RNN-GANs for Generating 8-bit Lofi Music

Nathan T. Hunsberger
Vanderbilt University
CS 8395: Representation Learning
nathan.t.hunsberger@vanderbilt.edu

Berke K. Lunstad
Vanderbilt University
CS 8395: Representation Learning
berke.k.lunstad@vanderbilt.edu

Shivam Vohra
Vanderbilt University
CS 8395: Representation Learning
shivam.vohra@vanderbilt.edu

Abstract

This paper highlights an attempt to reduce the size of encoding music such that a GAN can generate new music sequences. We explore a novel approach to sound encoding based on the Fast Fourier Transform to encode wav files into trainable csv data. We then develop an RNN-GAN focused on learning sequences of music and present it's results in generating 8-bit lofi style wav files.

1 Introduction

Generative Adversarial Networks (GANs) have seen increased interest amongst researchers and engineers since their introduction in 2014 [4]. They are popular for many reasons, namely their ability to generate new data that seemingly seems realistic. Generally, GANs have been used to generate image data. This is something GANs are particularly strong for, as they can recognize patterns in "real" images and use these patterns to generate "real-looking" images of their own. More recently, GANs have been used to generate other media, such as MuseGAN, which generates sequences of music using a handful of different instruments [3]. MuseGAN, and other music generating GANs, are an extension on top of the GAN architecture that introduce concepts taken from Recursive Neural Networks (RNNs) like feed-forward networks. We present an approach that uses a similar architecture to MuseGAN with a novel sound representation format, straying away from the traditional MIDI format that MuseGAN uses.

1.1 MIDI Music Format

MuseGAN, and other music generating GANs, commonly use the MIDI format to represent sound. MIDI files are unique from traditional music formats in that the actual audio data is not stored; rather, the files store musical notes and their timing on a series of channels. Each channel represents an instrument that is expected to play that the notes on the respective channel. MIDI has a few different formats, ranging from one single track to multiple synchronous tracks to multiple independent tracks. Tracks can represent a sequence of channels with different notes/timings for each instrument. This conjunction of tracks and channels allows for an accurate encoding/decoding of sound files. The drawback is that MIDI files can be relatively large to store, and they also sometimes fail to represent the full range of the original sound source. We explore an approach that focuses on a series of frequencies and amplitudes to represent sound as opposed to a series of notes/pitches. This approach is aimed to store music in a smaller encoding, as well as capture sound sequences (chords) efficiently.

1.2 8-Bit Lofi Sound Style

We chose to use 8-Bit Lofi style music as our training data for the RNN-GAN. There was a few reasons for using 8-bit Lofi, namely it's distinctive "bips" and "bleeps" that resemble a square wave, allowing us to encode sound in a square wave with relatively modest accuracy(Section 2). Our goal was to build somewhat nostalgic music that resembles early video game consoles, hence another reason for the 8-bit choice. Lofi ("lo-fi" standing for low fidelity) is a format of music with somewhat lower quality of sound and simpler sound sources. It is meant to be a simple, relaxing sound style, which we value both for the output of the GAN sounding soothing as well as being a style that is not overly difficult to replicate. The Lofi style is forgiving in that it simple chords/sound sequences that are repeated, so there should be a lot of sequences for the generator to learn, and the 8-Bit style makes our encoding via square waves relatively accurate. Additionally, the style has no true rules. Thus, if our GAN produces sound that relatively sounds 8-Bit in nature, it can pass as 8-Bit Lofi.

2 Encoding Sound

Due to our limited computational resources and the large nature of our audio formats, we were required to fine an alternative representation for sound. In order to accomplish this, we used the Fast Fourier transform to learn aspects of the sound and then attempted to create limited recreations of it. Initially we had wanted to use the midi format as it has been used successfully for music generation[10]. However, we had trouble finding large amounts of data to this end and thus needed to rely on our own methods of data compression and representation.

2.1 The Fast Fourier Transform

The Fast Fourier Transform (FFT) is a method for efficiently computing the discrete Fourier Transform. Due to the frequency with which we were required to run Fourier transforms to parse sound data, using a high efficiency algorithm to perform these was required to keep computational time reasonable. It works by being highly efficient at computing the discrete Fourier transform (DFT) which in turn is a close approximate for the FFT. In fact, due to the nature of the audio samples, we were already working with discrete data [1].

2.2 Encoding Attempts

Compressing sound took several attempts and due to our initial unfamiliarity with the nature of sound, several of these attempts fell significantly short of our initial goal. This section details the overarching progress through these attempts.

2.2.1 Single Frequency

The first attempt at sound compression was based on the flawed idea that a single frequency could represent sound. The audio was first divided into tenth second intervals, then FFT was run on these sections to grab information regarding frequencies and their amplitudes. Then attempts were made at playing both the average frequency and the highest amplitude frequency but neither of these were able to at all accurately reproduce the sound.

2.2.2 Small Number of Frequencies

Instead of recreating frequencies, the second attempt relied on recreating tones. Rather than representing sound via a single frequency, several frequencies would be played simultaneously to recreate tones of the music. These frequencies were selected by taking the FFT over an interval of time then taking the top number of frequencies for that interval. This approach worked well however it lost phase information. As a result, when the interval duration of our samples was decreased sufficiently to identify the sound, the recreated music became distorted.

Table 1: Generator Architecture

Layer	Input Size	Output Size
LSTM	(1000, 1)	(1000, 512)
Bidirectional LSTM	(1000, 512)	(1000, 512)
Dense	(1000, 512)	(1000, 256)
Leaky ReLU	(1000, 256)	(1000, 256)
Batch Normalization	(1000, 256)	(1000, 256)
Dense	(1000, 256)	(1000, 512)
Leaky ReLU	(1000, 512)	(1000, 512)
Batch Normalization	(1000, 512)	(1000, 512)
Dense	(1000, 512)	(1000, 1024)
Leaky ReLU	(1000, 1024)	(1000, 1024)
Batch Normalization	(1000, 1024)	(1000, 1024)
ReLU	(1000, 1024)	(1000, 1024)
Reshape	(1000, 1024)	(1024, 1)

2.2.3 Down Sampling

As disretizing phase information proved difficult, we attempted to simply down sample the music and train. This failed as we could not compress the music sufficiently to be able to recreate a second of music without overloading our computation resources.

2.2.4 Additional Attempts

Our final product attempted tone recreation however made several changes to more accurately represent 8-bit audio. First, square waves were used instead of sine waves to create a more electric tone. Next, random offsets were including to partially mediate clicking. Finally, the FFT results were convolved with a normal distribution to smooth out results slightly such that taking the highest amplitude frequencies would gravitate towards areas with several important frequencies. Attempts were also made towards blending sound via overlapping, however this resulted in further sound distortion.

3 RNN-GAN Architecture

Our GAN architecture resembles a traditional GAN architecture in that it is divided into an architecture for the Generator and architecture for the Discriminator. This allows us to later train them against each other, and hopefully force the Generator to begin outputting realistic sounding music. As discussed in Section 2, we encode wav files into their own latent space represented by csv files. Each row of the csv file contains frequencies and amplitudes for one timestep of the input wav file. We used Pytorch to represent each row as a 1D tensor, resembling: $[frequency_1, frequency_2, ..., frequency_8, amplitude_1, amplitude_2, ..., amplitude_8]$ [6]. Each frequency corresponds to an amplitude ($frequency_1$ corresponds to $amplitude_1$, and so on). This allows us to represent each csv file as a 2D tensor. This 2D tensor is the input shape for the Discriminator, as well as the output shape for the Generator. The latent space input to our Generator, as seen in Table 1, is shape $(1000, 1)$.

3.1 LSTM

As can be seen in Tables 1 and 2, we use LSTM layers at the beginning of the Generator and Discriminator architectures. The reason for this is to make use of an RNN-GAN structure, which allows our Generator and Discriminator to learn information about sequences of input, learning inputs in series [5]. The reason we want this is that sound can be thought of as sequences of frequencies played in succession, thus learning sequences of input is desirable. This will allow the Discriminator to better differentiate real and fake inputs, as real inputs will have frequencies that mesh together over sequences (or at least we hope our training data has strong melodies/chords). This forces our generator to try to learn sequencing information and generate songs with characteristics chords/note sequences.

Table 2: Discriminator Architecture

Layer	Input Size	Output Size
LSTM	(1024, 1)	(1024, 512)
Bidirectional LSTM	(1024, 512)	(1024, 512)
Dense	(1024, 512)	(1024, 512)
Leaky ReLU	(1024, 512)	(1024, 512)
Dense	(1024, 512)	(1024, 512)
Leaky ReLU	(1024, 512)	(1024, 512)
Sigmoid	(1024, 1)	(1024, 1)

One common issue with conventional RNNs is their tendency to not learn long-range dependencies due to vanishing gradients. We felt long-range dependencies were important for the task at hand, as modeling temporal sequences should take a wide-range of sequences over time into account. For this reason, we use Long Short-Term Memory (LSTM) layers as our method of learning sequences in the Generator and Discriminator [8]. LSTMs are well known for processing time series data, as they use a feedback approach to connect information over longer ranges than conventional RNNs. We also included a Bidirectional LSTM layer, which processes sequential data in both the forwards and reverse directions, utilizing information from both directions to better model sequential data [2].

After the LSTM layers, both the Generator and Discriminator architectures make use of LeakyReLU hidden layers. Additionally, the Generator uses Batch Normalization layers, which is meant to normalize training by re-centering and re-scaling data as it moves through the network.

4 Training

We trained the Generator and Discriminator for 250 Epochs in Google Colaboratory on a T4 GPU using 30 minutes of 8-Bit Lofi wav files. We based our model on another state-of-the-art RNN-GAN model that generates music for the MIDI format [9]. We used a sequence length of 64, meaning the generator generates 64 timesteps of frequencies and amplitudes. It also means network input is sequences of length 64. We would have liked to use larger sequence lengths, as larger sequence lengths mean there is more sequencing information for the network to learn and there is longer output songs from the network, but using a sequence length of larger than 64 crashed our instance's RAM. Thus, we were forced to stick to a sequence length of 64. Additionally, we had to flatten the input tensors. As discussed in Section 3, we represent inputs as 2D tensors. Thus, we each input is shape (64,16), 64 being the sequence length and 16 being 8 frequencies + 8 amplitudes. We flattened this into a (1024,1) tensor for input into the network.

4.1 GAN Training Issues

We ran into numerous issues training the Generator and Discriminator. In the beginning, the Generator dominated the Discriminator, and the Discriminator was never able to learn enough information to accurately detect real and fake information. This lead to our Generator generating frequencies above 20kHz, which is the frequency limit for human hearing [7]. Clearly, we needed to modify training such that the Generator works more slowly, and the Discriminator works faster. In their Vanilla GAN paper, Goodfellow et. al. describe training the Discriminator k times for every time the Generator is trained once (where k $\geq$ 1)[4]. This k term is called the critic, and we use a critic of 2 to help the Discriminator "beat" the Generator. Training a GAN is a power balance between the Generator and Discriminator, and by increasing the critic value, we are giving more power to the Discriminator at the beginning of training. This allows the Discriminator to better understand real vs fake data at the beginning, which in turn forces the Generator to better learn the latent space in order to then "beat" the Discriminator. This phenomenon of the Discriminator performing better at the start can be seen in our final training loss graph in Figure 1, where Generator loss starts out low, then takes a big spike near epoch 50, as the Discriminator is learning faster than the Generator.

Other methods we employed to force the Generator to train slower are lowering the learning rate of our optimizer from $1e^{-3}$ to $2e^{-5}$, which takes smaller gradient steps. We also increased the momentum term of our BatchNormalization layers in the Generator architecture from 0.8 to 0.99,

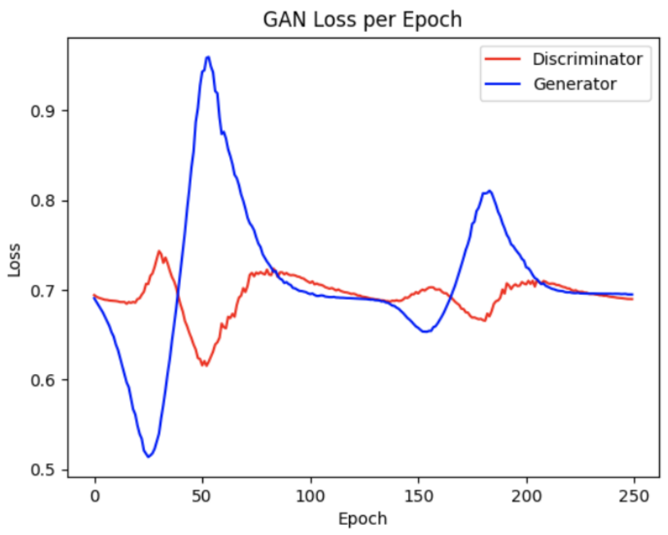

Figure 1: GAN Training Loss

which increases the "lag" of the normalization. This allows the BatchNorm layers to slow training down, as more of the running mean/variance are used in calculating the mean/variance of the next running mean/variance.

Making these sequences of changes allowed our GAN training to progress more smoothly, generating sounds within the range of human hearing and amplitude values that weren't so loud as to hurt the listener's ears.

5 Results

Our GAN output, shape (1024,1), was reshaped to (64,16), written to a CSV, then decoded to a wav file. The results can be heard on our Github under the $Results/$ directory (https://github.com/nathanhunsberger/cs8395-lofi-gan). As can be heard in the resulting songs, the GAN output is mediocre at best. It contains distinctive chords and chiptunes that we were going for with 8-Bit Lofi, but it is not as smooth as we would have hoped. The GAN output has occasional amplitude or frequency spikes that are uncharacteristic of 8-Bit Lofi, which is much smoother and relaxed in nature. We believe these results to be limited by the nature of our encoding. Encoding sound with frequencies and amplitudes fails to included things like phase information, which is why the GAN output is so choppy. Thus, we believe the Generator is making the most of a mediocre sound encoding, still outputting sounds that are distinctly 8-Bit Lofi, just not as pleasing to the ear as we had hoped for. We believe that under a better sound encoding, like MIDI for example, our GAN would produce sounds that are much more pleasing to the ear, while still being characeristically 8-Bit Lofi.

6 Future Work

In the future, reworking our RNN-GAN to support better encodings of sound would be desired. This could a pre-established encoding like MIDI, or our own, custom encoding that includes things like phase shift information. Additionally, including some prior information on the smoothness of sound to our decoder would smooth out the Generator's sound outputs. This would entail making the decoder smooth out the resulting sound, producing sounds that are closer to 8-Bit Lofi than we are able to generate right now. This approach would solve some issues we've had with the Generator producing sharp changes in frequency and amplitude.

7 Conclusion

This paper presents a novel way to encode sound: sequences of frequencies and amplitudes. We present the benefits of this approach, largely being a very reduced size of the encoding space while still encoding enough information to distinctly reconstruct the input sound. Using this sound encoding we train an RNN-GAN to learn sequences of chords and melodies of 8-Bit Lofi. We explore the benefits proposed of using LSTM layers inside Generator and Discriminator architectures to process inputs in sequences, learning information about the input sound spread over a long-range. Ultimately, we are able to decode the Generator output and produce wav files that resemble 8-Bit Lofi. Our encoding of sound is not perfect in that it loses information like phase shifts, thus our resulting sounds are far from perfect. However, our approach demonstrates an ability to encode sound to a small enough space such that it can be trained on a GAN, while still decoding it to a distinctive genre. This demonstrates the power and utility RNN-GANs provide.

References

[1] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, D.E. Nelson, C.M. Rader, and P.D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967.

[2] Zhiyong Cui, Ruimin Ke, Ziyuan Pu, and Yinhai Wang. Deep bidirectional and unidirectional lstm recurrent neural network for network-wide traffic speed prediction, 2019.

[3] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment, 2017.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[5] Olof Mogren. C-rnn-gan: Continuous recurrent neural networks with adversarial training, 2016.

[6] Adam Paszke, Sam Gross, and Massa. Pytorch: An imperative style, high-performance deep learning library, 2019.

[7] R. J. PUMPHREY. Upper limit of frequency for human hearing. *Nature*, 166(4222):571–571, 1950.

[8] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Interspeech*, 2014.

[9] Seyedsaleh. Seyedsaleh/music-generator: Multi-instrument music generation using c-rnn-gan with midi format input, 2021.

[10] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation, 2017.