Exploring the Space of ByteBeat Music: A Genetic Algorithm

Gus Xia & Roberto Noel
New York University Shanghai
1555 Century Avenue, Shanghai, China
Independent Study: Music Information Retrieval

***Abstract*— In recent years, sound engineers and computer scientists have been employing various Machine Learning techniques to generate original compositions. Some of the most notable algorithms have been able to mimic the creative styles of the world's most famous composers such as Bach, Beethoven and Mozart. In this paper we explore the application of Machine Learning for generating ByteBeat music, a class of music that was discovered in 2011, which is similar to the chiptune music found in early video games. We take an approach based on a genetic algorithm, which starts with a randomly generated set of tracks. These tracks iteratively reproduce, mutate and die, with the ultimate goal of optimizing towards a fitness value, which aims to measure the musicality of each song. We describe several approaches to each step in this process, including random generation, mutation, crossover, feature extraction, and fitness. After this, we walk through the current algorithm as it stands, as well as future improvements.**

A. Authors & Contributors

(Gus Xia and Roberto Noel, 2019).

I. Introduction

Over the past few decades, computer scientists and sound engineers have transformed the music industry, discovering new techniques for feature extraction, style classification, and much more. One of the most impressive outcomes from this research has been the use of Machine Learning to simulate human creativity, generating original compositions that are nearly indistinguishable from those of the greats. Despite Computer Music having been around for nearly 80 years now, new developments in Artificial Intelligence have drastically accelerated progress in the field. We are now at a new frontier of discovering the range of possibilities available to us with these techniques.

On September 6, 2011, a Finnish low-tech artist, Ville-Matias Heikkila, known as "Viznut" online [1], released a video showing how short C programs can generate sounds which far outgrow their size in complexity. Since then, the subject has gone viral, with YouTube channel "Computerphile" uploading a video about it, and hundreds of others composing their own songs. These tracks sound similar to the chiptune music found in early video games [2], a style of music constrained to simple wave forms such as square waves, which can be easily reproduced by low-power sound chips. Viznut's discovery, however, differs from other kinds of lo-fi music in that every song boils down to a simple, wave-generating expression in C.

This style of music generation is now dubbed "ByteBeat." Recently, many deterministic adaptations of the style have appeared. Though these are interesting, the logic behind them is quite trivial. The least understood versions of these expressions, have a constrained set of operators, excluding certain conditionals, trig functions, and lists.

The exploration of this less deterministic approach has been mainly trial and error [1], with some experimenters gaining an intuitive sense of how these programs work. Some interesting patterns have emerged through this experimentation such as the "Serpinsky Harmony" which will be elaborated on in the related work section of this paper.

The objective of this study is to explore this space in a more structured, procedural way. One method that has worked particularly well for discovering mathematical conjectures is the "Ramanujan Machine" [3], an ongoing project which uses the Meet-In-The-Middle-RF (MITM) Algo-

rithm to generate and check the validity of conjectures in real-time. In this project, we take a similar real-time approach to discovering new ByteBeat music.

Instead of using MITM however, we use a genetic algorithm (GA) based on a fitness value that measures the musicality of each track. To do this, we break our research into four phases. First, we design a method to randomly generate a dataset of ByteBeat tracks. Second, we apply Music Information Retrieval (MIR) techniques to extract relevant features from these tracks. Third, we design a method for the mutation and crossover of their underlying expressions, and lastly, we craft a fitness value around the extracted features. Once all these phases are complete, we combine all the pieces to create a real-time GA which works to discover new, computer generated music. In the next section, we will go over some related works that will inspire some of the future developments of the project.

II. Related Works

A. What we know about ByteBeat

Despite much of the current exploration of the genre being done through trial and error, we have gained some significant insights surrounding the general properties of the music-generating expressions. Viznut compiled several of these findings in his 2011 paper *"Discovering novel computer music techniques by exploring the space of short computer programs"* [1]. He first lays out the technical framework of how the expressions are turned into raw PCM audio, and then gets into the patterns that have been found through experimentation with certain operators.

The technical framework of these programs is relatively simple. It consists of a loop, incrementing a time value "t" which is later modified by some expression. At each iteration. the resulting number is modulated by the "putchar" function, which constrains the output to 256 possible characters. This results in a wave, quantized at 8-bits, whose shape is ultimately determined by which ever expression is chosen. The track's 8-bit quantization, along with its tendency to contain saw and square waves, contributes to the genre's chiptune-like character.

Understanding this framework is important, but it still leaves us with several questions about the nature of the expressions and their outputs. To research this, Viznut collected a sample of 71 expressions created by the followers of his work. Many of the authors of these expressions merely discovered them by chance, admitting to not understanding the logic behind their creations. Others, however, took a more deterministic approach to their pieces, but ended up writing long lines of code that were more structured and traditional. One interesting pattern that did emerge from this collection however, is the presence of bitwise operators.

A common structure in Viznut's dataset, is the combination of fast and slow changing expressions. The fast moving expression acts as the carrier wave, while the slow moving expression creates a melody. A great example of this is a class of expressions which have been dubbed "Sierpinski harmonies". These consist of a bit-wise "and" combined with a bit-shift. Take "t & t >>8" for example. This expression results in a melody with mostly octave intervals.

Building on the idea of combining a carrier wave (in this case "t") with a slower expression, Viznut also found that we could emulate a more traditional style of synthesis using a tone generator with a variable wavelength. A simple solution to this is to multiply the carrier wave ("t") with another expression which yields suitable pitch values. This is the strategy behind one of the most famous expressions discovered separately by several enthusiasts, the The Forty-Two Melody: "t * (42 & t >>10)." The sub-expression in parentheses can be replaced by any other tone-generator to create original melodies.

Another common characteristic of expressions in the dataset is the presence of powers of two. Viznut hypothesizes that this is due to the structural ratios that we often find in popular music.

Its important to note that all of these observations come from a limited domain of knowledge about the genre, and in order to properly explore the space, we shouldn't exclusively constrain our experiment to previously encountered patterns.

B. What is a genetic algorithm?

A great way to conduct this kind of exploration was invented by Dr. John H. Holland from the University of Michigan in the 1960's [4]. Known as the father of genetic algorithms, Holland designed a methodology for optimizing computers to complete tasks by leveraging the power of natural selection. A close colleague of Holland's, Melanie Mitchell, lays out the framework for these algorithms in her 1999 book titled *"An Introduction to Genetic Algorithms"* [4]. The algorithm works as follows. You begin with an initial population of "chromosomes" which represent some objects in your searchable space. A fitness value is then calculated for each resulting object, and n objects are selected with increasing probability based on this value. These objects then perform crossover and mutation, creating the next epoch. This is done iteratively until the experimenter is satisfied with the result.

C. Previous attempts at evolutionary composition

Researchers have been experimenting with genetic methods of music composition for decades now. One early attempt at this comes from John A. Biles, Associate Professor at the Rochester Institute of Technology. In his 1994 study, *"GenJam: A Genetic Algorithm for Generating Jazz Solos"* [5], Biles presents a method for exploring the automated improvisation of Jazz. In order to design a GA, you must have a way of measuring fitness, this is particularly difficult in music due to the subjectivity of the matter. Biles describes this as the *"Fitness Bottleneck."* Most GAs rely on an algorithmic method of calculating fitness, the benefit of this is that it means the worst case bottleneck is one of computational performance. Sometimes, however, it is hard to come up with an algorithm which can properly measure fitness, this is exactly what we encounter in the case of music. To circumvent this, Biles takes the "I know what I like" approach, and simply rates the resulting jazz solos himself. For the purposes of his experiment, this worked, though it limited the population size and number of generations of his GA.

Towsey et al. from the Queensland University of Technology cite Bile's in their 2001 study *"Towards Melodic Extension Using Genetic Algorithms"* [6]. They divide fitness values into three classes, *"the human critic, the rule-based critic, and the learning critic."* As you might have guessed Biles' fitness function falls under *"the human critic."* There are several issues with Biles' approach, including noise from human biases [6], and more importantly, the inevitable exhaustion of the listener. Though many researchers are proponents of combining all three classes of fitness functions, Biles himself has expressed dissatisfaction with this approach [6].

Towsey et al. experiment with a fourth type of function based on statistics extracted from a dataset of famous monophonic melodies. In their study, they use MIDI tools to analyze 36 songs along 21 melodic features. They then use PCA and k-means clustering to visualize the songs in 2D space, and propose a method which assigns higher fitness values to tracks which fall in high density areas of the plane. Due to the limited size of their dataset, they don't end up implementing this idea but propose it as a possible addition to existing methods.

D. Other measures of musicality

While musicality has historically been hard to measure due to its high-degree of subjectivity, some experiments have found success in niche use cases. One such study by Pati et al. from the Georgia Institute of Technology, *"Assessment of Student Music Performances Using Deep Neural Networks"* [7], experiments with several models to measure the quality of student performances. Pati et al. finds that a Deep Neural Network (DNN) architecture which takes pitch contours (PC) and mel spectograms (MEL) as inputs, greatly outperforms the baseline model - a Support Vector Regression (SVR) standard in many experiments preceding this one.

The DNN shows impressive results from a limited dataset of roughly 3000 performances provided by the Florida Bandmasters Association. To achieve this, researchers combine two DNN models. First, a fully convolutional model which takes in PC inputs, and second, a convolutional recurrent

neural network trained on MEL inputs. Due to the variable size of inputs, the implementation of this model is quite complex and time consuming, making it a viable option as a future improvement to our genetic algorithm.

Other attempts to measure the goodness of sounds use fixed sized inputs and simpler models. A study by Picas et al., *"A real-time system for measuring sound goodness in instrumental sounds"* tests several features extracted with the *"Essentia"* C++ package [8]. It then uses the highest correlated features as proxies for the following high-level descriptors: Dynamic Stability, Pitch Stability, Timbre Stability, Timbre Richness, and Attack Clarity. Lastly, it computes an overall goodness score of different instrumental sounds by averaging these values.

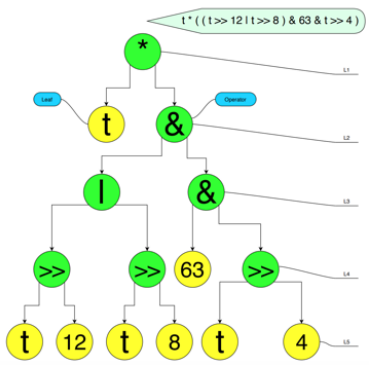
Though the scope of the Picas et al. experiment is specific to single notes on musical instruments, it shows that fixed-sized features can provide valuable information about the underlying music.

In the next section, we lay out a framework for designing a GA specific to ByteBeat, and explain how we go about measuring musicality in our experiment.

III. Implementation

A. Expression trees as chromosomes

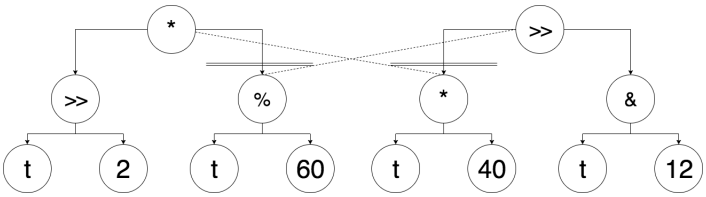
In the second chapter of *"An Introduction to Genetic Algorithms"*, Mitchell explains how early attempts at automatic programming used trees to represent Lisp programs. Similarly, since the foundation of every ByteBeat song is an expression, we can parse each expression into a tree. As seen in the figure below, the preorder traversal of this tree results in the original code.



This structure gives us the freedom to crossover and mutate our tracks using binary tree functions. It also allows us to randomly generate a dataset to base our fitness value around.

B. Crossover

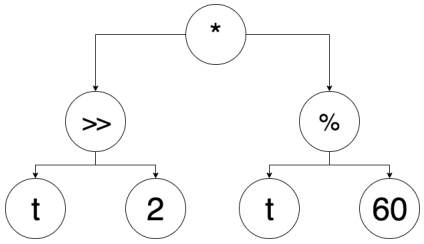
A key function in many GAs is the ability to combine chromosomes, in many cases, this allows for better exploration of the search space. We do this in our algorithm by swapping sub-trees as shown in the figure below.



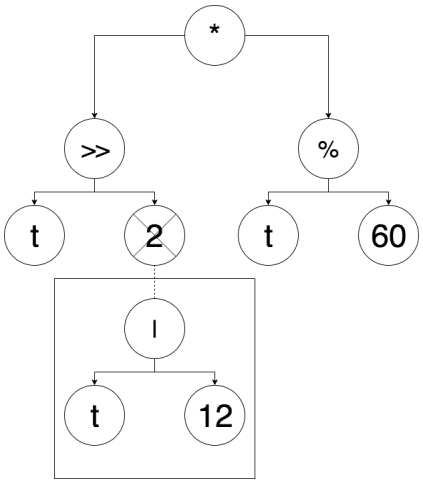
Our implementation always swaps sub-trees of equal heights, with a higher probability of crossover at lower depths.

C. Mutations

We use four basic transformations to mutate each generation of expressions: extension, trim, change of operator and change of operands. Each of them relies on a random function to select nodes, new operators, and new operands. To demonstrate, we begin with the tree representation of the following expression "$(t >> 2) * (t\%60)$."

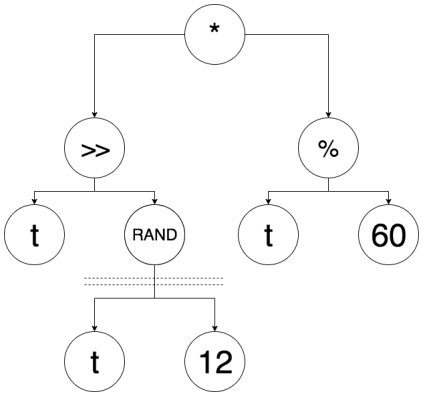


To perform an extension on this tree, we select a leaf node from the original structure, and replace it with a random three-node sub-tree.



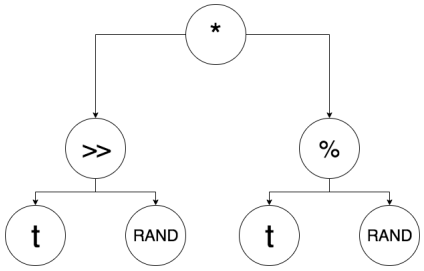
This transformation results in the following expression: "$(t >> (t \mid 12)) * (t\%60)$." Due to our research on Viznut's findings regarding structures such as the "Sierpinski Harmonies", we assign a higher probability for certain operators to root the new sub-tree. The current model favors multiplication and bit-shift functions.

Now that we have a longer tree, we can demonstrate the trim mutation, which requires at least nine nodes to execute. To do this, we randomly select an operator on the lowest level, and replace it with a number, cutting out its children. This number is chosen from a normal distribution around the average of the leaves.

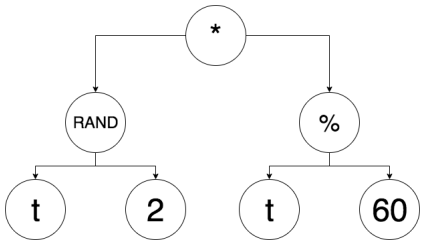


The resulting expression is now similar to the first tree we looked. This function is essential to our GA since it prevents our trees from growing infinitely with each iteration.

Sometimes we want to mutate the tree without modifying its size, this is where the other two transformations come into play. The change of operator function simply selects and edits an internal node.



The change of operand function randomizes all the non-"t" leaf nodes over a normal distribution.



D. Fitness Value: Dataset

To create a fitness value we started with a dataset of 1000 man-made and randomly generated ByteBeat expressions. We then listened to each resulting song and rated them on a 100 point scale. Once this was done, we extracted features from the dataset and experimented with different models to predict ratings.

To start, we used our extension transformation to randomly generate 1000 trees of different heights. Then we rendered the resulting tracks for feature extraction. We extracted two types of features: track-based, and tree-based.

The track-based class consists of the following spectral and rhythmic features extracted through the *Librosa* Python package:

- Tempo
- Average Spectral Centroid
- Standard Deviation of Spectral Centroid
- Average Spectral Rolloff
- Standard Deviation of Spectral Rolloff
- Average Zero-Crossing Rate
- Standard Deviation of Zero-Crossing Rate
- Average Spectral Flatness
- Standard Deviation of Spectral Flatness
- Average Bandwidth
- Average Spectral Contrast
- Standard Deviation of Spectral Contrast

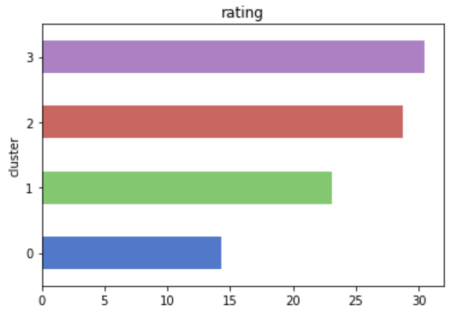
Though these features provide little information on the melodic structure of each track, they helped

us identify songs with musical qualities. For example, a track with a tempo of 0, would likely be a flat tone and would easily be disqualified from entering the next generation. Below is a correlation matrix between some of these features and the song ratings.

	rating	std_cent	std_rolloff	std_zcross	std_flt	tempo
rating	1	0.330144	0.253739	0.233578	0.113816	0.0441757
std_cent	0.330144	1	0.759427	0.722088	0.0882293	0.0319906
std_rolloff	0.253739	0.759427	1	0.276805	-0.123041	0.128588
std_zcross	0.233578	0.722088	0.276805	1	0.285604	-0.101615
std_flt	0.113816	0.0882293	-0.123041	0.285604	1	-0.0638587
tempo	0.0441757	0.0319906	0.128588	-0.101615	-0.0638587	1

By clustering the features, we gain a better idea of the factors that influence rating across the board. As you can see in the following figures, cluster 0 is the lowest rated and also has the lowest standard deviation of spectral centroid, spectral rolloff and zero-crossing rate. This is indicative of limited pitch variations throughout the song, which probably means there are many flat tones in the cluster.

	std_cent	std_rolloff	std_zcross	std_flt	tempo
0	108.905128	105.480447	0.042545	0.017163	103.158193
1	162.957629	269.037356	0.053495	0.016565	96.112109
2	419.448134	890.821973	0.121518	0.008364	110.902296
3	281.729569	488.702248	0.104280	0.014053	128.952315



The tree-based features help provide insight into the relationship between the trees and their resulting sounds. We have two types of tree features, structure-descriptive and content-descriptive. The structure descriptive features are:

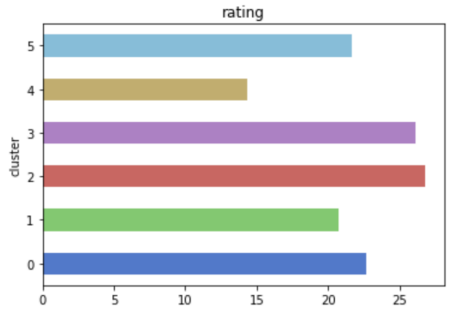
- Height: height of the tree.
- Leaves: number of leaves
- t-count: number of ”t” leaves
- Operators: number of operators

The content-descriptive features are:

- Count of each operator: 9 columns counting each operator.
- Powers of 2: number of leaf nodes that are powers of 2 (based on Viznut’s research.)

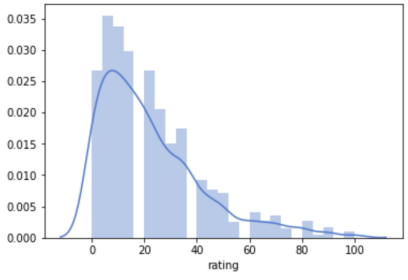
We ran a similar analysis to see how these tree features affected ratings. While the structural features provided little insight, the content-based features revealed interesting patterns.

	>>	*	&	^	%	|	-	+	/	2_powers
0	1.394366	0.926056	1.183099	0.647887	0.306338	0.461268	0.264085	0.281690	0.059859	0.669014
1	1.742331	1.153374	3.509202	0.828221	0.257669	0.644172	0.325153	0.282209	0.018405	0.822086
2	3.867470	1.271084	2.030120	0.668675	0.240964	0.759036	0.240964	0.259036	0.030120	1.162651
3	2.035088	1.175439	1.298246	0.842105	0.482456	2.701754	0.307018	0.333333	0.052632	1.236842
4	2.223214	1.401786	2.107143	1.625000	0.473214	1.000000	0.571429	0.491071	0.017857	3.312500
5	1.813665	3.316770	1.732919	0.919255	0.242236	0.708075	0.285714	0.273292	0.068323	1.298137



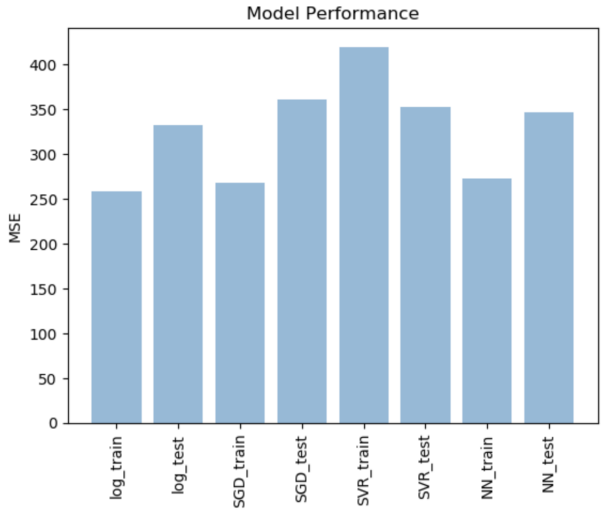
One of the clusters runs contrary to Viznut’s hypothesis about powers of two. The worst performing cluster has three powers of two on average. On the other hand, the best performing cluster has many bit-shifts and bit-wise ”&” operators, this is in line with what we have learned about the ”Sierpinski Harmonies.”

The last thing about our dataset that is important to take into account is the distribution of our ratings (displayed below), which ended up being quite negatively skewed. This turned out to be a sticking point in our development of a fitness value because it led to an unbalanced dataset. In the future work section, we explain how we plan to remediate this issue moving forward.



E. Fitness Value: Prediction Method

With the features extracted and the data labeled, we applied four machine learning models to try to predict the rating of each song: Logistic Regression, Stochastic Gradient Descent Regressor, Support Vector Regression, and Neural Network. We used cross validation to optimize the hyperparameters for each of these models, with the exception of the Neural Network, which we manually tuned. Due to the unbalanced nature of our dataset, none of these models were able to accurately predict ratings. The mean test MSE across all the models was 348, which means that predictions deviate from actual ratings by roughly 20 points on average. Below are the individual MSE's for each model.



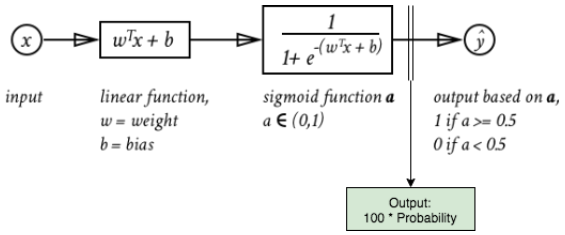
Due to the skewness of our ratings, we decided to see how our MSE compared to baseline models. For this we used five baseline measures: mean (predicts the mean every time), random (predicts a random number between 0 and 100), median (predicts the median every time), mode1 (predicts the first mode every time), and mode2 (predicts the second mode every time).

```
mean MSE: 493.574896
rand MSE: 3115.23
median MSE: 221.23
mode1 MSE: 0.13
mode2 MSE: 23.83
```

As we can see, our models perform much better than random, which is to be expected since our predictions follow a similar skew to the inputs. On the other hand, the mode baselines perform much better than our models because they completely ignore outliers, which is a common problem encountered in unbalanced data sets.

Although none of our models provided satisfactory results, we plan to address this issue in future iterations of our genetic algorithm. In order to move forward with the design of our GA, we used the our best performing model, logistic regression, as a fitness value.

Contrary to the other models, our logistic regression uses a discretized version of ratings as an input. To do this we gave a 0 value to any song with a rating less than or equal to 40 and a 1 value to any song with a rating greater than 40. We then used the probability outputs of the regression as our ratings. The resulting test MSE for this model is 330, which is the best out of the four models.



IV. Results & Conclusion

A. Tying it all together

Once we had all of the individual parts of the GA developed, we combined them into a test iteration. For this first version, we decided to hold-off on including crossover in our GA so we could see the results that would come from only mutations. Additionally, instead of selecting winning expressions probabilistically, we simply chose the songs in the top 50% of ratings to move on to the next generation.

B. Results

We tested this GA on a new set of 100 randomly generated expressions. One issue we faced was that our mutation method would sometimes result in un-renderable expressions due to arithmetic errors such as "divide by zero." This would lead to some "child" expressions dying off, causing each epoch to be smaller than the previous one. To solve this, we created a function called "stubborn mutate" which runs test renderings on each attempted mutation until it creates a valid child. Even with this

function, however, we still found expressions that were able to be rendered at a small scale, but would return errors when rendered for a longer time-window.

Despite this issue, we were able to evolve our initial 100 expressions for 20 epochs. The last epoch contained six expressions, two of which rendered songs that were much more musical than any of the original 100 (see QR codes below).

C. Conclusion & Future Work

With this first iteration done, we have gained significant insight into what can be done in the future to improve our GA. The fitness bottleneck remains the toughest sticking point of our algorithm, and there are several approaches we can take to get better results.

The first step towards addressing this is obtaining better data. We can start by crowd-sourcing ratings to obtain a more robust dataset, reducing idiosyncratic risk by diversification. We can also use data augmentation to generate more examples of ”good” songs in order to remediate the inevitable negative skew in our new dataset. To do this we would augment our data in a way that wouldn't affect the resulting rating, e.g. transforming the expression tree so the tree features change but the in-order traversal remains the same.

Once we have a good dataset, we can extract features that better describe the melodic structure of each track. One possibility for this would be to use Pati et al.'s approach, extracting the pitch contours and mel spectogram of each track. We would then use a convolutional neural network as our model to predict ratings.

After improving our method for generating fitness values, we can focus on the other parts of the GA. We must alter our ”stubborn mutate” method so that every parent chromosome creates a valid child, this will prevent our set of expressions from shrinking on each iteration. We can also add crossover and use a probabilistic selection method.

References

[1] Ville-Matias Heikkila 2011. Discovering novel computer music techniques by exploring the space of short computer programs.

[2] Kevin Holmes 2012. Meet Bytebeat: A Brand New Electronic Music Genre.

[3] Anonymous Authors 2019. The Ramanujan Machine: Automatically Generated Conjectures on Fundamental Constants.

[4] Melanie Mitchell 1996. An Introduction to Genetic Algorithms.

[5] John A. Biles 1994. GenJam: A Genetic Algorithm for Generating Jazz Solos.

[6] Towsey et al. 2001. Towards Melodic Extension Using Genetic Algorithms.

[7] Pati et al. 2018. Assessment of Student Music Performances Using Deep Neural Networks.

[8] Picas et al. 2015. A real-time system for measuring sound goodness in instrumental sounds.