CS230

Deep Pianist
(Category: Generative Modeling)

Roy Park[*]
Department of Computer Science
Stanford University
rpark3@stanford.edu

Hyun Soo Jeon[*]
Department of Computer Science
Stanford University
hsjeon@stanford.edu

Abstract

Songwriting indeed requires unexplainably intuitive methods to create melody, rhythms, and chord progressions. However, there clearly exist noticeable structural patterns within a genre, works by the same artist, or even music in general. We have created a Long-Short Term Memory (LSTM)-based generative model for music generation, aiming to capture this structural pattern in the process of creating a musical piece. Our model trained on a MIDI file-based text representation of music, to generate musical sequences. We show that this model successfully generates musical sequences but with room for improvement in terms of listenability.

1 Introduction

Music has always been an integral part of human life since ancient eras. While there is a general pattern in music that is considered intuitively desirable to human ears, humans have generated such patterns creatively drawing from millions of different notes to combine and create melodic sequences. Hence, music composition and production has long been reserved as a task for humans. Although the advancements in computer science have helped digitize and accelerate the process of producing music, modern music production still mainly relies on manual human work.
In this project, we attempt to create a neural network that learns the characteristics of a musical dataset, consisting of MIDI files, to generate musical pieces with similar styles and overtones to real music. Because the concept of "enjoyability" is a subjective matter, it is hard to generalize features or characteristics that make a music piece "enjoyable". In addition, since there exist so many features and different ways to represent music, there are various aspects and approaches to scrutinizing elements of music which can be extensively explored.

2 Related Work

There have been attempts to achieve the task of music generation using various forms of networks including, but not limited to, recurrent neural networks (RNNs), generative adversarial networks (GANs), and auto-encoders/transformers. Because music is inherently a sequential form of data, RNNs have been widely used as the starting point for various approaches; one of the earliest projects is MelodyRNN [1] from Google Brain, available not only with the basic model but also with more advanced models that aim to capture long-term patterns. GANs are also a popular choice for such generative tasks. MuseGAN [2], introduced in 2018, is one such network that is able to generate bars of music even without human input to start from. Using different representations, other than

[*]Equal contribution

CS230: Deep Learning, Autumn 2021, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

sequential ones, other approaches such as WaveNet [3] by DeepMind, a fully convolutional neural network which functions as a deep generative model of any raw audio waveforms including speech and music, have also been tried as well.
With recent advances in deep learning models, there also has been further developments in this area making use of different models, such as OpenAI's MuseNet [4] that uses a transformer model to predict each component that follows the previous sequence. MuseNet also allows for combinations of different music styles and genres.

3 Dataset

There are various methods to represent music in the digital world. While mp3 files may be the most recognizable audio form of music files, other non-audio file formats exist, among which MIDI (.mid or .midi) files are the most widely used. MIDI files have three advantages for their usage as datasets: first, as the standard format for instructing synthesizers to play music, they are computer-readable and widely generalizable; second, they contain "meta-messages" that provide information on the traits applicable to the entire song (e.g., key and tempo of the piece), as well as messages that contain instructions for playing each individual note; and third, because they are so widely used, there are relatively more datasets that are available online to use in training [5].
To elaborate on the information that MIDI files contain, the messages that are directly relevant to our project are those of type "note_on" or "note_off". For each note played, there is a corresponding pair of "note_on" and "note_off" messages that contain the following information:

- "Type": note_on or note_off, indicating the start and end of the note respectively
- "Channel": information for the synthesizer to distinguish between different instruments
- "Note": pitch of the played note, ranging from 0-127 with C4 corresponding to 60
- "Time": time interval relative to the last message sent, measured in MIDI ticks
- "Velocity": how loud the note is played, ranging from 0-127; note that some "note_off" messages are often represented as "note_on" messages with velocity 0

```
note_on channel=0 note=38 velocity=68 time=0
note_on channel=0 note=38 velocity=0 time=120
note_on channel=0 note=45 velocity=72 time=0
note_on channel=0 note=45 velocity=0 time=960
note_on channel=0 note=38 velocity=66 time=0
note_on channel=0 note=38 velocity=0 time=1920
MetaMessage('end_of_track', time=0)
```

During the initial phases of the development, we used small datasets available online that consisted of sample MIDI files (of the same genre, to keep consistency) to test that our data processing and network implementation steps worked as we intended. These small datasets included the Bach piano MIDI dataset (which is a subset of the Classical Music MIDI dataset [6]) and the Lo-Fi Hip Hop MIDIs dataset [7]. After the initial phases, we retrieved a larger dataset from the MAESTRO dataset (MIDI and Audio Edited for Synchronous TRacks and Organization) [8] for further development. This dataset contains over 1,200 MIDI files accumulated from ten years of International Piano-e-Competition—the repertoire consisting of classical piano pieces. We have converted over 1,200 MIDI files into compressed text files (the details follow) for convenience and to use them as inputs for our text generation model.

4 Methods

We have implemented a model based on Long Short-Term Memory (LSTM), using string representations of MIDI files.

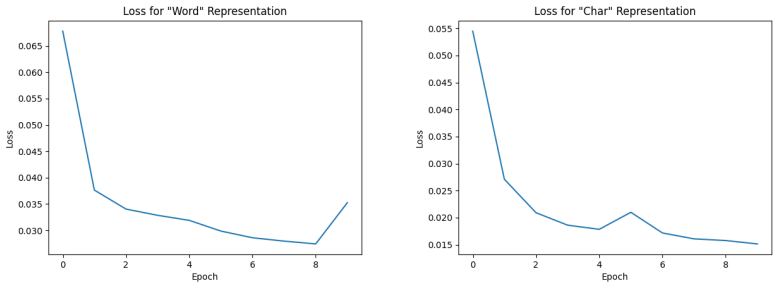
4.1 Data Preprocessing

We first implemented a Python script that reads in a MIDI file and converts the relevant data into 2D NumPy arrays. These NumPy arrays have rows representing the 128 notes that can occur in the MIDI range (0-127), each column representing the unit time step, and values representing the velocity of the note being played (0 if the note is not being played). Initially, we used each MIDI tick, the basic

unit of time in MIDI files, as one column. However, because MIDI ticks are very small–many MIDI files have 960 ticks per quarter note–this resulted in NumPy array representations with 10,000 or more columns even for tiny files. Thus, we rounded the start and end of each note to the nearest 16th note and used each 16th note as one column.
Inspired by Cary Huang's video [9], we compressed and stored the NumPy representations as text files, where strings representing the note configurations for each column (time step, in this case 16th note) were separated by a separator character, with each string containing character representations of pitches of the notes being played followed by character representations of their respective velocities. For example, if at a certain time step, C4(MIDI 60) was being played at a velocity of 70 and D4(MIDI 62) was being played at a velocity of 72, its corresponding string representation would be <>FH (chr(60), chr(62), chr(70), chr(72) respectively). This conversion reduced the size of the data files (text compared to MIDI), as well as the amount of computation in training data generation from data files in the subsequent training. Corresponding reverse functions that converted the text files to NumPy arrays, and subsequently to MIDI files that could be played on built-in media players like GarageBand or Windows Media Player, were implemented as well.

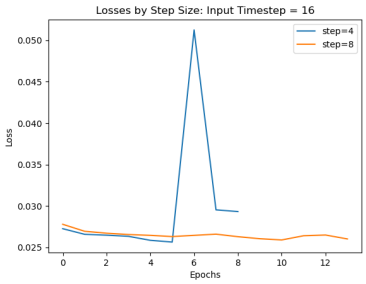
4.2 LSTM Model

With the string representation of data, the network now becomes more of a text generation model. The training data were drawn from all possible contiguous subsequences of length $input\_size + output\_size$. This is analogous to the sliding window in effect, where, for example with $input\_size = 16$ and $output\_size = 1$, an input of $str[0 : 16]$ results in an output of $str[16]$, an input of $str[1 : 17]$ results in an output of $str[17]$, and so on. The model allows for both the "word" representation and the "char" representation, where a "word" refers to the string of information on notes being played at one time step, and "char" refers to each character in the entire string representation. Regardless of which one we use, each possible "word" or "char" is assigned a unique index, which is used to convert the information to and from an array of float values that can be input into the model.
Our initial model was relatively simple, consisting of a single layer of LSTM, with 64 hidden units and the mean square error as the loss function. With the "word" representation, this resulted in consistent training errors of around 0.03-0.05 after 10 epochs for the small Bach piano dataset (batch size=32, input size=16, output size=4, resulting in 7,389 training examples). With the "char" representation, the training errors were slightly lower, typically around 0.02 (same conditions, resulting in 21,580 training examples). This phenomenon would have likely occurred because the "char" representation has a smaller vocabulary set and therefore less variation in the output.



4.3 Boolean-Vector Representation

As we moved onto the larger MAESTRO dataset, we found that the model constantly had a training loss of around 0.10-0.13. In search of a representation that may better reflect the characteristics of the dataset that we were interested in, we also tried an alternative boolean-vector representation of the dataset to use as the training data, where we focused only on whether each note was played or not rather than retaining their velocities. In this representation, each timestep was represented as a vector of length 128 (the number of possible notes in MIDI files), with 1 if the note was pressed at that time step and 0 if not. With the model itself similar to the LSTM-based text generation model, we tried out different numbers of LSTM and Dropout layers with different combinations of input windows and training batch sizes. However, the problem persisted where the training error did not decrease below a certain point, as can be seen in the plot below (except for what seems to be noise).



Furthermore, while generating music based on the trained model, our model simply converged to predicting a vector of all zeros just after a few iterations. Upon a closer inspection of the output, we realized that this may be partly due to the sheer abundance of "0"s (notes not played) compared to "1"s (notes played) in the training data: at any one timestep, there often were not more than a few notes being pressed, in which cases the rest of the 128-element vector was filled with zeroes. We tried various approaches to address this issue, such as setting a threshold value where only those with an output value above the threshold would be played, but this also largely resulted in the same note(s) being played for a long time, and did not prove successful.

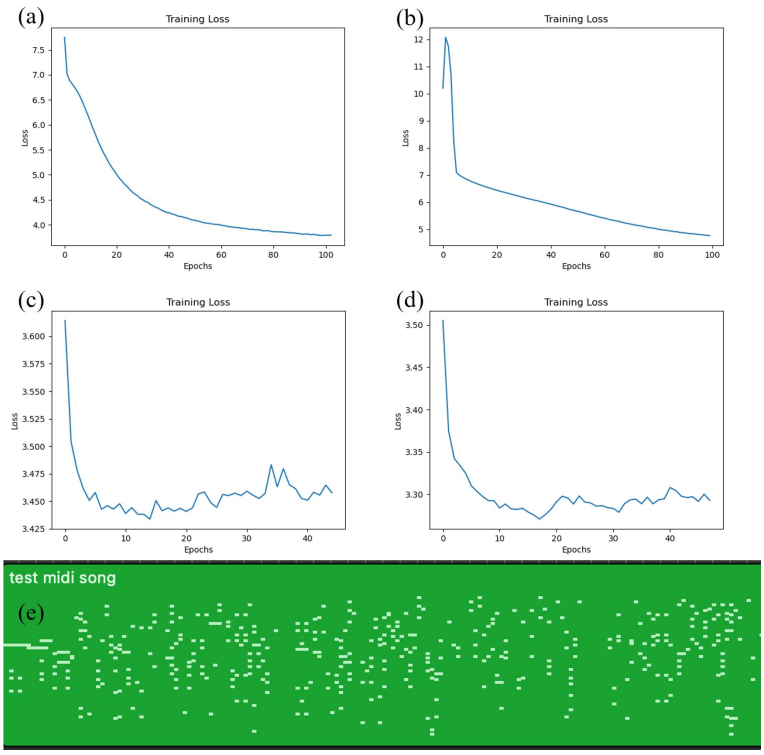
4.4 Refining the Text Generation Model into a Classification Model

Coming back to the text-generation model, we aimed to address the same problem, but with a different approach. We slightly changed the problem to a classification model, where the input values stayed the same but the output value became a one-hot encoding vector of length equal to the size of the vocabulary. Consequently, we added a softmax activation function at the end and changed our loss function accordingly, from mean-squared error to categorical cross entropy. We noticed that categorical cross entropy provided a much desired training behavior than the original model with mean-squared error, with the training error consistently decreasing over the epochs. During training, our loss value did oscillate but retained a general trend of gradual decrease.
Furthermore, because the output of the softmax function is in itself a probability distribution, we used NumPy's random.choice() function to draw the note to add to our generated music using the output vector as the probability distribution. This had the effect of providing irregularity and an imitated behavior of musical improvisation, and also prevented the result from playing only a few notes throughout the entire duration.

5 Experiments/Results/Discussion

We tested two different cases of the final model, addressed in section 3.4: one in which the model only generated a single note for each time step ("char" representation), and the other in which the model could generate one or more notes for a single time step ("word" representation).
When we implemented our model to generate just a single note each time step, we noticed that our model was able to pick up a pattern easily. For our sample track, accessible on "singlenote_generated.mid", we noticed that there seemed to be a repeating structure of several lower notes followed by a few higher notes. However, in terms of its suitability as a real-life music piece, the result was still poor.
When we implemented a model that generates one or more notes for a single time step, the generated track felt more sophisticated due to combinations of notes being played. Considering that the music generator now had over 100,000 combinations of notes to draw from based on the output probability distribution, we tried reducing the pool of combinations to draw from to the top 20 with the highest probabilities. While this did not affect the training process, the resulting track showed more occasions of longer notes, meaning that the generator drew the same combination for two or more contiguous timesteps.
It was rather difficult to find the optimal level of randomization to create sequences with both a well-organized structure and some unexpectedness of real-life music. From the choice_threshold values (the size of the output pool, from which the generated notes were drawn) that we have tried, our generation model either created a completely randomized or monotonously repetitive sequence of notes. Unfortunately, both rarely had a realistic musical structure. Although certain segments of the

generated music had reasonable musicality, the generated piece as a whole was generally not pleasant to human ears. We hope that further tuning of this threshold number of combinations to draw from could lead to a good balance between variety in music and longer notes.
The plots below show the training losses for different representations and models. (a) used the "word" representation with LSTM layer (128 hidden units) and one Dropout layer; (b) used the "word" representation with two LSTM layers (256 and 128 hidden units, respectively), each followed by a Dropout layer; (c) and (d) are the "char" equivalents of (a) and (b). The MIDI file in panel (e) was generated from the trained model in (b), with a choice_threshold value of 1024. The music corresponding to panel (e) is accessible on "threshold_generated.mid".



6 Conclusion/Future Work

Currently, LSTM is generally not considered the most suitable deep learning architecture for music generation. Most state-of-the-art deep learning models for music generations such as Open AI's MuseNet utilized transformers and auto-encoders. While we used LSTMs here as the starting point to make use of what we have learned in CS230, we hope to explore other types of models as well, as we learn about other types of models.
Also, we used MIDI files as our input data for our model, which restricts our training data to consist of only a single musical instrument. Hence, in order to generate more advanced music, we need to find ways to accommodate orchestration of multiple instruments. Most MIDI file datasets that we have found consisted only of a specific genre or had too many instruments, which was difficult for us to model into a single type of training dataset. Furthermore, if we had utilized audio files as our training data, we could have had the advantage to explore more diverse and advanced types and genres of music. However, due to the immense nature of audio datasets, we were not able to utilize raw audio data as our training data in this project.
Lastly, we could perform further data processing in order for our model to handle more specific tasks rather than handling the music generation process as a whole. For instance, we could collect chord information of each music data sample so that the model could generate only certain selections from particular set notes.

7 Contributions

All authors contributed equally throughout all steps of the project, including literature review and research, data preprocessing, and model analysis and tuning.

8 Code

The project code is located at https://github.com/hsjeon-k/deep-guitarist.git

References

[1] E. Waite, “Generating Long-Term Structure in Songs and Stories,” *Magenta*, Jul. 2016. https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn

[2] H.-W. Dong, W.-Y. Hsiao, L.-C. Yang, and Y.-H. Yang, “MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment,” arXiv:1709.06298 [eess.AS], Sep. 2017.

[3] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu, “WaveNet: A Generative Model for Raw Audio,” arXiv:1609.03499 [cs.SD], Sep. 2016.

[4] C. Payne, "MuseNet," *OpenAI*, Apr. 2019. https://openai.com/blog/musenet

[5] N. Kotecha and P. Young, "Generating Music using an LSTM Network," arXiv:1804.07300 [cs.SD], Apr. 2018.

[6] S. Rakshit, "Classical Music MIDI," Kaggle, May 2019. https://www.kaggle.com/soumikrakshit/classical-music-midi

[7] Z. Katsnelson, "Lo-Fi Hip Hop MIDIs," Kaggle, Apr. 2021. https://www.kaggle.com/zakarii/lofi-hip-hop-midi

[8] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck, "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset," arXiv:1810.12247v5 [cs.SD], Jan. 2019. Dataset located at https://magenta.tensorflow.org/datasets/maestro

[9] C. Huang (carykh), "Computer Evolves to Generate Baroque Music!" Mar. 2017. https://youtu.be/SacogDL_4JU