(), curr_sr, string)
461
462                 AudioSample.static_save(fake.detach(), curr_sr, '%s/fake_sample.wav' % (opt.outf))
463                 # plt.imsave('%s/G(z_opt).png' % (opt.outf), functions.convert_image_np(netG(Z_opt.detach(), z_prev).detach()), vmin=0, vmax=1)
464                 AudioSample.static_save(z_opt_to_save, curr_sr, '%s/G(z_opt).wav' % (opt.outf))
465
466                 # plt.imsave('%s/D_fake.png' % (opt.outf), functions.convert_image_np(D_fake_map))
467                 # plt.imsave('%s/D_real.png' % (opt.outf), functions.convert_image_np(D_real_map))
468                 # plt.imsave('%s/z_opt.png' % (opt.outf), functions.convert_image_np(z_opt.detach()), vmin=0, vmax=1)
469                 # plt.imsave('%s/prev.png' % (opt.outf), functions.convert_image_np(prev), vmin=0, vmax=1)
470                 # plt.imsave('%s/noise.png' % (opt.outf), functions.convert_image_np(noise), vmin=0, vmax=1)
471                 # plt.imsave('%s/z_prev.png' % (opt.outf), functions.convert_image_np(z_prev), vmin=0, vmax=1)
472
473                 torch.save(z_opt, '\%s/z_opt.pth' \% (opt.outf))
474
475             schedulerD.step()
476             schedulerG.step()
477             if opt.use_schedulers ¿ 0:
478                 schedulerD.step()
479                 schedulerG.step()
```

```
480
481         functions.save_networks(netG, netD, z_opt, opt)
482         return z_opt, in_s, netG
483
484         functions.save_networks(netG,netD,z_opt,opt)
485         return z_opt,in_s,netG
486
487 def draw_concat(Gs,Zs,reals,NoiseAmp,in_s,mode,m_noise,m_image,opt):
488 def draw_concat(Gs, Zs, reals, sr_list, NoiseAmp, in_s, mode, opt):
489         # This function generates output from the previous scale that is up-scaled for the current scale
490         # It does this by starting at the bottom of the pyramid and working it's way up. At each level it
491         # uses the already trained generators to create output, upscale it and feed it to the next level.
492         # The first option rand (random presumably) generates it's own random noise maps at each level which
493         # should result in an entirely new, random image. The Next option reconstruction (presumably) uses
494         # the saved noise maps for each level which (if I understand correctly) allows for a recreation of the
495         # image previously used. It is what is used for calculating the reconstruction loss.
496         G_z = in_s
497
498         pad_num = int(((opt.ker_size - 1) * opt.num_layer) / 2) * opt.dilation
499
500         if len(Gs) > 0:
501             # This mode generates it's own noise
502             if mode == 'rand':
503                 count = 0
504                 pad_noise = int(((opt.ker_size-1)*opt.num_layer)/2)
505                 if opt.mode == 'animation\_train':
506                     pad_noise = 0
507                 for G,Z_opt,real_curr,real_next,noise_amp in zip(Gs,Zs,reals,reals[1:],NoiseAmp):
508                 for G, Z_opt, real_curr, real_next, noise_amp in zip(Gs, Zs, reals, reals[1:], NoiseAmp):
509
510                     if opt.alt_pyramid_exp ¿ 0:
511                         level = count
512                         calc_ker_size=opt.ker_size_pyr[level]
513                         pad_num = int(((opt.ker_size - 1) * opt.num_layer) / 2) * opt.dilation
514
515                     if count == 0:
516                         z = functions.generate_noise([1, Z_opt.shape[2] - 2 * pad_noise, Z_opt.shape[3] - 2 * pad_noise], device=opt.device)
517                         z = z.expand(1, 3, z.shape[2], z.shape[3])
518                         z = functions.generate_noise([1, Z_opt.shape[2] - 2 * pad_num], device=opt.device)
519                         z = z.expand(1, 1, z.shape[2])
520                     else:
521                         z = functions.generate_noise([opt.nc_z,Z_opt.shape[2] - 2 * pad_noise, Z_opt.shape[3] - 2 * pad_noise], device=opt.device)
522                     z = m_noise(z)
523                     G_z = G_z[:,:,0:real_curr.shape[2],0:real_curr.shape[3]]
524                     G_z = m_image(G_z)
525                     z_in = noise_amp*z+G_z
526                     G_z = G(z_in.detach(),G_z)
527                     G_z = imresize(G_z,1/opt.scale_factor,opt)
528                     G_z = G_z[:,:,0:real_next.shape[2],0:real_next.shape[3]]
529                         z = functions.generate_noise([opt.nc_z, Z_opt.shape[2] - 2 * pad_num], device=opt.device)
530                     z = AudioSample.static_pad(z, pad_num, opt)
531                     G_z = G_z[:, :, 0:real_curr.shape[2]]
532                     G_z = AudioSample.static_pad(G_z, pad_num, opt)
```

```
533                     z_in = noise_amp * z + G_z
534                     G_z = G(z_in.detach(), G_z)
535                     if opt.normalize_generator_output ¿0:
536                         G_z=AudioSample.static_normalize(G_z)
537                     G_z = AudioSample.resample_to_julius_static(G_z, sr_list[count], sr_list[count + 1])
538 
539                     if opt.adjust_upsampled ¿ 0:
540                         if G_z.shape[2] ¡ real_next.shape[2]:
541                             dif = real_next.shape[2] - G_z.shape[2]
542                             G_z = torch.cat((G_z, torch.zeros([1,1,dif], dtype=torch.float32)), dim=2)
543 
544                     # G_z = imresize(G_z, 1 / opt.scale_factor, op t)
545                     G_z = G_z[:, :, 0:real_next.shape[2]]
546                     count += 1
547             # This function uses Z_opt for noise
548             if mode == 'rec':
549                 count = 0
550                 for G,Z_opt,real_curr,real_next,noise_amp in zip(Gs,Zs,reals,reals[1:],NoiseAmp):
551                     G_z = G_z[:, :, 0:real_curr.shape[2], 0:real_curr.shape[3]]
552                     G_z = m_image(G_z)
553                     z_in = noise_amp*Z_opt+G_z
554                     G_z = G(z_in.detach(),G_z)
555                     G_z = imresize(G_z,1/opt.scale_factor,opt)
556                     G_z = G_z[:,:,0:real_next.shape[2],0:real_next.shape[3]]
557                     #if count != (len(Gs)-1):
558                     # G_z = m_image(G_z)
559                 for G, Z_opt, real_curr, sr_curr, real_next, noise_amp in zip(Gs, Zs, reals, sr_list, reals[1:], NoiseAmp):
560 
561                     if opt.alt_pyramid_exp ¿ 0:
562                         level = count
563                         calc_ker_size=opt.ker_size_pyr[level]
564                         pad_num = int(((opt.ker_size - 1) * opt.num_layer) / 2) * opt.dilation
565 
566                     G_z = G_z[:, :, 0:real_curr.shape[2]]
567                     G_z = AudioSample.static_pad(G_z, pad_num, opt)
568                     z_in = noise_amp * Z_opt + G_z
569                     G_z = G(z_in.detach(), G_z)
570                     if opt.normalize_generator_output ¿0:
571                         G_z=AudioSample.static_normalize(G_z)
572                     G_z = AudioSample.resample_to_julius_static(G_z, sr_list[count], sr_list[count + 1])
573 
574                     if opt.adjust_upsampled ¿ 0:
575                         if G_z.shape[2] ¡ real_next.shape[2]:
576                             dif = real_next.shape[2] - G_z.shape[2]
577                             G_z = torch.cat((G_z, torch.zeros([1,1,dif], dtype=torch.float32)), dim=2)
578 
579                     # G_z = imresize(G_z, 1 / opt.scale_factor, opt)
580                     G_z = G_z[:, :, 0:real_next.shape[2]]
581                     # Lines that were commented out in the original function removed here
582                     count += 1
583         return G\z
584 
585     ...
```

```
586
587
588 train_single_scale(D_curr,G_curr,reals[:scale_num+1],Gs[:scale_num],Zs[:scale_num],in_s,NoiseAmp[:scale_num],opt,centers=
        centers)
589                 z_curr, in_s, G_curr = train_single_scale(D_curr, G_curr, reals[:scale_num + 1], Gs[:scale_num],
590                                                           Zs[:scale_num], in_s, NoiseAmp[:scale_num], opt, centers=centers)
591
592                 G_curr = functions.reset_grads(G_curr,False)
593                 G_curr = functions.reset_grads(G_curr, False)
594                 G_curr.eval()
595                 D_curr = functions.reset_grads(D_curr,False)
596                 D_curr = functions.reset_grads(D_curr, False)
597                 D_curr.eval()
598
599                 Gs[scale_num] = G_curr
600                 Zs[scale_num] = z_curr
601                 NoiseAmp[scale_num] = opt.noise_amp
602                 torch.save(Zs, '%s/Zs.pth' % (opt.out_))
603                 torch.save(Gs, '%s/Gs.pth' % (opt.out_))
604                 torch.save(reals, '%s/reals.pth' % (opt.out_))
605                 torch.save(NoiseAmp, '%s/NoiseAmp.pth' % (opt.out_))
606
607                 scale_num+=1
608                 scale_num += 1
609                 nfc_prev = opt.nfc
610             del D_curr,G_curr
611             del D_curr, G_curr
612     return
613
614
615 def init_models(opt):
616
617     #generator initialization:
618     # generator initialization:
619     netG = models.GeneratorConcatSkip2CleanAdd(opt).to(opt.device)
620     netG.apply(models.weights\init)
621     if opt.netG != '':
622         netG.load_state_dict(torch.load(opt.netG))
623     print(netG)
624
625     #discriminator initialization:
626
627     # discriminator initialization:
628     netD = models.WDiscriminator(opt).to(opt.device)
629     netD.apply(models.weights_init)
630     if opt.netD != '':
631         netD.load_state_dict(torch.load(opt.netD))
632     print(netD)
633
634
635     return netD, netG
```

Listing A.5 models.py

```
1 import torch.nn as nn
2 import numpy as np
```

```
import torch.nn.functional as F
from SinGAN.AudioSample import AudioSample


class ConvBlock(nn.Sequential):
    def __init__(self, in_channel, out_channel, ker_size, padd, stride):
    def __init__(self, in_channel, out_channel, ker_size, padd, stride, batch_norm, dilation, dropout=0,
                 use_RELU=False):
        super(ConvBlock, self).__init__()
        self.add_module('conv', nn.Conv2d(in_channel, out_channel, kernel_size=ker_size, stride=stride, padding=padd)),
        self.add_module('norm', nn.BatchNorm2d(out_channel)),
        self.add_module('LeakyRelu', nn.LeakyReLU(0.2, inplace=True))
        self.add_module('conv',
                        nn.Conv1d(in_channel, out_channel, dilation=dilation, kernel_size=ker_size, stride=stride,
                                  padding=padd)),
        # TODO: Get Batch norm working as an optional parameter
        if batch_norm:
            self.add_module('norm', nn.BatchNorm1d(out_channel)),
        if use_RELU:
            self.add_module('Relu', nn.LeakyReLU(inplace=True))
        else:
            self.add_module('LeakyRelu', nn.LeakyReLU(0.2, inplace=True))
        if dropout ¿ 0:
            self.add_module('Dropout', nn.Dropout(p=dropout))


def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv2d') != -1:
    if classname.find('Conv1d') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('Norm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)

class WDiscriminator(nn.Module):
    def __init__(self, opt):
        super(WDiscriminator, self).__init__()
        self.is_cuda = torch.cuda.is_available()
        N = int(opt.nfc)
        self.head = ConvBlock(opt.nc_im, N, opt.ker_size, opt.padd_size, 1)
        self.head = ConvBlock(opt.nc_aud, N, opt.ker_size, opt.padd_size, opt.stride, opt.batch_norm,
                              dilation=opt.dilation)
        self.body = nn.Sequential()
        for i in range(opt.num_layer - 2):
            N = int(opt.nfc / pow(2, (i + 1)))
            block = ConvBlock(max(2 * N, opt.min_nfc), max(N, opt.min_nfc), opt.ker_size, opt.padd_size, 1)
            if opt.change_channel_count ¿ 0:
                N = int(opt.nfc / pow(2, (i + 1)))
            else:
                N = 1
            block = ConvBlock(max(2 * N, opt.min_nfc), max(N, opt.min_nfc), opt.ker_size, opt.padd_size, opt.stride,
                              opt.batch_norm, dilation=opt.dilation, dropout=opt.dropout)
            self.body.add_module('block%d' % (i + 1), block)
        self.tail = nn.Conv2d(max(N, opt.min_nfc), 1, kernel_size=opt.ker_size, stride=1, padding=opt.padd_size)
```

```
58            self.tail = nn.Conv1d(max(N, opt.min_nfc), 1, kernel_size=opt.ker_size, stride=1, dilation=opt.dilation,
59                                  padding=opt.padd_size)
60
61        def forward(self, x):
62            x = self.head(x)
63            x = self.body(x)
64            x = self.tail(x)
65            x = self.head(x) # Layer 1
66            x = self.body(x) # Layers 2-4
67            x = self.tail(x) # Layer5
68            return x
69
70
71    class GeneratorConcatSkip2CleanAdd(nn.Module):
72        def __init__(self, opt):
73            self.opt = opt
74            super(GeneratorConcatSkip2CleanAdd, self).__init__()
75            self.is_cuda = torch.cuda.is\_available()
76            N = opt.nfc
77            self.head = ConvBlock(opt.nc_im, N, opt.ker_size, opt.padd_size,
78                                  1) # GenConvTransBlock(opt.nc_z,N,opt.ker_size,opt.padd_size,opt.stride)
79            self.head = ConvBlock(opt.nc_aud, N, opt.ker_size, opt.padd_size, opt.stride,
80                                  opt.batch_norm, dilation=opt.dilation,
81                                  use_RELU=opt.RELU_in_gen) # GenConvTransBlock(opt.nc_z,N,opt.ker_size,opt.padd_size,opt.stride)
82            self.body = nn.Sequential()
83            for i in range(opt.num_layer - 2):
84                N = int(opt.nfc / pow(2, (i + 1)))
85                block = ConvBlock(max(2 * N, opt.min_nfc), max(N, opt.min_nfc), opt.ker_size, opt.padd_size, 1)
86                if opt.change_channel_count ¿ 0:
87                    N = int(opt.nfc / pow(2, (i + 1)))
88                else:
89                    N = 1
90                block = ConvBlock(max(2 * N, opt.min_nfc), max(N, opt.min_nfc), opt.ker_size, opt.padd_size, opt.stride,
91                                  opt.batch_norm, dilation=opt.dilation, use_RELU=opt.RELU_in_gen)
92                self.body.add_module('block%d' % (i + 1), block)
93            self.tail = nn.Sequential(
94                nn.Conv2d(max(N, opt.min_nfc), opt.nc_im, kernel_size=opt.ker_size, stride=1, padding=opt.padd_size),
95                nn.Conv1d(max(N, opt.min_nfc), opt.nc_aud, kernel_size=opt.ker_size, stride=opt.stride,
96                          dilation=opt.dilation, padding=opt.padd_size),
97                nn.Tanh()
98            )
99
100   def forward(self, x, y):
101           x = self.head(x)
102           x = self.body(x)
103           x = self.tail(x)
104           # These next two lines exist because y may (and seems to often be, at least for images) bigger
105           # Then x. This trims off some off from each edge to make them the same size. Assumes it's the same
106           # amount bigger in each dimension
107
108           ind = int((y.shape[2] - x.shape[2]) / 2)
109           y = y[:, :, ind:(y.shape[2] - ind), ind:(y.shape[3] - ind)]
110           y = y[:, :, ind:(y.shape[2] - ind)]
111           return x + y
```

Listing A.6 manipulate.py

```
#This is the relevant part of the file only
def SinGAN_generate(Gs,Zs,reals,NoiseAmp,opt,in_s=None,scale_v=1,scale_h=1,n=0,gen_start_scale=0,num_samples=50):

def SinGAN_generate(Gs, Zs, reals, NoiseAmp, opt, in_s=None, scale_v=1, scale_h=1, n=0, gen_start_scale=0, num_samples=50):
    #if torch.is_tensor(in_s) == False:
    if in\_s is None:
        in_s = torch.full(reals[0].shape, 0, device=opt.device)
        in_s = torch.full(reals[0].shape, 0, dtype=torch.float32, device=opt.device)
    images_cur = []
    for G, Z_opt, noise_amp in zip(Gs, Zs, NoiseAmp):
        pad1 = ((opt.ker_size-1)*opt.num_layer)/2
        m = nn.ZeroPad2d(int(pad1))
        pad1 = int(((opt.ker_size - 1) * opt.num_layer) / 2) * opt.dilation

        if opt.alt_pyramid_exp ¿ 0:
            level = n
            calc_ker_size=opt.ker_size_pyr[level]
            pad1 = int(((opt.ker_size - 1) * opt.num_layer) / 2) * opt.dilation

        # m = nn.ZeroPad2d(int(pad1))
        nzx = (Z_opt.shape[2]-pad1*2)*scale_v
        nzy = (Z_opt.shape[3]-pad1*2)*scale_h
        # nzy = (Z_opt.shape[3]-pad1*2)*scale_h

        images_prev = images_cur
        images_cur = []



        for i in range(0,num_samples,1):

            if n == 0:
                z_curr = functions.generate_noise([1,nzx,nzy], device=opt.device)
                z_curr = z_curr.expand(1,3,z_curr.shape[2],z_curr.shape[3])
                z_curr = m(z_curr)
                z_curr = functions.generate_noise([1, nzx], device=opt.device)
                z_curr = z_curr.expand(1, 1, z_curr.shape[2])
                z_curr = AudioSample.static_pad(z_curr, pad1, opt)
            else:
                z_curr = functions.generate_noise([opt.nc_z,nzx,nzy], device=opt.device)
                z_curr = m(z_curr)
                z_curr = functions.generate_noise([opt.nc_aud, nzx], device=opt.device)
                z_curr = AudioSample.static_pad(z_curr, pad1, opt)

            if images_prev == []:
                I_prev = m(in_s)
                I_prev = AudioSample.static_pad(in_s, pad1, opt)
                #I_prev = m(I_prev)
                #I_prev = I_prev[:,:,0:z_curr.shape[2],0:z_curr.shape[3]]
                #I_prev = functions.upsampling(I_prev,z_curr.shape[2],z_curr.shape[3])
            else:
                I_prev = images_prev[i]
                I_prev = imresize(I_prev,1/opt.scale_factor, opt)
```

```
54                  #I_prev = imresize(I_prev,1/opt.scale_factor, opt)
55                  I_prev = AudioSample.resample_to_julius_static(I_prev, opt.SR_pyr[n-1], opt.SR_pyr[n])
56                  if opt.mode != "SR":
57                      I_prev = I_prev[:, :, 0:round(scale_v * reals[n].shape[2]), 0:round(scale_h * reals[n].shape[3])]
58                      I_prev = m(I_prev)
59                      I_prev = I_prev[:,:,0:z_curr.shape[2],0:z_curr.shape[3]]
60                      I_prev = functions.upsampling(I_prev,z_curr.shape[2],z_curr.shape[3])
61                      I_prev = I_prev[:, :, 0:round(scale_v * reals[n].shape[2])]
62                      I_prev = AudioSample.static_pad(I_prev, pad1, opt)
63                      I_prev = I_prev[:,:,0:z_curr.shape[2]]
64                      #I_prev = functions.upsampling(I_prev,z_curr.shape[2],z_curr.shape[3])
65                      #I_prev = AudioSample.resample_to_julius_static(I_prev, opt.SR_pyr[n - 1], opt.SR_pyr[n])
66                      if z_curr.shape != I_prev.shape:
67
68                          if I_prev.shape[2] ¡ z_curr.shape[2]:
69                              dif = z_curr.shape[2] - I_prev.shape[2]
70                              I_prev = torch.cat((I_prev, torch.zeros([1, 1, dif], dtype=torch.float32)), dim=2)
71                  else:
72                      I_prev = m(I_prev)
73                      I_prev = AudioSample.static_pad(I_prev, pad1, opt)
74
75              if n < gen_start_scale:
76                  z_curr = Z_opt
77
78              z_in = noise_amp*(z_curr)+I_prev
79              z_in = noise_amp*(z_curr)+I_prev #z_curr: 582, I_prev: 492
80              I_curr = G(z_in.detach(),I_prev)
81
82              if n == len(reals)-1:
83                  if opt.mode == 'train':
84                      dir2save = '\%s/RandomSamples/\%s/gen_start_scale=\%d' \% (opt.out, opt.input_name[:-4], gen_start_scale)
85                  else:
86                      dir2save = functions.generate_dir2save(opt)
87                      if opt.mode == 'random_samples':
88                          dir2save = '%s/RandomSamples/%s/gen_start_scale=%d' % (opt.out, opt.input_name[:-4], opt.gen_start_scale)
89                  try:
90                      os.makedirs(dir2save)
91                  except OSError:
92                      pass
93                  if (opt.mode != "harmonization") & (opt.mode != "editing") & (opt.mode != "SR") & (opt.mode != "paint2image")
                        :
94                      plt.imsave('%s/%d.png' % (dir2save, i), functions.convert_image_np(I_curr.detach()), vmin=0,vmax=1)
95                      #plt.imsave('%s/%d.png' % (dir2save, i), functions.convert_image_np(I_curr.detach()), vmin=0,vmax=1)
96                      AudioSample.static_save(I_curr.detach(), opt.SR_pyr[-1], '%s/%d.wav' % (dir2save, i))
97                      #plt.imsave('\%s/\%d_\%d.png' \% (dir2save,i,n),functions.convert_image_np(I_curr.detach()), vmin=0, vmax
                            =1)
98                      #plt.imsave('\%s/in_s.png' \% (dir2save), functions.convert_image_np(in_s), vmin=0,vmax=1)
99              images_cur.append(I_curr)
100         n+=1
101     return I_curr.detach()
```

Listing A.7 random_samples.py

```
1   from config import get\_arguments
```

```
2  from SinGAN.manipulate import *
3  from SinGAN.training import *
4  from SinGAN.imresize import imresize
5  import SinGAN.functions as functions
6
7
8  if __name__ == '__main__':
9      parser = get_arguments()
10     parser.add_argument('–input_dir', help='input image dir', default='Input/Images')
11     parser.add_argument('–input_dir', help='input image dir', default='Input/Audio')
12     parser.add_argument('--input_name', help='input image name', required=True)
13     parser.add_argument('--mode', help='random_samples | random_samples_arbitrary_sizes', default='train', required=True)
14     # for random_samples:
15     parser.add_argument('--gen_start_scale', type=int, help='generation start scale', default=0)
16     # for random_samples_arbitrary_sizes:
17     parser.add_argument('--scale_h', type=float, help='horizontal resize factor for random samples', default=1.5)
18     parser.add_argument('--scale_v', type=float, help='vertical resize factor for random samples', default=1)
19
20     opt = parser.parse_args()
21     opt = functions.post_config(opt)
22     Gs = []
23     Zs = []
24     reals = []
25     NoiseAmp = []
26     dir2save = functions.generate_dir2save(opt)
27     opt.SR_pyr=[800, 1600, 2150, 2850, 3825, 5100, 6750, 9000, 12000, 16000, 24000, 32000]
28
29
30
31 + if opt.mode == 'random_samples':
32         dir2save = '%s/RandomSamples/%s/gen_start_scale=%d' % (opt.out, opt.input_name[:-4], opt.gen_start_scale)
33     else:
34         dir2save = functions.generate_dir2save(opt)
35     if dir2save is None:
36         print('task does not exist')
37     elif (os.path.exists(dir2save)):
38         if opt.mode == 'random_samples':
39             print('random samples for image \%s , start scale=\%d, already exist' \% (opt.input_name, opt.gen_start_scale))
40         elif opt.mode == 'random_samples_arbitrary_sizes':
41             print('random samples for image \%s at size: scale_h=\%f, scale_v=\%f, already exist' \% (opt.input_name, opt.
                    scale_h, opt.scale_v))
42     else:
43         try:
44             os.makedirs(dir2save)
45         except OSError:
46             pass
47
48         if opt.mode == 'random_samples':
49             real = functions.read_image(opt)
50             functions.adjust_scales2image(real, opt)
51             inputpath = opt.input_dir + "/" + opt.input_name
52             real = AudioSample(opt, inputpath, sr=16000)
53             functions.adjust_scales2data(real, opt)
54             opt.stop_scale = len(opt.SR_pyr) - 1
55             Gs, Zs, reals, NoiseAmp = functions.load_trained_pyramid(opt)
56             in_s = functions.generate_in2coarsest(reals,1,1,opt)
```

```
57                in_s = functions.generate_in2coarsest(reals, 1, 1, opt)
58                SinGAN_generate(Gs, Zs, reals, NoiseAmp, opt, gen_start_scale=opt.gen_start_scale)
59
60
61            elif opt.mode == 'random_samples_arbitrary_sizes':
62                real = functions.read_image(opt)
63                functions.adjust_scales2image(real, opt)
64                Gs, Zs, reals, NoiseAmp = functions.load_trained_pyramid(opt)
65                in_s = functions.generate_in2coarsest(reals,opt.scale_v,opt.scale_h,opt)
66                SinGAN_generate(Gs, Zs, reals, NoiseAmp, opt, in_s, scale_v=opt.scale_v, scale_h=opt.scale_h)
```

Listing A.8 functions.py

```
1   #This is the relevant part of the file only
2
3   def generate_noise(size,num_samp=1,device='cuda',type='gaussian', scale=1):
4
5   +def generate_noise(size, num_samp=1, device='cuda', type='gaussian', scale=1):
6       #We're gonna start by only modifying gaussian because that seems to be the default
7       if type == 'gaussian':
8           noise = torch.randn(num_samp, size[0], round(size[1]/scale), round(size[2]/scale), device=device)
9           noise = upsampling(noise,size[1], size[2])
10      if type =='gaussian_mixture':
11          noise1 = torch.randn(num_samp, size[0], size[1], size[2], device=device)+5
12          noise2 = torch.randn(num_samp, size[0], size[1], size[2], device=device)
13          noise = noise1+noise2
14          noise = torch.randn(num_samp, size[0], round(size[1] / scale), device=device)
15          noise = upsampling(noise, size[1])
16      """
17      if type == 'gaussian_mixture':
18          noise1 = torch.randn(num_samp, size[0], size[1], device=device) + 5
19          noise2 = torch.randn(num_samp, size[0], size[1], device=device)
20          noise = noise1 + noise2
21      if type == 'uniform':
22          noise = torch.randn(num_samp, size[0], size[1], size[2], device=device)
23          noise = torch.randn(num_samp, size[0], size[1], device=device)
24      """
25      return noise
26
27
28  ....
29
30
31
32  def np2torch(x,opt):
33      if opt.nc_im == 3:
34          x = x[:,:,:,None]
35          x = x.transpose((3, 2, 0, 1))/255
36      else:
37          x = color.rgb2gray(x)
38          x = x[:,:,None,None]
39          x = x.transpose(3, 2, 0, 1)
40  # Function is an eddited version of np2torch
41  # Added by Levi Pfantz on 10/14/2020
```

```
42  def np2torch(x, is_not_cuda=False):
43      x = torch.from_numpy(x)
44      if not(opt.not_cuda):
45      if not (is_not_cuda):
46          x = move_to_gpu(x)
47      x = x.type(torch.cuda.FloatTensor) if not(opt.not_cuda) else x.type(torch.FloatTensor)
48      #x = x.type(torch.FloatTensor)
49      x = norm(x)
50      # FloatTensor is a 32bit float data type. I'm going to recommend all input audio is in 32 bit float.
51      x = x.type(torch.cuda.FloatTensor) if not (is_not_cuda) else x.type(torch.FloatTensor)
52      # x = x.type(torch.FloatTensor)
53      # x = norm(x)
54      return x
55
56
57  ....
58
59
60  def adjust_scales2data(real_,opt):
61      #opt.num_scales = int((math.log(math.pow(opt.min_size / (real_.shape[2]), 1), opt.scale_factor_init))) + 1
62      opt.num_scales = math.ceil((math.log(math.pow(opt.min_size / (min(real_.shape[2], real_.shape[3])), 1), opt.scale_factor_init))) + 1
63      scale2stop = math.ceil(math.log(min([opt.max_size, max([real_.shape[2], real_.shape[3]])]) / max([real_.shape[2], real_.shape[3]]),opt.scale_factor_init))
64
65  def adjust_scales2data(real, opt):
66      real_ = real.data
67      sr = real.sr
68      # opt.num_scales = int((math.log(math.pow(opt.min_size / (real_.shape[2]), 1), opt.scale_factor_init))) + 1
69      opt.num_scales = math.ceil((math.log(math.pow(opt.min_size / (real_.shape[2]), 1), opt.scale_factor_init))) + 1
70      scale2stop = math.ceil(math.log(min([opt.max_size, real_.shape[2]]) / real_.shape[2], opt.scale_factor_init))
71      opt.stop_scale = opt.num\scales - scale2stop
72      opt.scale1 = min(opt.max_size / max([real_.shape[2], real_.shape[3]]),1) # min(250/max([real_.shape[0],real_.shape[1]]),1)
73      real = imresize(real_, opt.scale1, opt)
74      #opt.scale_factor = math.pow(opt.min_size / (real.shape[2]), 1 / (opt.stop_scale))
75      opt.scale_factor = math.pow(opt.min_size/(min(real.shape[2],real.shape[3])),1/(opt.stop_scale))
76      scale2stop = math.ceil(math.log(min([opt.max_size, max([real_.shape[2], real_.shape[3]])]) / max([real_.shape[2], real_.shape[3]]),opt.scale_factor_init))
77      opt.scale1 = min(opt.max_size / real_.shape[2], 1) # min(250/max([real_.shape[0],real_.shape[1]]),1)
78      realsize = int(sr * opt.scale1)
79      # opt.scale_factor = math.pow(opt.min_size / (real.shape[2]), 1 / (opt.stop_scale))
80      opt.scale_factor = math.pow(opt.min_size / (realsize), 1 / (opt.stop_scale))
81      scale2stop = math.ceil(math.log(min([opt.max_size, realsize]) / realsize, opt.scale_factor_init))
82      opt.stop_scale = opt.num_scales - scale2stop
83      return real
84      return opt.scale1, opt.scale_factor
85
86
87  ....
88
89
90  def creat_reals_pyramid(real,reals,opt):
91      real = real[:,0:3,:,:]
92      for i in range(0,opt.stop_scale+1,1):
93          scale = math.pow(opt.scale_factor,opt.stop_scale-i)
94          curr_real = imresize(real,scale,opt)
95          reals.append(curr_real)
```

```
96          return reals
97
98  def creat_reals_pyramid(real, reals, opt, verbose=False):
99
100         reals.append(real.data)
101         for i in range(0, opt.stop_scale, 1):
102             curr_sr=opt.SR_pyr[-(1+i)]
103             new_sr=opt.SR_pyr[-(2+i)]
104             result=AudioSample.resample_to_julius_static(reals[-1], curr_sr, new_sr)
105             if result.shape[2] % 2 != 0 and opt.make_input_tensor_even ¿ 0:
106                 result = result[:, :, 0:result.shape[2] - 1]
107             reals.append(result)
108
109             if verbose:
110                 print("On level:", i, "curr_sr is", curr_sr, "new sr is: ",new_sr, "justed addes shape is: ", reals[-1].shape)
111         reals.reverse()
112         if True:
113             if os.path.exists('Audio_pyramid'):
114                 shutil.rmtree('Audio_pyramid')
115             os.makedirs('Audio_pyramid')
116             for x in range(len(reals)):
117                 AudioSample.static_save(reals[x], opt.SR_pyr[x], 'Audio_pyramid/%s.wav' % (str(x)))
118         return reals
119
120 def creat_reals_pyramid_torch(real, reals, opt, verbose=False):
121         sr_list=[]
122         for i in range(0, opt.stop_scale + 1, 1):
123             scale = math.pow(opt.scale_factor, opt.stop_scale - i)
124             curr_real = real.clone() # for some reason curr_real = real.clone().resample_by(scale) causes a bug...
125             curr_real.resample_by(scale)
126             reals.append(curr_real.data)
127             sr_list.append(curr_real.sr)
128             if verbose:
129                 print("On level:", i, "New scale is", scale, "new sr is: ",curr_real.sr, "curr_real.shape is: ", curr_real.data.shape)
130         return reals, sr_list
```

Listing A.9 AudioSample.py

```
1
2   import torchaudio
3   import torch
4   import julius
5
6
7   class AudioSample:
8
9       def __init__(self, opt, path, clone=False, data=None, sr=None):
10          self.opt = opt
11          self.path = path
12          if not clone:
13              if opt.audio_backend == "soundfile":
14                  self.data, self.sr = torchaudio.load(path, normalization=opt.norm)
15              elif opt.audio_backend == "sox_io":
16                  self.data, self.sr = torchaudio.load(path, normalize=opt.norm)
```

```
17              if not opt.not_cuda and torch.cuda.is_available():
18                  self.data = self.data.to(torch.device('cuda'))
19
20              # Data is stored in a tensor of shape 1 (batch size) x audio channels x width (floats in channel)
21              self.data = self.data[0].view(1, 1, -1)
22              if self.sr != sr:
23                  self.resample_to_julius(sr)
24
25              if self.data.shape[2] % 2 != 0 and opt.make_input_tensor_even > 0:
26                  self.data=self.data[:,:,0:self.data.shape[2]-1]
27          else:
28              self.data = data
29              self.sr = sr
30
31      @staticmethod
32      def static_save(data_in, sr_in, path_in):
33          data_in = data_in.to(torch.device('cpu'))
34          torchaudio.save(path_in, data_in.view(1, -1), sr_in, 32)
35
36
37
38      @staticmethod
39      def static_pad(data_in, pad_each_side_by, opt):
40          if opt.pad_with_noise < 1:
41              part1 = torch.zeros(1, data_in.shape[1], pad_each_side_by, dtype=torch.float32)
42              part2 = part1
43          else:
44              if opt.not_cuda > 0:
45                  device = 'cpu'
46              else:
47                  device = 'cuda'
48
49              part1 =torch.randn(1, data_in.shape[1], pad_each_side_by, device=device)
50              part2 = torch.randn(1, data_in.shape[1], pad_each_side_by, device=device)
51          data_in = torch.cat((part1, data_in, part2), dim=2)
52          # data_in=torch.cat((data_in, part1), dim=2)
53          return data_in
54
55      @staticmethod
56      def resample_to_julius_static(data_in, sr_in, new_sr):
57          return julius.resample_frac(data_in, sr_in, new_sr)
58
59      def resample_to_julius(self, target):
60          self.data = julius.resample_frac(self.data, self.sr, target)
61          self.sr = target
62
63
64      def resample_by(self, scale):
65
66          saved_shape = self.data.shape
67          if saved_shape[1] == 1:
68              newsr = int(self.sr * scale)
69              self.data = (torchaudio.transforms.Resample(orig_freq=self.sr, new_freq=newsr)(self.data)[0, 0, :]).view(1,
70                                                                                                                        1,
71                                                                                                                        -1)
72          # Things are mostly only setup for one channel. This is the exception
73          elif saved_shape[1] == 2:
```

```
74              newsr = int(self.sr * scale)
75              chan1 = (torchaudio.transforms.Resample(orig_freq=self.sr, new_freq=newsr)(self.data)[0, 0, :]).view(1, 1,
76                                                                                                                    -1)
77              chan2 = (torchaudio.transforms.Resample(orig_freq=self.sr, new_freq=newsr)(self.data)[0, 1, :]).view(1, 1,
78                                                                                                                    -1)
79              self.data = torch.cat(chan1, chan2, dim=1)
80          self.sr = newsr
81
82
83
84      @staticmethod
85      def static_normalize(data_in):
86
87          factor=torch.max(torch.abs(data_in))
88          if factor > 1:
89              out = data_in/factor
90              return out
91          else:
92              return data_in
93
94      def clone(self):
95          return AudioSample(self.opt, self.path, clone=True, data=self.data, sr=self.sr)
```