Neural Network Music Generation Comparing Symbolic and Visual Audio Representations

Karim Arem
Cornell Tech
ka295@cornell.edu

Rory Connolly
Cornell Tech
rlc367@cornell.edu

Marika Cusick
Cornell Tech
mmc265@cornell.edu

Benjamin Hwang
Cornell Tech
bwh57@cornell.edu

Abstract

Music generation is a difficult task for Neural Networks. Unlike applications of Neural Nets in areas like computer vision, music poses the challenge of being a temporal medium that contains long term structure. Hence we propose utilizing cutting edge techniques and a completely novel approach to tackle this problem from two different angles: composition of symbolic representations of music and image generation as a means of creating new instances of music. Leveraging existing approaches to these problems we aim to apply the more traditional frameworks in a novel way by trying to optimize these models for inputs other than traditional classical piano music. We intend to produce Electronic Dance Music (EDM) and see whether or not we can tune our models to produce comparable results to previous literature. For our novel image generation approach we will use the traditional piano inputs and compare the results to the cutting edge traditional results outlined below.

1. Introduction

Institutions like Google have produced new frameworks (e.g. Magenta) to experiment with music generation. This has paved the way for more people to experiment with new applications and approaches of using Neural Networks to generate new pieces of music. One approach includes using encoded MIDI (Musical Instrument Digital Interface) files as a symbolic representation to generate new digital arrangements. Another approach is to use raw waveform representations of music to generate new waveforms that can then be processed as audio. Both are the common current practices for generating music, but we propose using image generation of visualizations of music as a new alternative to the these techniques.

1.1. Current Problems in Music Generation

Music is notoriously difficult to model because of its complex structure. While models can generally understand music at a short time scale, they often fail to capture the longer structure that exists in musics which is what we generally use to recognize music as a song [3]. While this issue has been explored more deeply in the symbolic representation realm, MIDI has its limitations. While MIDI can capture melodic structures it is difficult to bring current machine learning techniques to bear without losing the temporal meaning of passages in a song. To combat this we have decided to evaluate songs on a holistic level as well as on a note by note basis. Namely, we are going to train a model on songs in their entirety based on visual representation as an alternative to our other recurrent network methods.

1.2. Current Techniques

As mentioned in the introduction, we are evaluating two primary approaches for generating music with neural networks; using symbolic audio representations like MIDI for input, and using the visual representations of entire songs as image input to train instead. Here we will discuss the technical implementations of each approach, as well as some of the strengths and weaknesses for previous attempts where applicable. In the case of visual learning we believe this to be a completely novel approach, so this project will shed light on an entirely new means of audio generation.

1.2.1 Visual Audio Models

Most music is represented digitally in an encoded data format that allows for playback of the given audio content. Among the more popular coding formats are MP3, WAV, FLAC, TIFF, and MIDI. Since these forms of audio storage are used so widely with audio data it makes sense that these are the focus of current machine learning techniques as applied to music.

However, there are means of representing audio data

aside from encoding and decoding waveform information. For instance, pre-dating computers and audio recordings of any kind was the written representation of music on a staff, colloquially known as sheet music.

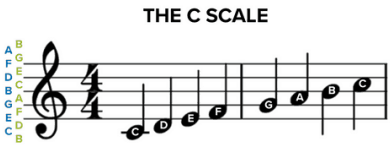


Figure 1: C-scale represented in sheet music

This is the most popular visual representation of music for human musicians and is the ubiquitous format for people who are looking to play a previously arranged piece of music on an instrument. There are other forms of visualizing data that are not as directly tied to playing musical instruments however, and we will investigate one of these for the purposes of this experiment.

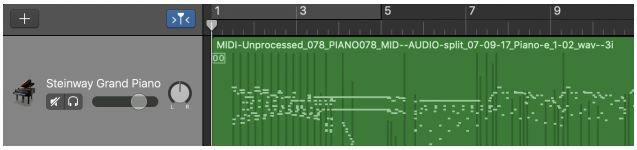


Figure 2: Visual representation of MIDI as a piano roll

We have talked about different formats for data storage as well as a visual format suitable for human reading, but there are other visualizations of music that are available which are worth exploring. For our purposes we will look into the visual representation of music as it is stored in the 128 pitch values available in the MIDI data format as this will allow for easier conversion from visual to audio format for evaluation. This visual representation will be the basis of our experimentation where we will utilize state of the art computer vision techniques to look to generate novel music visually.

1.2.2 Symbolic Audio Models

Most deep learning approaches for music generation have been based upon symbolic representations of music. A common way to represent music is through MIDI (Musical Instrument Digital Interface) [6]. MIDI is a technical standard that describes a protocol that allows for interoperability between various electronic musical instruments, software, and devices. Further information on the parameters within MIDI files have been provided in the data description. One drawback of using the MIDI messages is that it does not effectively preserve the notion of multiples notes being played at once through different tracks, thus making it difficult for the model to learn to play multiple notes at once. [1]

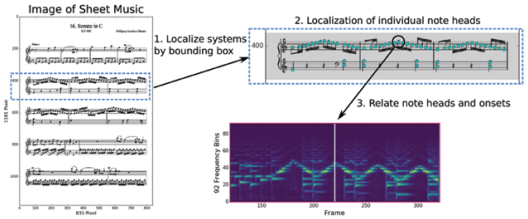
Although understanding temporal dependencies is traditionally done by recurrent neural networks, many researchers have found that this is not as successful in music generation due to large temporal dependencies in music, such as key and time signatures [6]. As a result of this, researchers have gravitated towards the use of Long Term Short Memory networks, which are a variant of recurrent neural networks. These models add an extra parameter, allowing them to maintain parameters over time. One notable example of this type of network for music generation is DeepBatch, a probabilistic neural network trained on chorale harmonizations by Johann Sebastian Bach [4]. (3)

2. Proposed Work

We propose to generate music using symbolic audio techniques in two different ways. The first using MIDI data to directly generate new MIDI files, and the new approach using GANs image generation to create a visual representation of music that is translated back into a MIDI file. Both approaches using the same corresponding audio and MIDI dataset. After generation, we will do a comparison in order to determine which technique was more successful in producing realistic music.

Implementation of the initial visual audio representation algorithm will be done using one of the FMA[2] or MAESTRO [5] datasets. These are appealing since they provide a large number of identical audio samples in both MIDI and WAV formats.

2.1. Approach 1: Image Generation of MIDI visualizations



After extensive research on the current music generation techniques using neural networks, the main hurdle that these techniques cannot get over is the fact that music is temporal in nature, but also has global structure. Using sequential modeling techniques like RNNs ,LSTMs, and various encoders aim to solve this problem by retaining memory of previous portions of the sequence, but this becomes memory intensive relatively quickly and there are other associated problems with training like catastrophic forgetting that may occur.

Hence, we propose an entirely different approach for generating music. As mentioned above, music can be translated into a visual representation using sheet music or MIDI.

By using MIDI representations of music we can visualize both the minute details (intervals between notes) and the overall scale of the music (the overall patterns over the length of a piece of music). Given that all of this information can be captured by an image, we have decided to generate image representations of music to see whether or not we could generate convincing MIDI visualizations that could then be transformed back into MIDI files that could be played.

In order to generate the new MIDI images we decided to experiment with various GANs architectures. We started with a simple Vanilla GAN and then progressed to a WGAN and finally a DCGAN architecture.

2.1.1 Visual GAN Architectures

All GAN approaches we employ will follow the same high level composition whereby we create two distinct models; a discriminator and a generator. Our discriminator function is fed a number of images that were created using real MIDI files which are labeled as real along with the same number of generated images that were created by the generator model which are labeled as false. The loss for our discriminator is based on correct labeling of both sets, and the loss for the discriminator is based on the accuracy of the discriminators classification of its generated artifacts. By replaying generated images into our discriminator and correcting for the calculated loss we should be able to refine our generator to the point where it can generate images that are indistinguishable from those created directly from real music files.

The differentiating factors for the different approaches will be the architecture of the models that we use for the generator and discriminator in each attempt.

Vanilla GAN

The Vanilla GAN is our simplest architecture having both models constructed of linear, fully connected layers between neurons of each subsequent representation of the data passing through the model. For both generator and discriminator we construct 4 linear layers to transform noisy input data vectors into images or to transform images into predictions, respectively. Here are the specifics of the layers in question:

```
Sequential(
  (0): Linear(in_features=100, out_features=256, bias=True)
  (1): ReLU()
  (2): Linear(in_features=256, out_features=512, bias=True)
  (3): Linear(in_features=512, out_features=1024, bias=True)
  (4): ReLU()
  (5): Linear(in_features=1024, out_features=128000, bias=True)
  (6): Tanh()
)
```

Figure 3: Vanilla GAN: Generator Architecture

WGAN Architecture

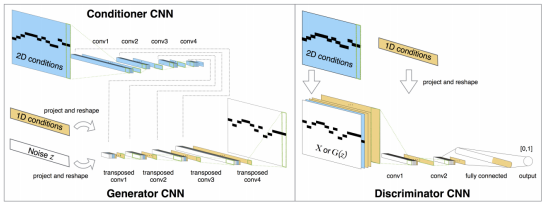
```
Sequential(
  (0): Linear(in_features=128000, out_features=1024, bias=True)
  (1): ReLU()
  (2): Linear(in_features=1024, out_features=512, bias=True)
  (3): Linear(in_features=512, out_features=256, bias=True)
  (4): ReLU()
  (5): Linear(in_features=256, out_features=1, bias=True)
  (6): Sigmoid()
)
```

Figure 4: Vanilla GAN: Discriminator Architecture

The WGAN architecture was introduced to try and improve the disparity between Generator and Discriminator training. By removing the Sigmoid activation function to ubound the outputs at the end of the discriminator and introducing a clamp that limits the weight magnitude.

DCGAN Architecture

After failing to achieve a higher level of detail, we decided to use a more state-of-the-art model architecture. DCGANs utilize deeper convolutional networks for both the discriminator and generator. The general architecture is described below:



This works similar to our Vanilla GAN generator except instead of upsampling using linear layers we are utilizing transpose convolutions to up-sample our noise vector into the proper shape. It is worth noting that for this training, we are upsampling into a rectangular shape with significantly longer widths. This is to try and capture the length and resolution of the full piece but also trying to limit the amount of white space for the note intervals. This proved to be challenging because we are aware of the artifacts that exist because of the upsampling to a much higher dimension.

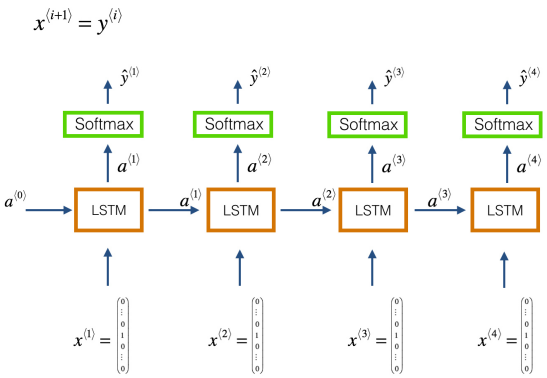
Our modified Generator and Discriminator Architectures are shown below:

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 32, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Conv2d(32, 64, kernel_size=(1, 8), stride=(1, 8), bias=False)
    (3): LeakyReLU(negative_slope=0.2, inplace)
    (4): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.2, inplace)
    (7): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (8): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): LeakyReLU(negative_slope=0.2, inplace)
    (10): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (11): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): LeakyReLU(negative_slope=0.2, inplace)
    (13): Conv2d(512, 1, kernel_size=(7, 7), stride=(2, 2), bias=False)
    (14): Sigmoid()
  )
)
```

Figure 5: DCGAN: Discriminator Architecture

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(4, 4), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace)
    (12): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(1, 4), padding=(1, 1), bias=False)
    (13): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU(inplace)
    (15): ConvTranspose2d(32, 3, kernel_size=(2, 14), stride=(2, 4), padding=(1, 1), bias=False)
    (16): Tanh()
  )
)
```

Figure 6: DCGAN: Generator Architecture



2.2. Approach 2: MIDI Representation

The model requires a symbolic representation of a fixed length, which is achieved by dividing the MIDI file into bars. Each song is represented as a matrix of shape $h * w$ representing the song's piano roll. h represents the number of MIDI notes which is 88 (since there are 88 keys in a piano) to consider and w is the number of time steps in a bar. w varies from song to song.

2.2.1 Overall Architecture

At a high level, our input music sequence is encoded through a linear layer at first, a batch normalization step and dropout step. Then an LSTM takes steps at every embedding of each element of the input music sequence to give an output at each step. The outputs are fed in to a fully connected linear layer to bring back the output sequence to a $h * w * batchsize$ tensor. Each sequence is then fed in to a softmax and we therefore get a probability for each note being played for each given sequence during a certain time period. We then are able to predict using various temperature values whether a note gets played for a certain time sequence.

We are therefore trying to apply a simple sequence to sequence model that is popularly used in areas like language, to music. To keep things simpler, we have added padding to our sequences that are less than 50 time steps long and cut off sequences that are longer than 50 time steps long. Below we can see a picture of our model architecture. As mentioned more in depth in the experiments section, we also experimented with using a GRU model to replace the LSTM as they have a similar structure but the GRU is more efficient computationally.

3. Dataset

In order to generate MIDI files, we used a web crawler that scraped MIDI files from the TheoryTab website. This crawler takes the MIDI xml files from the website, coverts these files into an event (roman) json file, event (symbol) json file, and piano roll MIDI file. For the purposes of our work, we only used the piano roll MIDI files.

Because we were interested in EDM Music, we identified EDM artists that we would like to emulate. The TheoryTab website formats artists by their first letter, so we decided to query for the letters a,d,m,and s, as this maximized our identified artist list. This resulted in a total of 31 artists, and 440 piano roll MIDI files. The artists we have chosen for training include Avicii, Major Lazer, Skrillex, and DJ Snake. After training and evaluating on the development set, we decided (whether or not it would worthwhile to increase the number of training songs for our final model.)

On the TheoryTab website, each song may have a number of different MIDI files to export into a piano roll. As an example, Avicii's song "Wake Me Up" has 5 different MIDI files, which include the intro, verse, pre-chorus, chrous, and bridge. The TheoryTab crawler also gives the key and no key version of every MIDI file.

For each of the songs in our training data, we have set a max length for the number of time steps in our song. Through this methodology, we can batch our data because all of our songs will have the same shape, the max number of time steps by the total number of MIDI notes (88). For the MIDI files that have less than our max length of time steps, we pad our data with zeros. For initial training, we will use a max length of 50. However, we will experiment with a variety of different maximum lengths in order to see if this improves our results.

In our training set, the minimum number of time steps for a given file is 19, and the maximum number is 450. The mean number of time steps is 98.

Our validation set is created by taking advantage of the songs that have a timestep length longer than our set maximum length. For the initial training, as our maximum length

is 50, the validation set comprises of the songs with time steps that range from 51 to 100.

3.1. Visual data set

We chose to use a well established data set for the initial attempt at generating new audio from visual representations of music, in this case the Maestro (MIDI and Audio Edited for Synchronous TRacks and Organization) data set. This is comprised of over 172 hours of classical piano music broken up into 1184 individual song files.

Our intention is to eventually move on to incorporate more sources of data across various genres but this dataset is well curated, clean, and reliably used for machine learning projects so it made sense to use for our first attempt at this novel approach to creating music with machine learning algorithms.

4. Experimentation

4.1. MIDI Generation

Our procedure for experimentation will involve changing the architecture of the network, tweaking hyper-parameters, and inputting various types of training data. Although our model does provide a training and validation loss at the end of every epoch, we believe that there should also be a qualitative evaluation approach for each experiment. For this reason, we decided to create a rating scale for every generated clip. We will rate the clip on how true it is to EDM music, how pleasant it is, and how interesting it is. We decided to rank each of these on a scale of 1-5. This rating scale is extremely similar to the one used in MidiNet's evaluation process. However, we changed the measure of being realistic to how true it is to EDM music.

Each of our clips will be played on the Online MIDI Sequencer. It is important to note that the songs played on Online MIDI sequencer do not sound like their original version because there is only one instrument playing at a time. As a baseline, we will rank one of the real EDM songs according to these metrics. The rating score for this EDM is given below:

Baseline Model Rating

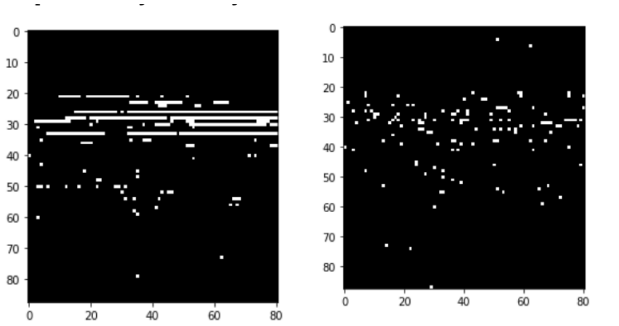
Song	True to EDM	Pleasant	Interesting
Wake Me Up	5	5	5

In our initial set of experiments, we tweaked the model architecture and hyper-parameters. Thus, we kept our training set constant. We trained on the entire set of songs, with a batch size of 50 and maximum time-step length of 50. Every experiment is trained for 10 epochs. The rating scores in the table are in the order of True to EDM, Pleasant, and Interesting.

4.1.1 Model Experimentation Results

Model Architecture Experimentation Results

Layers, Hidden Size, LR	Dev Loss	Rating Scores
2 layers, 512, 1e-3	0.0496	3,2,4
2 layers, 100, 1e-3	0.0675	3,3,4
2 layers, 800, 1e-3	0.05168	1,1,1
3 layers, 100, 1e-3	0.09734	3, 1, 3
1 layer, 100, 1e-3	0.0593	1, 3, 3



(a) LSTM Model One Layer (b) LSTM Model Three Layers

Figure 7: LSTM MIDI Output Images

We found that increasing the number of hidden units in our LSTM model made the quality of the music significantly worse, even though it was able to achieve a relatively low validation loss. This proves that the validation loss may not be an accurate indicator for assessing the quality of the model. We found that the model had the overall best rating for the hidden size 100.

Additionally, we found that increasing then number of layers drastically changed the MIDI output. As evident in figure 1, increasing the number of layers causes the LSTM model to produce notes that are extremely short, while the model with one layer produces notes that are much longer. While the model with only one layer is able to produce more aesthetically pleasing music, it is less true to EDM. On the other hand, the model with three layers produces music that is closer to true EDM music, but is not as pleasing. Based on this, we decided that it would be benefical to go for a middle-ground of two layers.

We decided to run an additional set of experiments on hidden size, and number of layers using the GRU architecture. Below is a table highlighting the models' performances.

GRU Model Architecture Experimentation Results

Hidden Size, # Layers, LR	Dev Loss	Rating Scores
1 layers, 100, 1e-3	0.0812	2,2,4
2 layers, 100, 1e-3	0.0683	2,1,1
3 layers, 100, 1e-3	0.0717	1,1,2
1 layers, 512, 1e-3	0.0716	1,1,1
2 layer, 512, 1e-3	0.0708	2,2,2
3 layer, 512, 1e-3	0.0872	2,1.5,2
1 layers, 800, 1e-3	0.0730	1.5,2,2
2 layer, 800, 1e-3	0.0727	2,1,4
3 layer, 800, 1e-3	0.0961	1,2,3

Overall the using a GRU architecture instead of an LSTM did not seem to be able to produce any better results. It is however interesting to see that the more we added layers the larger range of pitches we were able to generate in our music which made it less enjoyable to listen to.

4.1.2 Training Data Experimentation Results

After tweaking the model architecture, we decided to conduct a second set of experiments involving the training data. We will experiment with the size of the training data and the type of training data.

Model Architecture Experimentation Results

Num. Songs	Data Type	Dev Loss	Rating Scores
440	All Songs	0.0675	3,3,4
260	Chorus	0.0810	1,1,1
100	Instrumental	0.0720	1,1,1
75	Verse	0.08884	4,2,3
440	No Key	0.05682	1,3,2

The verse version of the MIDI files lead to music that was true to EDM, as the notes were extremely short and somewhat sporadic. However, the music was not pleasant and seemed almost choppy. When inputting the no-key version of the MIDI files into the model, the model produces notes that extremely long, which is unlike typical EDM music. In addition, in the middle of the song, there was a long pause.

4.2. MIDI Visualization Generation

As earlier we will go through each of our models and the corresponding results that came about from each experiment. Most of the tweaking came in the form of choosing varying learning rates, optimizers, and layers for the architectures.

4.2.1 Model Experimentation Results

Vanilla GAN

The Vanilla GAN yielded the most sporadic results of all the approaches but still had its merits. The model was generally able to focus notes into the central band on the pitch (Y) axis of our visual audio representations, which is in keeping with the behavior of human composed songs. However, the model was unable to establish the inter-dependencies of musical structure that we were hoping to capture through this exercise.

Figure 8: Vanilla GAN: Experiment Results

The training loss did not quite converge throughout the training but we did see a decided shift towards a central plateau where the two models seemed to settle.

The transformations of noise input to generated music images at this point were as shown above. As mentioned, there is no great musical form to the sounds and the notes are played in a somewhat haphazard manner, but the high level visuals of the generated songs still correlate to the real songs that were transformed using the method described previously indicating that the model was able to learn some general characteristics of visual song representations.

Figure 9: Real MIDI File Visualization

WGAN: Experiment Results

WGAN results improved upon the Vanilla GAN outputs but overall the results were still not detailed enough to distinguish a local pattern; however the generator seemed to produce much more sure results by given higher intensities to the output notes. Overall we still observed a global band structure indicating that the model is still creating results that have a good general form.

DCGAN: Experiment Results

In order to address this level of detail problem, we then moved onto a more powerful GAN model: DCGAN. The DCGAN has a deeper architecture and also contains convo-

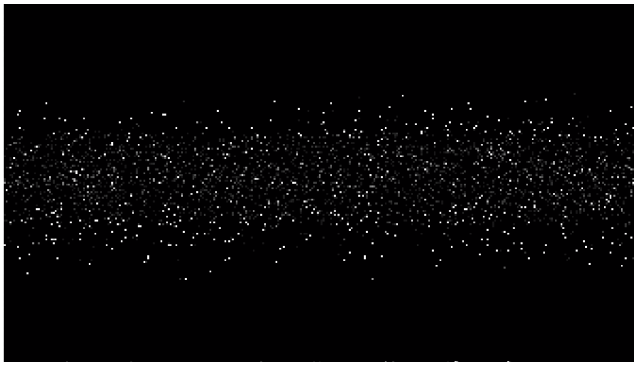


Figure 10: WGAN: Experiment Results

lutions which help deal with image based data. Training this architecture proved difficult but overall we were able to get the training loss for both the Generator and Discriminator to converge without complete collapse.

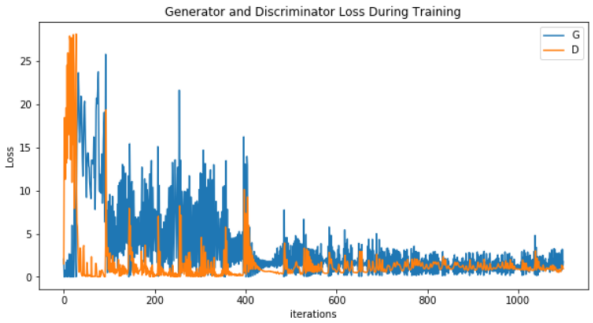


Figure 11: DCGAN: Training Results

Overall the results were very exciting as compared to the Vanilla and WGAN. For these generations we used colored representations. For comparison, below is a real piano piece in MIDI:

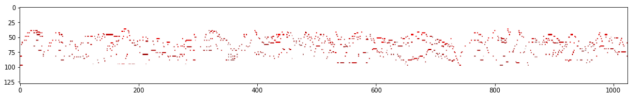


Figure 12: Real Piano Visualization for Reference

Looking at the results, we have made considerable and exciting progress! After 400 epochs of training (we are using a smaller sub-sample of the dataset due to high variability in track length) we can not only see a global structure but also some local definition. The most important take away of this is that our approach is able to maintain a global structure which other techniques fail to do. There is still work to be done in getting a higher resolution and sharper image convergence, but this is certainly a good direction.

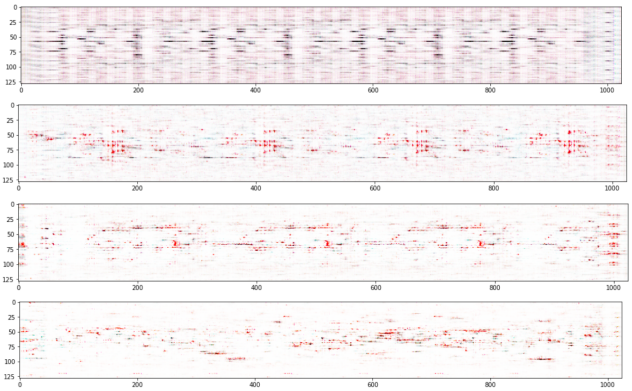


Figure 13: DCGAN: Experiment Results: 100-400 Epochs of Training

5. Final Model

For our final model, we decided to use the LSTM model (2 layer, 100 hidden units) with all of the songs used as training. This led to the best combination of our rating scores. The MIDI output of this final model is given in Figure 2.

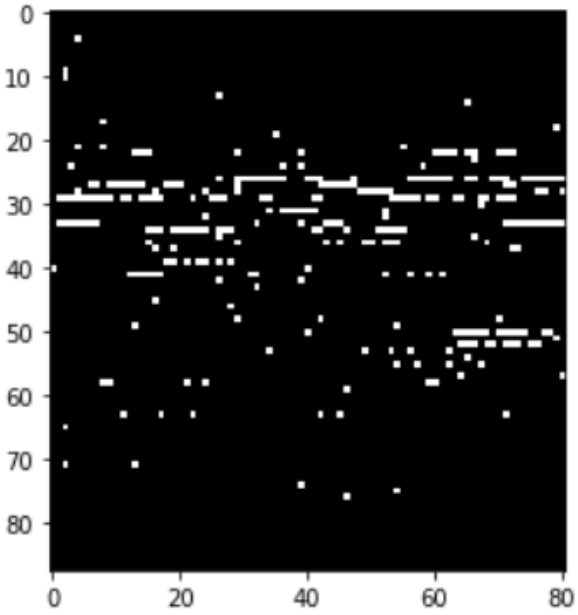


Figure 14: LSTM Model (Two Layers Hidden Unit = 100), All Songs

5.1. Final Model User Research

After the creation of the final model, we decided to do some user-testing in order to see how people responded to our generated song. We asked 6 users to rate the original

baseline song (Wake Me Up by Avicii) and our generated song based on the rating scores established throughout experimentation. The average scores are given in the table below. Evidently, we cannot produce EDM music that is as pleasant, interesting, or true to EDM as Avicii.

Model Architecture Experimentation Results

Song	True to EDM	Pleasant	Interesting
Wake Me Up	4.83	5	5
Song by K&M	3.66	3.33	4.16

6. Discussion

6.1. MIDI Generation

After thorough experimentation with the LSTM model, we found that music generation is as difficult as described in our introduction. Although we were able to produce a number of song samples, none of our samples were able to produce a rhythm and pattern that was constant throughout. In several of our samples, there were a series of time-steps in which the music quality was relatively pleasant. However, this would soon deteriorate with either a random break in song or a harsh note.

On top of the difficulty of music generation, we made our task even more difficult with our selection of training data. Most researchers have experimented with piano data because it has the simplicity of one instrument and a single piano-roll channel. We selected electronic dance music, which typically combines a number of different instruments and has multiple channels. Because our model only outputs one channel, it is a challenge to replicate the training music.

During experimentation with our LSTM model, OpenAI released a model called MuseNet, which has the ability to generate remarkably eloquent 4-minute musical compositions with 10 different instruments by using a transformer model. MuseNet uses the recomputed and optimized kernels of the Sparse Transformer to train a 72-layer network with 24 attention heads, which is why it is able to remember long-term structure in a piece. Despite the obvious advantages of this approach, the Transformer model requires an extraordinary amount of memory, due to the number of attention matrices. The MuseNet model required 590 GB of stored data and 9.2 GB of recomputed data in the backwards pass. Unfortunately, we do not have this bandwidth.

6.2. MIDI Visualization Generation

Upon testing various different architectures and tuning hyper-parameters we have made significant headway into creating an introductory proof-of-concept. Still there are many general procedure items that we need to work on.

First area of improvement is handling of the dataset. We used the MAESTRO dataset since it contained only piano music and a single voice for simplicity sake in generation. One issue with this dataset however, is that when converting everything into a visualization, there are a lot of pieces of varying lengths and tempos, meaning that if you standardize the visual output to a single output size, the notes can become very squished or elongated for long and short pieces respectively. We tried to control for this by filtering out pieces that were longer than 5 minutes, but this cut our usable image data to only 400 examples. Ideally, for the next phase of this we would work on standardizing the length of the pieces so that we can get resolution and interpret-ability of the results.

Second area is the image processing. We used a larger dimensionality to help visualize our results, but this of course uses up much more data and computing resources which unfortunately were limited in this exercise. We experimented using varying cropping as well and in order to help capture the full length scale of the music we created rectangular outputs with significantly longer width dimensions (128 x 1024). This proved an issue because when upsampling the noise into the final image, there were visible artifacts created as a result. We tried to increase our kernel sizes and adding additional layers, but unfortunately due to computational resource restrictions we were limited in how much we could add. These combinations of image processing need further experimentation and exploration with more powerful computing.

Lastly, image to MIDI translation needs more experimentation. We wrote our own custom MIDI image to MIDI file converter, but there are some issues with this conversion. Due to image compression and varying time scales, there is information lost when converting images to MIDI. While our scripts can capture a good deal of the music, this is key for translating our generated images into pieces that can be interpreted audibly.

7. Listen to our music!

You may listen to the music generated by our models by downloading the files from our github repository and inputting them into any online sequencer website such as this link.

References

[1] J.-P. Briot, G. Hadjeres, and F. Pachet. Deep learning techniques for music generation a survey, 2018. https://arxiv.org/pdf/1709.01620.pdf. 2

[2] M. Defferrard, K. Benzi, P. Vandergheynst, and X. Bresson. Fma: A dataset for music analysis, 2017. https://arxiv.org/pdf/1612.01840.pdf. 2

[3] S. Dieleman, A. van den Oord, and K. Simonyan. The challenge of realistic music generation: modelling raw audio at scale. 2018. https://arxiv.org/pdf/1806.10474.pdf. 1

[4] G. Hadjeres, F. Pachet, and F. Nielsen. Deepbach: a steerable model for bach chorales generation, 2017. https://arxiv.org/pdf/1612.01010.pdf. 2

[5] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck. Enabling factorized piano music modeling and generation with the maestro dataset, 2019. https://arxiv.org/pdf/1810.12247.pdf. 2

[6] R. Manzelli, V. Thakkar, A. Siahkamari, and B. Kulis. Conditioning deep generative raw audio models for structured automatic music, 2018. https://arxiv.org/pdf/1806.09905.pdf. 2