CS 7643 Project Report: Making Music with LSTMs

Daeil Cha
Georgia Tech
dcha32@gatech.edu

Daniel Dias
Georgia Tech
ddias3@gatech.edu

Chitwan Kaudan
Georgia Tech
ckaudan3@gatech.edu

Abstract

Generating polyphonic music using recurrent neural networks (RNN/LSTMs) is a fairly common task; however, there is no clear consensus on which data representations and model architectures are the best. This work compares two different approaches to music generation: one where music is discretized into a Time Steps x Note Range matrix, dubbed the Note State Matrix approach; the other where each note, chord and rest element is treated as a unique "word" in a dictionary, dubbed the Vocabulary approach. This paper finds that it's easier to train a generalized model using the vocabulary approach and that having training data that represents a cohesive style is crucial to generating pleasant sounding music. Feel free to checkout our Github repo for implementation details.

1. Introduction

1.1. Motivation

Our objective was to generate music by training deep learning networks on a dataset of popular Western songs from the last 70 years. We believe music generation has numerous implications in musicology and in the entertainment industry; given the growing plethora of streaming television and film content, the demand for music to accompany said media will only grow in lockstep. Thus, automatic music generation could help content creators turn around projects faster, reduce costs, and more easily consider a multitude of styles. We also think this work could pave the way for new music genres.

1.2. Music Generation Today

Music generation via deep learning is an evolving field of study with no established standard of practice. Separate research approaches have included feedforward networks, recurrent networks (including Long Short Term Memory networks), autoencoders, and generative adversarial networks, all with varying results. Much of the existing work is trained on MIDI transcriptions of classical piano compositions of specific composers, which makes the learning process more straightforward as the data is contiguous and follows a relatively consistent style of musical patterns. MIDI is distinct from physical audio; MIDI is a technical standard used by music producers that essentially is the symbolic transcription of musical notes (somewhat akin to sheet music, but in digital form). The vast majority of deep learning systems in this domain use this symbolic representation of music. For more information on MIDI, follow this link.

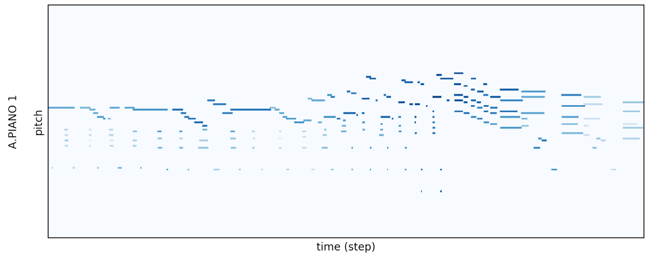

Figure 1. Piano-roll

The encoding of MIDI information has various approaches. One common representation is the Note State matrix, which we outline in the Approach section. Briefly, it is a matrix of binary vectors with dimension (time steps) x (possible playable notes). Another similar representation is the piano-roll, which is a matrix of notes over time steps; this is useful for visualize both input and output (see Figure 1). It is possible to encode multiple piano-rolls of different instruments; this is called a multi-track piano-roll. A third form of encoding is to transform every possible note, chord, and rest into tokens to create a vocabulary (akin to a natural language processing approach); the neural network then tries to predict the next token given a sequence of tokens.

The evaluation of generated music is inherently difficult. Given the subjective and personal nature of music, there seems to be no systematically robust method of evaluation in this domain. Some of the existing research surveys the opinions of composers on a model's output, but even that approach has flaws given the inherent biases of each composer. That said, personal ratings of music must ultimately be relied upon as that is arguably one principle goal of music – to evoke an emotional response in the listener.

1.3. Lakh MIDI Dataset

We used the Lakh MIDI dataset, which contains 176,581 MIDI files [5]. Besides note information, many of the MIDI files contained meta-data such as artist and song name, year published, key, mode, and others. Most of the files were multi-track MIDI files, wherein individual instruments (or 'stems') were embedded into one MIDI file. We observed that many individual stems had long spaces where no notes were played, which is common among Western popular music as instruments take turns playing. As such, we also used a subset of the dataset called the Slakh dataset, which holds 2,100 songs. We consider this more robust because Slakh only selects songs from Lakh that have at least all of the following: piano, bass, guitar, and drums, where each instrument plays at least 50 notes [4]. Slakh is beneficial also because each song's instrument is already presented as a separate MIDI file, which helped us because we planned to utilize individual instruments in some of our experimentation. We expect a model to train more easily from this standardized dataset compared to Lakh.

While both Lakh and Slakh contain a large amount of data, we should mention some limitations. The Lakh MIDI files were collected by a matching algorithm that paired a known dataset of audio music files with 176,141 MIDI files found 'in the wild' via a large-scale web scrape. Thus, the reliability of the MIDI files as ground truth depends on the methods used to transcribe the songs from their original audio counterparts. Further, the reliability of the Slakh dataset depends on the accuracy of the meta-data embedded in the original files, like instrument name and key/mode. If any of these are wrong (notes, instrument, key/mode), then our model's performance would suffer.

2. Approach

We decided to pursue a LSTM based approach because we felt they will be easier to train than other models such as GANs. During our literature review for LSTM based music generation models, we realized there was no one superior approach to encoding MIDI files as trainable data. We decided to pursue two drastically different approaches, a Note State Matrix Approach and a Vocabulary approach, and qualitatively compare the music each of them generated.

2.1. Note State Matrix (NSM) Approach

$$\textit{Note Range [78]} \left\{ \underbrace{\begin{bmatrix} [a,p] & \cdots & [a,p] \\ \vdots & & \vdots \\ [a,p] & \cdots & [a,p] \end{bmatrix}}_{\textit{Timesteps [128]}} \right. \tag{1}$$

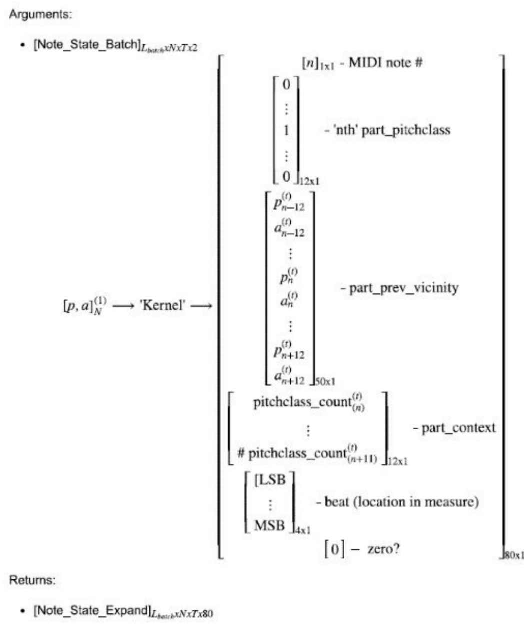
Arguments:
• [Note_State_Batch]$_{L_{batch} x N x T x 2}$
$[n]_{1x1}$ - MIDI note #
- 'nth' part_pitchclass
- part_prev_vicinity
$[p,a]_N^{(1)}$ ⟶ 'Kernel' ⟶
pitchclass_count$^{(t)}_{(n)}$
# pitchclass_count$^{(t)}_{(n+11)}$
- part_context
[LSB
MSB
- beat (location in measure)
[0] – zero?
80x1
Returns:
• [Note_State_Expand]$_{L_{batch} x N x T x 80}$

Figure 2. Input Kernel as described in [3]

One of the approaches in this paper was to take the MIDI files and discretize the notes into a matrix. Notes are guaranteed to be in a music file, and each note is associated with a duration and pitch. These values are used to create a matrix known as the Note State Matrix. The discretization we used was to break up each musical measure into 16 timesteps. At a timestep, all 78 notes are polled to determine their value in the matrix. Each value is a tribool that we represented with 2 values: articulation and sustain. The 2 values are each a single bit, and together they can represent 4 values; however, it is impossible to sustain a rest, so one of the four possible combinations is ignored and remapped. For this network, only 8 measures of each song were used. Ultimately, the note state matrix was b batch size by 78 notes by 128 timesteps by 2.

For the NSM approach we used the model architecture from this paper [3] on our dataset. Since converting to NSM is a common task, we leveraged code from an existing github repo to convert from MIDI to NSM and from NSM back to MIDI [6].

The model used is based on the work of [3]. It's called a bi-axial LSTM network, but the name is vague and doesn't accurately capture its design philosophy. The idea is that the data is shaped as $b \times N \times T \times 80$ where b is the batch size, N is the number of notes, T is the number of timesteps, and 80 is the size of each vector of the input kernel for a single note at one timestep. This input kernel will be explained in the next few paragraphs. The network first splits this matrix along the note axis to create N different 3-dimensional arrays; every single one is pushed through a different LSTM network that each have their own weight matrices. Then, all are re-concatenated into a new matrix of shape $b \times N \times T \times h$ where h is the length of the hidden vector outputed by the time-axis part of the overall network. h is a hyperparameter. The next step is the note-axis part where the 4-dimensional array is split along the note dimension into T different 3-dimensional arrays; like the first half, each sub-array is fed into a different LSTM network that also each have their own weight matrices. The sequence axis for this LSTM is the pitch of the note where the first input is the lowest pitch note ending with the highest. The output of this network is a length 2 vector that is supposed to be identical to the original note state matrix. These LSTMs of this section output a 3-dimensional array and are all concatenated along the note axis to end up with a final output of dimensions $b \times N \times T \times 2$. In an effort to decrease the memory footprint of the network, the length-2 array for each note at every timestep is returned as is.

This approach was based on the work in [3]. One of the key differences between this approach and merely supplying in data into an LSTM is that when sequential data is supplied into an LSTM, the data is collapsed along the time sequence axis. The idea is that the LSTM will learn information found along this axis. Music, however, has important information in both the time axis and the pitch axis. The way we tried to get around this is by creating a vector that is referred to as a Note State Expand.

Figure 2 shows a single length-80 vector for one note at a single timestep. This is a vector that encodes information about the notes around itself. In this vector, exact pitch is remapped into what [3] calls a pitch class. Musical pitches have what is known as an octave; this is when the frequency doubles, it sounds to be the same pitch to our ears. The note state expand first has the actual MIDI number for the pitch, then a 1-hot encoding of the pitch class, and then looks at every single pitch around itself to encode if other notes are played along side. It does this for both the articulation and sustain bits. Lastly, the pitch classes are aggregated and stored. In [3] there are other bits of data, but we wound up discarding them and filling them with zeros.

After the expand is built, the first LSTM uses the music's time axis to learn and collapse the data. Since the data is first split along the note dimension and then recombined, it traverses along the time dimension of the data, but ends up with a 4-dimensional array with the final dimension the same length as the hidden vector. This is done again for the second LSTM but split along the time dimension and then recombined. The output of the second LSTM is a 2 dimensional array, which is then recombined to create an array the same shape as the original note state matrix, $b \times N \times T \times 2$.

The loss measured for this approach was essentially a reconstruction loss. The expected note state matrix was the same as the input shifted one timestep into into the future; the first timestep was all zeros. The network was measured at how well it could predict the next timestep given the current and previous time steps. The network is also trained to predict silence in the first timestep since the output was all zeros in the first timestep for all expected data. A mean-squared error was used on the final output against the expected note state matrices. The output values were not modified after the note-axis LSTM; they were used as is in the loss function.

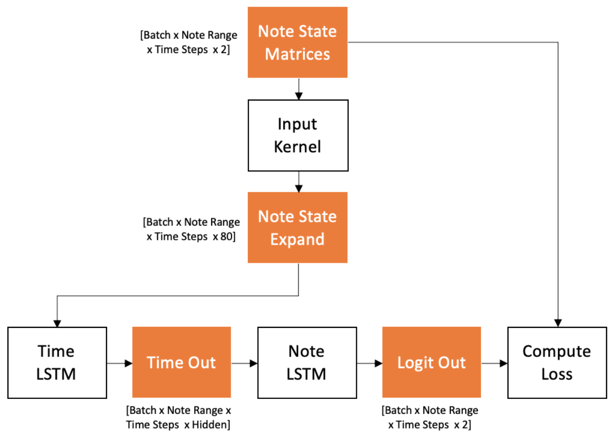
[Batch x Note Range x Time Steps x 2]
Note State Matrices
Input Kernel
[Batch x Note Range x Time Steps x 80]
Note State Expand
Time LSTM
Time Out
[Batch x Note Range x Time Steps x Hidden]
Note LSTM
Logit Out
[Batch x Note Range x Time Steps x 2]
Compute Loss

Figure 3. NSM Model Architecture

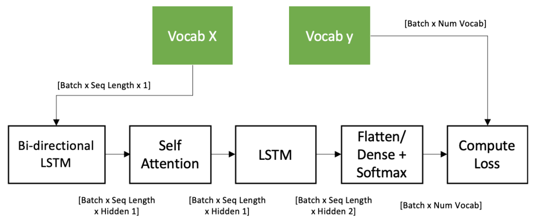

Figure 4. V Model Architecture

2.2. Vocabulary (V) Approach

The Vocabulary based approach to embedding music parallels a similar task in natural language processing. First, we mapped our target musical features into a numerical form, then set the collection of these features as a vocabulary. In our case, we mapped notes & duration, chords & duration, and rests & duration. Each song represents a sequences of features, and a model is iteratively trained on samples of these sequences to predict the next feature of each respective sequence. This is analogous to n-gram tokenization and next-word prediction in natural language processing. The model architecture we utilized was based on the method described here [1]. While we wrote our own data processing and modeling scripts, we did leveraged the code from the article to convert our model's output to the

MIDI file format [2].

Specifically, the procedure starts with converting a MIDI file into a 'Stream' object via the Music21 library in Python, a powerful toolkit for music information retrieval. That object contains the features of the song, including notes, chords, and rest objects and their respective durations; these are then mapped to numeric features and sampled into 50-unit sequences. In turn, we used these sequences to train a model. Given the aptitude of recurrent neural networks with temporal data (such as music), our architecture started with a bi-directional LSTM with 1024 nodes. To compensate for the fixed length of the memory cell in an LSTM, we also incorporated an Attention mechanism into our architecture. The intuition for this is to guide the LSTM better with contextual understanding of the inputs. From the Attention layer, the model flows into another LSTM with 512 nodes, a Dense layer with 256 nodes, and finally a Dense layer that performs softmax predictions that assigns a probability for the next combination of feature-duration to be played. The 'generation' component of the model is simply the feature (note, chord, or rest) with the highest probability.

The loss function we used in this approach is categorical cross entropy, and the optimization method we used was RMSprop (which is commonly used for RNNs). Dropout was used at several points in order to prevent overfitting. For a visual representation of this architecture, see Figure 4.

2.3. What's New & Unanticipated Challenges

What's novel about our project is our use of the pop music from the lakh MIDI dataset. Every paper we read only used classical music from composers like Bach and each composition had only one instrument or track, namely Piano. The lakh MIDI dataset had multiple tracks for each song (e.g. Piano, Voice, Drums, Bass) and hundreds of unique tracks overall. For the note matrix approach, we aggregated all tracks into one note state matrix. For the vocabulary approach, we first tried to append the track to each note/duration, chord/duration, rest/duration pair (e.g. Piano, F#, 0.25) but this made our vocabulary size unmanageable (500K+). We then decided to extract only one track for each song and we experimented with extracting Vocal and Piano. In addition, because the lakh dataset had a large diversity of songs (compared to classical music), we decided to transform all songs in a major key to C major and all songs in a minor key to A minor. We felt this would help our generated music sound more cohesive. This is because the same sequence of notes can have different contextual meanings depending on the key of the respective songs from which each sequence originates.

We anticipated that training a model that generalizes to test data would be a challenge. The paper we followed had mixed results for validation loss, where validation loss jumped around quite a lot and never really decreased substantially [3]. We did notice that the paper chose random batches of songs and random sequence start locations at each iteration which is why their loss plots have a large variance. We wanted to experiment with always starting at the beginning of each song and and sequentially choosing batches to see if we could find a more generalized fit. We also decided to conduct a more elaborate qualitative analysis of the quality of the music the model produced when it was given data it had never seen before.

What we failed to anticipate was the messy nature of the lakh dataset. Since the dataset was scrapped from several sources, the quality of the MIDI files was very inconsistent. We found several duplicate songs, incorrectly formatted MIDIs that outputted the wrong notes/chords when parsed, and unidentified tracks. The data pre-processing part of our project took much longer than we had initially anticipated.

We also did not anticipate a class imbalance problem. Our data had a lot of rest elements and we our initial LSTM models would only predict rests. Neither the paper or article directly addressed this issue. The paper vaguely mentions that sometimes they sampled from the 2nd and 3rd highest class to generate "better output" but didn't provide an explicit framework. We addressed this problem by assigning weights inversely proportional to the frequency of each class during the loss calculation.

3. Experiments and Results

We measured the success of our experiments using both quantitative and qualitative techniques. For quantitative measures, we used traditional metrics of training loss over time and/or validation loss over time to debug our models and, if needed, tune our hyper-parameters.

Since the data representations of our two approaches differed so dramatically, we felt it would not be fair to compare quantitative metrics across the two approaches. Instead, we generated new music and analyzed the quality of the music produced across the two approaches. To assess the quality of the new music, we first visualized our MIDI output as a piano roll. The piano roll helped us assess if the model was able to produce a diverse range of notes and chords and play them for different durations. We were also able to analyze if the generated music contained any repeating melodies or followed a non-trivial musical structure.

Beyond the piano roll, we synthesized the generated MIDI files into mp3. To convert the MIDI output into audio, we used the music production software Logic Pro X, and we mastered the audio for clarity with the following signal chain: Compression ¿ Saturation ¿ Limiting. We also truncated long silences for some of the samples, but all the notes are untouched from our models' outputs. We used the audio to listen for any cohesive melodic structure in the

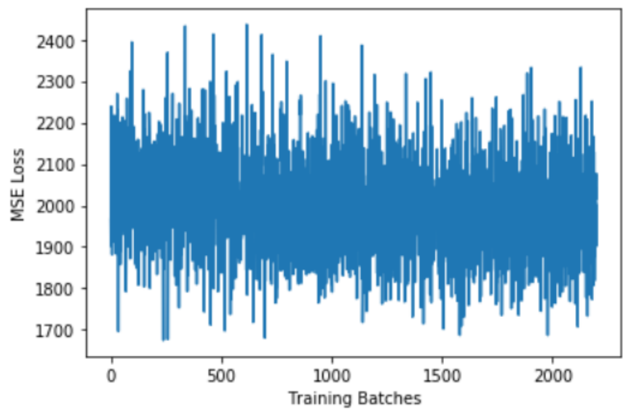

Figure 5. NSM Training Loss

piece. One of our team members is a classically trained pianist, vocalist, and composer. Our two other members have training in musical instruments as well.

3.1. Note State Matrix Results

3.1.1 Quantitative Metrics

There were a few hyperparameters used in the NSM model. One of the hyper parameters was the length of the hidden vector used in the time-axis LSTM. This hyperparameter was simply chosen at 36 because it was smaller than 80 and greater than 2. Due to the very long training time, no other hidden vector lengths were experimented with. Another hyperparameter is the length of the music used. This affected the model and number of parameters. 256 and 128 were both used; the latter was settled on because of both training and memory concerns. 8 measures is enough for music to create a motif (16 timesteps per measure).

When the note state matrix was 256 timesteps, as it was early in development, training a single epoch took about an hour. Therefore, we decided not to cross validate since training took so incredibly long.

With a smaller dataset used later of only 128 timesteps, training took about 1 hour per 10 epochs; 1 epoch included 350 songs pre-converted to an NSM. The model was trained for 50 epochs before we decided that it was not learning any more meaningful information. Training for that long took about 4.5 hours. As shown in Figure 5, the loss per training step fluctuated wildly; there is barely a downward trend. The small improvement in overall loss is likely due to overfitting. but there was not validation to properly determine.

Testing was done by using the other 50 songs separated before training and to calculate the loss when supplied into the network. Test error were not any better than the training error. Evaluation was done the same way as training; the output was used as is and measured against the expected value using the same criterion as used in training, mean-squared error. Snapshots of the model parameters were taken; table 1 shows the error after the first snapshot and the final model. Overall, neither the test error nor the training error showed very promising results.

Number of Epochs	Average Test Loss
10	1958.30
50	1913.71

Table 1. Test Loss

The final song was generated by taking the final note state matrix and thresholding each value to either 0 or 1 by using a threshold of 0.5. The idea behind doing this is that the MSE Loss should push each value to either 0 or 1.

3.1.2 Qualitative Evaluation

Qualitatively, the results were not good. None of the test data output the same exact matrix, but none of the final generated music was distinguishable from one another. They all sounded like the exact same music. The music itself was not very enjoyable to listen to, but at least it did seem to have some kind of rhythm. It sounded like someone mashing on a piano almost randomly.

Given these results, I would argue that the model wound up learning very little information from the individual music itself, but instead learned that there are a lot of rests. Each time step contains 78 notes and often, music can have as few as just 1 note played at time, leaving the input matrix to be very sparse. The final test output likely was due to the model learning the average of all the songs. An MSE loss function doesn't weigh the notes in the input any more strongly than the many rests. A possible change to this approach would be to change the note state matrix to only use pitch classes out of the gate from the MIDI conversion.

Here is a YouTube link to the music this network always outputted.

3.2. Vocabulary Results

3.2.1 Quantitative Metrics

For the vocabulary approach, we first tried to figure out which track, Piano or Vocal, was better to train on. We extracted the Piano and Vocal tracks from 50 songs in our dataset and trained them both on a simple 1 layer LSTM. This experiment revealed a class imbalance problem in our data. Since we were extracting specific tracks from songs, our data was sparse in that it had a lot of rest measures (times when other instruments were playing). Our 1 layer LSTM would only predict rest measures for Vocal and mostly rest measures with some chords for Piano. We decided to pick Piano since it was less sparse than voice.

We then trained 35 songs of Piano data on the architecture from the article depicted in Figure 4. To combat the class imbalance problem we added class weights to the loss calculation step. The weight of a class was inversely proportional to its frequency in the training dataset. This weighting procedure did help our model predict something other than rest. Initially, similar to the article, our Hidden 1 dimension was 1024 nodes, Hidden 2 was 512, and we used a RMSprop optimizer with a learning rate of 0.01 [1]. By the 10th epoch, our model was massively over-fitting on training data while validation loss was very variable and trending slightly upwards.

To reduce overfitting, we cut down on the capacity of our model, switching to 512 for Hidden 1 and 256 for Hidden 2. We also introduced dropout layers after the Bi-directional LSTM and the LSTM layers. We ran the model for 10 epochs and experimented with a few proportions for the dropout layers and found Dropout near 30% gave decent validation losses. To reduce the spiking losses, we reduced our learning rate from 0.01 to 0.0001, where we saw a somewhat steady decline in validation loss. Once we saw promising results for these hyper parameters for 10 epochs, we ran this model for 100 epochs. Table 2 contains the final hyperparameters used and Figure 6 contains our loss plots.

Hyperparameter	**Value**
Hidden 1 Dim	512
Hidden 2 Dim	256
Dropout %	30
Batch Size	50
Sequence Length	50
Learning Rate	0.0001
Loss	Categorical Cross Entropy

Table 2. Final hyperparameters of V model

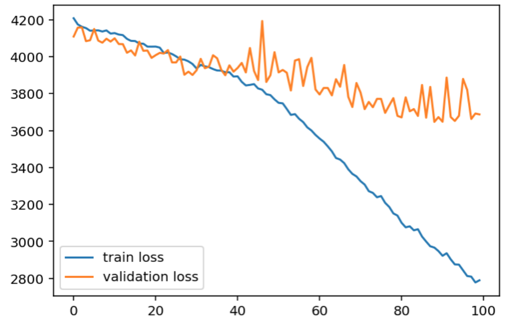

Figure 6. Training and validation loss for V Model

As is clear in Figure 6, after epoch 50 our model begins to over-fit pretty severely to our training data. Our validation loss is trending downwards but begins to become a lot more variable after epoch 50. Just by analyzing the loss curves, we believe the V model at epoch 50 does a better job at generalizing to music it hasn't been trained on than the NSM model.

3.2.2 Qualitative Evaluation

We generated music using test data and our model at various epochs and plotted their Piano Rolls. Figures 7 and 8 show the piano roll for the same test input at epoch 1 and epoch 50. Note, that the original test input is not included and the figures display only the model outputs.

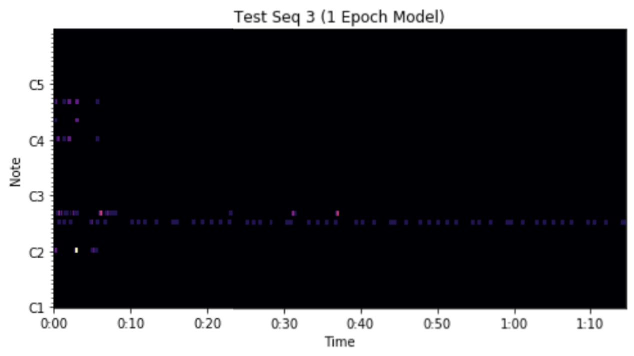

Figure 7. Test sequence 3 output using 1 epoch model

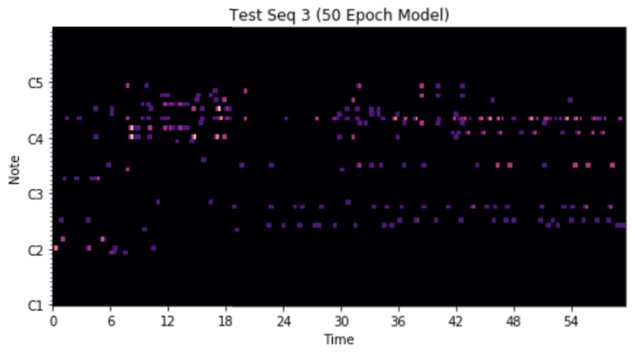

Figure 8. Test sequence 3 output using 50 epoch model

In the first few epochs, the model's predictions are very sparse and it predicts one chord over and over again. By epoch 50, we have a much "fuller" prediction with a greater variety of predicted notes and chords. For some predictions we noticed our model outputs really long rest sequences. We realized this is because there are really long rest sequences in our training data such as songs where the piano plays in the beginning and then again at the end.

If you listen to the mp3 version of the output in our YouTube playlist, you'll notice that all the outputs have a similar strange "style". The model really likes predicting a staccato F#2 sequence and you will hear it in the background of each output. It also like throwing in singular really high pitched notes. The "style" it learned was the aggregated style of our training data.

Student Name	Contributed Aspects	Details
Daeil Cha	Implementation and Analysis	Implemented V model
Daeil Cha	Implementation and Analysis	Implemented NSM model
Chitwan Kaudan	Data Pre-processing and Analysis	Parsed MIDI files into correct formats, tuned V model

Table 3. Contributions of team members.

We wondered if we training on data that had more cohesive style would give us better sounding outputs. We found the Slakh dataset which is a cleaned version of the Lakh dataset that extracted contiguous instrumentals from songs and organized them by decade. We trained the model in Table 2 on Piano 1970-1979 and, sure enough, we got more pleasant sounding music. We also trained the same model on drums and bass guitar and you can listen to them at here.

4. Conclusion

First and foremost, we learned that music generation via deep learning architectures requires careful consideration of the input data. Past research used standardized, limited datasets of piano pieces from classical composers, and the reason for this became clear during the course of our project: a model would otherwise be confounded by long silences as well as the simultaneous incorporation of contextually distinct musical elements (like melodies versus accompanying chords). We also realized that the evaluation of our output is difficult because the concept of music itself, let alone 'good' music, is not well defined. Further, we concluded that different architectures are needed to capture the other foundational components of music. Here, our model focused on progressions from note to note, which intuitively lends itself well to learning melodies and chord progressions. But what about capturing broader patterns like song structure, variations on a theme, or separate movements in pieces? These would require different approaches from ours.

On a more general note, while this work would be useful to content creators, the fact that it relies directly on the intellectual property of others could create legal issues, especially if the output resembles the original works too closely. So assuming one wants to make this commercially viable, a critical step for any future work would be to ensure that the generated songs are somewhat distinct from the training data (perhaps by requiring a minimum error). Finally, an exciting frontier for this type of work would be the creation of new music genres.

5. Work Division

Please see Table 3.

References

[1] Alex Issa Jake Nimergood Isabelle Rogers Arjun Singh Anushree Biradar, Michael Herrington. Generating original classical music with an lstm neural network and attention. 2019. 3, 6

[2] Alex Issa. Create midi files. 2019. 4

[3] Nikhil Kotecha and Paul Young. Generating music using an LSTM network. *CoRR*, abs/1804.07300, 2018. 2, 3, 4

[4] Ethan Manilow, Gordon Wichern, Prem Seetharaman, and Jonathan Le Roux. Cutting music source separation some Slakh: A dataset to study the impact of training data quality and quantity. In *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*. IEEE, 2019. 2

[5] Colin Raffel. *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching*. PhD thesis, Columbia University, 2016. 2

[6] Dan Shiebler. Midi manipulation. 2016. 2