Deconstructing Mozart: A GAN-Style Approach to Raw Audio Processing and Generation

DongGyun Kim

Advisor: Prof. Niraj Jha

June 2018

Submitted in partial fulfillment

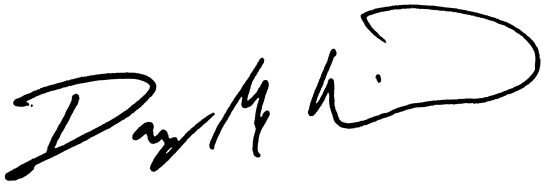
of the requirements for the degree of

Bachelor of Science in Engineering

Department of Electrical Engineering

Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.



DongGyun Kim

Deconstructing Mozart: A GAN-Style Approach to Raw Audio Processing and Generation

DongGyun Kim

Abstract

We propose a new system for raw audio processing and generation which combines the success of Generative Adversarial Networks (GANs) and WaveNet, a generative model created by DeepMind. This system will generate piano music using only sample piano audio files that are given to it. The use of GANs together with raw audio, rather than derived features, makes this project unique in the realm of machine learning. Most previous works that compose music through artificial means utilize representations of music, rather than raw audio, due to computational complexity, and previous works that use GANs often solve the problem of image synthesis. Current progress of the fully implemented system with WaveNet and GANs are able to nearly mimic the frequency domain characteristics of the inputted data.

Acknowledgments

There are a lot of people to thank for their support on this paper. Any research project, particularly those driven by undergraduates, requires a great deal of guidance and help from those with a deeper understanding of the topics. In this regard, I couldn't have found a better advisor and research associate than Professor Niraj Jha and Ozge Akmandor. Their advice, week by week and email by email, helped me scope out an appropriate project and helped light the way through uncharted lands.

I'd also to thank my parents as well for their encouragement in the most difficult times on this project. I really could not have accomplished what I have, both in this paper and at Princeton, without them.

Contents

Chapter 1

Introduction

An incredibly important step towards artificial intelligence is unsupervised learning, the modeling of real, observational phenomena into structures of a lower dimension without human supervision. At this time, most machine learning algorithms (classified as supervised learning) require a great deal of manual work before the algorithms can begin to learn. They attempt to classify according to labels that must be created by humans: numerical digits, images of cats and dogs, musical composers — all of these must be collected and labeled before the prediction and classification of new data can begin. Could it be possible for algorithms to move past the need for human labels and instead model the behavior of the system?

To this end, there are emerging approaches slowly starting to achieve this lofty goal of artificial intelligence. And while the most common approach to unsupervised learning is through various clustering algorithms (i.e., k-Means), there are exciting strides being made with a new model of unsupervised learning coined "GANs" (generative adversarial networks). GANs were first introduced by Goodfellow et. al. in 2014 and have since empirically achieved remarkable success in canonical machine learning tasks [1]. GANs have already achieved success in the realm of image generation, being able to generate superficially realistic images with recognizable characteristics [2].The

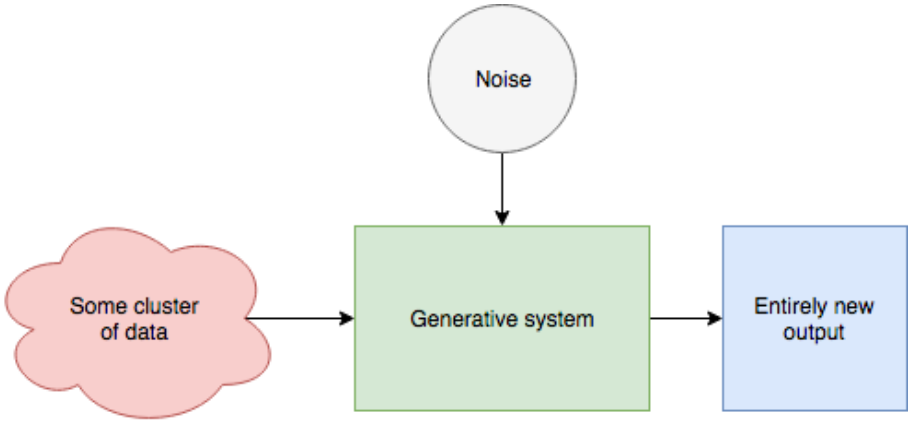


Figure 1.1: High level representation of audio generation scheme.

question must then be asked: if images can be generated, why not music?

1.1 Overview

This project will focus on the generation of piano music using a collection of raw audio samples (i.e., files of type .wav or .m4a), as shown in Figure 1.1. While this project will specifically focus on generating music in the style of composers, we note that the unsupervised nature of GANs means that a successful generative system will be able to generalize whatever input is given to it: the decision to narrow the audio to a more recognizable set of human labels, such as music of the same composer and instrument, was only made to make a more successful and cohesive output, not as a result of a technical limitation of this project.

As a result, this project is unique in a number of ways. While there are already a number of machine learning projects that can "compose" in certain musical genres, few have utilized the power of GANs, and even fewer use raw audio. By being able to use raw audio, as opposed to derived features of audio or representations of music, we unlock far greater flexibility and face much larger challenges. In particular, harnessing the power of WaveNet, introduced in Section 2.2 within our GAN structure is particularly unique.

There are already a number of applications that use GANs to learn on images, one of which we'll see in Section 1.2, but there are some important differences between raw image data and raw audio data that make audio much more difficult to learn. On the surface, it may appear that audio is simpler to process — while images are in two dimensions, audio only exists in one. However, audio presents much greater difficulty for traditional learning algorithms due to the sheer size of the data. In the case of the .wav audio format, for example, audio can be sampled at 1 Hz to 4.3 GHz (up to 4.3×10^9 data points per second of audio) and has 16 bits of information at each data point (65,536 quantized levels, or "channels"). Beyond that, music presents an additional challenge because of the information and structure present at many timescales. Pieces do not generally get composed through a rambling melody but rather by a strict harmonic structure within adjacent notes, measures and sections. It is not enough to generate music given only the last few milliseconds of data — music has a much broader structure, often at the scale of seconds or minutes, necessitating an increase in the size of the receptive field.

In the traditional convolutional neural network model, increasing the receptive field — the size of input considered before generating a new output point — is an extremely costly operation. A model to effectively process data at this scale would have repercussions that reach further than just music generation and classification. Being able to process large amounts of raw data is an important step for more accurately modeling complex systems for a vast number of applications.

1.2 Motivation

As mentioned earlier, GANs have been used to create incredibly photo-realistic images. Recent developments in both hardware and software have yielded incredible results, some of which we can see in Figure 1.2.

Figure 1.2: Faces generated by Nvidia's new GAN algorithm [3].

While these photos generated by Nvidia don't match the resolution of a top-of-the-line camera, they are "sharp, detailed, and, in many cases, remarkably convincing" [3]. With enough computing time, Nvidia trained its algorithms over thousands of celebrity images and was able to generate images that could fool humans.

From a high level view, generating music in the style of a composer is not much different than the ability to generate photo-realistic celebrity photos. The images generated by Nvidia's GANs aren't fitted to a particular person or object. Instead, the neural networks learned to model the distribution of the photo database and create something entirely new. The algorithm created by Nvidia doesn't specialize in faces; it can generate horses, buses, bicycles, etc. through the recognition of common features [3]. In the same way, this project aims to train a GAN-type network to model the distribution of the music of a composer (say, Mozart), and then generate music in that style.

Just as faces have groups of recognizable features organized in an understandable way — ears, eyes, nose — music is the same. Mozart's music is made up of recognizable chords, phrase fragments and structures put together in ways idiomatic of his time period and of style of composing. One such idiomatic structure, Sonata-Allegro form, is detailed in Figure 1.3. While this form looks at the scale of many minutes

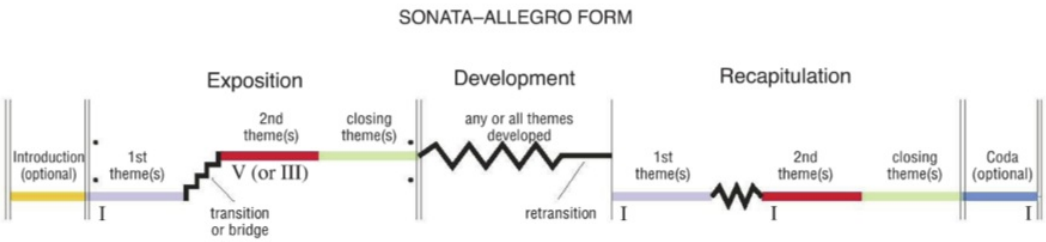


Figure 1.3: A macro-structure in many of Mozart's works [4].

(which is difficult to generate in a reasonable amount of time), the music of Mozart also often follows a set of rules referred to "tonal syntax". Tonal syntax dictates both the construction of each chord and the relationship between chords. In the music of Mozart and his contemporaries, tonal syntax was often strictly adhered to.

It is not particularly unique to compose music through the use of machine learning algorithms. However, it is quite novel to do music composition through the modeling of raw audio, as opposed to the use of representations of music (i.e., MIDI files). There are numerous projects that have achieved a good deal of success in music generation using these music representation methods through a manipulation of different neural-network structures, such as recurrent neural networks (RNNs) [5] and long short-term memory (LSTM) [6]. It is important to note that using note representation greatly reduces the amount of data present in each piece of music, thus requiring less complex algorithms and computing time. This project is motivated by both the success that GANs have had in creating realistic images and the uniqueness and flexibility of modeling raw audio.

Interesting parallels to the design in this paper can be found in "Synthesizing Audio with Generative Adversarial Networks" by Donahue et al [7]. Donahue's paper proposes two structures which utilize the learning ability GANs to the problem of modeling raw audio, named "SpecGAN" and "WaveGAN". These structures model the audio samples in the frequency-domain and time-domain, respectively. Although the building blocks of their proposed solutions are different than the ones proposed

in this paper, they aim to solve a very similar problem and do so with a good deal of empirical success. However, "SpecGAN" produces only a spectrogram representation for the audio samples, which then can be approximately inverted, and both "SpecGAN" and "WaveGAN" were unable to produce competitive results with that of WaveNet [8] and other similar audio generative models.

Chapter 2

Background Work

This chapter will detail background work essential to the design of this project.

2.1 Generative Adversarial Networks

There are numerous works that this paper will draw from. The most fundamental is the work of Goodfellow et al. which develops the foundation of the inner-workings of GANs. In short, GANs pit two different neural networks against each other. Nag describes GANs as follows:

> The models play two distinct (literally, adversarial) roles. Given some real data set R, G is the generator, trying to create fake data that looks just like the genuine data, while D is the discriminator, getting data from either the real set or G and labeling the difference. Goodfellow's metaphor (and a fine one it is) was that G was like a team of forgers trying to match real paintings with their output, while D was the team of detectives trying to tell the difference. (Except that in this case, the forgers G never get to see the original data — only the judgments of D. They're like blind forgers) [9].

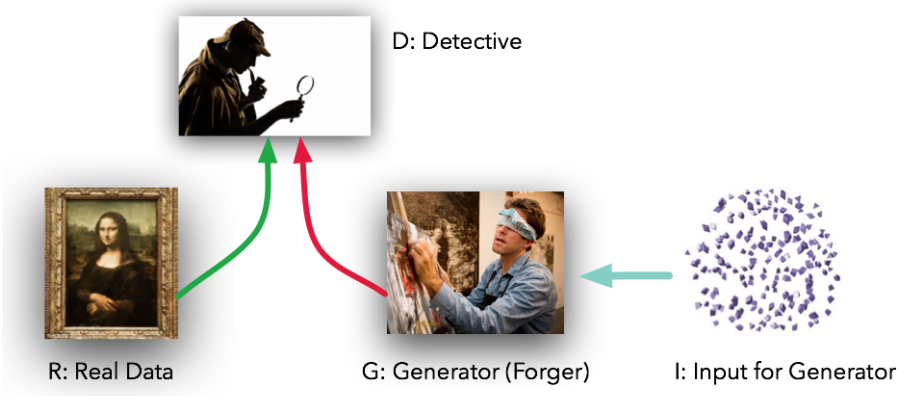


Figure 2.1: Generative Adversarial Networks metaphor visualized [9].

This metaphor is additionally illustrated in Figure 2.1.

While there are a number of specific technical ways to implement GANs, there are a few standard required steps for its learning, regardless of implementation. In a classical GAN, the discriminator network takes in an input (which can be either real data or generated data) and outputs a 1 or 0: real or fake, while the generator network takes a random input (usually sampled from a normal distribution) and generates an output the same size as discriminator's input. Through training, the discriminator alternates between receiving both real and fake data. In the case the discriminator actually believes the input is real when it is in fact fake, or vice-versa, gradient descent is performed on the weights of its network. In the case where the discriminator believes the input is fake and it is actually fake, the generator then likewise adjusts its weights with gradient descent. Figure 2.2 summarizes this training process.

This description is a high level understanding of the training process. More officially, the ascending in the stochastic gradient update performed by the discriminator, given noise samples $\{z_1, \ldots, z_m\}$ and real data $\{x_1, \ldots x_m\}$ is given by

$$\nabla\theta_d \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x_i\right) + \log\left(1 - D\left(G\left(z_i\right)\right)\right)\right] [1].$$

Similarly, the descending in the stochastic gradient update for the generator, given

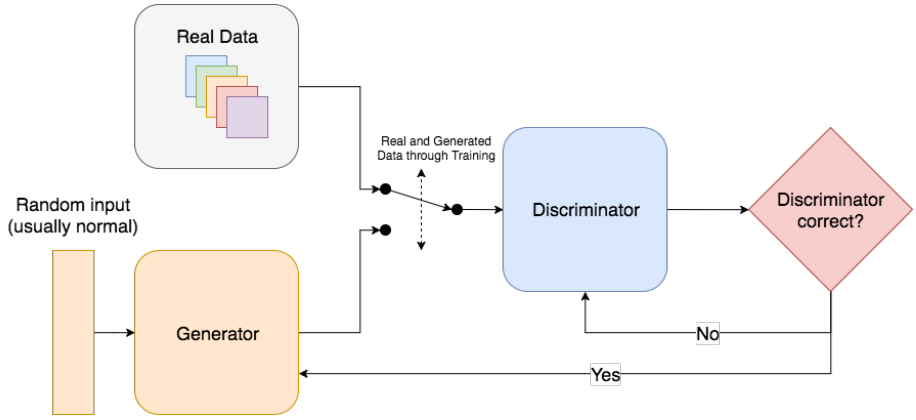


Figure 2.2: Generative Adversarial Networks training flowchart.

noise samples $\{z_1, \ldots, z_m\}$ is given by

$$\nabla\theta_d \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z_i\right)\right)\right) [1].$$

2.2 WaveNet

Another important foundational work for this project is WaveNet. WaveNet was developed by Google's DeepMind to generate speech to mimic the human voice and improve the quality of TTS (text to speech) systems [10]. However, it is the structural advantages that WaveNet presents in audio processing that make it well suited for this project, particularly in regard to the exponential increase in the size of receptive field over the number of hidden layers to capture features at a greater timescale.

At its core, WaveNet is a stack of dilated convolutional networks, which are similar to traditional convolutional neural networks (CNNs). Unlike CNNs, the neuron connections in Dilated CNNs between layers are spaced out exponentially. By dilating the convolutions at an exponential rate, the receptive field grows exponentially, rather than linearly, by the number of hidden layers [8]. Below, in Figure 2.3, we see this increase in receptive field by comparing causal convolutional layers and dilated convolutional layers. On the left of Figure 2.3, we have the "traditional" CNN. For

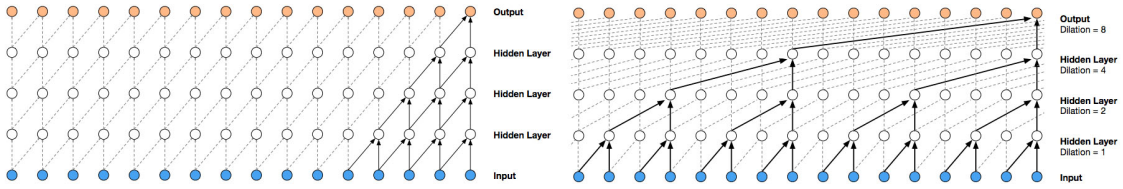


Figure 2.3: Causal layer of CNN vs. dilated layer of CNN [8].

that network, we have a receptive field of size $\#layers + 2$. For the dilated CNN, we instead of a receptive field of size $2^{\#layers+1}$. This exponential increase in the size of the receptive field makes WaveNet much more attractive for audio processing.

Another key feature of WaveNet is the μ-law softmax layer which reduces the number of bits per data point from 16 bits to 8 bits. The applied transformation on a signal x_t is described by

$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)}$$

in [8], where $-1 < x_t < 1$ and $\mu = 255$ (the desired number of target levels). This layer reduces the data from 65,536 (corresponding to the 16 bits per sample in wav files) to 256 possible levels non-linearly, which provides for a much better reconstruction of the original signal. Reducing the number of bits per data point by a factor of 256 greatly improves the rate at which prior probabilities can be calculated.

The architecture of WaveNet utilizes a stack of dilated CNNs connected through residual and skip connections. The input waveform goes through a causal convolution to prevent the information from the samples being replicated through the network. Following that, the convolved input is one-hot encoded to 256 levels as generated by the μ-law companding step and sent to the stack of layers. In each layer, the input goes through a dilated convolution, which travels through a gated activation unit. The dimensionality is reduced through a 1x1 convolution and summed with the residual. The layers are then connected through skip-connections and go to post-processing to

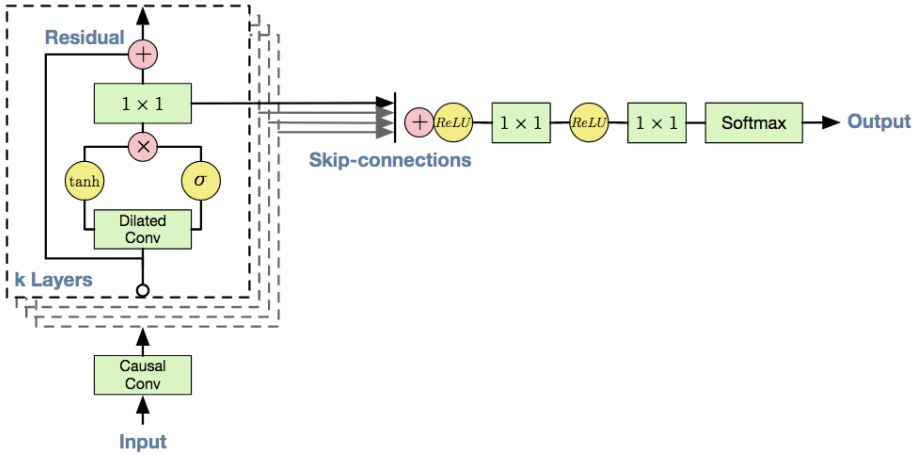


Figure 2.4: WaveNet architecture [8].

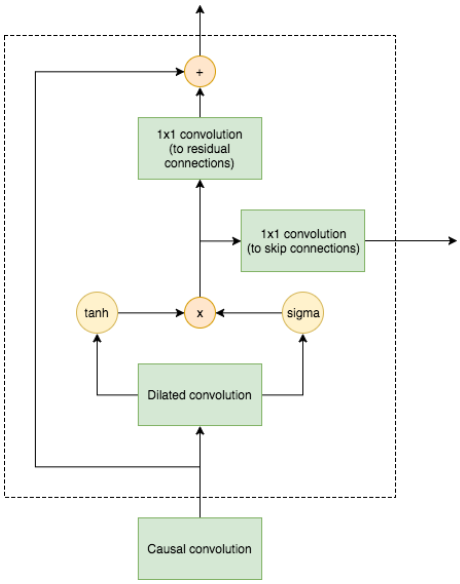


Figure 2.5: Detailed residual level architecture.

yield the output. We can see the full architecture in Figure 2.4. It is worth noting in this architecture that, at each residual level, there are actually two different 1x1 convolutions happening after the gated activation unit, as shown in Figure 2.5.

In the form presented in [8], WaveNet computes predictions incrementally prior to training, resulting in slow prediction time. Recent developments in WaveNet attempt to speed up the prediction time by parallizing the prediction with a "student-teacher" model, similar to the general structure of GANs [11]. Due to the slow speed of the prediction function, as well as its non-differentiability, we will instead utilize the

raw output of WaveNet, rather than the generated waveforms, thus harnessing its structural power.

Chapter 3

Methodology

This project aims to bring together the advantages of GANs and WaveNet in a unique way. While WaveNet was originally designed to train first on real data then incrementally generate output based on the trained probabilities, it will instead be used as the generator, taking an input of white noise and generating a set of values from that input. That set of values, over the course of training, will eventually produce a Mozart-type, piano-like waveform. This high level architecture is visualized in Figure 3.1. The true data input to the discriminator network will be randomly selected from real data from selected audio samples, and fake data will be generated by the WaveNet generator network.

Just like traditional GANs, the network will train the generator and discriminator with real and fake (generated) data. Some preprocessing will have to take place to create the correct dimensions for the data inputs and outputs. Before the training and generation process, we must select some number of data points to output from the generator. This size will be fixed throughout the training loop, and will determine the length of the audio sample that is composed. This value is also used as both the input size of the discriminator network and the size of the slices of the audio from the real dataset. As an additional preprocessing step, the program will trim out silence

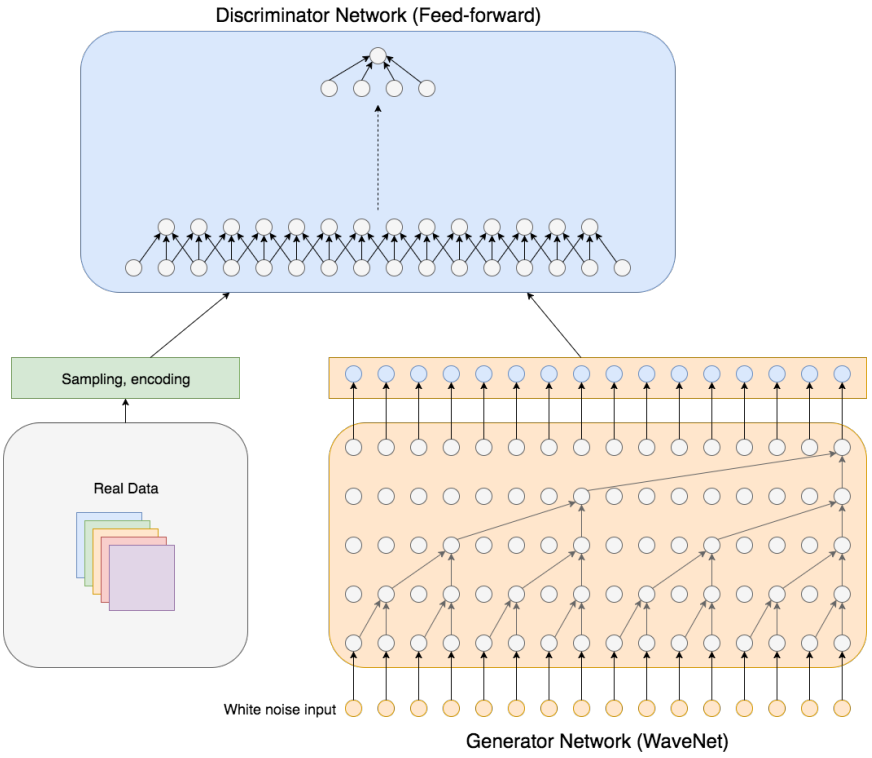


Figure 3.1: Desired final architecture.

at the beginning and ending of the real data files, as well as standardize the all inputs to the discriminator network using the mean and standard deviation of that specific slice.

3.1 Discriminator Feasibility

Before the full system is implemented, it's important to determine the feasibility of neural networks to learn to label the complicated structure of raw audio. We start by evaluating the ability of a standalone feed-forward neural network discriminator of distinguishing between Mozart's piano music and all other music (orchestral, vocal, pop). To accomplish this, we can run an independent training loop, as described in Figure 3.2, which runs separately from the main GAN code. This will help tweak the numerous parameters of the neural network. Having a well tuned discriminator that can independently perform classification will be critical to the improvement of

Table 3.1: Data set classifications

True Data	"Loose" False Data	"Tight" False Data
Mozart piano sonatas	Symphonic works	Beethoven piano sonatas
Misc. Mozart piano works	Vocal works	Misc. other piano works
	Pop songs	

the generator.

For this discriminator training loop, we will consider "true" data to be our target generated data: Mozart piano music. To test out the learning ability of the system, we will consider the "false" data to be a collection of classical music that does not feature the piano. Considering computational resources, the initial testing feed-forward network will have a smaller receptive field for about one second of data. Since one second of piano should be nearly identical between the true data and the tight false data, we will use the loose false data for the first round of training. Essentially, this will train the network to tell the difference between piano and non-piano.

While distinguishing between piano and non-piano is not the end objective of the project, it is an important first step. Having a discriminator network be able to distinguish between the two means that the generator network will be forced to generate something that sounds at least like a piano, just not yet like Mozart. We hypothesize that being able to generate piano music that sounds like Mozart (as opposed to that of a different composer) will require a much larger receptive field than one second. However, receptive field increase can only happen after full implementation of the GAN.

3.2 Simple GAN

Another important step to take with this project is a "proof of concept" using a simple GAN setup, using only two neural networks. It is expected that a simple GAN will not be able to model the distribution of the dataset as well as the proposed Wavenet

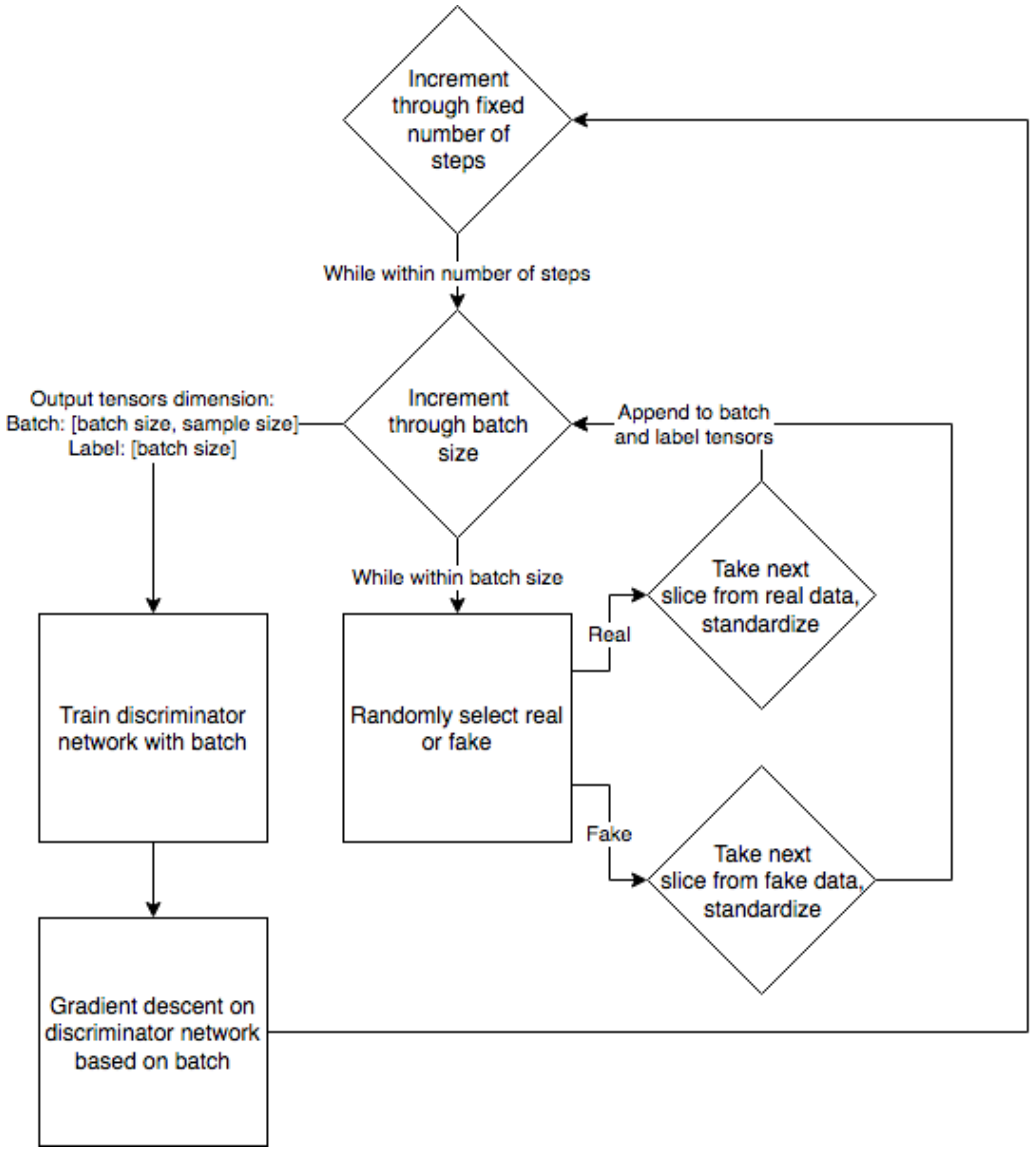


Figure 3.2: Discriminator training flow chart.

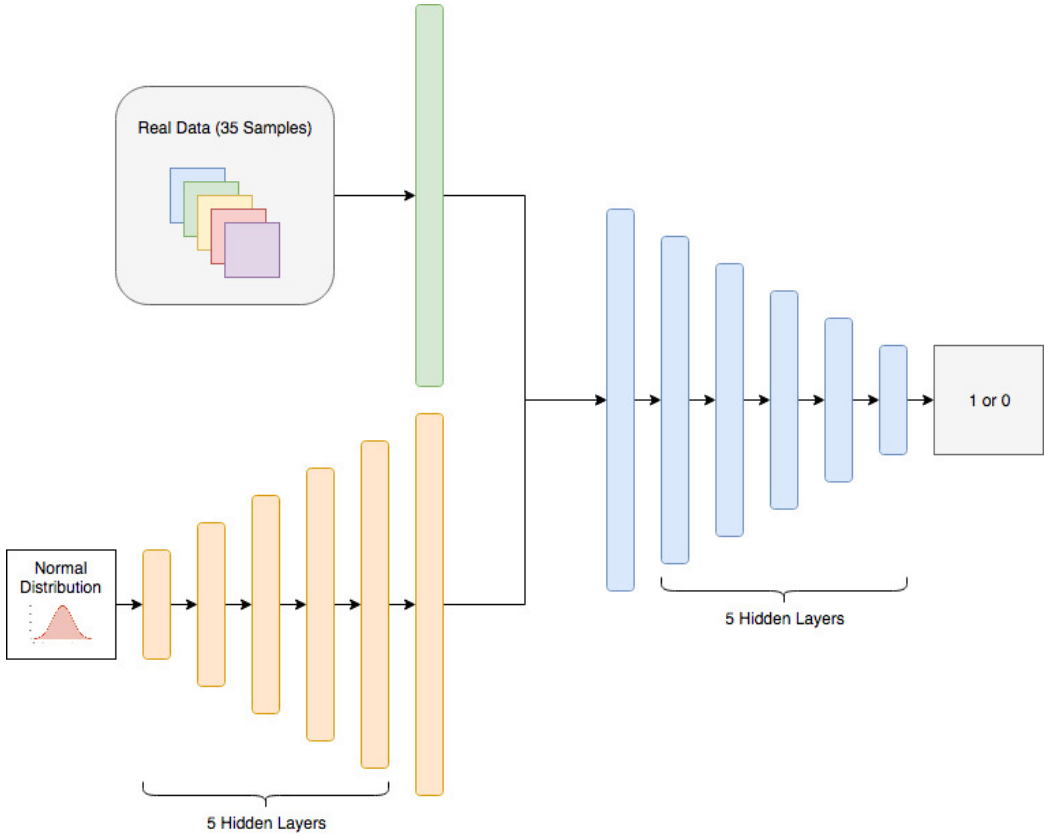


Figure 3.3: Simple GAN block diagram.

+ GAN structure.

The Simple GAN consists of two 5-layer networks (the choice of 5-layers is not arbitrary, and was based on the initial results from Section 4.1). When compared to the design of Figure 3.1, the Simple GAN setup is relatively more straightforward, as shown Figure 3.3. The two networks are both simple feed-forward networks, although the generator takes in an input size of 100 random samples to increase the randomness throughout the network. Due to initial results having too many weights being zeroed out, the activation function for both networks chosen to be a leaky ReLU:

$$f(x) = \begin{cases} x & x > 0 \\ 0.2x & \text{otherwise} \end{cases}$$

We will minimize our objective loss function with stochastic gradient descent.

We implement the GAN learning with TensorFlow. To start, we define two variables that will be inputs to the discriminator network: a placeholder input `X` which will be sampled from real data and the sample from the generator. These variables, when passed through the discriminator network, in turn yield the probabilities and logits[1].

```
X = tf.placeholder(tf.float32, shape=[None, w1], name='X')
G_sample = generator(Z)
D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)
```

The logits from the real and fake data are used to calculate the loss figures from the output of the discriminator. The loss figure from the real input is quite intuitive.

```
D_loss_real = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(D_logit_real),
        logits=D_logit_real
    )
)
D_loss_fake = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.zeros_like(D_logit_fake),
        logits=D_logit_fake
    )
)
D_loss = D_loss_real + D_loss_fake
```

The discriminator should label the logits from `D_logit_real` as true, or 1 and the logits from `D_logit_fake` as fake, or 0. In each case, the mean of the cross entropy

[1]Logit is the inverse of the sigmoidal "logistic" function. In this case, as the output of the network is a probability, the logit is the log-odds.

between the label and the logits is used to define the loss, and the total discriminator loss is defined as the sum of the two losses.

The generator loss is less intuitive: the goal of the generator is to make the discriminator believe that the label should be true, or 1. Loss should then be defined as the distance (distance being any metric, not necessarily Euclidean) from the label 1, given fake data. This loss is the objective function to minimize for our gradient descent solver. We need to define our generator loss to be the mean of the cross entropy between a label of true, or 1 and the fake logits, as defined below.

```
G_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(D_logit_fake),
        logits=D_logit_fake
    )
)
```

Once these loss figures have been defined, we set up the gradient descent solvers, being careful to only adjust the weights of the discriminator, defined as `theta_D`, when minimizing the discriminator loss, and only adjusting the weights of the generator, defined as `theta_G` whem minimizing the generator loss.

```
D_solver = tf.train.GradientDescentOptimizer(learning_rate=3e-3)
    .minimize(D_loss, var_list=theta_D)
G_solver = tf.train.GradientDescentOptimizer(learning_rate=3e-3)
    .minimize(G_loss, var_list=theta_G)
```

Finally, we can run these solvers to perform gradient descent. Empirically, we saw from initial runs that the discriminator often outperforms the generator, i.e., that the discriminator loss converges quickly, leaving the generator unable to improve itself. To resolve this, we run the generator solver more times at each epoch than the discriminator.

```
for g_batch in range(20):
    _, G_loss_curr = sess.run([G_solver, G_loss],
        feed_dict={Z: sample_Z(1, 100)}
    )

for d_batch in range(1):
    _, D_loss_curr = sess.run([D_solver, D_loss],
        feed_dict={X: batch_data, Z: sample_Z(1, 100)}
    )
```

3.3 WaveNet GAN

There are a number of more fine implementation details that are not fully explained with only Figure 3.1. To start, we must evaluate the details of the generator network. In its original implementation, WaveNet incrementally makes predictions and, as mentioned before, there are additional attempts to improve this speed, such the work of van den Oord et al. in "Parallel WaveNet: Fast High-Fidelity Speech Synthesis" [11]. Perhaps more crucial than speed, however, is that the generative aspect of the WaveNet implementation makes training the generator weights difficult, as incremental predictions will create a new set of weights based on past probabilities.

As a result, we use only the structure of the WaveNet training for our generative model. This does however introduce a few new factors into the training process. The network itself produces a size (number of samples requested) $\times$ 256 matrix[2]. Each column vector of the output (which has dimension 1×256) contains logit values for each output level. This output requires processing before it can be compared with the standardized real data. The processing block diagram is shown in Figure 3.4. While

[2]Where 256 is the number of quantization channels, determined by the parameter μ in the μ-law softmax layer of WaveNet.

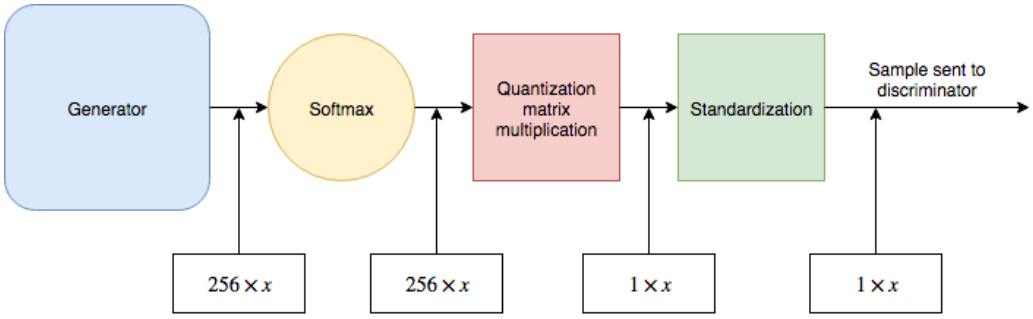


Figure 3.4: Processing steps for WaveNet output.

the original WaveNet structure used a random selection of levels (a non-differentiable operation), we process using this flow chart to ensure that gradients can still be computed between the generator and the output of the discriminator.

The softmax layer converts the logits to pure probabilities, which we will use as weights to simulate the μ-law encoding. This encoding step is achieved through a simple matrix multiplication:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,x} \\ \vdots & \ddots & \\ a_{256,1} & & a_{256,x} \end{bmatrix}_{256\times x} \times \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 254 \\ 255 \end{bmatrix}_{256\times 1}$$

The implementation of this processing is quite straightforward:

```
levels = []
for i in range(quantization_channels):
    levels.append(i)

levels_tensor = tf.reshape(
    tf.constant(levels, dtype=tf.float32),
    [quantization_channels, 1]
```

```
)
G_pre_stand = tf.matmul(tf.nn.softmax(w_prediction), levels_tensor)
mean = tf.reduce_mean(G_pre_stand)
std = reduce_std(G_pre_stand)
G_sample = tf.map_fn(lambda x: (x - mean)/std, G_pre_stand)
```

This `G_sample` variable is then used in the same way as it was in the Simple GAN: as the input to the discriminator network to generate the loss figure for minimization. Again, due to initial results, we will train the generator at a higher frequency than the discriminator.

As before, the discriminator network is set up with 5 hidden layers, using leaky ReLUs as the activation function for the neurons and the sum of the losses from the real and fake logits as the objective to minimize.

3.4 Success Metrics

It is difficult to measure "success" in this project objectively outside of the surveying of human subjects. However, instead of surveying human subjects, we will look at two specific criteria that do not necessarily indicate generation of music that would fool humans, but indicate a baseline success of the system.

1. We want to see loss figures for both the discriminator and generator converging to near-zero. This would indicate the system's ability to prevent convergence to spurious local minima.

2. We would like to see strong peaks in the frequency spectrum of the generated output. This can be achieved by both looking at the Fast Fourier Transform and the spectrogram.

The second metric is particularly important due to the nature of our desired data. In Western music, notes are constructed according to a fixed set of frequencies, as

Table 3.2: Notes and their corresponding frequencies [12]. The letter refers to the pitch name (A, B, C, D, E, F, G), and the number refers to the octave of that note.

A0 (lowest piano note)	**C3**	**C4**	**C5**	**C8** (highest piano note)
27.50 Hz	130.8 Hz	261.6 Hz	523.3 Hz	4186 Hz

shown Table 3.2. As a result, we should see peaks at some of the frequencies that correspond to notes. High amplitudes at those frequencies correspond to the audibility of notes. Looking at the spectrogram should further confirm this, with frequency peaks changing over time, representing changes in note values. Ideally, we would see multiple strong peaks at the same time, meaning the chords (rather than just individual notes) are being generated.

Consider for example the spectrum (Figure 3.5) and spectrogram (Figure 3.6) of a small sample from one of our "true" data points. In this example, we observe a peak around 520 Hz, which corresponds to the C5 frequency in Table 3.2. We further see the high intensity bands of frequency in the lower frequencies of the spectrogram, mainly below 0.1. In this particular spectrogram, we have a sampling frequency of 44.1×10^3 Hz, meaning the cutoff for the highest intensity (around 0.1) corresponds to frequencies in the range of

$$\frac{44.1 \times 10^3 \text{ Hz}}{2} \times 0.1 = 2.205 \times 10^3 \text{ Hz}.$$

This freqeuncy corresponds to approximately a C#7, which is well within the range of a piano [12].

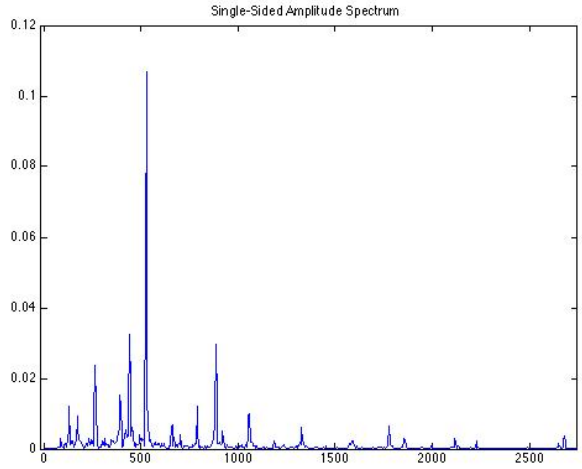


Figure 3.5: Spectrum from a small sample of a Mozart piano work.

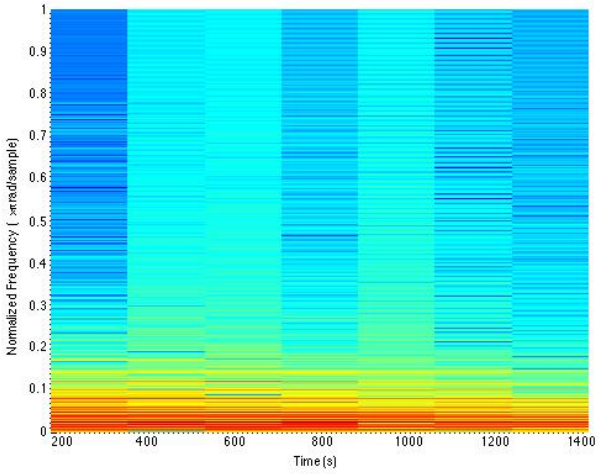


Figure 3.6: Spectrogram from a small sample of a Mozart piano work.

Chapter 4

Results

4.1 Initial Discriminator Testing

As a preliminary test, the discriminator was set up with three layers, 10008 total neurons and tested on 28 samples of Mozart piano sonatas (i.e., only true data). Data was considered and trained on one sample per batch, as described in Section 3.1.This first run was quite promising, showing a dramatic decrease in the loss figure (see Figure 4.1). However, this rapid reduction in the loss appears to be due to the lack of negative samples. Without the non-Mozart data, the discriminator network simply learned the distribution of the inputted samples, overfitting to the positive data. Regardless, this first run was a good indicator of the discriminator network's ability to perform gradient descent and a demonstration of the correct setup for the neural network. Notice that, as expected, the starting loss figure is quite close to our figure of 0.693 nats as being the "random" classifier.

The next steps in the discriminator tuning was to implement training on true and false information with random batching, according to the flow chart described in the methodology. First, we start with a learning rate sweep on the different depths of the neural network. We first observe the cumulative average loss on each for different

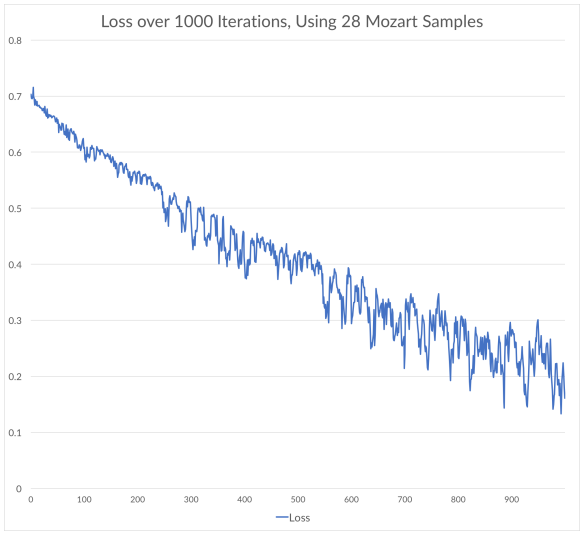


Figure 4.1: Loss with 28 positive samples only

learning rates over a number of layer depths in Figures 4.2, 4.3 and 4.4. These plots paint a promising picture for all of the networks at the learning rate of 0.01, particularly as the depth of the network increases. However, having a higher learning rate means the loss is unstable, as we see in Figures 4.5, 4.6 and 4.7.

While the higher learning rate showed very exciting results for the average loss, it created a great deal of oscillation in the step by step loss. This instability will be a problem in the GAN system, as spikes in the loss of the discriminator will adversely affect the generator. We notice as well that the loss at our lower rate of 0.001 hovered around our 0.693 figure, both on average and step to step, meaning it was ineffective in classifying between real and fake data.

4.2 Simple GAN results

In the case of the Simple GAN described in Section 3.2, the discriminator's learning ability was often greater than the generator's, even in the case where the generator's learning batch size was twenty times the size of the discriminator's learning batch. In addition, both the `AdamOptimizer` and `GradientDescentOptimizer` provided by

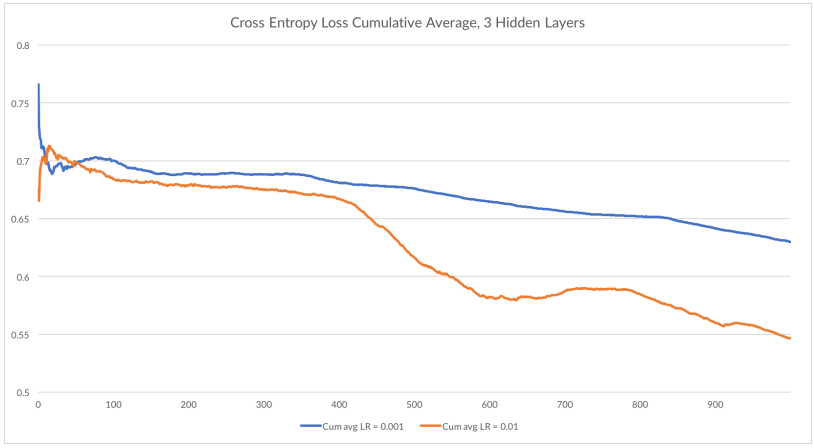


Figure 4.2: 3 hidden layers, cumulative average loss

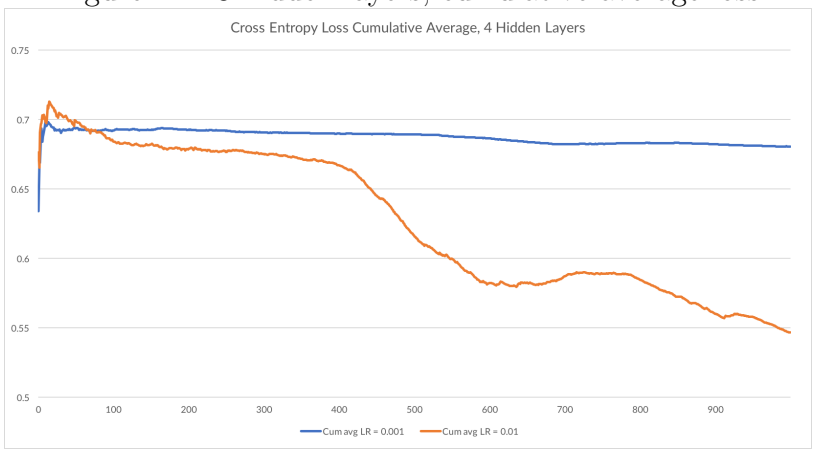


Figure 4.3: 4 hidden layers, cumulative average loss

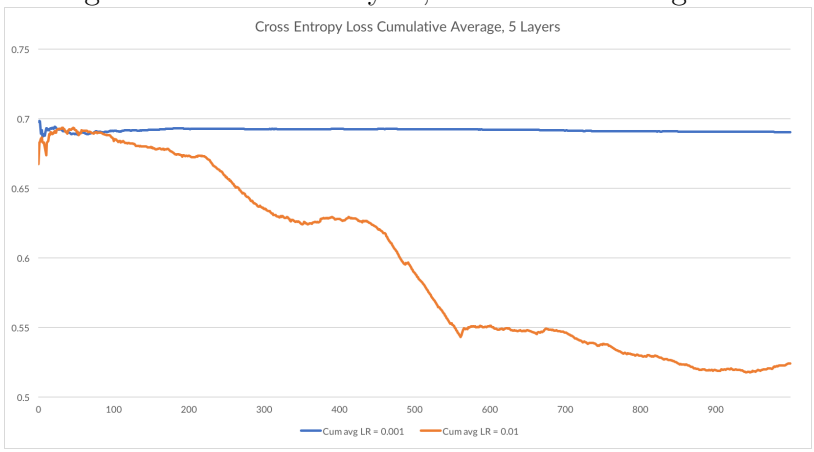


Figure 4.4: 5 hidden layers, cumulative average loss

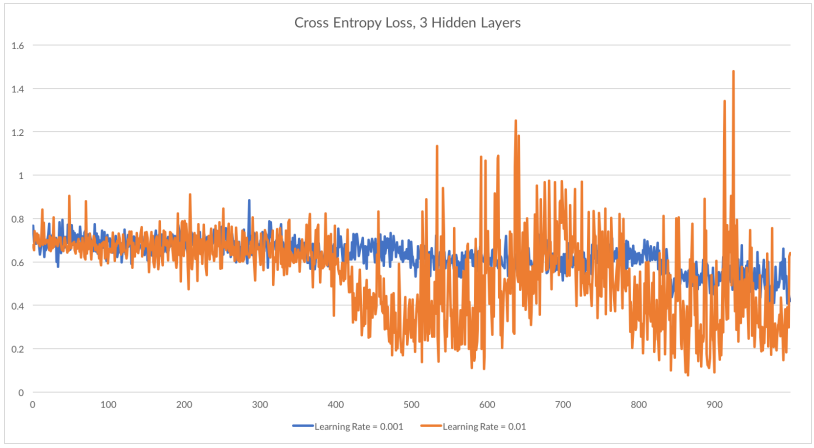


Figure 4.5: 3 hidden layers, loss per step

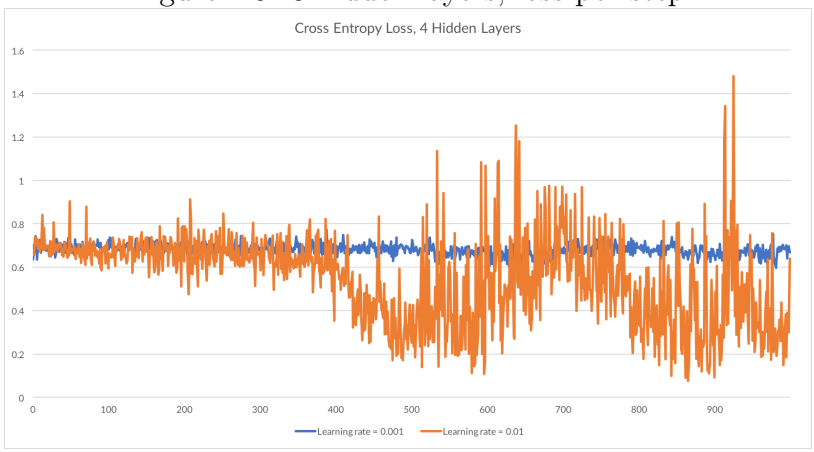


Figure 4.6: 4 hidden layers, loss per step

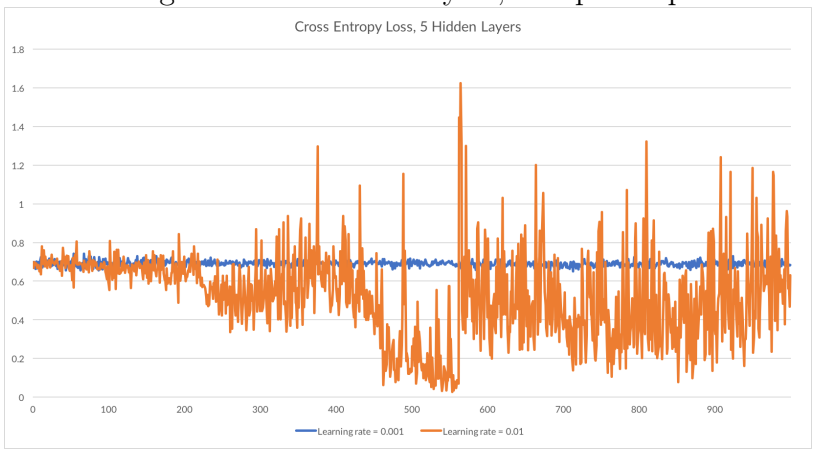


Figure 4.7: 5 hidden layers, loss per step

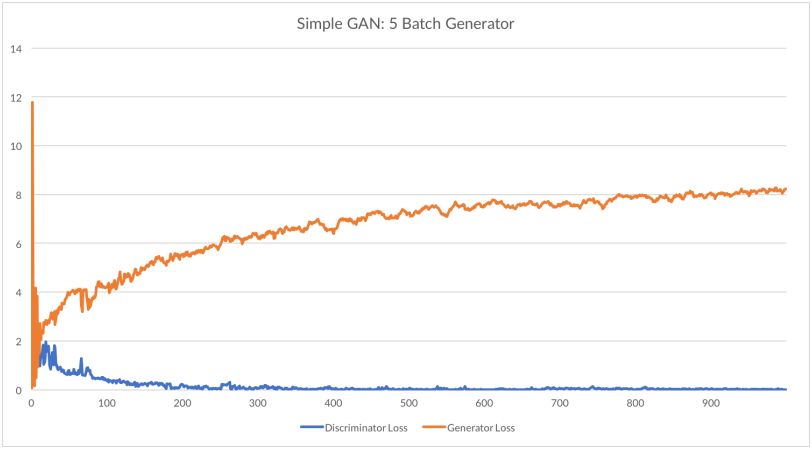


Figure 4.8: Generator Trained 5× the Discriminator

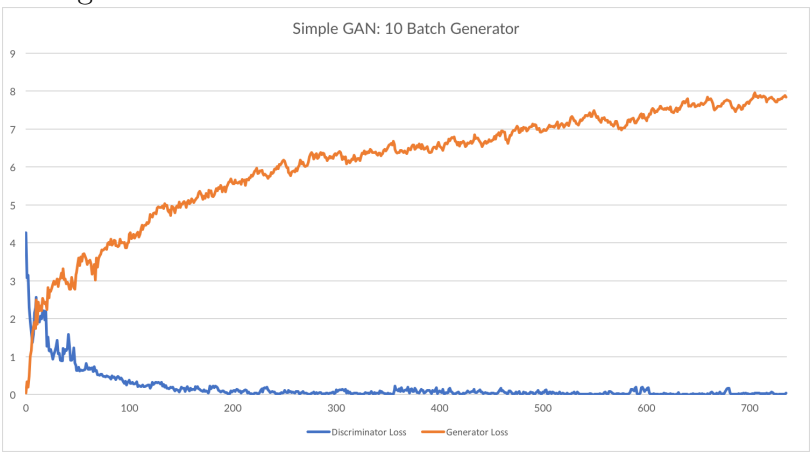


Figure 4.9: Generator Trained 10× the Discriminator

TensorFlow resulted in spurious local minima for the generator's loss function. We can see these results in Figures 4.8 and 4.9. In either scenario, both of the networks seemed to converge quickly, with the discriminator loss hovering near zero, and the generator loss converging to a spurious local minimum.

Oftentimes, this generative system was essentially unsuccessful in generating anything that sounded like piano music. There was one instance in the 10× batch however — either through luck of the stochastic gradient descent or through some sort of overfitting — where an early iteration did produce something that was very clearly mimicking the sound of a piano, albeit covered with some noise. Looking at the spectrum of this "nearly" piano signal in Figure 4.10 shows us the frequency peaks that

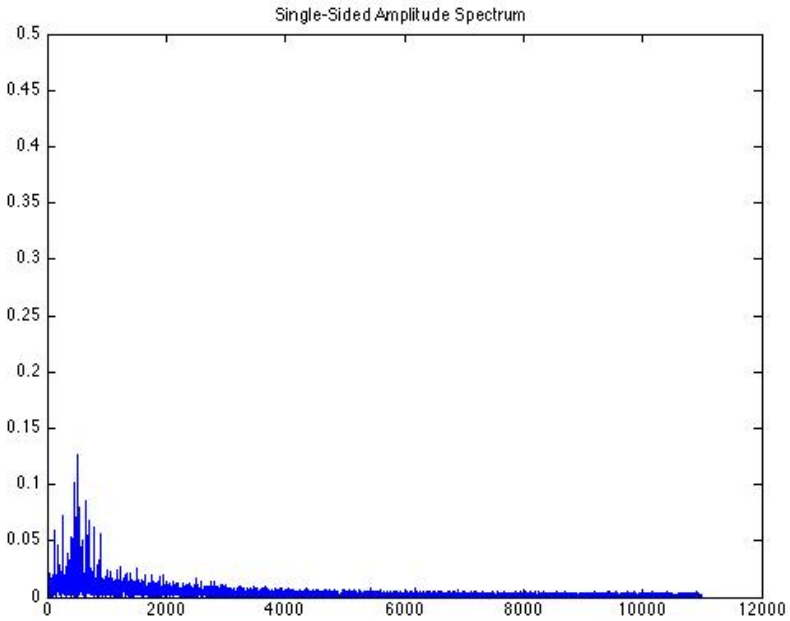


Figure 4.10: "Nearly" Piano Spectrum

we were looking for. We further improve the sound quality of this spectrum with a 15th order Chebyshev low pass filter, as shown in Figure 4.11. Additionally, we can see that the outputted waveform moved through several pitches in the spectrogram in Figure 4.12.

On a number of other occasions, piano-sounding pitches were produced at various stages of the training process. Their respective frequency spectra and spectrograms were similar to that of 4.10, but were not easily reproducible, due to the stochastic nature of our learning algorithm.

Unfortunately, most iterations of the Simple GAN were not this successful in creating piano-like sounds. We hypothesize that the number of nodes and layers was not sufficient for capturing the complexity of the audio files. In Figure 4.13, we see the spectrum generated after training the Simple GAN for 980 iterations. In particular, we see that are no peaks anywhere in the spectrum.

Despite the number of unsuccessful samples generated by Simple GAN, we con-

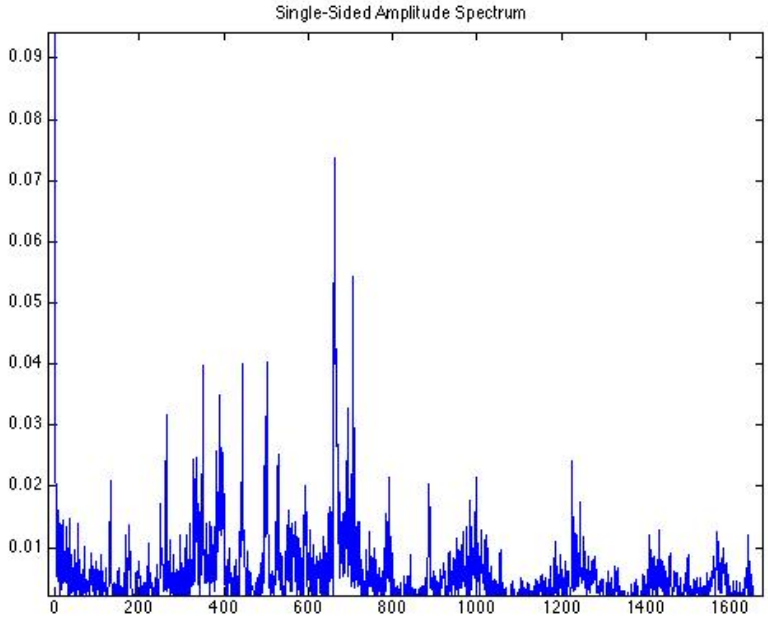


Figure 4.11: Filtered "Nearly" Piano Spectrum (rescaled)

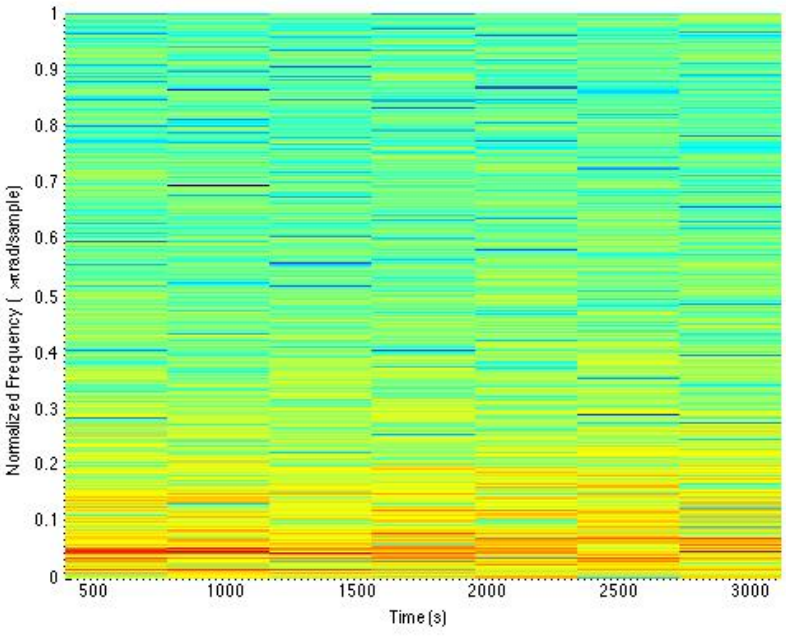


Figure 4.12: Spectrogram of "Nearly" Piano Spectrum

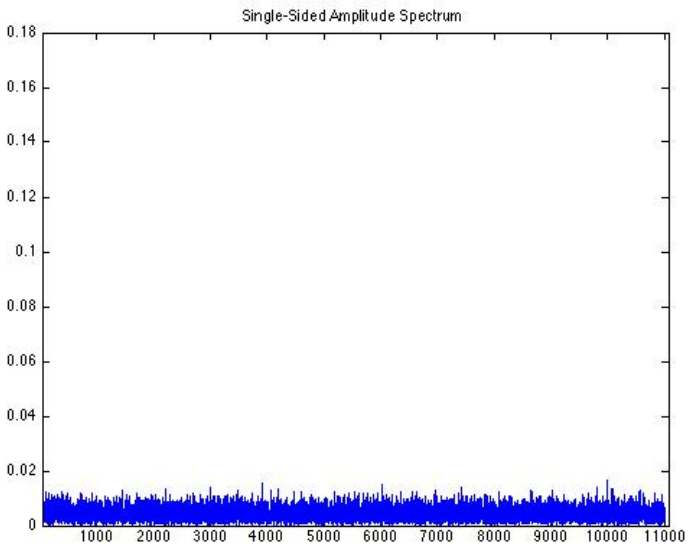


Figure 4.13: Spectrum From Unsuccessful Simple GAN Sample After 980 Iterations

tinue implementation of the WaveNet GAN, as we expect that the Simple GAN's generator was the largest factor in the system's inability to learn and mimic the model, considering the success of the discriminator alone to distinguish between real and fake data, as shown in Section 4.1.

4.3 WaveNet GAN results

The WaveNet GAN was relatively more successful in generating according to our success metrics, although not necessarily so to the human ear. Perhaps most important feature to note is the success of the system in the frequency domain relative to the Simple GAN. We start by running the full WaveNet GAN using 5 times frequency of generator training over 1000 iterations. Figure 4.14 shows the development of spectra the generator samples. Early samples at 100 iterations do not show a clear tendency towards modes at frequencies, yet starting at 300 iterations, we begin to see clear peaks in the spectrum. By the 900th iteration, there is a clear peak at a single frequency point in the spectrum.

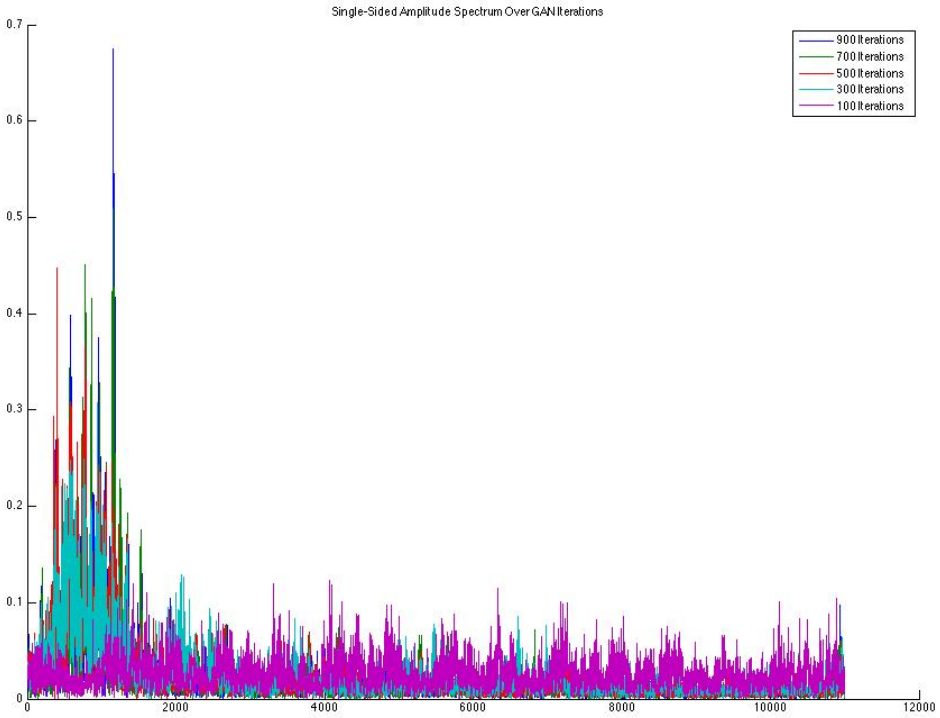


Figure 4.14: Spectra from GAN Samples

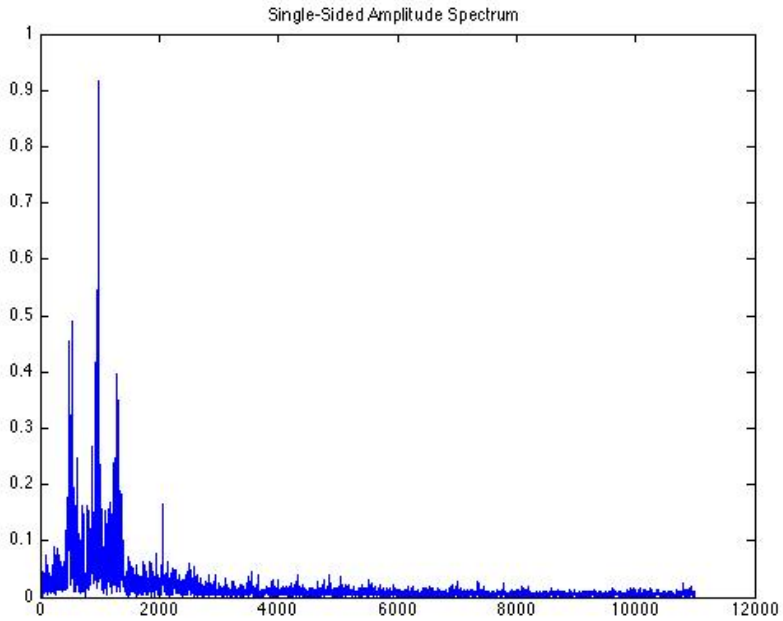


Figure 4.15: Spectra from 2060th Iteration

With 900 iterations and a 5× frequency of generator training, we see that the generator is still unable to fully mimic the model. We increase the number of iterations to 2000 and the frequency of generator training relative to the discriminator training to 8. In Figure 4.15, we see the spectrum following this training method at the 2000th iteration. Although the resulting waveform is hardly piano-like to the ear, we still see that there is a marked improvement in the spectrum.

We also examine the loss figures of this training loop. While Figure 4.16 does seem to show a convergence in loss, looking at the training losses from iterations 100 to 2000 in Figure 4.17 shows that the discriminator and generator are still unable to fully converge. However, these loss figures show much more promising tendency towards convergence to zero compared to those of the Simple GAN.

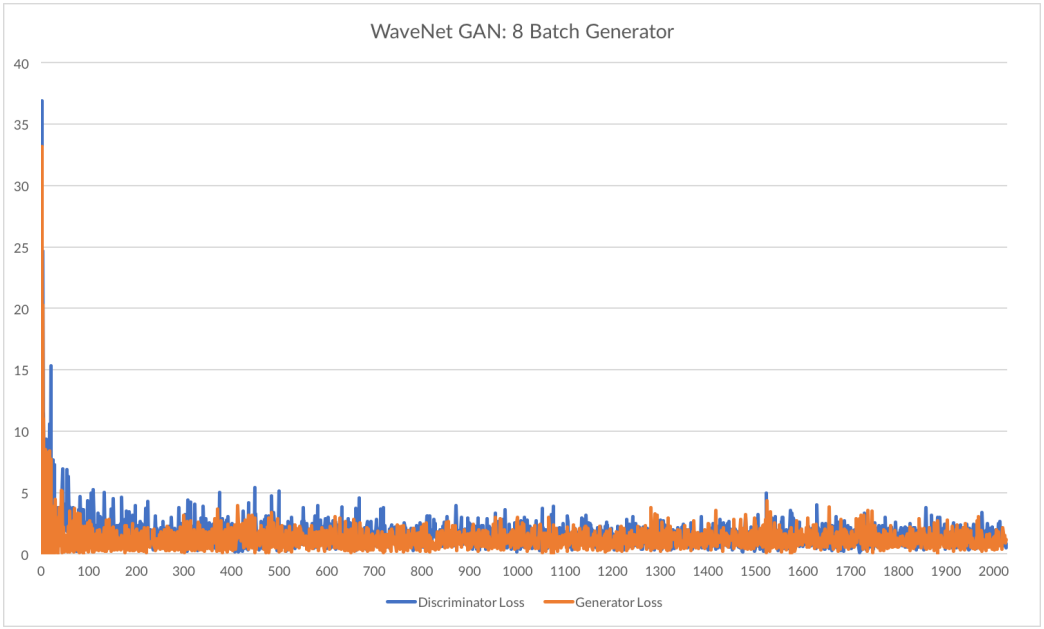


Figure 4.16: Discriminator and Generator Training Losses

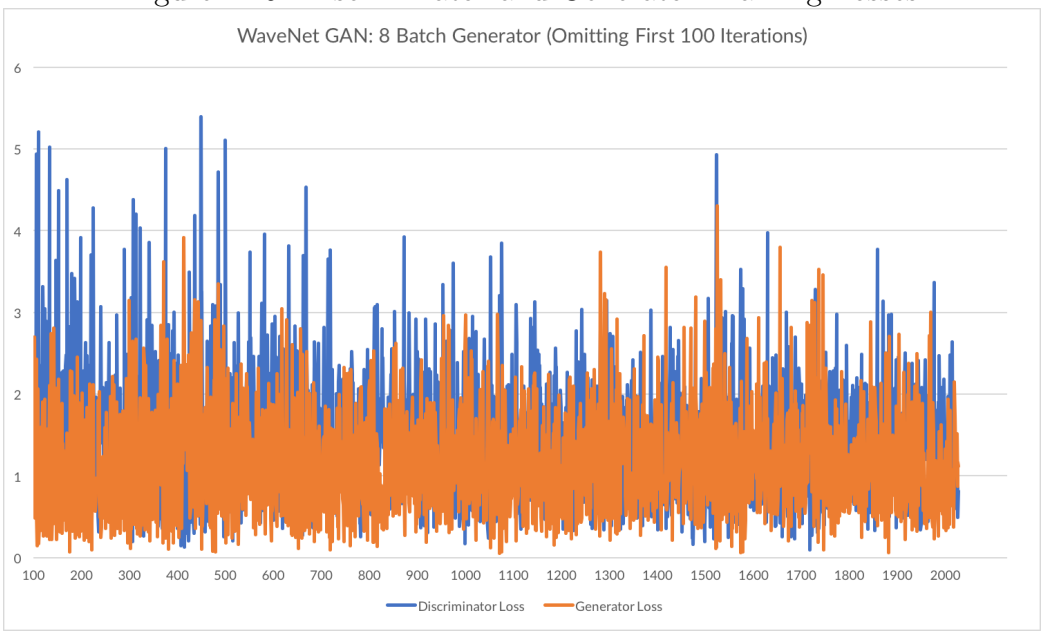


Figure 4.17: Discriminator and Generator Training Losses, Omitting First 100 Steps for Greater Clarity

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The results, particularly from WaveNet GAN, indicate a promising first start on the road to authentic audio generation, and towards further extending the applications of GANs. While the generation was ultimately not successful in creating specifically piano-esque sound, we have shown that Generative Adversarial Networks are capable of creating samples that show tendencies in the frequency domain, not just the time domain.

In fact, this is a particularly novel conclusion: in the case of image processing and generation, which is the standard demonstration of the power of GANs, the main focus is the probability distribution of the "time" domain, which in the case of images, is the value of the pixels. This project shows that, with the proper network structures, GANs can also be used to mimic the frequency domain given only time-based data.

It appears that the mode-collapse issue inherent to GANs is a problem in both time and frequency domains. While the WaveNet GAN was able to create a pitched waveform, it was generally unable to produce something that sounds like a piano, due to the fact that the sound of a piano is more complex than a single pitch at a

frequency point. Every instrument has a different set of overtones that are produced upon striking a key or plucking a string. The production of overtones is what gives different instruments a distinctive timbre aside from the primary pitch that is produced. Among other issues, production of a realistic piano sound will be contingent on the ability to deal with the mode collapse issue.

5.2 Proposed Improvements

5.2.1 Increased Depth and Computation Time

The samples produced for the WaveNet GAN ran for a maximum of 20 hours, given computational resource constraints, while similar projects, such as those in [7] and [13] ran on more powerful hardware for a period of 4 and 18 days, respectively. It is possible that simply running the code for a longer period of time would yield more meaningful and realistic results with the existing implementation of the WaveNet GAN. While we did not see a huge improvement in the loss figures for the discriminator and generator between iterations 300 and 900, we did see an improvement in its spectrum over that time, so further computation may yield better results.

In addition, increasing both the number of quantization channels for the generator and increasing the depths of the generator and discriminator may have yielded better results. These steps, coupled with additional tuning and rework of other hyperparameters (such as nodes per layer and learning rate), could also lead to increased realism and quality, but are again dependent on more processing power.

As seen in Figure 4.17, the training losses for both the discriminator and generator were unable to converge. In fact, empirical results show that stochastic gradient descent with ReLU activation functions are usually unable to converge to ground truth parameters, even for less complex networks [14]. In these scenarios, we must consider either changing our learning algorithm or changing the landscape of our

objective function.

It may be possible to redesign the loss function of the generator and the discriminator to have guarantees of no spurious local minima while maintaining the ability to estimate gradients using samples. Some initial attempts to make objective functions with these promises can be found in "Learning One-hidden-layer Neural Networks with Landscape Design", though their designed objective function works only under very specific assumptions and conditions [15]. Further, it may be possible to utilize other learning algorithms which can prevent convergence to spurious local minima.

5.2.2 Signal Processing for Generator Improvement

Given the nature of our desired generated samples, we can consider adding traditional elements of digital signal processing to the output of our generator. As was previously seen in Section 4.2, passing the generated sample through a low pass filter improved the quality of our signal. However, filters and other signal processing steps (aside from the data reshaping in Figure 3.4) were never used in the training process. The addition of a signal processing step within training, like in Figure 5.1, could improve the generator's output step by step and therefore further challenge the discriminator.

It is also possible to factor the output of the discriminator's decision into adaptive filters for the signal processing block if the analytic and differentiable solutions for the filters are known (which is the case for many less-complex filters). In this way, the signal processing step is factored into the landscape of the objective filter. While this project used filters to remove noise, and a low pass filter to remove noise may seem most intuitive at first, other elements may be taken into consideration: Donahue et al. used an upsampling procedure based on a learned filter from the generator's parameters [7].

An interesting result from [7] was that the "WaveGAN" network, which trained on the time domain, was more successful in mimicking the frequency domain of the

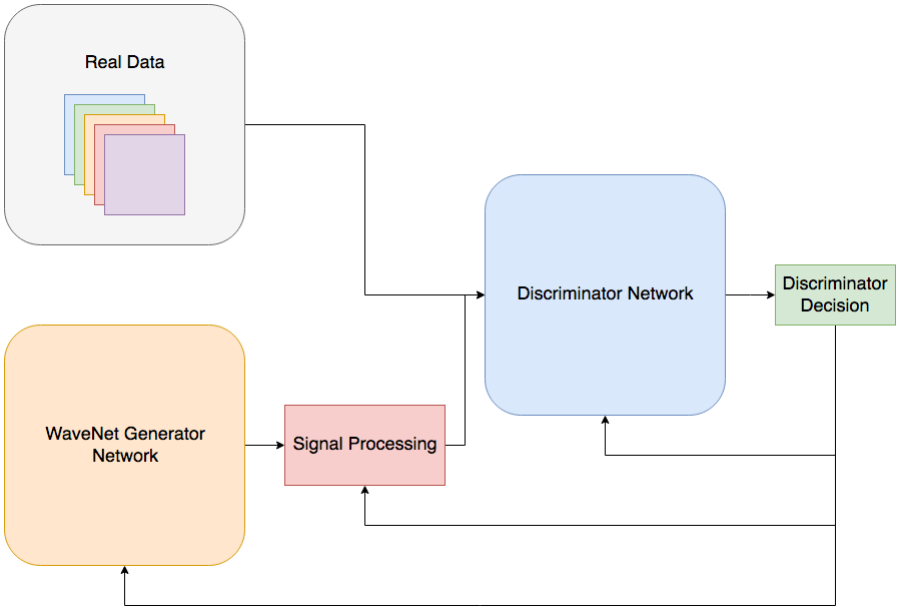


Figure 5.1: Block diagram with DSP step.

true data than their "SpecGAN" implementation, which trained on the frequency domain using spectrograms as images. So while learning on the frequency domain may not immediately be successful (i.e., through spectrograms), frequency domain based processing on time domain data may still overall improve the system.

5.2.3 Potential Mode Collapse Solutions

Mode collapse is a well known and critical problem in GANs. Although our real data distribution is complex and multimodal, yet the generator fails to capture the complexity of the data and produces samples that are identical or nearly identical. We faced this problem in our training, with later iterations failing to improve the generated waveform.

There has been much discussion on how to fix the mode collapse issue in the training of GANs, and employing some of the following proposed solutions may improve the results of this project.

1. Multiple generators (coined Mixture GAN): instead of just one generator net-

work, multiple generators are used along with a discriminator network and a classifier network. The output of a generator is randomly selected as the fake input to the discriminator. As with standard GANs, the discriminator attempts to distinguish fake and real while the classifier network attempts to determine which generator network was randomly selected as the fake data [16]. It would be particularly interesting to see if there was a relationship between the number of large frequency modes and number of generators used.

2. Wasserstein GAN (WGAN): WGAN uses the Wasserstein metric as a loss function, and has empirically been shown to prevent mode collapse [17].

3. Improve diversity through minibatches: because the discriminator processes each example independently, there is no coordination between its gradients. When GANs are near collapse, the gradient of the discriminator may point in similar directions for many points. By having the discriminator look at multiple examples in combination, we can potentially avoid collapse of the generator [2].

5.2.4 Progressive Growing

The work referenced to in Section 1.2 (celebrity image generation) is fully explained in "Progressive Growing of GANs for Improved Quality, Stability, and Variation" [13], in which they propose a system in which the resolution of the generator and discriminator grows with the progression of training (shown in Figure 5.2). There are many benefits to the progressive training, such as increased stability of generation early in training and reduced training time.

Progressive growing takes advantage of slowly increasing the resolution in images, and we must therefore find an analog to resolution for audio. One proposed definition is to divide the audio sample into the number of desired levels (i.e. 16 to mimic the 4×4 image), and take the average for each divided segment. Since audio waveforms

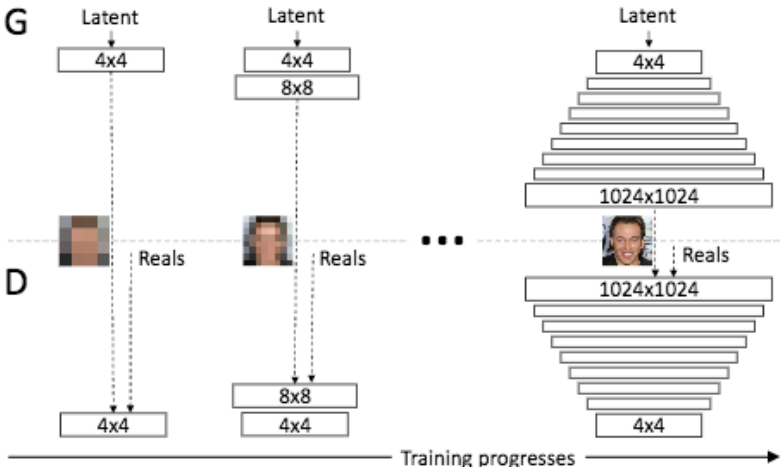


Figure 5.2: Progressive growing through training loop.

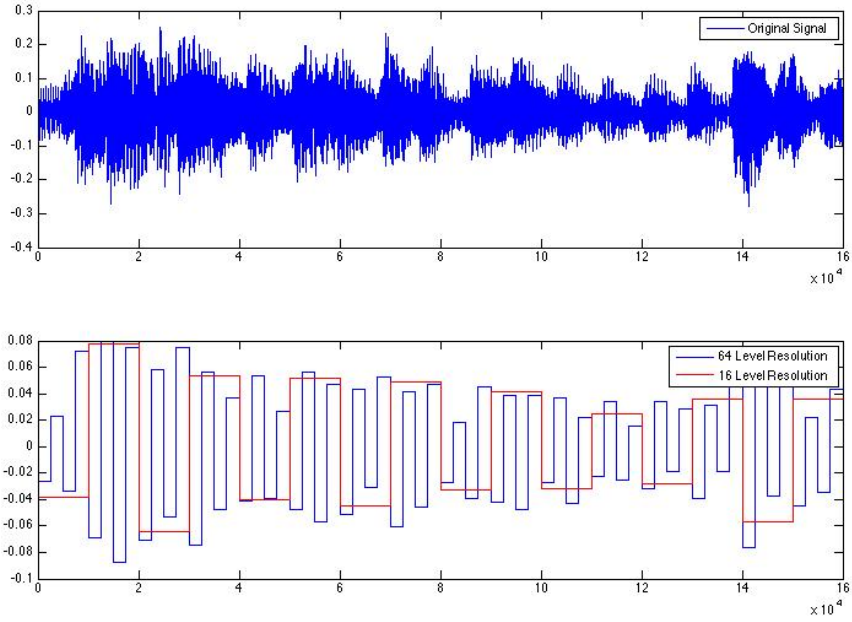


Figure 5.3: 16 and 64 "Pixel" Resolution for Sample Waveform.

often approximately average to 0, we take the average of the absolute value of the points in a segment, and flip the sign between each "pixel", i.e., for some signal x with length l and a desired p number of pixels, the held value of the k-th pixel is defined to be

$$f_k = (-1)^k \left(\frac{\sum_{i=0}^{l/p} \left| x_{i+(k \times l/p)} \right|}{\frac{l}{p}} \right),$$

where the value of f_k is held for a duration of $\frac{l}{p}$ samples. We can see this in practice on a real signal in Figure 5.3.

5.3 Final Discussion

We present WaveNet GAN, which is one of the first applications of GANs on raw audio waveforms. We have seen that WaveNet GAN can produce samples with a meaningful frequency distribution. While WaveNet GAN did not capture the distribution of an instrument or a composer, it did show that GANs have the possibility to capture these complex datasets and we propose a number of future steps towards improving our system.

Appendix A

Code Listing

A.1 Simple GAN Code

```
from __future__ import print_function
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from datetime import datetime
from random import *
import argparse
import json
import os
import sys
import time

import librosa
import tensorflow as tf
```

```
import numpy as np

from wavenet import AudioReader

def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / tf.sqrt(in_dim / 2.)

    return tf.random_normal(shape=size, stddev=xavier_stddev)

def standardize(samples):
    mean = np.mean(samples)
    std = np.std(samples)

    standardized = []

    for d in samples:
        standardized.append(float(d-mean)/std)

    return standardized

def process(samples):
    reshape = []
    for d in samples:
        reshape.append(d[0])

    return reshape

def write_wav(waveform, sample_rate, filename):
```

```
    y = np.array(waveform)
    librosa.output.write_wav(filename, y, sample_rate)
    print('Updated␣wav␣file␣at␣{}'.format(filename))

w1 = 22000
w2 = 18335
w3 = 14668
w4 = 11001
w5 = 7334
w6 = 3667
w7 = 1

# Discriminator Variables
X = tf.placeholder(tf.float32, shape=[None, w1], name='X')

D_W1 = tf.Variable(xavier_init([w1, w2]), name='D_W1')
D_b1 = tf.Variable(tf.zeros(shape=[w2]), name='D_b1')

D_W2 = tf.Variable(xavier_init([w2, w3]), name='D_W2')
D_b2 = tf.Variable(tf.zeros(shape=[w3]), name='D_b2')

D_W3 = tf.Variable(xavier_init([w3, w4]), name='D_W3')
D_b3 = tf.Variable(tf.zeros(shape=[w4]), name='D_b3')

D_W4 = tf.Variable(xavier_init([w4, w5]), name='D_W4')
D_b4 = tf.Variable(tf.zeros(shape=[w5]), name='D_b4')

D_W5 = tf.Variable(xavier_init([w5, w6]), name='D_W5')
D_b5 = tf.Variable(tf.zeros(shape=[w6]), name='D_b5')
```

```
D_W6 = tf.Variable(xavier_init([w6, w7]), name='D_W6')
D_b6 = tf.Variable(tf.zeros(shape=[w7]), name='D_b6')

theta_D = [D_W1, D_W2, D_W3, D_W4, D_W5, D_W6,
    D_b1, D_b2, D_b3, D_b4, D_b5, D_b6]

# Generator Variables
Z = tf.placeholder(tf.float32, shape=[None, 100], name='Z')

G_W1 = tf.Variable(xavier_init([100, w6]), name='G_W1')
G_b1 = tf.Variable(tf.zeros(shape=[w6]), name='G_b1')

G_W2 = tf.Variable(xavier_init([w6, w5]), name='G_W2')
G_b2 = tf.Variable(tf.zeros(shape=[w5]), name='G_b2')

G_W3 = tf.Variable(xavier_init([w5, w4]), name='G_W3')
G_b3 = tf.Variable(tf.zeros(shape=[w4]), name='G_b3')

G_W4 = tf.Variable(xavier_init([w4, w3]), name='G_W4')
G_b4 = tf.Variable(tf.zeros(shape=[w3]), name='G_b4')

G_W5 = tf.Variable(xavier_init([w3, w2]), name='G_W5')
G_b5 = tf.Variable(tf.zeros(shape=[w2]), name='G_b5')

G_W6 = tf.Variable(xavier_init([w2, w1]), name='G_W6')
G_b6 = tf.Variable(tf.zeros(shape=[w1]), name='G_b6')

theta_G = [G_W1, G_W2, G_W3, G_W4, G_W5, G_W6,
```

```
    G_b1, G_b2, G_b3, G_b4, G_b5, G_b6]

# Generator network
def generator(z):
    G_h1 = tf.nn.leaky_relu(tf.matmul(z, G_W1) + G_b1)
    G_h2 = tf.nn.leaky_relu(tf.matmul(G_h1, G_W2) + G_b2)
    G_h3 = tf.nn.leaky_relu(tf.matmul(G_h2, G_W3) + G_b3)
    G_h4 = tf.nn.leaky_relu(tf.matmul(G_h3, G_W4) + G_b4)
    G_h5 = tf.nn.leaky_relu(tf.matmul(G_h4, G_W5) + G_b5)

    G_log_prob = tf.matmul(G_h5, G_W6) + G_b6
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob

# Discriminator network
def discriminator(x):
    D_h1 = tf.nn.leaky_relu(tf.matmul(x, D_W1) + D_b1)
    D_h2 = tf.nn.leaky_relu(tf.matmul(D_h1, D_W2) + D_b2)
    D_h3 = tf.nn.leaky_relu(tf.matmul(D_h2, D_W3) + D_b3)
    D_h4 = tf.nn.leaky_relu(tf.matmul(D_h3, D_W4) + D_b4)
    D_h5 = tf.nn.leaky_relu(tf.matmul(D_h4, D_W5) + D_b5)

    D_logit = tf.matmul(D_h5, D_W6) + D_b6
    D_prob = tf.nn.sigmoid(D_logit)

    return D_prob, D_logit

# Uniform sampler
```

```
def sample_Z(m, n):
    return np.random.uniform(-1., 1., size=[m, n])

# Define real and fake logits
G_sample = generator(Z)
D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)

# Define loss figures to minimize for networks
D_loss_real = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(D_logit_real),
        logits=D_logit_real
    )
)
D_loss_fake = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.zeros_like(D_logit_fake),
        logits=D_logit_fake
    )
)
D_loss = D_loss_real + D_loss_fake
G_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(D_logit_fake),
        logits=D_logit_fake
    )
)
```

```
# Only update D(X)'s parameters, so var_list = theta_D
D_solver = tf.train.GradientDescentOptimizer(
    learning_rate=3e-3)
    .minimize(D_loss, var_list=theta_D)
# Only update G(X)'s parameters, so var_list = theta_G
G_solver = tf.train.GradientDescentOptimizer(
    learning_rate=3e-3)
    .minimize(G_loss, var_list=theta_G)

coord = tf.train.Coordinator()
sess = tf.Session()

# Set up real data sampler
directory = './sampleTrue'
reader = AudioReader(directory, coord, sample_rate = 22000,
    gc_enabled=False, receptive_field=1000,
    sample_size=21000, silence_threshold=0.05)
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
reader.start_threads(sess)

init = tf.global_variables_initializer()
sess.run(init)

# Main training loop
for it in range(1000):
    batch_data = []

    start_time = time.time()
```

```
# Sample real data
data = sess.run(reader.dequeue(1))
while (len(data[0]) < w1):
    data = sess.run(reader.dequeue(1))

data = np.array(data[0])
samples = process(data)
batch_data.append(samples)

# Train generator
for g_batch in range(20):
    _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample

# Train discriminator
for d_batch in range(1):
    _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: batch_

duration = time.time() - start_time

# Produce sample on every 20th iteration
if (it % 20 == 0):
    waveform = []
    waveform = np.reshape(sess.run(G_sample, feed_dict={Z: sample_Z(1
    print(waveform)
    name = 'simplegenerate-' + str(it) + '.wav'
    write_wav(waveform, 22000, name)
```

A.2 WaveNet GAN Code

```
from __future__ import print_function
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from datetime import datetime
from random import *
import argparse
import json
import os
import sys
import time

import librosa
import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import ffnn

from wavenet import WaveNetModel, AudioReader, optimizer_factory

# GAN params
NUM_EPOCHS = 10

# Wavenet params
BATCH_SIZE = 1
```

```
DATA_DIRECTORY = './VCTK-Corpus'
LOGDIR_ROOT = './logdir'
CHECKPOINT_EVERY = 50
NUM_STEPS = int(1e5)
LEARNING_RATE = 3e-2
WAVENET_PARAMS = './wavenet_params.json'
STARTED_DATESTRING = "{0:%Y-%m-%dT%H-%M-%S}".format(datetime.now())
SAMPLE_SIZE = 100000
L2_REGULARIZATION_STRENGTH = 0
SILENCE_THRESHOLD = 0.3
EPSILON = 0.001
MOMENTUM = 0.9
MAX_TO_KEEP = 5
METADATA = False

def placeholder_inputs(batch_size):
    # Generate placeholder variables for input tensors
    inputs_placeholder = tf.placeholder(tf.float32,
        shape=(batch_size, w1),
        name='inputs_placeholder'
    )
    labels_placeholder = tf.placeholder(tf.int32,
        shape=(batch_size),
        name='labels_placeholder'
    )
    return inputs_placeholder, labels_placeholder

def fill_feed_dict(batch_data, label_data, inputs_pl, labels_pl):
    # Feed dict for placeholders from placeholder_inputs()
```

```
    feed_dict = {
        inputs_pl: batch_data,
        labels_pl: label_data
    }

    return feed_dict

def get_generator_input_sampler():
    return lambda mu, sigma, n: np.random.normal(
                mu, sigma, size=[1, n]
            )

def standardize(samples):
    mean = np.mean(samples)
    std = np.std(samples)

    standardized = []

    for d in samples:
        standardized.append(float(d-mean)/std)

    return standardized

def process(samples):
    standardized = standardize(samples)

    return standardized
```

```
def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / tf.sqrt(in_dim / 2.)

    return tf.random_normal(shape=size, stddev=xavier_stddev)

def get_arguments():
    # omitted for clarity

def discriminator(x):
    D_h1 = tf.nn.leaky_relu(tf.matmul(x, D_W1) + D_b1)
    D_h2 = tf.nn.leaky_relu(tf.matmul(D_h1, D_W2) + D_b2)
    D_h3 = tf.nn.leaky_relu(tf.matmul(D_h2, D_W3) + D_b3)
    D_h4 = tf.nn.leaky_relu(tf.matmul(D_h3, D_W4) + D_b4)
    D_h5 = tf.nn.leaky_relu(tf.matmul(D_h4, D_W5) + D_b5)

    D_logit = tf.matmul(D_h5, D_W6) + D_b6
    D_prob = tf.nn.sigmoid(D_logit)

    return D_prob, D_logit

def write_wav(waveform, sample_rate, filename):
    y = np.array(waveform)
    librosa.output.write_wav(filename, y, sample_rate)
    print('Updated␣wav␣file␣at␣{}'.format(filename))

def reduce_var(x, axis=None, keepdims=False):
    m = tf.reduce_mean(x, axis=axis, keep_dims=True)
    devs_squared = tf.square(x - m)
```

```
    return tf.reduce_mean(devs_squared,
            axis=axis,
            keep_dims=keepdims
        )

def reduce_std(x, axis=None, keepdims=False):
    return tf.sqrt(reduce_var(x, axis=axis, keepdims=keepdims))

with tf.Graph().as_default():
    coord = tf.train.Coordinator()
    sess = tf.Session()

    w1 = 22000
    w2 = 18335
    w3 = 14668
    w4 = 11001
    w5 = 7334
    w6 = 3667
    w7 = 1

    # Discriminator Variables
    D_W1 = tf.Variable(xavier_init([w1, w2]), name='D_W1')
    D_b1 = tf.Variable(tf.zeros(shape=[w2]), name='D_b1')

    D_W2 = tf.Variable(xavier_init([w2, w3]), name='D_W2')
    D_b2 = tf.Variable(tf.zeros(shape=[w3]), name='D_b2')

    D_W3 = tf.Variable(xavier_init([w3, w4]), name='D_W3')
    D_b3 = tf.Variable(tf.zeros(shape=[w4]), name='D_b3')
```

```
D_W4 = tf.Variable(xavier_init([w4, w5]), name='D_W4')
D_b4 = tf.Variable(tf.zeros(shape=[w5]), name='D_b4')

D_W5 = tf.Variable(xavier_init([w5, w6]), name='D_W5')
D_b5 = tf.Variable(tf.zeros(shape=[w6]), name='D_b5')

D_W6 = tf.Variable(xavier_init([w6, w7]), name='D_W6')
D_b6 = tf.Variable(tf.zeros(shape=[w7]), name='D_b6')

theta_D = [D_W1, D_W2, D_W3, D_W4, D_W5, D_W6,
    D_b1, D_b2, D_b3, D_b4, D_b5, D_b6]

args = get_arguments()

# Load parameters from wavenet params json file
with open(args.wavenet_params, 'r') as f:
    wavenet_params = json.load(f)

quantization_channels = wavenet_params['quantization_channels']

# Intialize generator WaveNet
G = WaveNetModel(
    batch_size=1,
    dilations=wavenet_params["dilations"],
    filter_width=wavenet_params["filter_width"],
    residual_channels=wavenet_params["residual_channels"],
    dilation_channels=wavenet_params["dilation_channels"],
    skip_channels=wavenet_params["skip_channels"],
```

```
    quantization_channels=
        wavenet_params["quantization_channels"],
    use_biases=wavenet_params["use_biases"],
    initial_filter_width=
        wavenet_params["initial_filter_width"])

gi_sampler = get_generator_input_sampler()

# White noise generator params
white_mean = 0
white_sigma = 1
white_length = 27117

Z = tf.placeholder(tf.float32,
    shape=[None, white_length], name='Z'
)

# Initialize generator
_, w_prediction = G.loss(input_batch=Z, name='generator')

theta_G = tf.trainable_variables(scope='wavenet')

X = tf.placeholder(tf.float32, shape=[None, w1], name='X')

init = tf.global_variables_initializer()
sess.run(init)

# Quantization matrix multiplication
levels = []
```

```
for i in range(quantization_channels):
    levels.append(i)

levels_tensor = tf.reshape(
    tf.constant(levels, dtype=tf.float32),
    [quantization_channels, 1]
)
G_pre_stand = tf.matmul(tf.nn.softmax(w_prediction),
    levels_tensor
)

# Standardization
mean = tf.reduce_mean(G_pre_stand)
std = reduce_std(G_pre_stand)
G_sample = tf.map_fn(lambda x: (x - mean)/std, G_pre_stand)

# Define real and fake logits
D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(
    tf.reshape(G_sample, [1, w1])
)

# Define loss figures to minimize for networks
D_loss_real = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(D_logit_real),
        logits=D_logit_real
    )
)
```

```
D_loss_fake = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.zeros_like(D_logit_fake),
        logits=D_logit_fake
    )
)
D_loss = D_loss_real + D_loss_fake
G_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(D_logit_fake),
        logits=D_logit_fake
    )
)

# Only update D(X)'s parameters, so var_list = theta_D
D_solver = tf.train.GradientDescentOptimizer(
    learning_rate=3e-3)
    .minimize(D_loss, var_list=theta_D)
# Only update G(X)'s parameters, so var_list = theta_G
G_solver = tf.train.GradientDescentOptimizer(
    learning_rate=3e-3)
    .minimize(G_loss, var_list=theta_G)

# Set up real data sampler
directory = './sampleTrue'
reader = AudioReader(directory, coord, sample_rate = 22000,
    gc_enabled=False, receptive_field=1000,
    sample_size=21000, silence_threshold=0.05
)
```

```
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
reader.start_threads(sess)

# Main training loop
for it in range(5000):
    batch_data = []

    start_time = time.time()

    # Sample real data
    data = sess.run(reader.dequeue(1))
    while (len(data[0]) < w1):
        data = sess.run(reader.dequeue(1))

    data = np.array(data[0])
    samples = process(data)
    batch_data.append(samples)

    # Train generator
    for g_batch in range(8):
        white_noise = gi_sampler(
            white_mean, white_sigma, white_length
        )
        _, G_loss_curr = sess.run([G_solver, G_loss],
            feed_dict={Z: white_noise}
        )

    # Train discriminator
    for d_batch in range(1):
```

```
        _, D_loss_curr = sess.run([D_solver, D_loss],
            feed_dict={X: batch_data, Z: white_noise}
        )

    duration = time.time() - start_time

    # Produce sample on every 20th iteration
    if (it % 20 == 0):
        white_noise = gi_sampler(
            white_mean, white_sigma, white_length
        )
        waveform = []
        waveform = np.reshape(sess.run(G_sample,
            feed_dict={Z: white_noise}), [w1]
        )
        name = 'fullgenerate-' + str(it) + '.wav'
        write_wav(waveform, 22000, name)
```

A.3 WaveNet Model

The WaveNet model was developed by Igor Babuschkin and is available freely on GitHub: `https://github.com/ibab/tensorflow-wavenet`. From this implementation, `wavenet/model.py` was used prominently in this project, with only one small change in the `loss` function. Lines 680 to the end of `model.py` have been adjusted to:

```
else:
    # L2 regularization for all trainable parameters
    l2_loss = tf.add_n([tf.nn.l2_loss(v)
        for v in tf.trainable_variables()
```

```
        if not('bias' in v.name)])

    # Add the regularization term to the loss
    total_loss = (reduced_loss + l2_regularization_strength * l2_loss)

    tf.summary.scalar('l2_loss', l2_loss)
    tf.summary.scalar('total_loss', total_loss)

    return [total_loss, prediction]
```

Bibliography

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," Jun. 2014, arXiv:1406.2661 [stat.ML].

[2] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved Techniques for Training GANs," Jun. 2016, arXiv:1606.03498 [cs.LG].

[3] C. Metz and K. Collins. (2018, Jan.) How an A.I. 'Cat-and-Mouse Game' Generates Believable Fake Photos. The New York Times. [Online]. Available: {https://www.nytimes.com/interactive/2018/01/02/technology/ai-generated-photos.html}

[4] Sonata-Allegro Form. [Online]. Available: {https://rampages.us/mhis243/lectures/lesson-8/sonata-allegro-form/}

[5] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "SampleRNN: An Unconditional End-to-End Neural Audio Generation Model," Dec. 2016, arXiv:1612.07837 [cs.SD].

[6] C. Walder, "Modelling Symbolic Music: Beyond the Piano Roll," Jun. 2016, arXiv:1606.01368 [cs.SD].

[7] C. Donahue, J. McAuley, and M. Puckette, "Synthesizing Audio with Generative Adversarial Networks," Feb. 2018, arXiv:1802.04208 [cs.SD].

[8] A. van den 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," Sep. 2016, arXiv:1609.03499v2 [cs.SD].

[9] D. Nag. (2017, Feb.) Generative Adversarial Networks (GANs) in 50 lines of code (PyTorch). Medium. [Online]. Available: {https://medium.com/@devnag/generative-adversarial-networks-gans-in-50-lines-of-code-pytorch-e81b79659e3f}

[10] A. van der Oord and Zenm H. (2016, Sep.) WaveNet: A Generative Model for Raw Audio. DeepMind. [Online]. Available: {https://deepmind.com/blog/wavenet-generative-model-raw-audio/}

[11] A. van den Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. van den Driessche, E. Lockhart, L. C. Cobo, F. Stimberg, N. Casagrande, D. Grewe, S. Noury, S. Dieleman, E. Elsen, N. Kalchbrenner, H. Zen, A. Graves, H. King, T. Walters, D. Belov, and D. Hassabis, "Parallel WaveNet: Fast High-Fidelity Speech Synthesis," Nov. 2017, arXiv:1711.10433 [cs.LG].

[12] Note Frequencies. seventhstring. [Online]. Available: {https://www.seventhstring.com/resources/notefrequencies.html}

[13] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive Growing of GANs for Improved Quality, Stability, and Variation," Oct. 2017, arXiv:1710.10196 [cs.NE].

[14] R. Livni, S. Shalev-Shwartz, and O. Shamir, "On the Computational Efficiency of Training Neural Networks," Oct. 2014, arXiv:1410.1141 [cs.LG].

[15] R. Ge, J. D. Lee, and T. Ma, "Learning One-hidden-layer Neural Networks with Landscape Design," Nov. 2017, arXiv:1711.00501 [cs.LG].

[16] Q. Hoang, T. Dinh Nguyen, T. Le, and D. Phung, "Multi-Generator Generative Adversarial Nets," Aug. 2017, arXiv:1708.02556 [cs.LG].

[17] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," Jan. 2017, arXiv:1701.07875 [stat.ML].