An Analysis of Implementing a GAN to Generate MIDI Music

Sofia Arora*, **Angus Dassler***, **Thomas Earls***, **Milo Ferrara***, **Niko Kopparapu***
Sachin Mathew*†
*Governor's School of New Jersey Program in Engineering & Technology, Rutgers University–New Brunswick, NJ, USA
†Corresponding Author
Emails: {sofiaaarora, angusmd18, earlsthomas9813, miloferrara, nikokopparapu, sachmathew}@gmail.com
+Sofia Arora, Angus Dassler, Thomas Earls, Milo Ferrara, & Niko Kopparapu all contributed to this project equally.

2022 IEEE MIT Undergraduate Research Technology Conference (URTC) | 978-1-6654-7345-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/URTC56832.2022.10002181

***Abstract*—Music generated with machine learning has the potential to serve as inspiration and accompaniment for all kinds of musicians. Although neural network technology has been applied to music generation before, these applications have often implemented large and complex networks. This research aimed to use a generative adversarial network (GAN) to generate multi-instrumental, complex music while implementing a simpler structure than past models. This required a data pre-processing algorithm to simplify the encoding of MIDI files. A simple GAN was developed in PyTorch, and generator outputs were scaled to values within the bounds of the MIDI Standard. The GAN ultimately produced MIDI data, but the results were not distinctly musical and lacked a clear melody, perhaps because the network lacked sufficient complexity to process or replicate the temporal structure of music.**

I. Introduction

A. Overview

As machine learning technology rapidly advances, research is emerging to study its uses in generating content that resembles creative works, such as music. Aside from making music more available to listeners, generative music technology has practical applications to inspire musicians and producers, generate accompaniment for artists, and allow researchers to generate large amounts of musical data for experimentation. However, music requires complex methods of data representation to account for the temporal nature of notes [1]. It is also difficult to generate data that sounds musical because of the intricate interplay of harmony, melody, and rhythm in music theory.

The complex nature of musical files favors the use of a generative adversarial network (GAN) to construct neural network architecture that can accurately produce musical output. These architectures—composed of a generator and discriminator network—allow for complex, realistic output with simple training. The generator network adjusts its output based on feedback from the discriminator until its output is, to the discriminator, indistinguishable from the training data.

B. Objective

The objective of this research was to develop a GAN that can be trained on examples of musical and non-musical MIDI files and produce files that sound like music to a human ear, replicating the intricacies of polyphonic composition. Defining musicality is difficult—it is a quality recognizable to humans but complex to quantify and format with a simple algorithm. The key characteristics are reasonably easy to identify: music is usually repetitive and makes use of melody and harmony. But the nebulous nature of these qualities makes them difficult for neural networks to easily identify. This research tested the effectiveness of simpler GAN architectures in creating MIDI music files.

II. Background

A. Music Theory and the MIDI File Format

The MIDI file format consists of messages that are read sequentially. The most common messages tell the synthesizer to turn a note on or off, defining a specific channel, pitch, time delay, and velocity (analogous to volume). As they are purely a series of instructions, MIDI files do not include any actual sound information. They are only commands for the computer or synthesizer to execute, so the exact sound produced will vary from platform to platform. The small size of MIDI files makes them excellent for machine learning because they can be practically stored and processed in large quantities [2].

MIDI files can hold a maximum of 16 channels, each of which contains data for one instrument. Notes can be played concurrently both within and across channels. MIDI files can also hold up to 65,536 tracks that each contain a separate stream of sequential MIDI messages. Track 1, for example, might encode the left hand part of a piano piece while Track 2 encodes the right hand part. Tracks are usually kept separate for production purposes, but they can be merged to simplify the file. The General MIDI Standard supports 128 different instruments that are divided into 16 instrument families [3].

B. Neural Networks

A neural network is a series of algorithms that observe and recognize relationships in a set of data, mimicking the processes of the human brain. It consists of layers of nodes that each contain one value. The nodes in different layers are connected, and each connection has a weight and bias that determines its strength, or significance, in computing the next output node.

A neural network undergoes a learning process: with each pass through the network, the actual output is compared to the desired output obtained from real data. This feedback is then used to adjust the values of the weights and biases to develop an accurate model in an iterative training process called backpropagation [4].

A loss function calculates the error in a model so that the network can improve based on the differences between the actual and predicted output. The neural network's overall goal is to optimize the set of weights and biases by minimizing the loss function through gradient descent, moving down the calculated gradient toward a local minimum.

C. Generative Adversarial Networks

A generative adversarial network (GAN) is an artificial intelligence structure that consists of two neural networks, known as the generator and discriminator, that compete against each other in order for each to improve. The structure was proposed in 2014 by Ian Goodfellow as an alternative to more computationally complex structures such as Markov chains [5].

One common analogy used to describe a GAN structure characterizes the generator as an art forger and the discriminator as a detective [6]. If the art forger is unconvincing, the detective will easily know that their art is fake. As a result, the art forger will improve their techniques to try to fool the detective. In the process, the detective will improve at distinguishing fake works from real works, and the cycle continues. With enough cycles, the art forger starts producing convincing, realistic works.

Similarly, the discriminator is given both training data and data produced by the generator, and tries to determine the probability that a given data sample is not the output of the generator. Based on discriminator feedback, the generator improves its model in order to produce output closer to the training data. The two neural networks are connected by the loss function that calculates their accuracy. The GAN uses a backpropagation algorithm to calculate the gradient of the loss function and then adjusts the weights in the direction of steepest descent. With enough training and effective gradient descent, the GAN approaches a local minimum of the loss function such that the generator's outputs become more like the training data and the discriminator improves its ability to distinguish between the two. In this structure, the generator has no direct connection to the training dataset—it only learns from the interaction it has with the discriminator [6]. This process is illustrated in Figure 1.

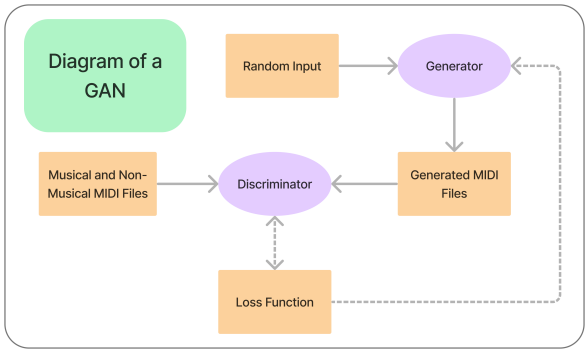

Fig. 1: Flowchart of the data flow of a GAN.

III. Experimental Procedure

A. Data Acquisition

The discriminator was first trained on examples of musical data and non-musical data in order to determine if the generator neural network was producing correct outputs. The Lakh MIDI Dataset [7], which consists of tens of thousands of songs from thousands of artists spanning hundreds of genres, was used as a training set after considerable pre-processing.

B. Data Pre-Processing

The goal of this data pre-processing was to parse the Lakh MIDI Dataset and convert each MIDI file into a numerical array of uniform size. This array could then be input into the neural network, with each value corresponding to a node in the network. However, MIDI files in the dataset were inconsistent and required significant pre-processing to standardize.

1) On/Off Messages

MIDI files have two ways to end a note. A "Note Off" message can be sent with any velocity, or a "Note On" message can sent with a velocity value of zero to end a note. These function identically to the synthesizer or computer playing the file but are encoded differently. Thus, the dataset was parsed to change every "Note On" message where velocity is zero into a standard "Note Off" message.

2) Reducing Tracks

Additionally, MIDI files often include multiple tracks. To simplify each file for input into the neural network, these tracks were merged into one track. First, duplicate tracks were removed by removing any tracks with a message length identical to an earlier track. Then, another algorithm was used to simplify the remaining tracks into one final track, making use of the families of instruments defined in the General MIDI 1 Standard [3].

Each family was designated one channel in a track, and all instruments in a given family were collapsed to one representative instrument from that family. For example, if a song contained a harpsichord, those messages would be instead encoded as a standard piano in the first channel. This meant that each channel corresponded to one representative instrument from one instrument family, and each channel contained a predictable instrument on one file. This significantly simplified encoding of the files as instruments by removing the need to specify an instrument, yet still retained the melody and musical character of each song.

3) Slicing

Once each song file was simplified into one track with 16 channels, it was sliced into segments containing 256 messages each. Depending on the duration of the notes encoded, these segments could have different play times, but the fixed number of messages meant the files were of a fixed size and thus constitute suitable input for a neural network.

4) Encoding

Each message contained five essential values: a 0 or 1 representing "Note On" or "Note Off," channel, pitch, velocity,

and time (where time is the amount of time before the next message will be read). Thus, a given segment of a MIDI file could be represented as an 256 by 5 array: 256 distinct messages, each containing 5 identifying values. This was encoded to a CSV file to more easily be read as a neural network's inputs. Figure 2 shows an example of a few rows of one of these CSVs, describing each note with its five parameters.

Note On/Note Off	Channel	Pitch	Velocity	Time
1	2	65	85	12
1	1	61	72	15
0	2	65	85	12
1	9	93	62	5
0	1	61	72	15

Fig. 2: An example of 5 MIDI messages represented in an array. This shows when a note is turned on or off and the note's channel, pitch, velocity and amount of time before the next message should be played.

5) Non-Musical Data Generation

Non-musical data was randomly generated using a Python script. The Lakh MIDI dataset was first analyzed to create a weighted set of the most commonly used instruments in songs. The channel of each note in the non-musical file was chosen based on a weighted random variable based on each family's overall usage in the Lakh dataset. This distribution was generated by analyzing the entire dataset of approximately 18,000 songs and counting the number of times each instrument was used. Then the distribution across the families was calculated as the sum of the values of all instruments in each family, giving a weight corresponding to each family. A random average volume was chosen for each track. Each note's individual volume was based on a normal distribution centered on this track volume. The pitch was also calculated by a normal distribution centered on 64, E4. The sequence of "Note On" and "Note Off" messages was generated such that there are chords of random length between one and four notes. With so many random variables, the resulting files consisted of real-sounding instruments, but lacked the rhythm or melodic line that could classify them as music. This process, repeated for a sequence of 128 notes on and 128 notes off, produces a single MIDI file that can serve as a negative example for the network—its complete lack of melody and repeating motifs makes it clearly distinguishable as not musical to human ears.

C. Developing a Generative Adversarial Network

1) Generator

The generator was constructed with ten linear layers, an input layer of size 100, and an output layer of size 1280, which could be later processed into a 256 by 5 array representing the parameters of messages in a MIDI file. Its forward propagation algorithm was structured with a Leaky ReLU activation function. Leaky ReLU is a linear function with a smaller slope for negative values.

2) Discriminator

The discriminator was constructed with five linear layers, an input layer of size 1280, and a single output node that would return what it deemed as the probability its input was real music. Its forward propagation algorithm was structured with a Leaky ReLU function and dropout with probability 0.3 applied to each layer. Dropout layers randomly zero a small proportion of weights, which prevents overfitting—a common problem where neural networks become trained specifically to a small range of values and struggle with new data.

3) Loss

Mean Squared Error loss was chosen as the loss function for the generator and discriminator. The function, shown below as Equation 1, finds the difference between expected and actual outputs, squares it to remove negative values, and finds the mean over many nodes.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}\left(Y_i - \hat{Y}_i\right)^2 \quad (1)$$

where $\left(Y_i - \hat{Y}_i\right)$ represents the difference between the actual and predicted output matrices of the network.

D. Data Scaling

Since the MIDI file format requires inputs to be within a certain numerical range, data scaling was implemented to ensure that the generator's outputs matched those ranges. The raw generator outputs were between -1 and 1, and were generally closer to 0, so these values passed through a function that scaled them to the appropriate ranges. For example, all the outputs that corresponded to pitch or velocity were scaled between 0 and 127, the bounds for pitch and velocity in the MIDI Standard. The scaling functions used were transformations of the sigmoid function, are shown below in Figure 3. They were then rounded to obtain the integer values needed for MIDI file translation.

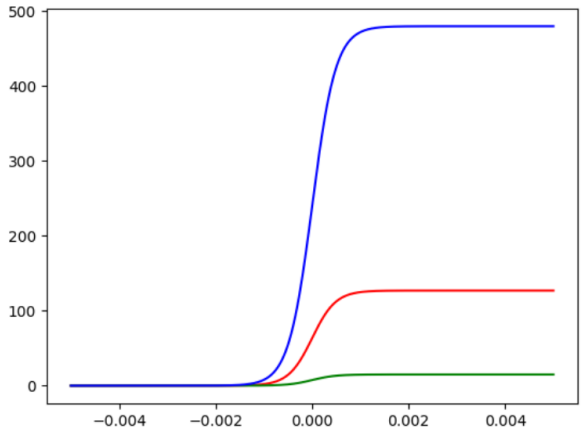

Fig. 3: Three data scaling functions that were applied to the generator's output. The blue function was used for time values, red for pitch and velocity, and green for channel.

IV. Results

This research constructed an operational GAN that produced MIDI data. The results were not musical, but displayed rudimentary elements of music such as pitch patterns and repeated notes.

A. Generated Musical Data

Figures 4 and 5 show an example musical file compared to an output MIDI file of the generator, respectively. The sample of music is shown after undergoing the pre-processing steps outlined earlier in this paper. The generated music has far fewer notes, mostly consisting of very long notes layered on top of one another as shown in Figure 4. This is possibly because of the pairing of "Note On" and "Note Off" messages in MIDI. To turn a note off that is currently playing on a given channel, a "Note Off" message must be sent on the same channel with the same pitch as the original note—which the generator did not always produce, given there are 16 possible channels and 128 possible pitches. Another contributing factor is that many instruments defined in the MIDI Standard continue playing indefinitely once they are started. Some, like instruments in the piano family, have a maximum length regardless of how long the note is set to be sustained depending on their velocity values. However, most of the MIDI instruments will play a continuous tone until they receive a note off message. Some of the MIDI files displayed the beginnings of melody, but the loud background instruments mostly overpowered them.

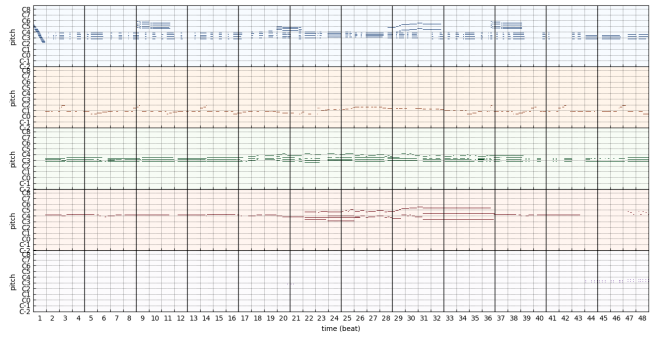

Fig. 4: Piano roll representation of ABBA's "Dancing Queen" after pre-processing [7].

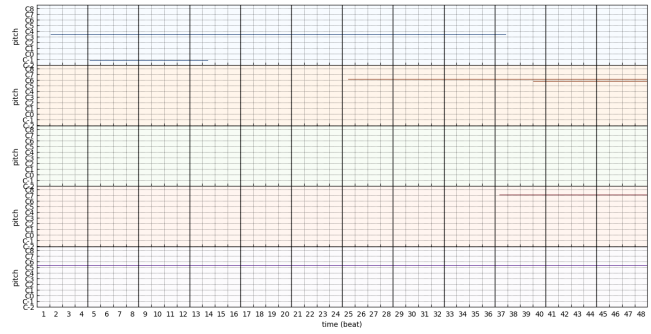

Fig. 5: Piano roll representation of a generator output.

The relative lack of musicality in the outputs was largely due to two factors that obstructed the generator's learning. The discriminator often outperformed the generator to such an extent that the generator was unable to learn. Even after this issue was addressed, the losses of the generator and discriminator quickly plateaued, minimizing the effects of further training.

B. Discriminator Overperformance

Initially, both the generator and discriminator had three hidden layers. Prior to the implementation of data scaling, this led to the discriminator's ability to discern non-musical and musical data far surpassing the generator's ability to produce convincing output. As a result, the discriminator always recognized the generator's output as non-musical, so the generator had no information on how to improve. Without some indication of which weights are more significant to the output, and thus which weights should be strengthened, the generator's loss rapidly increased while the discriminator's loss rapidly decreased, as shown in Figure 6.

Epoch	Discriminator Loss	Generator Loss
1	0.619	1.031
2	0.03	6.615
3	0.002	11.455
4	0.001	14.709
5	0.0007	14.709

Fig. 6: Losses of the discriminator and generator over the first five epochs in an early implementation of the GAN (before data scaling).

Several steps were taken to remedy the generator outpacing the discriminator. First, the number of hidden layers in the generator was increased to eight to allow it to produce more complex samples, as a better generator would be able to compete with the overperforming discriminator. Additionally, the technique of layer freezing was implemented: if the loss of the discriminator was less than 70% the loss of the generator, it was frozen such that it would not learn and surpass the generator any further. This would allow the generator time to catch up to the discriminator's performance before the dual training continued. To strengthen the generator further, an alternating training series was used. For each epoch of training the discriminator underwent, the generator was trained five times. These solutions slightly decreased the effects of discriminator overperformance, but the issue persisted.

The process of data scaling remedied this issue. By scaling the data to values in a range that could be converted into a MIDI file, the generator's output more closely resembled the training data. The discriminator's task became harder: instead of distinguishing between MIDI training data and distinctly non-MIDI generator outputs, its task was to distinguish between musical MIDI data from the training set and generated MIDI data. This task required the discriminator to recognize patterns in music and look for those patterns in generator outputs, as opposed to simply recognizing that a given generator output does not correspond to the MIDI Standard. As a result, instead of losses diverging rapidly as shown in Figure 6, the generator and discriminator were of

comparable magnitude after 5 epochs, as shown in Figure 7.

Epoch	Discriminator Loss	Generator Loss
1	0.495	1.022
2	0.255	1.042
3	0.212	1.035
4	0.194	1.024
5	0.165	1.011

Fig. 7: Losses of the discriminator and generator over the first five epochs in an later implementation of the GAN (after data scaling).

C. Loss Plateau

The loss values of both the discriminator and generator plateaued fairly quickly, as shown in Figure 8. This occurs when the loss function converges at a local minimum or a saddle point, where the gradient approaches zero. This minimum is not the most optimal model—that would be the absolute minimum—but since the gradient becomes zero or approaches it, the network's weights adjust by very little when it tries to learn. As a result, the model stagnates quickly. This is a common issue with training neural networks, and it was possibly caused by a small learning rate. If the learning rate is too small, the model takes equally small steps during gradient descent and can become almost trapped in a sub-optimal local minimum.

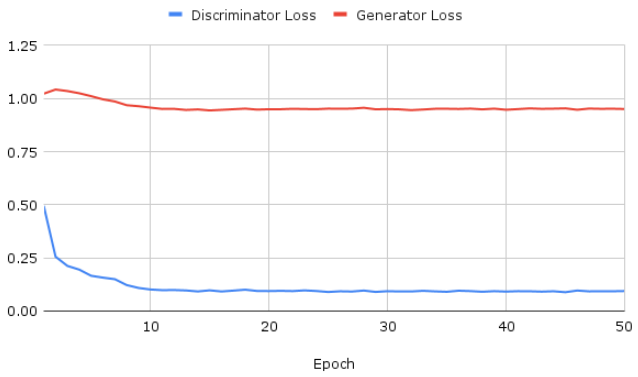

Fig. 8: Discriminator and Generator Loss as shown over 50 epochs of training. Loss plateaued after 10 epochs.

V. Conclusions

This project demonstrates the complexity required to create realistic musical MIDI files. While this simple implementation of a GAN was able to produce usable MIDI data that displayed pattern and pieces of melody, it ultimately lacked the ability to process the recurrent and intricate patterns present in music. In future improvements, the GAN model could utilize recurrent neural networks (RNN) with bidirectional long short-term memory (LSTM) cells to eliminate the issues present with understanding music complexity. The addition of LSTM cells would effectively allow the networks to remember previous outputs, improving their ability to recognize patterns that occur sequentially in music.

Although the functionality of the GAN as a whole was limited, the data processing algorithms applied to the musical data were successful. The reduction in size of a full MIDI file after processing was marginal, but the data contained within the files was significantly simpler and well standardized for machine learning purposes, while still retaining complexity and musicality. This procedure could have potential for further experiments in music generation through machine learning. The marginal loss in musical quality is compensated by the extreme simplicity of the file after processing and the fixed, adjustable size.

A. Github

Code described in this paper as well as some example MIDI format results can be found at the following repositories:
https://github.com/MF270/gset-22-midi
https://github.com/MF270/gset-22-midi-gan

Acknowledgement

The authors of this paper would like to sincerely and gratefully thank the following: the Rutgers School of Engineering, Rutgers University, and the NJ Office of the Secretary of Higher Education; Governor's School Alumni for their invaluable support; Governor's School Director Dean Jean Patrick Antoine for making this research possible; Project Mentor Sachin Mathew for their crucial guidance all throughout the research process; Head Residential Teaching Advisor Ian Joshua Origenes, Research Coordinator June Lee, and Residential Teaching Advisor Hamzah A. Farooqi for their guidance; and everyone involved in the Governor's School for Engineering and Technology for their indispensable support.

References

[1] S. Walter, G. Mougeot, Y. Sun, L. Jiang, K.-M. Chao, and H. Cai, "MidiPGAN: A Progressive GAN Approach to MIDI Generation," IEEE Xplore, May 01, 2021.

[2] de Oliveira and R. Oliveira, "Understanding MIDI: A Painless Tutorial on Midi Format," Research Gate, pp. 1–7, May 2017, Accessed: Jul. 15, 2022. [Online].

[3] "GM 1 Sound Set," MIDI Association, 1991. https://www.midi.org/specifications-old/item/gm-level-1-sound-set

[4] R. E. Uhrig, "Introduction To Artificial Neural Networks," Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics, pp. 33–37, 1995, doi: 10.1109/iecon.1995.483329.

[5] I. Goodfellow et al., "Generative Adversarial Nets," Arxiv, 2014. Accessed: Jul. 15, 2022. [Online]. Available: https://arxiv.org/pdf/1406.2661.pdf

[6] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative Adversarial Networks: An Overview," IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 53–65, Jan. 2018, doi: 10.1109/msp.2017.2765202.

[7] Colinraffel, "The Lakh MIDI Dataset v0.1," colinraffel.com. https://colinraffel.com/projects/lmd/ (accessed Jul. 15, 2022).