A Sequence-to-Sequence LSTM Network for Music Generation

Tianyang Zhan[1], **Sheng Yee Siow**[2], **Brandon Jones Gunaman**[3], **Songwen Su**[4], **Sophia Chen**[5], **Sammy Jia**[6], **Ryan Chak Hin Chan**[7], **Hin Shing Mai**[8], **Zesheng Xing**[9], and **Qixuan Feng**[10]

Department of Computer Science,
University of California, Davis

[1]tzhan@ucdavis.edu, [2]ssiow@ucdavis.edu, [3]bjgunaman@ucdavis.edu, [4]swsu@ucdavis.edu,
[5]sxmchen@ucdavis.edu, [6]sajia@ucdavis.edu, [7]rchchan@ucdavis.edu, [8]hmai@ucdavis.edu,
[9]zsxing@ucdavis.edu, [10]qfeng@ucdavis.edu

Abstract

Traditionally, music is generated manually by designing sequences of music notes in a temporal order. In recent years, researchers have been using neural networks to generate music. Recurrent Neural Networks have been used to capture the temporal relationship between music notes, but there are several limitations. In this paper, we propose a sequence-to-sequence network with Long Short-Term Memory (LSTM) units and a new data processing algorithm for processing music in musical instrument digital interface (MIDI) format. We show that our model can generate long sequences of unique, polyphonic music with notes of varied duration. (Source code available at: https://github.com/TianyangZhan/AutoMusicGeneration)

1 Introduction

Music can be defined as the sound produced when a combination of notes are exquisitely arranged together in a temporal order. The temporal relationship between these sequences of notes and certain properties of each note determines the cohesion, harmony, quality and performance of a piece of music. To generate music using Artificial Intelligence models, we need to determine what kind of music input we want to train our network with, what kind of music output we want to produce, how to capture the temporal relationship between note sequences, and how to capture the properties of each note.

There have been a number of frameworks developed to accomplish music generation with different methods such as generative grammar and hidden Markov models. Recently, with the rise in popularity of Artificial Neural Networks, researchers and programmers have tried using neural networks for the music generation tasks. Due to the fact that musical notes and features are dependent on the notes and features that appeared before it, Recurrent Neural Networks (RNNs), especially Long Short-Term Memory (LSTM) models are used in music generation because these models can "memorize". To train the AI model above, we can use monodic music (single note playing at a given time) or polyphonic music (multiple notes being played at the same instance). The models using generative grammar like hidden Markov models and genetic algorithms are generally incoherent and has too much repetition in the output. The standard RNN models also have several limitations - some can only produce monodic music; some cannot generate music with varied note duration; and some cannot produce long sequences of melodic music.

In order to address the above limitations, we propose a sequence-to-sequence LSTM network for music generation. By using LSTM we can capture the relationship between long sequence of notes. The sequence-to-sequence network allows us to have a encoder-decoder structure to capture the

relation between input and output sequences and to generate longer music. We also choose to encode music in a matrix instead of vectors, which enables our model to handle representations of polyphonic music). To have varied note duration in our output, we also design pre-processing and post-processing algorithms to encode music with preserved temporal information.

2 Related Work

To accomplish music generation using Markov models, most people model the transition between one note and the next, the probability that transition occurs and the duration each node is played. The inputs and outputs of this model are usually MIDI files. Markov-Music [15] is a Markov model based music generation model. The limitations of the Markov model is that the transitions used in this type of model and probabilities used are usually more or less too specific to the genre of music used for training, and thus cannot capture the complexity of most music, and possibly even the music used for training. The model used in here is also only capable of producing monodic music.

Genetic algorithms are complex algorithms used to solve robust optimization problems. It involves working with a population of individuals where each individual has certain qualities such that they can qualify as good individuals. During each iteration of the process good individuals are selected out to "reproduce" new ones. Genetic algorithms for music generation however, only produces short music (4 bars) with compositions being represented only by one array(represents the pitch and duration. Resulting in the production of only short and monodic music [14].

A Recurrent Neural Network (RNN) is a type of neural network where each node can receive input from previous outputs of nodes in past states and also from the outputs and inputs of the current state. Unlike the Feed Forward Neural Network (FFNN), it has some form of memory and uses this to generate output for the current state. However, simple RNNs only take into account of the memory from the previous state, which is not optimal for learning from long music sequences. LSTM is a special type of RNN structure that contains multiplicative gates that enable them to retain memory for long input sequences, making them useful for learning the sequential patterns that are present in musical data.

Melody RNN is an standard LSTM framework that focuses on generating basic melodies. It uses one-hot encoding to represent the extracted melodies as input to the model and it can only generate monodic sequence with fixed note duration [8]. Polyphony RNN [2] and Pianoroll RNN-NADE [9] both extend Melody RNN by adding support for simultaneous notes and varying note duration. Polyphony RNN employs language modelling using LSTM to generate polyphonic music. It uses special symbols to represent music as a single stream of note events and these notes are sorted in descending order by pitch within each step. Pianoroll RNN-NADE also employs language modelling to generate polyphonic music using LSTM combined with an architecture called RNN-NADE [13]. To model inputs consisting of simultaneous notes, it uses "pianoroll" as input. A pianoroll is a binary matrix with pitches as rows and the steps as columns. A 1 in the matrix means that the pitch is active and a 0 means that the pitch is off. A NADE [13] is used to sample the multiple outputs. The single-layered LSTM network proposed by Mangal and others [7] can generate sequences of polyphonic music with a note matrix. However, all of the above structures suffer from the problem of overfitting the input music and the inability to generate coherent long music sequences. The music generator models use notes to model notes, so they quickly forget the sequences that they are primed with. Because of this, later parts of the composition sound completely different to how the composition started. The aforementioned models also experience difficulty with balancing the note density throughout the generated output. Too many or too few notes are played within a part of the output very frequently. Playing too many notes causes the music to sound chaotic or messy, while playing too few notes can make the music too slow or create awkward pauses in the composition. Because of these two flaws, the music generators have difficulty with producing harmonically sound music that is longer than thirty seconds.

3 Data

3.1 Data Format

Musical Instrument Digital Interface format (MIDI) is a communications protocol for electronic musical instruments. A MIDI file contains information like tempo, note pitches, velocity, and

note duration. MIDI allows the computer to process music data directly, instead of extracting the information from audio contents. (Hewahi, Alsaigal & AlJanahi, 2019). Figure 1 shows the Python representation of a MIDI file.

```
midi.Pattern(format=0, resolution=480, tracks=\
    [midi.Track(\
        [midi.SetTempoEvent(tick=0, data=[])
         midi.NoteOnEvent(tick=0, channel=0, data=[67, 127]),
         midi.NoteOnEvent(tick=0, channel=0. data=[71, 127]),
         midi.NoteOnEvent(tick=0, channel=0, data=[74, 127]),
         midi.NoteOffEvent(tick=100, channel=0, data=[67, 127]),
         midi.NoteOffEvent(tick=0, channel=0. data=[71, 127]),
         midi.NoteOffEvent(tick=0, channel=0, data=[74, 127]),
         midi.EndOfTrackEvent(tick=1, data=[])
        ]
    )]
)
```

Figure 1: MIDI File Structure

3.2 Dataset

The dataset we use for this project is MAESTRO (MIDI and Audio Edited for Synchronous Tracks and Organization) from Tensorflow Magenta:

`https://magenta.tensorflow.org/datasets/maestro`

The dataset is composed of over 200 hours of virtuosic classical piano performances captured with fine alignment ($\sim$3 ms) between note labels and audio waveforms [3]. The music data contains a single track with a single instrument which is easier to process as no isolation of other instruments is required when constructing our piano roll matrix. In addition, the dataset of classical piano music has more note variation compared to other music styles. This characteristic makes classical piano music a good choice for our training data as they contain high densities of different note sequences to be learnt. Most importantly, the MAESTRO data set has the largest amount of easily accessible MIDI files among all other data sets that we could find online.

4 Method

4.1 The Proposed Framework

We propose the architecture shown in Figure 2 as our framework for music generation. Our objective is to predict which notes are played and not played at each time slice t. To learn and generate polyphonic music, we formulate the problem as a multi-class classification problem where the classes are the unique notes. We encoded the data from MIDI files into sequences of data and use them to train the network. To generate new music, we sample a music sequence randomly as the seed/primer and use the model's prediction to generate new music.

4.2 Data Preprocessing

Although all songs inside our dataset are classical piano music, there are different periods among classical piano music. These periods include the Baroque period (1600-1750), the Classical period (1750-1820), the Romantic era (1820-1900) and the Contemporary (1900-). Music from different periods adhere to different composition styles that are vastly different from other periods. To improve the music that our model generates, we filtered our data by year and by composer and selected 265 songs that have varied chord progressions while sharing some similarity in composition styles.

To obtain a MIDI file in the format shown in Figure 1 above, we have used a Python package named MIDO to parse the MIDI files of the MAESTRO dataset. As we are used to passing matrices into

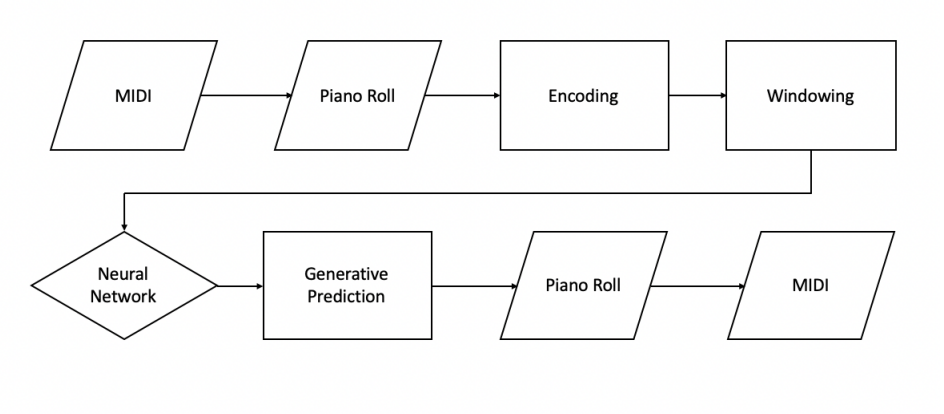

Figure 2: Framework Structure

Artificial Neural Networks, we convert MIDI representations of tracks into a 2-dimensional piano roll matrix. A piano roll matrix is a 2-dimensional matrix that has notes as the rows and time ticks as the columns. The value 1 in each entry in the matrix indicates that the note is being played during that time tick whereas the value of 0 corresponds to the note not being played during that time tick. The reason we choose to encode the MIDI input in this architecture is because it is simple to visualize and represent a dimension in the matrix as time slices. Output generation also becomes much easier under this architecture as we simply have to set zeroes and ones to correspond if a note is played.

Here are some of the technical music terms that are required to understand the conversion from a MIDI file to a piano roll matrix:

- Beat: Fundamental unit of time in music (quarter notes)
- Note: Frequency of the note played (note 60 in MIDI = C5 on piano, which corresponds to 261.625 Hz)
- Tempo : Beats per minute (number of quarter notes played per minute)
- Resolution: Density of time ticks in MIDI (measured in Ticks per Beat)

For this conversion to work, we have to determine the width (number of columns) of our piano roll matrix using the following method.

1. Find the total (cumulative) number of ticks of the track.
   - Ticks of the current track event are relative to the ticks of the previous track event.
   - m represents the number of notes played the whole track.
   - Total number of ticks = $\sum_{i=1}^{m}$ [ticks(NoteOnEvent$_i$) + ticks(NoteOffEvent$_i$) + ticks(EndOfTrackEvent)]
2. Find the span per time slice, or a chosen constant to discretely slice time in music to represent each column in piano roll matrix.
   - We chose 0.02s as the span of one time slice here because there may be many variations in music within one second.
3. Find the ticks per second: tempo (in Beats/second) $\times$ resolution (in Ticks/Beat)
4. Find the ticks per time slice: $\frac{\text{ticks}}{\text{second}} \times \frac{\text{total time in seconds}}{\text{span per time slice}}$
5. Find the number of time slices: $\frac{\text{total ticks}}{\text{ticks per time slice}}$
6. Find the piano roll width: ceiling(Number of time slices)

With these parameters all defined, we can populate the correct ones and zeroes into each entry of our piano roll matrix by parsing the notes and ticks involved in the NoteOnEvents and in the NoteOffEvents. The MIDI file is then converted into a matrix with each column corresponding to

each time slice and each row corresponding to a note from note 33 (A2) to note 81 (A6) using MIDO. This piano roll matrix represents the MIDI file of one song.

We encode our matrix by taking its transpose such that the columns now correspond to the notes (33-81) whereas the rows represent the time slices because the features (categories) has to be the columns of the matrix when they are passed into a LSTM model, and it makes more sense for the notes to be the features we are predicting. We then repeat this process for all of the songs in our training data and then we append them into a list.

4.3 A Sequence-to-Sequence Network

Instead of the standard RNN models like the one-layer network designed by Mangal and others [7], we propose a Sequence-to-Sequence (Seq2Seq) model. The Seq2Seq model is an encoder-decoder structure that can map sequences of different lengths to each other. The encoder processes its input sequence and compresses this information into a context vector of fixed length that serves as a compact representation of the input sequence. The context vector then serves as the input of the decoder. The decoder then learns from the encoded data to generate its output. This type of network structure has been commonly used in tasks like machine translation where the machine needs to take into account of information present from words before the current word (input). As in language translation, in music, the notes played during a given time period depend on several preceding notes, and the Seq2Seq models are able to generate output sequences after seeing the entire input. In our model, we use three different types of layers.

LSTM layer: the Recurrent Neural Net layer that can capture long-range dependencies from the input sequence.

Dropout layer: a regularization technique that consists of setting a fraction of input units to 0 at each update during the training to prevent overfitting.

Dense layer: a fully connected neural network layer where each input node is connected to each output node.

We apply grid search to determine the best configuration of the model and decide on the model with 5 LSTM layers with 512 nodes per layer and 2 fully connected Dense layers with 49 nodes (the number of unique notes). After the first LSTM layer we perform batch normalization to make training faster and add dropout layers with 0.2 dropout rate to avoid overfitting. Between the encoder and decode we apply RepeatVector to transform the flattened embedded vector of size (batch, latent_dim) to a sequence vector of size (batch, timesteps, latent_dim).

Our model structure is shown in Figure 3. For the final two Dense layers, we use Softmax as the activation functions so each of the output vector is a probabilistic distribution. In Figure 3, let **n** be the batch input size; let **w** be the time step length; let **k** be the length of output predicting window.

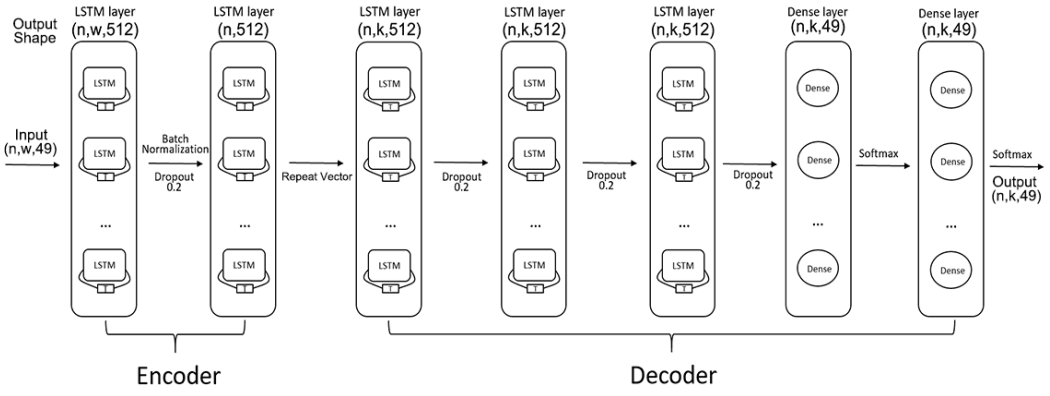

Figure 3: Network Structure

4.4 Training the Network

Training Data. In order to learn from our training data and generate new music, we build a Seq2Seq LSTM network. To train our model, we need the input and output to be a pair of note sequences. To do this, we transform the encoded data into training samples using the sliding window method on the time axis of the data. For this section, we define length as the count of time slices in the encoded data. The algorithm to generate our training data is described below.

1. Let the sliding window length (training input sequence length) = **w**.
2. Let the output sequence length = **k**.
3. Let the training data (a matrix) = **D** where the rows are the time slices.
4. Extract the first set of training data input (X) of length **w**, (**D[0:w]**).
5. Extract the first set of true output data (Y) of length **k**, (**D[w+1:w+k]**).
6. Slide the window by length **w**.
7. For the i^{th} window:
   - Extract the next set of training data input (X) of length **w**, (**D[(i-1)·w+1:i·w]**).
   - Extract the next set of true output labels (Y) of length **k**, (**D[(i·w+1):i·w+k]**).
8. After sliding through our training data, we feed our X's and Y's to train our network.

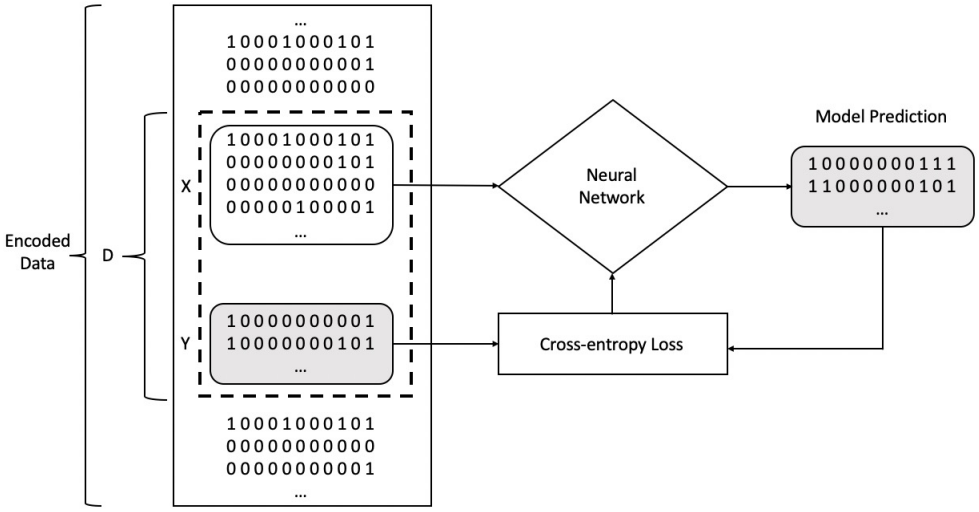

Figure 4: Data Flowchart

Figure 4 shows how the training data is fed into the network.

Loss Function. We choose the categorical cross-entropy as the loss function because it is the most suitable loss metric for multi-class classification problems. Let y_i be the label value for note i and $\hat{y}_i$ be the predicted value for note i, then we can write the loss function as:

$$Loss(y, \hat{y}) = -\sum_{i} y_i \cdot log(\hat{y}_i) \tag{1}$$

4.5 Generating Music

Generating New Sequences. After we trained the model, we save the model structure and the weights that achieve the lowest loss. To generate music, the network is fed with a seed music sequence of length **w**. This can either be a matrix of noise, or a random sample from the unseen dataset. Then we use the same network structure with pre-trained weights to predict the next sequence

of notes of length **k**. Each time after the model generates a prediction, we append the generated sequence to both the result matrix and the input matrix. Then we shift the sliding window on the input matrix by **k**. The algorithm for music generation is described as:

1. Let sliding window length (predictive input sequence length) = **w**.
2. Let single predicted output sequence length = **k**.
3. Let output iteration = **l**.
4. Use a primer/seed of length **w**, unseen by the model, as the initial predictive input sequence to generate an output sequence of length **k**.
5. Concatenate the initial output sequence of length **k** to the primer of length **w** to form a sequence of **w + k** (referred to as the *cumulative sequence*).
6. Take the latest sequence of length **w** from the *cumulative sequence* as the next predictive input sequence to generate another output sequence of **k**.
7. Append the output sequence of length **k** to the *cumulative sequence*.
8. Repeat steps 5-6 above until a desired length (**l** $\cdot$ **k**) is reached.
9. We collect the cumulative output sequence of length (**l** $\cdot$ **k**) to be our generated output.

Converting to MIDI file. The output sequence is a probability distribution of notes at each time steps. For a note to be played at time t, we define a threshold θ such that if the likelihood for playing a note needs to be higher than θ. In order to handle varied note duration, we design the following algorithm calculate the note on/off event when converting encoded matrix back to a MIDI file. Let E be note on/off event matrix where E_{it} = 0 means note i is not played time t and E_{it} = 1 means note i is played at time t. Let P_{it} be the prediction output for note i at time t, and let θ be the threshold for playing a note. Then E_{it} can be calculated as,

$$E_{it} = \begin{cases} 1, \text{ if } P_{it} \geq \theta \text{ and } E_{i(t-1)} = 0 \\ 0, \text{ otherwise} \end{cases}$$

5 Results

5.1 Model Parameters

Our model has a few important parameters listed below:

- Sliding window length (X sequence length), **w = 100**
- Output sequence length (Y sequence length), **k = 10**
- Threshold for playing a node, θ **= 0.1**
- To make training faster, we use mini-batch gradient descent with **batch size = 64**

5.2 Result Evaluation

We reserve a set of 250 MIDI files as our training data and a separate set of 50 files as the validation set. After training the model for about 750 epochs, we plot the training and testing loss as shown in Figure 5. We observe that the training loss and validation loss both decrease consistently. Based on the plot, we can conclude that our model is learning from the input and is not overfitting to the training data. Moreover, we notice the validation loss remain significantly higher than the training loss. We think this is acceptable since the nature of music composition requires many unique elements in each music piece. Therefore, the model does not need to learn a way to represent all music, but to learn the general rules for music composition. To let the model learn more sequence variations, we can use a larger training dataset. However, the difference between each song in the training data can also increase the error and make training more difficult.

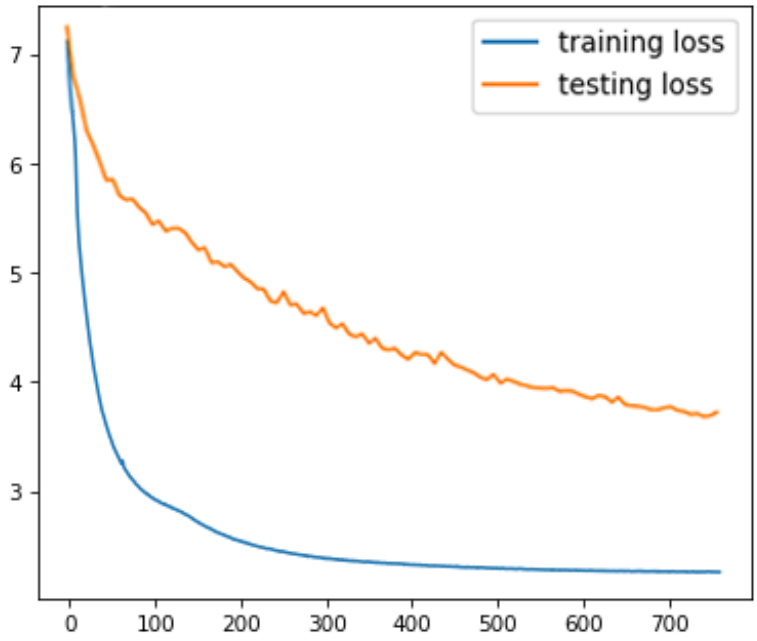

Figure 5: Training and testing Loss

Variation of Duration. As discussed above in the **Related Work** section, most models do not sound natural because of having a constant duration in between notes. This may lead to chaotic sounds (constant high density of notes) or monotonic sounds (constant low density of notes).

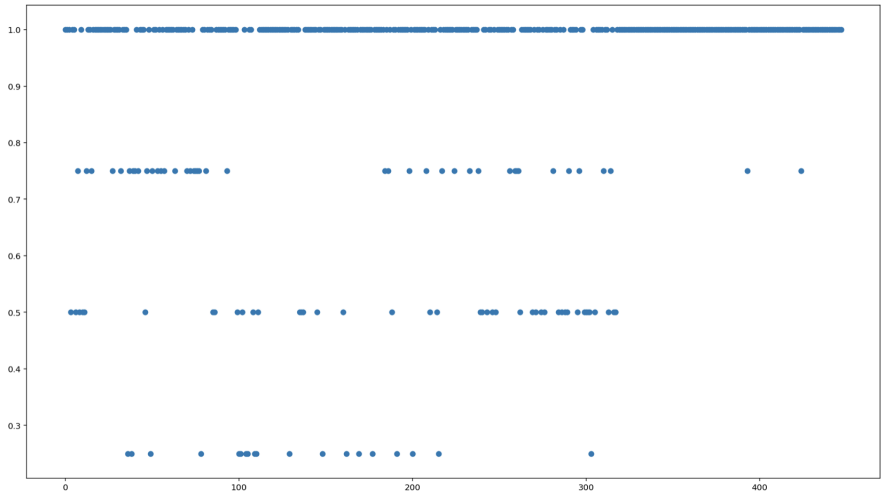

Figure 6: Scatter Plot for Normalized Note Duration in one of the outputs of our model

Our model is able to achieve a decent variation of notes as shown in Figure 6, (**duration against** i^{th} **note**). The plot above shows the duration of the first 500 notes normalized between 0 and 1 in our MIDI output. We are able to generate notes that fall under four discrete categories (full-note, three-quarter-note, half-note and quarter-note).

Generating Longer Melodic Sequences. We define melodic sequences as sequences that have variation in combinations of notes. Therefore, the duration of melodic sequence corresponds to the continuous length of time during which these melodic sequences are maintained in the output of a

model. It turns out that sequences are great encapsulation of this characteristic as reflected in our results table below.

Model	Architecture Used	Duration of Melodic Sequence
Melody RNN	LSTM	12 ± 3
Polyphony RNN	LSTM Language Modelling	25 ± 5
Pianoroll RNN-NADE	LSTM Language Modelling Piano Roll	35 ± 5
Our Model	LSTM Piano Roll Seq2Seq	80 ± 23

Table 1: *Duration of output sequence* is defined as the length of the sequence (in seconds) that has note variation. Duration (mean $\pm$ standard deviation) is calculated from 10 output samples from each model. The *Architecture Used* column describes what architecture the models are based on. We identify these sequences by listening to the output of a model and by looking at the output MIDI file.

From the table above, we have a comparison of how each architecture increases the duration of melodic sequence. The Seq2Seq architecture is, by far, the best performer as there is an increase of 45 seconds in the mean of the duration of melodic sequence generated upon the inclusion of the Seq2Seq architecture. A Seq2Seq architecture generates a sequence of outputs in a generative iteration and therefore has more coherence compared to the output of architectures that generates one note at a time. This property help reduce the cumulative error in each prediction, thus allowing the model to generate longer music sequences. We also notice that our model has a large standard deviation. We suspect the reason to be the style of the prior music is drastically different from that of our training data. We plan to explore this problem in future work.

Generating Polyphonic Music. Figure 7 shows part of our output matrix in a piano roll form. Think of this figure as a picture formed by many small pixels. Each black pixel represents a note that is on (equivalent to a "1" in the matrix), and each white pixel represents a note that is off (equivalent to a "0" in the matrix). If there are more than one pixels that are black in the same vertical section of the figure, there are more than one notes played at this time frame. We call this type of music polyphonic music. As shown in the slice between the two red lines drawn in the Figure 7, we can see that the music we have generated is polyphonic. This is essential because most music composed by humans is polyphonic.

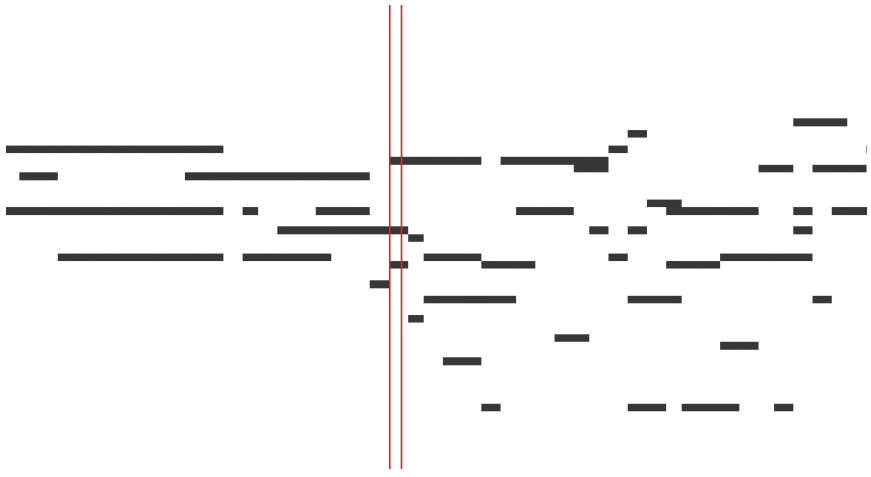

Figure 7: Output Piano Roll

Human Evaluation. To evaluate our output result, we also apply music theory to conduct human evaluation. Pieces of the Baroque era primarily used *J.S. Bach's The Art of Fugue* to compose polyphonic music. *The Art of Fugue* is a sequence of contrapuntal movements based on one theme or variations of imitation of the base theme (Golomb, 2006). Our Seq2Seq model generates outputs that are of continuous sequences, in which one sequence is similar to the sequences of movements in Baroque (1600-1750). We are able to produce this type of output because of our the sliding-window characteristic of as described in **Section 4.5** above. As the current output sequence depends on a fixed length predictive input that comes before it, the rhythm of the output can get cyclical, albeit with different note combinations.

6 Limitations and Future Works

6.1 Note Duration

Our output has variations in the duration of the notes, but it is restricted to regular rhythmic patterns primarily due to the rounding by using the ceiling function when calculating the time slice to construct the piano roll matrix. In our model, time is sliced in a discrete manner such that many of the irregular variations, such as triplets, are rounded into the previous time slice.

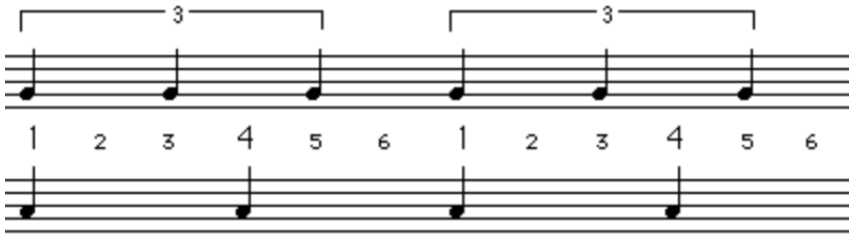

Figure 8: Comparison of triplets and quarter notes

Figure 8 shows the timing of triplets with respect to its duplet counterparts. If these triplets fall in between two adjacent time slices, one of the three notes will end up in the later time slice (two of the three other notes will end up in the earlier time slice) due to the ceiling function that we have applied when calculating the number of ticks for the piano roll. This rounding makes time slices larger than they actually are in the continuous time space, and thus breaking any triplet characteristic. Moreover, this rounding may contribute to noises in the model during training. A model that implements a continuous form of time slicing would be required to encapsulate this type of characteristic.

6.2 Dynamics

Change of loudness is an essential component to express mood and contrast in music since music was discovered.

In classical piano music, the gradual changes in loudness is expressed in Figure 9 with the legend below.

- p: *piano* (soft)
- <: *crescendo* (gradually louder)
- f: *forte* (loud)
- >: *diminuendo* (gradually softer)

We have chosen to omit this expression in our model because the change of loudness in parts are independent to the transition of combinations of notes (what our model is primarily generating). Therefore, our output lacks this expression and sounds very monotonous.

Other forms of expressing mood are *slurs*, *Staccatos*, and sustaining pedals. Sustaining pedals and *slurs* are the most essential in incorporating livelihood and animation in between note transitions.

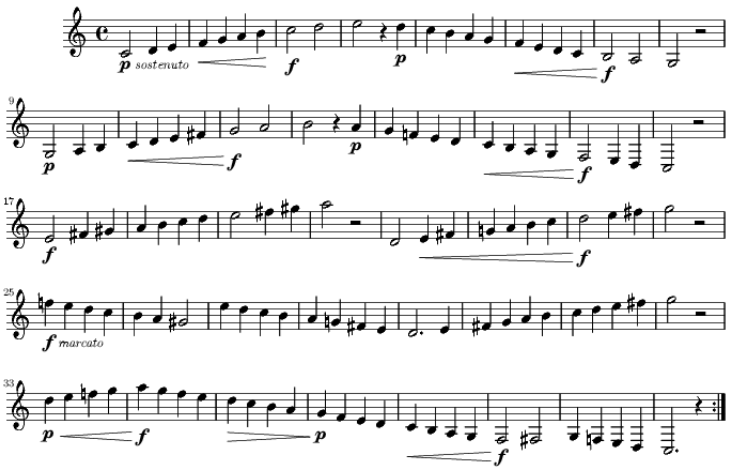

Figure 9: Various Dynamics in Classical music

Our model does not encapsulate these characteristics from the MIDI training data. *Staccatos* are independent of the transition of notes and the change in loudness whereas *slurs* and sustaining pedals are dependent on the transition of notes but are unable to be represented in a MIDI file easily.

Due to its independence, we would need at least two other separate neural networks to train the change in loudness and *slurs*. It would be computationally difficult to train three separate networks independently and combine them to generate one output. As for the sustaining pedal implementation, we have found that a Pianoteq file would be an adequate structure to work with should we want to encapsulate such information (Nettheim, 2015). However, Python currently does not offer a library to work with Pianoteq files and Pianoteq training data is not common as of the period of this project.

6.3 Change in Key

Changes in key in classical piano music is very common and happens multiple times in one track. The best examples of music pieces that changes the key fall under the Classical era because they adhere to the sonata-form (Larsen, 1967). The sonata-form consists of three major movements in one piece and these transitions from one movement to another usually involves a change from a major key to a minor key.

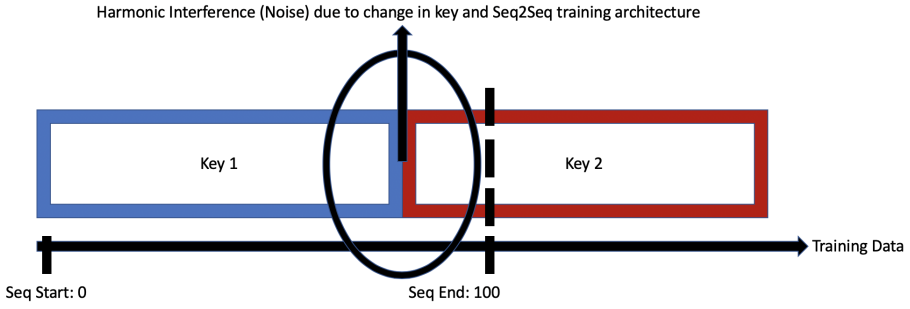

Figure 10: Harmonic Interference (Noise) due to change in key in a Seq2Seq training architecture

These changes in keys in our training data contributes to noisy data due to our Seq2Seq architecture. A movement-to-movement architecture would be a better model for this but would take too much

computation power to train due to the length of one movement. During training, we take the previous 100 time slices to predict the next 10 time slices. One could argue that the neural network could learn this. However, we think that it does not occur often enough compared to the normal same-key-to-same-key sequences for the neural network to learn this characteristic in a Seq2Seq structure.

7 Conclusion

The area of music generation using artificial neural networks has been getting more and more attention recently. The standard recurrent neural network with LSTM layers is mostly used structure but cannot produce music that resembles the songs composed by humans. In this project, we design a Seq2Seq LSTM network that address the problems like monodic output, fixed note duration, and short output length. We show that our algorithms for conversion between music and encoded matrices can produce polyphonic music with varied note duration. More importantly, we show that using the Seq2Seq model can help generate longer music sequences. Future work will explore the ways to improve the proposed network and test the model performance on different genres of music.

8 Acknowledgements

We acknowledge support for this project from all members of our team.

Topic Research: Ryan Chak Hin Chan, Sammy Jia, Hin Shing Mai, Zesheng Xing, Songwen Su, and Qixuan Feng

Related Works Research: Sammy Jia, Hin Shing Mai, Ryan Chak Hin Chan, Brandon Jones Gunaman, and Zesheng Xing

Data Preprocessing: Sheng Yee Siow, Tianyang Zhan, Sophia Chen, and Ryan Chak Hin Chan

Model Development: Tianyang Zhan, Sophia Chen, Songwen Su, Brandon Jones Gunaman, and Sheng Yee Siow

Model Tuning: Brandon Jones Gunaman, Zesheng Xing, Tianyang Zhan, Sophia Chen, Hin Shing Mai, and Qixuan Feng

Model Evaluation: Songwen Su, Sheng Yee Siow, Sammy Jia, and Qixuan Feng

Report Write-Up: All Members

Presentation Preparation: All Members

References

[1] Golomb, U. (2006) "Johann Sebastian Bach's The Art of Fugue", u: Goldberg Early Music Magazine, 48 (February 2006), 64-73

[2] Hawthorne C., Parunashvili S., Cífka O., Liang F., Roberts, A. (2017) "Polyphony RNN". GitHub Repository, 2017. https://github.com/tensorflow/magenta/tree/master/magenta/models/polyphony_rnn/

[3] Hawthorne, C., Stasyuk, A.,Roberts, A., Simon, I., Huang, C.A., Dielemant, S., Elsen, E., Engel, J. & Eck, D. (2019) Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset

[4] Hewahi, N., AlSaigal, S. & AlJanahi, S. (2019) Generation of music pieces using machine learning: long short-term memory neural networks approach, Arab Journal of Basic and Applied Sciences, 26:1, 397-413, DOI: 10.1080/25765299.2019.1649972

[5] Huang C. A., Simon I., & Dinculescu M. (2018) "Music Transformer: Generating Music with Long-Term Structure. Magenta Blog, December 13, 2018. https://magenta.tensorflow.org/music-transformer

[6] Larsen, J.P. (1967) "Some Observations on the Development and Characteristics of Vienna Classical Instrumental Music", u: Studia Musicologica Academiae Scientiarum Hungaricae, T. 9, Fasc. 1/2 (1967), pp. 115-139

[7] Mangal, S., Modak, R., & Joshi, P. (2019) "LSTM Based Music Generation System." IARJSET 6.5 (2019): 47–54.

[8] Nettheim, N. (2015) "Studies in the Computer Rendition of Piano Pedalling", u: Israel Studies in Musicology Online, Vol. 13, 2015-16

[9] Roberts, A., Cífka O., Hawthorne C. (2017) "Pianoroll RNN-NADE". GitHub Repository, 2017. https://github.com/tensorflow/magenta/blob/master/magenta/models/pianoroll_rnn_nade/

[10] Roberts, A., Simon I., Cífka O., May J., McDonald M., Hawthorne C. (2017) "Melody RNN". GitHub Repository, 2017. https://github.com/tensorflow/magenta/blob/master/magenta/models/melody_rnn/

[11] Simon, I., Oore S. (2017) "Performance RNN: Generating Music with Expressive Timing and Dynamics." Magenta Blog, (2017). https://magenta.tensorflow.org/performance-rnn

[12] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. (2017) "Attention Is All You Need". arXiv.org. December 6th, 2017. arXiv:1706.03762 [cs.CL].

[13] Benigno Uria, Marc-Alexandre Côté, Karol Gregor, Iain Murray, and Hugo Larochelle. Neural autoregressive distribution estimation. arXiv preprint arXiv:1605.02226, 2016.

[14] Matic, D., 2010. A genetic algorithm for composing music. Yugoslav J. Operat. Res., 20: 157-177. DOI: 10.2298/YJOR1001157M

[15] Lin, A. (2016) "Generating Music Using Markov Chains". Medium, 2016. https://medium.com/hackernoon/generating-music-using-markov-chains-40c3f3f46405#.1wukhwbrt