# Music Generation

Zachary Kieda
Advisor: Jesse Stiles

May 11, 2016

## 1 Overview

In this paper, we will discuss the music generation and processing for our project. First, we will discuss the midi format and the processing that we did to get higher level information about our midi files. Then, we will discuss the $n$-gram algorithm, and why we chose to use it above other algorithms (we tested some others, eg the LSTM model, but we were not getting good results). We will then discuss what data we will be learning on based on the MIDI format, and why we made those choices (and what information in the file format we chose to ignore). Then, we will discuss how we modified the algorithm as an attempt to generate patterns and behavior that resembles music.

## 2 Midi Format and Processing

The midi format has several unique features that make it extensible, but also low-level. The midi file format defines several channels or tracks that define a stream of bits for notes. Midi can contain percussion, notes, and a variety of effects. Here, we are interested in the sequence of notes, and how we can find a high level representation while also preserving the effects that are used. However, we do not wish to learn on the effects due to the exponential explosion of training data required.

For each track we pay attention to the notes to bundle them into a sequence of high level note information. Each note in midi is defined by a note ON and a note OFF. Additional data in between determines the duration of the note, the polyphonic key pressure, and more. We transform this data to

a higher level representation that includes the duration of the note, distilling the note ON/OFF, and what happens in between into a single structure.

In order to perform this we do a scan-line sweep across the track. The 'Status' is a collection of current information or properties that we keep track of while scanning. When we reach a note ON, we place the note into our status, and on the note OFF we remove the note from our status and place it into our sequence of notes. When we reach a control change, pitch wheel change, or polyphonic key pressure midi signal, we store the message in our status. This is later retrieved when we reach a note OFF, where we copy these signals and place them into our note bundle. We store the bundle with the velocity specified by note ON in our data structure. The exception is the channel pressure message, which we store in all notes; we only have one instance of channel pressure per channel. We then store the time till the next ON message.

For each structured note, we modify the note byte based on the key to find the current octave and the tone of the note. First, we store the octave that the note is in based on the current key. Then, we store the tone in a 4-bit sequence which has been adjusted based on the key of the note (this only has eight possible values for our octave). We do this because we can learn a smaller set of data based on the notes in the octave. This also allows our algorithm to be in the same key while learning data, such that we're not jumping between two octaves.

Along with this sequence, we store the general key that the track is in and the beats per minute. We make the assumption that all tracks are in 4/4 timing and that the beats per minute or key does not change in the middle of the track. This gives us a high level stream of data that we can use to recreate the original track piece under these assumptions.

## 3 Music Learning Algorithms

In our final version, we are using the $n$-gram model to process and generate music. Typically, the $n$-gram model is used for text analysis when sequential data is important to learn. Naturally, this translates to music generation.

### 3.1 $n$-Gram Model

Suppose we have an alphabet space $\Sigma$. An $n$-gram is in the space $W \in \Sigma^n$ is an $n$ tuple of information. For a midi file, we can traverse through the generated music obtaining every consecutive $n$ sequence of notes. Gathering all sequences allows us to view the $n$ gram as a probabilistic event – certain sequences occur more often than others, or some transitions are more or less likely.

Given a sequence $x_{i-(n-1)}, x_{i-(n-2)}, \cdots x_{i-1}$, there are different possibilities for $x_i$ that have different likelihoods based on the $x_i$ we have seen occur after the sequence $x_{i-(n-1} \cdots x_{i-1}$. When $X_i$ is the random variable for the next letter, each $y \in \Sigma$ has a corresponding probability $P(X_i = y|x_{i-(n-1)}, x_{i-(n-2)}, \cdots x_{i-1})$

Consider the example $\Sigma = \{A, B, C\}$. Suppose we want to analyze 2-grams. Suppose our training data is $AABAAC$. All 2-grams are shown in the following table,

| 2-Gram | No. Times |
|---|---|
| AA | 2 |
| AB | 1 |
| AC | 1 |
| BA | 1 |

Suppose we know that $x_0 = A$. Then what is the probability for each next letter? The total number of times that $A$ appears as a letter with a following one is 4. $P(X_1 = A|x_0 = A) = 2/4$, since $AA$ occurs two out of the four times. In a similar fashion $P(X_1 = B|x_0 = A) = 1/4$ and $P(X_1 = C|x_0 = A) = 1/4$. When we are producing text from this alphabet, we would randomly select a letter from this discrete probability distribution.

We use a heavily modified version of this algorithm to produce music. Note that this is a discrete method of learning information. Therefore, it does not work well (or at all) for learning values that follow a real-number distribution, so learning on the real-number velocity or other information will not provide a satisfying result. Therefore, we modify the algorithm a bit to account for this, and to account for the duration of notes.

### 3.2 Changes From Original Version

Previously, we used an ensemble of the $n$-gram model and the LSTM Models,

but the ensemble was weighing the $n$-gram model more than we thought due to the uncertainty in the LSTM Model. This is due to the amount of data required to train a recursive neural network – we might be able to get better performance if we trained on tens of thousands of musically similar midi files all in 4/4 timing and according to our format assumptions. However, doing so is out of the scope of our project.

# 4 Algorithm Modifications

We make some special modifications to allow us to train a smaller sample size, and to add in pauses.

We want to learn durations of notes in our model. So, for every note that is being played, we get its duration. Then, we round it down to 1/4, 1/2, 1, 2, or 4 beats in our track. This allows us to give an idea of the duration of our notes without a large exponential explosion. If we chose to represent the duration as a floating point like binary number combination with a min precision of 1/4 and max of 4, we would have to represent the number with five bits meaning $2^5 = 32$ possibilities just for duration! So, instead, we just stick to our discrete five possible values and lose some precision.

Before, we mentioned that we place all of the notes in an octave based on the current key. We just learn the tone in the octave with eight possibilities in our $n$-gram sequence. This gives us a total of $5 \cdot 8 = 40$ different values for the size of $\Sigma$. We keep a constant octave when learning, but hold a separate learning process for octave transitions. We run a separate $n$-gram sequence for the change of our octave. We learn five different values for our octave values – we only allow a track to go up or down 1-2 octaves at a time or stay the same. We only allow an octave change every eight beats in our track, and the probability of an octave change is also based on the $n$-gram probability of an octave change in our original sequence.

We engineer a secondary feature during the octave change. When an octave has changed in our program, we weight the last eight beats greater, meaning that we have a greater chance to repeat the last eight beats of information. After this has concluded we remove this weighting from the track.

When the track is normally playing, we also update our own $n$-gram sequence in real time. We re-weight our heuristic such that the four beats of

time are more likely to be similar. We follow an exponentially decreasing function, such that the last four beats are the most important, and the previous four are a fraction important as the previous four, and so on. Every four beats we update this heuristic, and every time we want to generate a note we combine the training data with the live data. Note that we have to keep the training data with the live data, otherwise the live data would also diminish and we would lose information about our generation process (and make it more and more likely over time to repeat the same thing).

Finally, we use the original notes with the effects that was a part of the original midi file. We keep track of all of the original notes that are generated so we can get some effects without learning them. Since each track is a linked list of notes, if we make a map from the notes to particular nodes we can find where they are in the track. Suppose that our algorithm generates note $x_i$. For simplicity, we look at notes $x_{i-1}$ to $x_{i-4}$ that were generated and their positional data from their original track. We look up $x_i$ up in the list of notes that fit, then, we score each plausible note based on their relative position to the tracks in $x_{i-1}$ to $x_{i-4}$ by traversing $x_{i-k}$ plus or minus four notes. The score is higher based on the closeness of the relative position of the two notes. We then probabilistically select from one of the notes based on the score.

For generating music, we have five distinct tracks which have fixed instrument values. We generate independent tracks for each of these. However, we also want to keep a cap on the complexity of an individual track over intervals. We consider in interval that has more notes more complex. We do this every four beats to tell us of the complexity of the track we're generating. When selecting the duration till the next note, we linearly correlate the waiting interval to the complexity displayed in the last four beats. Again, we select an interval of $1/4, 1/2, 1, 2$ or 4 beats to wait.

We also have engineered a feature that allows us to make an instrument mute for a period of time. We randomize the number of beats to rest as a probability distribution $g(x)$ with the mode at 16 beats, a minima of 4 and a maxima of 32. This is merely a stylistic choice and is not learned. The probability that this track will be muted is $g(x) \cdot (\texttt{generalComplexity} - \texttt{thisTrackComplexity})$. The general complexity is the complexity of all tracks. Note that if this track is the only one that is playing the probability that this track will also go silent is zero. Also, note that if a track is silent, this will allow the other tracks to become more complex, allowing for an

engineered ‘solo’ feature.

**Note:** Some of these features are still being worked on (eg the key change and using beats instead of number of notes). They’ll probably be available by around 6am on Thursday.