musG_dev, the Deep Music Generator

By

Terry Wang, Rima Mittal, Joshua Goldberg

Supervisor: Yuri Balasanov

A Capstone Project

Submitted to the University of Chicago in partial fulfillment
of the requirements for the degree of

Master of Science in Analytics

Graham School of Continuing Liberal and Professional Studies

March, 2020

The Capstone Project committee for Terry Wang, Rima Mittal, Joshua Goldberg

Certifies that this is the approved version of the following capstone project report:

musG_dev, the Deep Music Generator

Approved by Supervising Committee:

Yuri Balasanov

Abstract

We propose the application of Deep Learning to help humans generate new music. There have been recent attempts around this area (Google's Magenta) that focused mainly on producing music through automation. However, our approach is to form a partnership between humans and AI (artificial intelligence) in the music generation process. With this augmentation mindset, there are two phases of work: 1) train a Deep Learning system, with a Generative Adversarial Network and 130,000 midi files, that focuses on music generation; 2) incorporate user feedback back into the system to align model training with individualistic taste.

While maintaining the spirit of our approach, we narrowed the scope to the first step as an obtainable checkpoint. Our training concluded with sample outputs we consider interesting, and most importantly, music. We note a few obstacles that continue to make the first phase of work challenging: mode collapse during model training (i.e., learning ceases), model architectural design, and hyper parameter selection.

Keywords: deep learning, music generation, generative adversarial model, midi

Executive Summary

We propose the application of deep learning to generate new music ideas to aid individuals in their music creation process. This approach in using deep learning differs from current approaches that emphasize automation which removes human input in the music generation process. The purpose is to have an interaction between human and AI. As such, it should be seen as an augmentation. The generated sample results can be perceived as musical. However, we include a few ideas on the future improvement of the model training process.

Our primary model is a GAN with two components: a generator and a discriminator (also called critic). The generator takes a random input and outputs a midi sample (a file type for music). The discriminator takes a midi sample as the input and outputs a classification of "real" or "fake". For real samples, we used a midi data repository containing 130,000 midi files across many different genres of music. For this project, we specifically used Baroque music. We structured our data with two main components in mind: the pitch and the duration of the note. The pitch was one-hot encoded with 128 possible pitches. The duration of the note was broken out by start time and end time, which are positive float vectors. This structure seems to perform the best out of the many we tried.

In order to set up the model training for success, the tempo of the music had to be made constant and varying keys had to be tranposed to C major and A minor. Additionally, data had to be further pre-processed so the deep learning system can interpret and learn the structure in the data. To perform these steps, our work includes a modified version of the

python package pretty_midi. We adapted pretty_midi's main engine to manipulate midi files with varying tempo and key changes, adapt midi files into data structures usable by python, and reverse-adapt our output back into a midi that can be used by music production software. Lastly, we provided an assortment of utility functions and model selections, which we made into a python package together with the pretty_midi.

Acknowledgements

We would like to thank our mentor, Prof. Yuri Balasanov, for his guidance and kind support during our investigations.

Prof. Mildred Rey provided valuable feedback to improve the style and content of this paper. We truly appreciate her help.

Profs. Lawrence Zbikowski and Sam Pluta provided additional feedback. Thank you.

Table of Contents

List of Figures

List of Tables

Introduction

Perfecting the creative process is a goal across all forms of art (ex. music, photography, painting, singing, poetry, writing). Finding a place to start can be the most challenging (writer's block). A musician can go down a melodic path that leads to a dead end. The process of discovery and satisfaction by the creator is iterative trial and error. To use a model as an analogy, this would be similar to an infinite grid search for hyperparameters — something that is not practical in the field of machine learning. Astonishingly, musicians today have been able to create music with their bare intellect and network of inspiration.

While there are successful musicians in today's creative environment, we think there is always room for improvement in the creative process. This deep learning approach aims to be the catalyst for that improvement by handling the difficult and arduous task of discovery and ideation for musicians and individuals interested in creating music by generating new melodic ideas in a short amount of time based on the musician's preferences, dramatically enhancing the creative capacity of the musician.

The purpose of the project is to develop a deep learning-based music generator (instrumental music) that has a good understanding of the language of music and can generate human-like outputs. Ultimately, the generator supports musicians and non-musicians alike in developing and refining musical ideas. We divided the model phase of the project into two parts: Generation and Validation.

By doing this iteratively, the weights and parameters of the model will hopefully converge

to a point at which the music generated is satisfactory to the user so they can use the output of the model or save the weights for future use. We expect the algorithmic process to able to explore more music combinations than any individual and thus provide a more holistic and creative approach to music creation.

At the current stage, we are focused on taking the first step towards our stated goal, and have created a deep learning model that shows great promise of generating human-like musical output.

Background

We propose the application of Deep Learning models to help individuals generate interesting music ideas with the goal of aiding musicians to use these computer-generated musical ideas to enhance their music writing process. Our model focuses on generating melody instead of the sound. We use midi files consisting of music notes, as training data. In addition, we explore a unique model structure where the time dependency inherent in music is not explicitly given to the model but is inferred by the model through the deep neural network structure. Despite this, we have observed that the model was able to learn time-dependent musical structure, such as the pre-dominant to dominant to tonic chord progression.

We note that the model exhibits certain unconventional behavior. This can be attributed to the model using a different perspective to create music compared to humans. This behavior may be useful in inspiring musicians to create new music.

Historical Background

in 1957, A 17 seconds long melody named "The Silver Scale" was the first ever music generated by computer (nSFTMC, 2020) using synthesizers. Around the same time, musicians and statisticians started to attempt to decode the language of music using statistical methods. Some of the very early algorithms used stochastic Markov Chain models for generation and rule filtering as well as more advanced Bayesian models (Temperley, 2007). In 1990s, David Bowie built the Verbasizer, which implemented a random re-ordering of groups of words and sentences to produce potentially significant lyrical combinations (pro-

vide an example of this).

Since then, computer music has been gaining a lot of public attention. In present time, there is an entire industry built around AI generated music including Flow Machines (Flow Machine Authors, 2020), IBM Watson Beat (IBM Corporation, 2020) and Google's NSynth (Google AI, 2020).

Methodology

The purpose is to build an effective platform to help artists in their creative process and assist them in music composition. Note that artists do not have to be professionals. Any individual interested in creating music should find the platform useful and empowering. The first concrete step we need to take to achieve that goal is to build a music generation model that is able to generate music that is similar to what a human would compose in the style of music the model is trained on. In this section we will dive deep into the details of how we preprocess the training data and how we set up the model for music generation.

Data

The training data used to train our model is a collection of midi files that was compiled by a Reddit user see reddit page and put up for download. There are about 130,000 midi files from a wide range of genres included in this collection. Our training, however, centers around a small subset of the files that are of 4/4 rhythm, in order to reduce the chance of complication from training with a wider range of rhythmic patterns.

In addition, in our pre-processing script, we took steps to "normalize" the samples, in terms of making the tempo constant as well as transposing all songs to the key of C major (or A minor if in minor key). The efforts to normalize keys is not always successful. In some midi files, tempo and key changes are included as part of the metadata, which we can then extract and make changes accordingly. But in others, the metadata is not included in which

case we do not adjust the keys.

In order to understand how to convert midi files into a format that can be an input to a deep neural network, it is necessary to introduce the basic structure of midi files. Midi files (.mid) are digital records of musical "events" divided into various tracks, where each track contains a voice, an instrument, or a line of melody. For example, a midi file containing piano music might have two tracks representing the notes played by the two hands of the human piano player. Each time the human player plays a note, the midi file would record the following information: which key is pressed; the time the key is pressed; the time the key is released; how hard the key is pressed; what the pitch bend is. After collecting this information for each and every note in a song in a midi file, we would have most of the information needed to reproduce the song in its entirety down to some basic expressive elements. In the midi metadata, some other relevant information is stored, such as tempo, key, tempo changes, etc. Midi files are not music by themselves because they only contain the information needed to produce music: it is the digital equivalent of sheet music. Midi decoders and synthesizers are needed to translate midi files into music.

If we strip the midi file structure down to its most basic elements, it would look something like this:

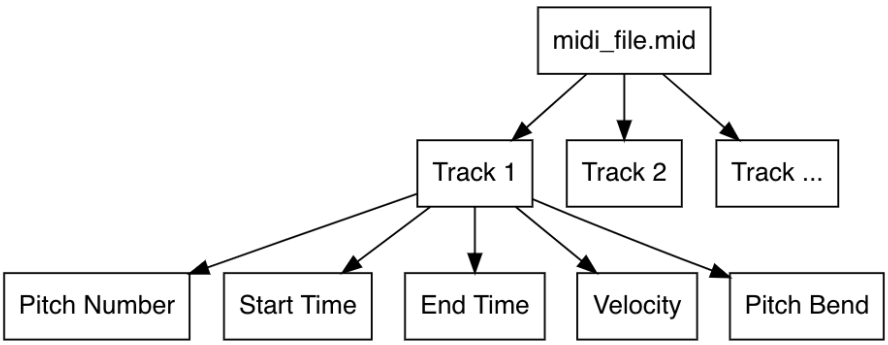

Figure 1: Midi File Structure

We can imagine the midi file to be a dictionary, where each track is indexed by its name, and in each track we have a tabular structure of data consisting of the following columns: pitch number, start time, end time, and pitch velocity (we will ignore pitch bend for now). For example, if we look at the first three notes played by the right hand in Beethoven's Moonlight Sonata, we can tabulate them as follows:

Table 1: Midi File Structure Example

Pitch	Start Time	End Time	Pitch Velocity
56	0.00	0.40	33
61	0.40	0.81	26
64	0.81	1.21	26

Table 1 tells us the following: At time 0.00 seconds, pitch 56 is played with velocity 33, and the note stopped at time 0.40 seconds; then at time 0.40 seconds pitch 61 is played with velocity 26, and the note stopped at time 0.81 seconds; and so on. We can see clearly how a piece of music can be represented digitally via the midi file structure.

Finally, different components of the midi file have different data types. The `Pitch` and `Pitch Velocity` columns take only an integer between 0-127, which is a discrete structure that required one-hot encoding. The `Start Time` and `End Time` columns are positive floats to represent the number of seconds since the start of the song. Since the goal of the model is to generate novel melody ideas, we decided to focus only on `Pitch`, `Start Time`, and `End Time` from the midi file information as well as limiting the output of the generator to 20 notes at a time. We made the decision to only focus on these variables in order to focus on learning the melody and not expressiveness as well as to cut down on complexity. Again, `Pitch` is the piano key pressed, and `Start Time` and `End Time` are time marks at which the keys are pressed. These features had an impact on the data pre-processing.

For data fed into the model, we use a processing script to rewrite a pre-selected subset of these midi files into numpy arrays in accordance with the above-mentioned structure: each song was represented by a series of numpy arrays, each representing a track, and each array will have the tabular structure of 130 columns: a 128-length one-hot vector for `Pitch`, one column for `Start Time`, and another column for `End Time`. We then use another training script to randomly make 20-note samples from the training data and arrange them into a numpy array. We need 20-note samples in order to make sure that the real and fake samples have the same dimensions.[1] Therefore, our training dataset has dimensions of (x, 130, 20) where x represents the number of real samples generated for training, the number 130

[1] One explanation is that the the fake input to the discriminator when training the discriminator is different from when training the adversarial model: the input to train the discriminator has a preprocess step to take argmax of the softmax vector and make a one-hot vector on the argmax index, meaning that it has the same format as the real samples, but during the adversarial stage this is not the case as the discriminator is fed output from the generator directly without preprocessing. This will be addressed with an update to the model. However, there is also a good chance that there are other reasons for the collapse in losses, as this phenomenon is observed in many other GAN models.

represents the three columns (`Pitch`, `Start Time`, `End Time`), and 20 represents the 20 notes sampled.

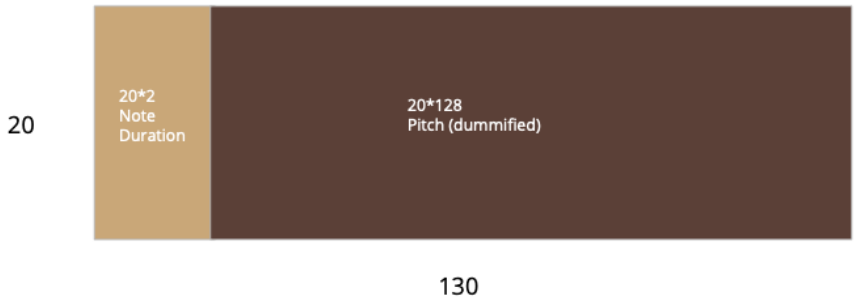

Figure 2: Training Data Structure

The decision to make the pitch a one-hot vector was made because in the Western musical tradition pitches do not exist on a linear or ordinal scale as we typically think of them. Musical theory does have a pattern and order, but for the purposes of modeling, this does not exist. One-hot encoding pitches will allow us to treat each pitch as and independent object. In addition, our model will be able to output softmax for pitches, therefore leading to pitch probabilities that can be further investigated.

Modeling Framework

Our model follows the Generative Adversarial Networks (GAN) architecture. GANs are a special case of deep learning models where the goal is to turn random noise into some kind of output (images or, in our case, music) which captures the essence of the training data. For example, they have been successfully deployed to generate realistic visual images.

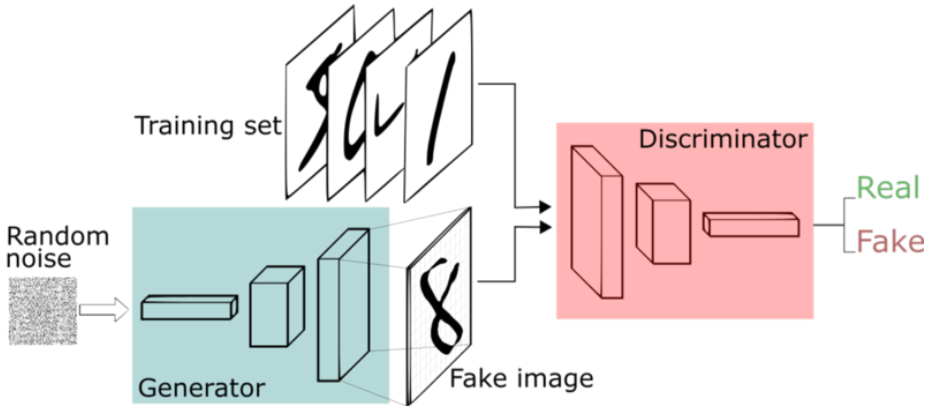

Figure 3: GAN Structure

The basic structure of the model has three parts: the generator, the discriminator, and the adversarial model, which combines the generator and the discriminator. The basic idea of the model is to train the discriminator with both real samples (from training data) and fake samples (generated by the generator), and then train the generator with the error from the discriminator. The goal is not to minimize the error of the generator or discriminator, but rather to achieve a balance between the generator and the discriminator so that they get better together.[2]

Generator

Our generator is a deep neural network with the following basic structure:

[2] p.113, (Foster, 2019)

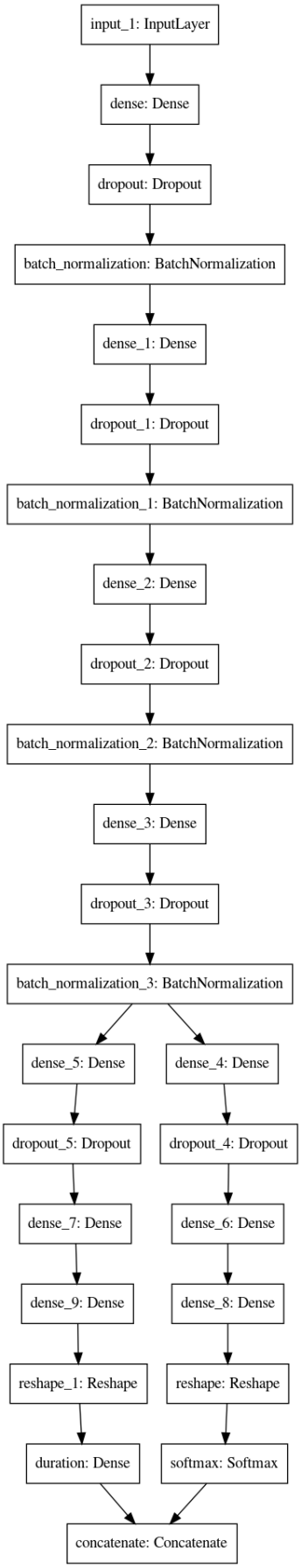

Figure 4: Generator Structure

The input of this model (figure 4) is a 128-length vector of Gaussian noise. The noise will go through four dense layers (currently set at 256 nodes each) with ReLU activation, dropout, and batch normalization. Then, the model splits in half, one each for pitch and duration generation. The pitch part of the model then upsamples the output from the last layer into 20×128 nodes, where 20 is for the 20 notes to generate, and 128 is for the one-hot vector of pitches. Similarly, we upsample duration from the same layer into 20×2 nodes, where 20 is for the 20 notes to generate and 2 is for start and end times. Afterwards, we reshaped these nodes into the correct shapes, (20, 128) for pitches and (20, 2) for duration, and apply the corresponding activation functions to the correct axis: softmax for pitch (for pitch onehot), and ReLU for duration (for outputting positive numbers). The generated pitch and duration vectors are then concatenated into one single array of dimensions (20, 130) as the output of the model.

Using this modeling structure, we ensure that pitch and duration of one generated music sample is generated by a single model and one single noise input. The dense layers preceding the split in the model will allow the model to learn latent features and rules of music before feeding that latent representation of the final product into the part of the model that turns latent vectors into pitch and duration.

Conventionally, music generator models are usually time-dependent models such as the LSTM, with some added features like the attention mechanism for model to learn how to emphasize on repeated patterns. While those models work, they are more complex than simple dense neural networks deployed in our approach. We observe that, even with a more simplistic (and thus less artificial) model, the generator used in this research is still able to learn basic musical structures.

Once the generator made its predictions, we fed them through another function we wrote to convert the softmax vector for pitch into a one-hot vector, so that the output would be identical to the real samples in terms of format.

Discriminator

The discriminator's job is to take a sample and judge whether this sample is real (composed by a human) or fake (generated by the generator). Our discriminator model is a simple dense neural network with input dimensions of 20×130 (for the 20-note sample, either generated or real) and outputs a float between 0 and 1 (using sigmoid activation) that represents the probability of whether the sample is fake (0) or real (1).

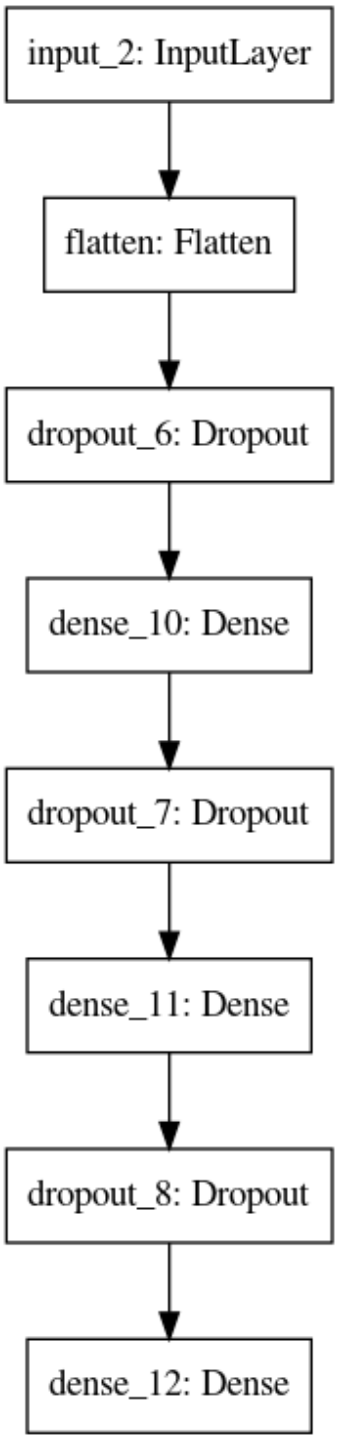

Figure 5: Discriminator Structure

As we can see, the model is a straightforward dense neural network (with dropout and batch normalization layers). The model is deliberately kept simple here because unequal speeds

at which the generator and discriminator learn was a big concern during the training. More often than not in this case, the discriminator learns faster than the generator. Surpsingly, we found that a simple discriminator model provided an environment that was conducive to long-term learning between both the generator and discriminator.

Adversarial Model

The adversarial model is simply the generator and discriminator organized in sequence as illustrated below:

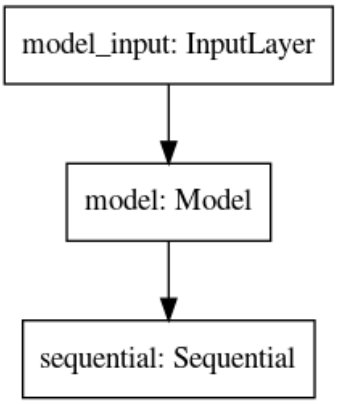

Figure 6: Adversarial Model Structure

However, in the adversarial model, the weights of the discriminator are frozen because the error from the discriminator must be fed only to the generator in order to train the generator. We will expand on this topic in the next section.

Training GAN models

Training GAN models is not a straightforward process and requires iterating through the following specialized steps:

1. **Train the discriminator with real samples.** We use a function to sample from the real training data. The size of this sample is usually 1/2 of the batch size parameter of the training script. We then feed these data and the target variable (a vector of 1s

indicating they are real samples) into the discriminator model. The resulting error is the error of misclassifying real samples as fake. We use binary crossentropy as the loss metric.

2. **Train the discriminator with fake samples.** We use a function to generate fake samples using the generator model that we set up. The size of this sample is 1/2 of the batch size parameter of the training script. We then feed these data and the target variable (a vector of 0s indicating they are fake) to the discriminator model. The resulting error is the error of misclassifying fake samples as real. We use binary crossentropy as the loss metric.
3. **Train the adversarial model**. We feed the adversarial model with a number of batches of noise as input, but mark the target variable as a vector of 1's, or telling the model that these are real samples. The adversarial model is the generator and discriminator organized in sequence, so the random noise input will be converted into fake samples of music before getting fed into the discriminator, where the output is a classification of whether the sample is fake or real. Since we technically mislabeled the data by using 1s as the target variable, the error was negatively correlated with how well the discriminator performs — the better the discriminator did, the larger the error, thus the more the weights will be adjusted in the generator, and vice versa. The discriminator is not affected because its weights were frozen when training the adversarial model. This was expected to balance the rate of learning for the generator and the discriminator and should have prevented either of them from outpeforming the other. In many cases this held true; however, as training persisted for an extended period of time, it became increasingly difficult to maintain this balance.

GAN models are notoriously fickle and require special care to train properly.[3] In our training script, we set up a mechanism to collect the errors from the discriminator and the generator in order to keep track of training progress and detect any frequently occurring

[3]For details see p115, Generative Deep Learning

learning issues, and most of our work focused on ways to address these issues. We will address our learning points in the Findings section.

Findings

We divide our findings into two separate sections: GAN Training and Assessment of Model Output.

GAN Training

GAN models are notoriously hard to train properly. During the training of our model, we encountered numerous issues and had some success in addressing them. The main issues are:

1. Loss balancing. In many cases, the losses from the discriminator and generator often collapsed to zero or become extremely large. To clarify, discriminator loss is loss from training the discriminator specifically and generator loss is the loss from training the adversarial model.
2. Mode collapse. The model learned to generate one kind of output that was able to fool the discriminator regardless of input.

Loss Balancing

Using the data we collected from the training process, we found that the GAN training can be summarized into three distinct stages:

1. The initial chaos. The generator and discriminator are trying to balance each other

out and the error can vary drastically from epoch to epoch. The loss of either the discriminator or generator loss can be much larger than the other. It often settles into a more balanced stage, but this is not a given as we observe some models being unable to get past this initial stage.

2. The stable equilibrium. The model enters into a stable stage characterized by a similar magnitude of losses between discriminator and generator across different epochs. Judging from the quality of the output, it is during this period when the model learned the most from training data and generated the most musical samples.
3. The final collapse. This stage is characterized by a steady increase in loss magnitude in either the generator or the discriminator or both until the losses of either model collapsing to zero. There were even rare cases where the losses from both the discriminator and generator went down to zero. In that scenario, learning stops and mode collapse set in.

In the following plot of average epoch losses from one of the training sessions, we can observe these three distinctive stages:

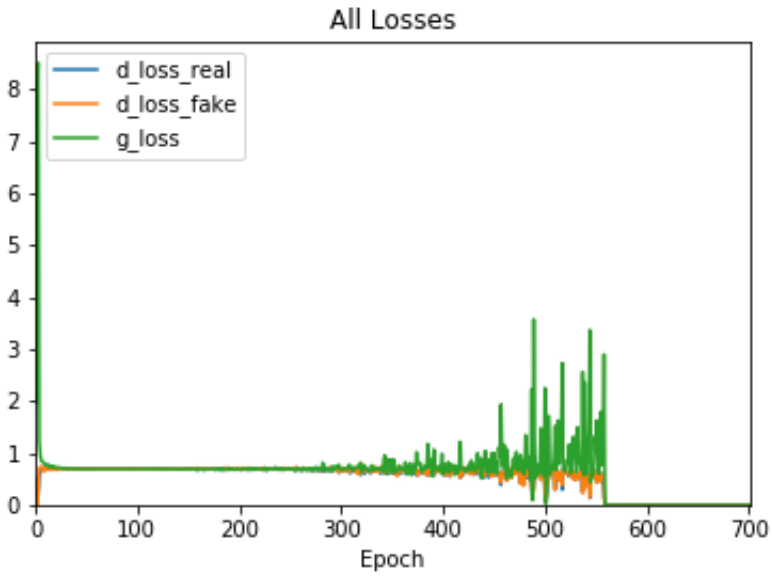

Figure 7: Training Loss Graph

During the first 25 epochs or so, we observed initially high but declining losses for both the discriminator and the generator. Thereafter, we enter the stable stage that lead to significant and high-quality learning where the loss stablized around 0.7 for both the discriminator and the generator. However, losses became unstable again starting around the 300th to 400th epoch mark and by the 500th epoch the generator loss increased to around 2 while the discriminator loss steadily declined. Finally, at around epoch 560, all losses went to zero and the model ceased learning.

It seems that the model showed the most promise during the stable equilibrium stage. While it is hard to show the quality of the generated samples in a mathematic way, we can clearly demonstrate this by hearing generated samples at different number of epochs. Here is the link to our github page to access the generated samples at every 100 epochs during the training session above. At the first epoch, the output from the generator has no sense of rhythm or melody and its output is simply random notes. After 100 epochs, it is clear that there are improvements in rhythm, but the melody is still lacking. At epoch 200 the notes sound much more pleasant as they generally come from the same key. We see more improvement at epoch 300 and 400, which we believe is the best sample generated. However, the improvement trends are reversed starting at epoch 500, and the samples generated become increasingly nonsensical.

This observation shows that the conventional GAN model training procedures that try to balance discriminator and generator learning have much to improve upon, as it fails to prevent the generator to learn how to generate samples that is able to produce zero loss both for the discriminator and at the adversarial stage[4].

[4]One explanation is that the the fake input to the discriminator when training the discriminator is different from when training the adversarial model: the input to train the discriminator has a preprocess step to take argmax of the softmax vector and make a one-hot vector on the argmax index, meaning that it has the same format as the real samples, but during the adversarial stage this is not the case as the discriminator is fed output from the generator directly without preprocessing. This will be addressed with an update to the model. However, there is also a good chance that there are other reasons for the collapse in losses, as this phenomenon is observed in many other GAN models.

It also seems that the loss collapse follows right after the wild oscillations in losses. Since the evidence seems to show that losses stability and generated sample quality are positively correlated, ideally we would want this stage of stable equilibrium to last as long as possible. To achieve that, we can either improve on the model structure, or revise the training procedures. We will discuss some of our ideas for improvement in a later section.

Mode Collapse

Mode collapse is a common problem for GAN models[5]. It happens when the generator finds one single pattern that is able to fool the discriminator and then mapping the random noise input to that pattern, so that all outputs from the generator are very similar.

An example of mode collapse is found in one of our training sessions. Judging from the samples, the model seems to learn relatively well during the first few hundred epochs, but starting from epoch 900, the generator will only generate the same sample over and over again. Use this [link] (`https://github.com/terrywang15/museG_dev/tree/master/2020-02-11%2022-13`) to download and listen to all the samples.

After some investigations, it seems that mode collapse is related to the loss issue described above: mode collapse only happened after all losses collapsed to near zero. Notice in the below graph recording the average epoch losses from the training session that generated the samples mentioned above, the generator started to generate the exact same samples around epoch 900, shortly after the collapse of losses:

[5] p.113, (Foster, 2019)

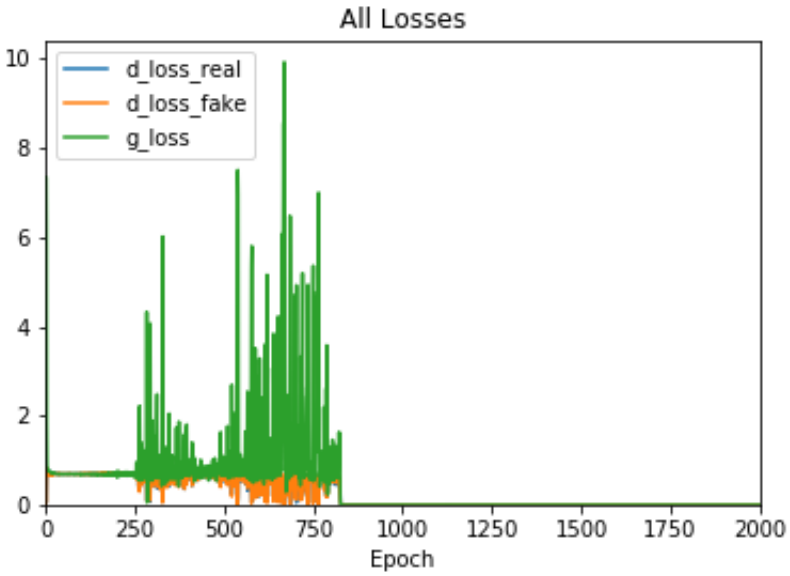

Figure 8: Mode Collapse Loss Graph

It is unknown at this point whether the collapse in losses and mode collapse are related or if we encountered a special case. However, it is clear that we will need to have a mechanism to monitor losses which will indicate to us whether the model is learning effectively.

Ideas for Improvements on Model Structure and Training

1. Model Structure Changes to Separate Pitch Loss and Duration Loss

One hypothesis that we pursued is that having one dedicated generator for pitch and duration would work better than our baseline model. The idea behind this is that, since pitch and duration are very different data types that differ in magnitude when comparing typical losses generated, it would make more sense to separate them so that loss is distributed proportional to how the model performs in terms of pitch and duration. This way, if a model does well in pitch and not duration, for example, the model would devote more losses to duration and less to pitch.

We proceeded to try this idea out, and had one generator each for pitch and duration. How-

ever, preliminary results are not very encouraging as the model has so far failed to generate any musical sounding samples.

2. Change to a Different Loss Metric

One idea that has worked well in music generation is the Wasserstein loss, which according to its authors is a more meaningful loss metric that correlates better with output quality as well as stablizes the GAN training process[6]. We have yet found time to implement this in our model, so we will leave it as a potential next step.

3. Automated Monitoring of Losses During Training

As we have observed, preserving and prolonging the long-term loss stability stage during training should be the goal. We propose to have the following system set up in the training script:

1. Record losses during every batch
2. At the end of each epoch, calculate average losses (for generator and discriminator) for that epoch, and record it
3. Wait until model "warms up" or enters the stable learning stage. We can manually define how this look like or use a metric (such as stationarity)
4. Establish a metric for trend stability. This can be the standard deviation of the new epoch's losses compared to the distribution of losses from the stable process
5. Stop training when losses shoot up for x number of epochs. We have observed that once the training enters the final collapsing stage it rarely finds its way back, so its better to terminate training rather than keep going in the hopes of finding the stable stage again

[6]For details see p115, Generative Deep Learning

4. Automated Model Structure Changes During Training

A more advanced idea that we have been considering is to combine automated loss monitoring with flexible model structures. The idea would go as follows:

1. Make a basic generator/discriminator model with as few layers as possible
2. Train model until it enters stable stage
3. Monitor losses, stop training when losses start to increase
4. Freeze current model weights
5. Add new layer to generator and/or discriminator (can have some conditions to trigger this for generator or discriminator)
6. Train this new model until loss start to increase. Repeat the process as needed or use a stopping criterion

This way, we hope to maximize learning during the stable stage for each layer while avoiding many of the problems that we have encountered during our training. We believe this is a modeling technique that we should attempt in our next steps.

Musical Assessment of Model Output

One of the most significant achievements of our generator is that it successfully learned many musical structures despite being fed random snippets of data, demonstrating that GAN model structure is quite flexible in dealing with different forms of data and formats. To illustrate the musicality of generated outputs, we will focus on one generated sample that showed the success of our generator.

The particular sample can be found via this link. The notes of the sample is shown in the following table:

Table 2: Generated Midi Sample

Pitch	Start Time	End Time
C5	0.000000	0.020455
A3	0.000000	1.888636
G4	0.000000	2.688636
F#3	0.500000	4.229545
D5	2.313636	3.797727
F#4	2.884091	2.979545
G3	3.020455	6.768182
E3	3.181818	3.677273
D4	3.827273	4.877273
D5	3.981818	4.375000
E5	4.145455	4.156818
C4	4.747727	5.856818
E4	5.190909	6.336364
A3	5.904545	6.263636
C5	5.938636	7.293182

Pitch	Start Time	End Time
E3	6.606818	7.634091
B3	8.020455	8.075000
D4	8.511364	8.545455

When listening to this sample, one can't help but notice how "musical" it sounds. This is because it has a structure that adheres to classical western music theory, as well as pattern repetition, things that are essential to human enjoyment of music (Levitin, 2006).

The sample begins with an ambiguous chord (A3, G4, C5) which is a transposed version of an A minor 7 chord with the fifth missing[7], but this chord is abruptly transitioned to the (F#3, A3, G4) chord, a very dissonant chord that, although jarring, start to pull the song in the direction of the E minor key because of the introduction of the F# note that is characteristic of the G major/E minor, so overall the beginning sequence establishes the key structure of this sample. The F#3 note also has a strong tendency to go up a half-step and resolve to G3, which we will learn that it does right after.

The next sequence of notes are very interesting. It starts with the melody line of D5, F#4, G3, and E3, a remarkably smooth transition that implies D major with the major third as the root (F#), which transitions to E minor (G3, E3)[8]. There are several remarkable things happening here: first, the generator chooses a melody line that spans a wide range of octaves, giving it a sound that is very characteristic of piano music; second, the melody smartly includes the note G3, which serves the dual purpose of resolving the jarring F#3 note from the beginning sequence and anticipating the next chord (E minor).

After this sequence, the sample has firmly established an E minor feel. Next, it features

[7] There are other interpretations for this chord of course, and this is by no means a professional assessment strictly based on music theory.

[8] See (Wikipedia contributors, 2020a). The D major chord should transition to G major but can also transition to E minor.

an additional transition to C major then A minor through a very interesting setup (D4, D5, E5, C4, E4, A3). The octave-skipping D notes at the beginning is again found in many piano music and serves to reinforce the the D note, a note that clashes somewhat with the E minor chord the sample is in at that point and has a tendency to resolve to E or go down to C. After introducing this tension, the sample deftly does both of these things: the leading D5 goes up to E5 as a glide, the D4 goes down to C4, exactly what we would expect. But notice how the two resolutions went in different directions: D4 went down a step, while D5 went up a step. This kind of arrangement is very common in Baroque era music, which generally shuns multiple voices moving in parallel[9]. Finally, an E4 is played after C4 to anchor the chord with a major third and removing ambiguities to what the implied chord is. This is followed by an A3 note which is again a smooth transition because it transitions to a new chord (A minor) by only adding one note (A3) while keeping the other notes (C4, E4) the same.

The ending sequence also features an interesting movement that could be interpreted as an attempt to make a key change in the context. The note sequence (C5, E3, B3, D4) can be understood as an attempt to move from the A minor or C major to a B major. Either way, it would entail a parallel shift of voices up or down a whole step, something that Baroque does not like. The generator's solution to this problem is to spread out the the two voices by almost 2 octaves (C5, E3) then bringing them together closer, with the C5 moving down to D4 and E3 moving up to B3. Also, the kind of resolving chords to something a half step lower (C chord to B chord) is commonly heard in Jazz and certainly makes sense in music theory. The reason that this sounds like a transition to a different key is because B is never heard throughout the sample up to that point, in addition to the sudden break in rhythmic pattern.

Speaking of rhythms, this generated sample also seems to have learned to produce coherent rhythmic structure. For example, in this sample melody line (D5, F#4, G3, E3) has a

[9] See (Wikipedia contributors, 2020b)

rhythmic pattern consisting of a long note followed by several short notes. The next melody line (D4, D5, E5, C4) references the structure of the preceding notes by reversing it: it now has several short notes followed by a long note. This kind of rhythmic construction is what gives a melody line a "human" feel, because it creates a pattern that repeats itself and at the same time is different in each repetition. Moreover, the rhythm and melody seem to work extremely well together, showing the success of our model structure which generates pitches and duration from the same latent space.

We now can see how the generator under the GAN structure achieved outputs of the following:

1. It learned basic music theory such as keys and chords;
2. It learned what rhythm sounds natural
3. It learned to consider past and future notes as a whole and try to create repeated patterns;
4. It learned highly abstract ideas such as chord progression and avoidance of clashing voices.

Even more remarkable is how this is achieved without relying on explicitly encoding any artificial structures, such as LSTM or the Attention mechanism, in the generator in order to "force" the model to learn self-referencing behavior and pattern recognition. It shows the remarkable flexibility of the deep neural network architecture. At the same time, more attention must be paid on the training steps in order for this model to learn as much useful information as possible.

Conclusion

The deep neural network structure implemented in our generator was able to generate human-like musical output that exhibits understanding of music theory, Baroque style, pattern recognition, and a preference to generate passages that is self-referencing. It shows great promise to become an additional tool for musicians to generate new musical ideas of certain styles.

The model in the current state is plagued by the inability to maintain stable learning and the problem of mode collapse, which are related problems. We propose that in our next steps, we keep iterating on the model structure to improve on these issues.

Next Steps

We see the following directions that we can take to further improve our model:

1. Make training data more consistent in terms of style and music instrument. We believe that the more consistent the input, the better and faster the generator can learn. However, this is a quite time-consuming task to sift through each and every midi file to find which composers, passages and instruments to include.
2. Upgrade model structure. While our relatively simplistic model structure is able to achieve quite a bit, we believe that adding additional complexity will marginally increase the performance of the model. We can also investigate the impact of increasing layers to our baseline generator or discriminator.
3. Implement new training script that is able to monitor losses. This would involve the development of a loss stability metric and stops training when loss instability is detected.
4. Implement flexible generator/discriminator in conjuction with loss monitoring. This idea is discussed in the Findings section and will enable each layer of the model to learn as much as possible before loss starts to increase, thus avoiding the problem of loss increase.

Appendix A

Additional Experimentations in Model Structure

We have performed additional experiments on our baseline model, which is shown in the Methodolody section. All changes in model structure are relative to the baseline model.

Increase Nodes per Layer in Generator from 256 to 512

For this run, we kept the basic model structure in place, with the only change being increasing the number of nodes in each layer in the generator to 512, from 256. Additional nodes should help the model break down the problem deeper and gain extra levels of abstraction of the music generation process.

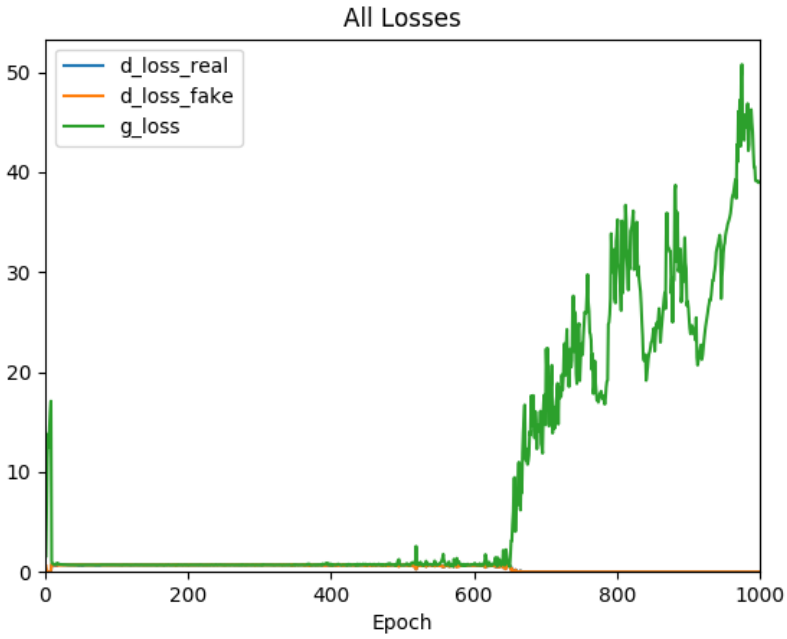

Figure A.1: Loss Graph, 512 Nodes

We see that during this run the stable learning stage ended just around 650 epochs, an improvement on the baseline run whose stable stage lasted until around 500 epochs. This could be a promising direction to improve the model, but addtional experimentation is needed to confirm this.

Adding a Dense Layer to Generator

For this run, we kept the basic model structure in place, with the only change being adding an additional Dense layer (along with its corresponding dropoff and batch normalization layers) to the generator. Additional layers should work similarly to adding nodes to existing dense layers and help model to learn higher levels of abstractions.

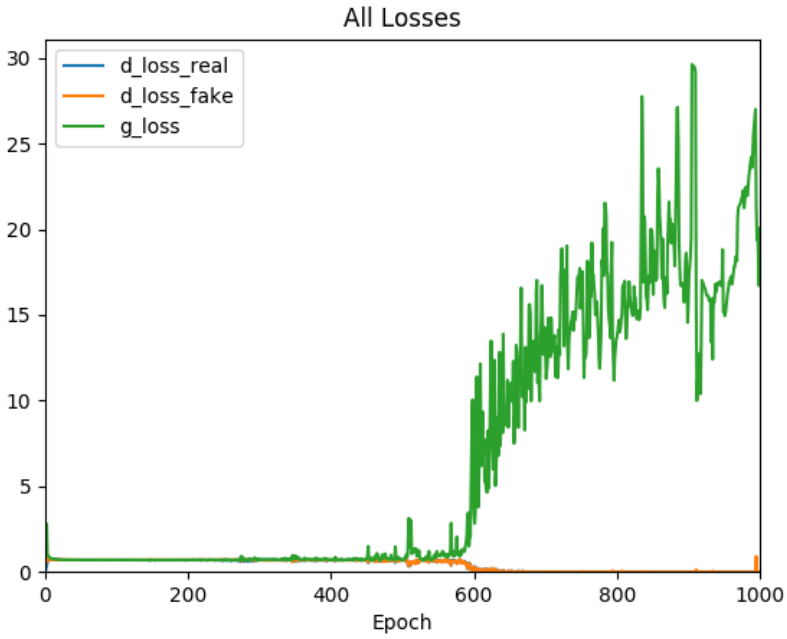

Figure A.2: Loss Graph, Extra Dense Layer

For this particular run, we see the stable stage ended just before 600 epochs, indicating that this could be a promising way to improve the model.

New Model Structure with Separate Generators for Pitch and Duration

For this run, we made a separate generator each for pitch and duration (with their corresponding discriminators). The model structure remain unchanged from the baseline model. This means that, instead of the generator outputting the entire 20-note sample, two separate generator will output one 20-note pitch sample and one 20-note duration sample. The reason for this experimentation was the hypothesis that pitch and duration are different datatypes with different magnitudes for error, therefore separating the amount of error fed into them is a better choice.

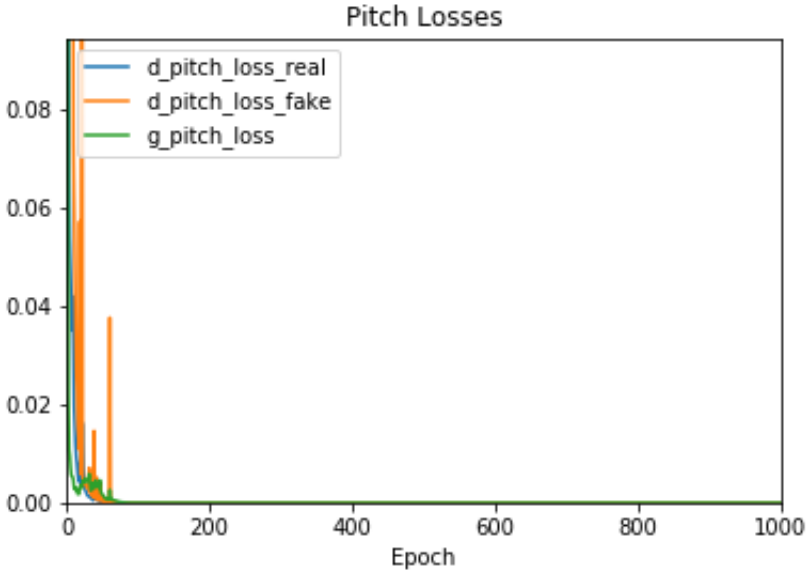

Figure A.3: Loss Graph, Pitch Generator

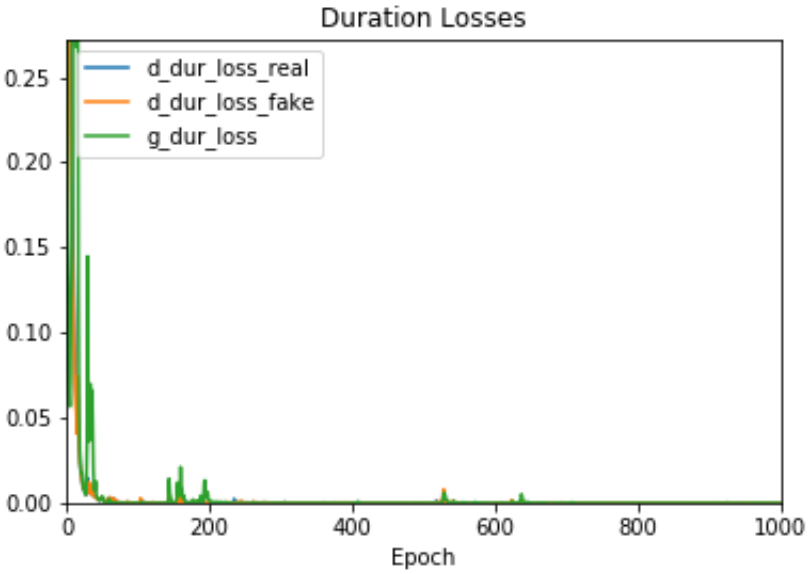

Figure A.4: Loss Graph, Duration Generator

However, under this setup the model completely failed to learn, as indicated by the loss graphs. This indicates that pitch and duration have intrinsic connections within themselves and cannot be generated as separate processes.

Training Using 30-note Samples

For this run, we tried to use 30-note samples as training data. Subsequently, we changed the generator output from 20-note samples to 30-note samples to match the training data.

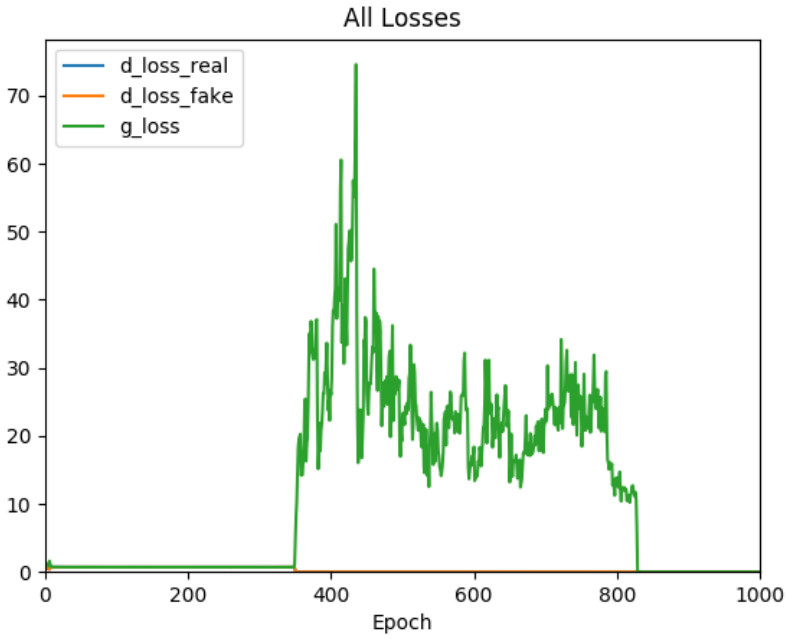

Figure A.5: Loss Graph, 30 Note Training Samples

Despite achieving around 350 epochs of stable learning, the model failed to learn properly from the data and the output does not show musicality.

References

Brannon Dorsey. (2017). Using machine learning to create new melodies. Retrieved from https://brangerbriz.com/blog/using-machine-learning-to-create-new-melodies

Flow Machine Authors. (2020). Flow machines - ai music-making. Retrieved from https://www.flow-machines.com

Foster, D. (2019). *Generative deep learning*. Boston, MA: O'Reilly Media, Inc.

Google AI. (2020). NSynth super. Retrieved from https://nsynthsuper.withgoogle.com

Huang, A., & Wu, R. (2016). Deep Learning for Music. *arXiv E-Prints*, arXiv:1606.04930.

Huang, C.-Z. A., Vaswani, A., Uszkoreit, J., Shazeer, N., Simon, I., Hawthorne, C., ... Eck, D. (2018). Music Transformer. *arXiv E-Prints*, arXiv:1809.04281.

IBM Corporation. (2020). IBM watson beat | ibm. Retrieved from https://www.ibm.com/case-studies/ibm-watson-beat

Jason Brownlee. (2017). How to use the keras functional api for deep learning. Retrieved from https://machinelearningmastery.com/keras-functional-api-deep-learning/

Keras Authors. (2019a). How to use the keras functional api for deep learning. Retrieved

from https://keras.io/getting-started/functional-api-guide/

Keras Authors. (2019b). Layer wrappers - keras documentation. Retrieved from https://keras.io/layers/wrappers/

Levitin, D. J. (2006). *This is your brain on music: The science of a human obsession*. New York, NY: Dutton Penguin.

Magenta Authors. (2019). GitHub - tensorflow/magenta: Magenta: Music and art generation with machine intelligence. Retrieved from https://github.com/tensorflow/magenta

nSFTMC. (2020). Newman guttman – the silver scale/pitch variations. Retrieved from http://sfsound.org/tape/Guttman.html

Rowel Atienza. (2017). GAN by example using keras on tensorflow backend. Retrieved from https://towardsdatascience.com/gan-by-example-using-keras-on-tensorflow-backend-1a6d515a60d0

Temperley, D. (2007). *Music and probability*. Cambridge, MA: The MIT Press.

Wikipedia contributors. (2020a). Circle of fifths — Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Circle_of_fifths

Wikipedia contributors. (2020b). Counterpoint — Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Counterpoint

Wikipedia contributors. (2020c). Generative adversarial network — Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Generative_adversarial_network