Bi-axial Recurrent Neural Network for Music Composing

E4040.2020Fall.QMSS.report

Yang Liu yl4318, Yunong Li yl4319, Chengwei Wang cw3210
Columbia University

Abstract

The goal of this project is to design a neural network architecture that can generate new music based on training pieces. The methodology that comes from Daniel D Johnson's work[2] uses a bi-axial RNN architecture to model music pieces both time-wisely and note-wisely. We used a subset of the MIDI data from Piano-midi.de that was used in the original work and trained an architecture similar to what was presented in the article with minor modifications. The results are basically the same with respect to the final training and validation loss. And our model output music pieces that are melodic, harmonic, rhythmic and pleasing to listen to.

1. Introduction

Music, especially classical music, is just a sequence of different pitches in time. The key goal of music modeling is to learn and predict music styles and generate new music content on this basis. This is not a simple task. The model needs to have the memory of the past in order to predict the future. In addition, the model must learn the theme of music and transform the theme into different variations. The next challenge is to understand the inner structure of the work and create a composition that has a progressive relationship and echoes in the structure.

Probabilistic models are most commonly used in training music. The music is modeled as a probability distribution, which maps a metric or note sequence based on the probability of the note appearing in the main body of the training music. These probability models do not need the input of information about the probability distribution characteristics of the training data in advance. The algorithm finds patterns from the structure and progression of the music. After the model is trained, new music is generated by sampling from the distribution using learned notes probability weights.

The most straightforward way of modeling music and making automatic generation is to use a neural network such as RNN, which is capable of predicting the value of a sequence at time t using values from time t-1. After fitting the model on training sequences, the probability weights are learned for each note and can be used to generate new sequences. However, simple feedforward networks have failed at this task for many reasons. Music pieces have spatial and temporal textures, which can only be spotted by looking at the global range. Simple feedforward neural networks can only look at a limited range of the piece and cannot keep track of temporally distant events that indicate global music structure. The design of these networks also results in 'vanishing gradients' problem when doing backpropagation through time. This problem limits the networks' performance when long-term dependencies exist in the sequence, as is the case with music.

Such limitations are overcome by LSTM cells. LSTM tries to keep a constant error stream that changes with time and protects the error stream from undesired interference. Each LSTM cell has a fixed self-connection unit and is surrounded by a bunch of other non-linear units, which are responsible for controlling the flow of information in and out of the cell. The multiplying input gate unit learns how to protect the flow from being interfered with by irrelevant inputs. Similarly, the multiplication output gate unit will learn how to protect other units from interference through the currently irrelevant stored content. When the storage unit's content is out of date, the forget gate is trained to learn to reset the memory unit. The learning is done through the gradient descent method.

Extensive literature supports the promising ability of LSTM in the field of music composing. Douglas and Jurgen(2002)[1] used LSTM architecture in blues improvisation. They found that LSTM does not rely on regularities in the melody to learn the chord structure, and they used LSTM to generate new pieces of bebop-jazz variations of standard 12-bar blues. Their work successfully learns temporal structure, but the output is pretty restricted in terms of the form. Also, they did not make a design that is able to distinguish between playing a note and holding it, so the network cannot rearticulate held notes. Nicolas et al. (2012) came up with an RNN based architecture containing two parts: the first part is just an RNN used to model recurrent relationships in timesteps; the second part is a restricted Baltzman machine that is used to model the conditional probability of one note being played given other played notes. The output of this work seems to have chords repeating and the progression of time is not apparent.

Above mentioned models use RNN and LSTM to model the recurrent connections in time as the music progresses. Another essential feature of music seems to be neglected. Unlike modeling language where each character or term is uniquely defined, specific in its meaning and functions somehow similarly in different sequences, the parallel harmonic or chord structure of different tonality indicates that we should look at the relative positions when modeling music of each note instead of absolute pitches. Otherwise, the model would learn from scratch if given precisely the same input transposing one semitone up, which is inefficient and clearly not the ideal model we want.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

When designing the architecture of the model used in Daniel D Johnson's work[2], the main focuses were:

- Have some understanding of time signature by giving the timesteps in reference to a time signature
- Be time-invariant and be able to compose indefinitely
- Be (mostly) note-invariant, the structure of the neural network to be almost identical for each note
- Allow multiple notes to be played simultaneously, and allow selection of coherent chords
- Allow the same note to be repeated

He proposed a bi-axial RNN architecture inspired by the kernel of convolutional neural networks. In this architecture, he achieved both time-invariant and note-invariant by stacking RNN or, more specifically, LSTM layers along the time axis and the note axis, with input being a sliding window of two-octave length around each note. On the time axis, each note is independently trained, while on the note axis, each timestep is independently trained. Together, the architecture can find patterns and achieve both time-invariant and note-invariant.

A boolean vector of two components represents the state of each note in each timestep, the first one being whether the note is played at this time step, the second one is whether the note is articulated at this time step. The key difference between being played and being articulated is that a note can be held from the last time step, making it played but not articulated at this time step. By keeping the two components throughout the whole music piece, holding a note and repeating the same note become different events in the model.

2.2 Key Results of the Original Paper

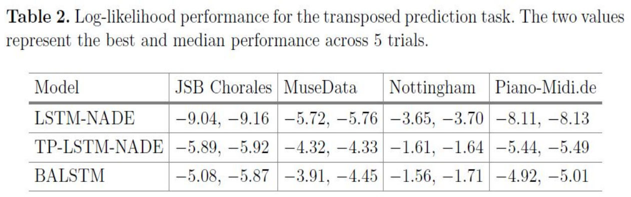
Table 2. Log-likelihood performance for the transposed prediction task. The two values represent the best and median performance across 5 trials.

Model	JSB Chorales	MuseData	Nottingham	Piano-Midi.de
LSTM-NADE	−9.04, −9.16	−5.72, −5.76	−3.65, −3.70	−8.11, −8.13
TP-LSTM-NADE	−5.89, −5.92	−4.32, −4.33	−1.61, −1.64	−5.44, −5.49
BALSTM	−5.08, −5.87	−3.91, −4.45	−1.56, −1.71	−4.92, −5.01

Fig 1 Log-likelihood of the bi-axial LSTM model in the original paper

The published paper trained two LSTM layers in the time-axis direction with 200 nodes each, and two LSTM layers in the note-axis direction with 100 nodes each, with a dropout of 0.5 added at each layer and optimizer being RMSprop, which is a slightly different architecture as described in Daniel's personal blog. The results are shown in figure 1. The bi-axial LSTM model trained on Piano-Midi.de data achieved a training log-likelihood of -4.92 and a testing log-likelihood of -5.01.

3. Methodology (of the Students' Project)

3.1. Objectives and Technical Challenges

This project's objective is basically identical to that of the original paper--building the RNN architecture that can find time-invariant and note-invariant patterns and allow for chords and re-articulation. We aim to build a model that could achieve roughly the same performance in terms of log-likelihood. And we have faced multiple problems in each stage, from data importing to model implementation.

First of all, we encountered some problems with understanding the concept of tick in the midi file and how to convert the midi format music into a matrix or any other format easy for the machine to interpret. We then figured out that notes recorded in the file with "note_on" represent notes that are played at that moment if tick = 0 or later if the tick is larger than 0. Therefore notes consecutively appearing with tick = 0 are played simultaneously.

After transforming the midi format into more structured ones, we confronted the problem of the non-rectangular shape of input data, combining pitch class, previous vicinity, previous context, and beat. We later found that we could flatten them before feeding them into our model and solve the problem.

Afterward, we were confused by the genius yet the original model's baffling design by training the time sequence first and then the note sequence later. We were not sure how to realize this in TensorFlow. We have proposed several ideas related to this implementation, including manually connecting two models (time layer one and note layer one) and then use TensorFlow's gradient tape to record the gradient calculated from the note layer and pass them to the time layer model. However, we later learned that we could manually set up a customized layer and transform data structure and shape in that layer without weight tuning by setting tensor's argument trainable to False. We then use three customized layers to transform input (midi-matrix) data into an expanded version with pitch class and previous information. Moreover, another layer exchanges the position of time series and note sequences to achieve the transition from time layers to note layers within one TensorFlow's sequential model.

3.2.Problem Formulation and Design Description

When comparing our design to the original paper, we initially expanded the range of notes to 0-127, instead of the lowerBound=24 and upperBound=102, which was set in the original paper. Though the author did not elaborate on why he set so, we assume he intended to cut out the range of notes (0-23 and 103-127) that are randomly used to limit the amount of input dataset and speed up the training process. However, we decided to remove this limitation because we intended to let the model learn this truth itself from the training set and make the entire decision based on its results with the least amount of manual intervention.

The original paper dumped all of the MIDI files from Piano-MIDI.de with composers from different times and styles for the data selection. We hand-picked four representing composers of romantic music: Beethoven, Chopin, Mozart, and Brahms. Even though the four of them also vary a lot in terms of the formats and styles, a model trained on this subset should be able to capture the typical patterns of romantic music pieces and reproduce them in an automated fashion. The subsetting of data makes the training process less time-consuming and the model more interpretable as for the styles.

We basically replicated the model architecture described in the original article. Figure 2 shows the workflow of our model. The MIDI files are processed into a NumPy array of state vectors as the baseline input. Then it is fed into the Bi-axial LSTM model and goes through eight layers in total.

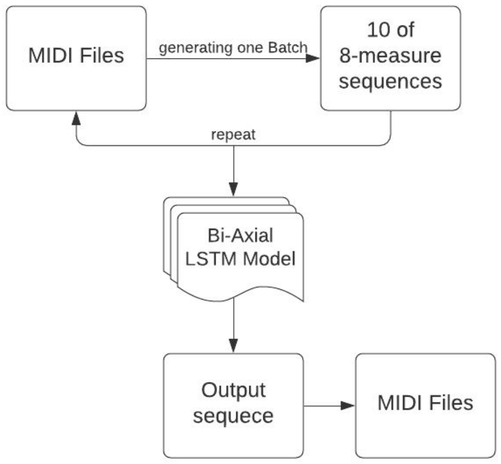

Fig 2 Flowcharts of the software design

4. Implementation

In this part, we would like to explain the process of generating batches as model inputs, training the deep learning model, and developing new musical pieces. We implemented the model on the Google Cloud Platform with Mido and TensorFlow Python packages.

4.1. Deep Learning Network

The data we used in this project were collected from the Classical Piano Midi Page website, including pieces created by Beethoven, Brahms, Chopin, and Mozart. The data were all in MIDI file format. MIDI (Musical Instrument Digital Interface) is a technical standard that describes a communications protocol, digital interface, and electrical connectors that connect a wide variety of electronic musical instruments, computers, and related audio devices for playing, editing, and recording music. MIDI carries event messages; data that specify the instructions for music, including a note's notation, pitch, velocity (which is heard typically as loudness or softness of volume); vibrato; panning to the right or left of stereo; and clock signals (which set tempo) [ref]. In our MIDI files, we have the records of the note, velocity, and time. We added an additional dimension following Johnson's original article, which is a binary variable indicating whether a note was articulated or sustained at a particular time step. For example, the first time step when playing a note is represented as [1, 1], sustaining a previous note is [1, 0], and resting is [0, 0]. The flow of generating the input batch is presented in Fig 3.

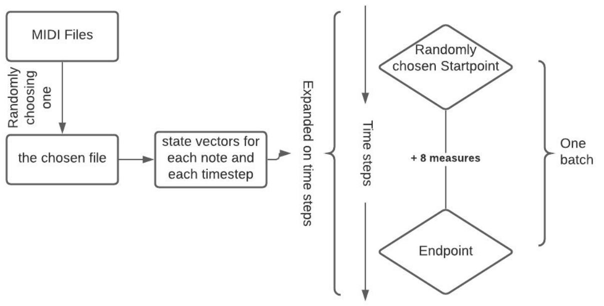

Fig 3 Flowcharts of generating batches

The deep learning network was built based on the dataset mentioned above. Our model was designed to generate new music pieces from the knowledge it learned from the original dataset. Our model's target is to predict each note's play-articulate state pair at each time step, which is derived from the probability distribution of these variables. The probability distribution is a conditional probability distribution by nature since it needs to consider the notes played in the previous time steps. To predict notes' state by time step, we used a bi-axisal LSTM model just as Johnson's implementation[ref]. The first LSTM stack is implemented on the time-axis. The second LSTM stack is recurrent along the note-axis, which scans up from low notes to high notes. Each LSTM stack consists of two independent LSTM layers, and all of the LSTM layers are applied with a dropout of 0.5, which is stated to be the optimum dropout rate from empirical analysis. The overall workflow of the deep learning network is shown in Fig 4.

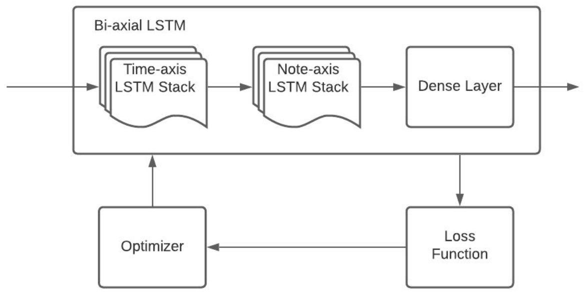

Fig 4 Flowcharts of the deep learning network

In the first time-axis LSTM stack, the targeting objects are specific notes. After we pre-processed the original input dataset, we possess records for the time steps that a given note was played or articulated. Therefore, we can use these records to compute the probabilities that this specific note would be played or articulated at each time step and conduct similar computation on every note in parallel in the first LSTM stack. That is to say, the first two LSTM layers have connections across time steps but are independent across notes.

Similarly, in the second note-axis LSTM stack, it computes the probabilities for each time step considering the connection between different notes. Notice that the note-axis LSTM stack's input shape would have two additional dimensions that keep the state of the previous (half-step lower) note (whether the note was played or articulated).

After the last LSTM layer, we implemented a Dense layer to produce two output values in the range of 0 to 1. The first one is the play probability, which is the probability that the note is chosen to be played. The second one is the articulate probability, which is the probability that this note is articulated if it was played previously.

As for the loss function, we used the binary cross-entropy function. This function penalizes the model when a note is articulated while it has never been played previously. This logic makes sense since an articulated note must be played before by definition.

4.2. Software Design

Our code's overall structure can be split into four parts:

- reading MIDI files and transforming them into the input matrix
- training and validating the model with the training set and validation set
- generating new note matrices with the trained model
- converting the note matrix into new MIDI files

The flow chart of our work is shown in Fig 5.

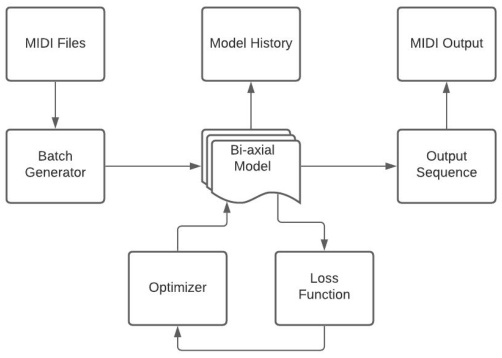

Fig 5 Flowcharts of the software design

First of all, we downloaded the MIDI files from the website, then randomly split them into the training set and validation set, stored separately in "train" and "validation" folders. Then we can read these MIDI files with functions in the Mido Python package. After that, we wrote a function to transform the MIDI messages into a 4-D NumPy array that contains each note's play-articulate state on each time step. The note matrix is presented as in Fig 6. In this function, we firstly initialized the note matrix with all zeros, and then wrote the state in the matrix by the following logic: if the message is "note_on", we can simply write the state as [1, 1]; if the message is "note_off", we write it as [0, 0]; when the time step comes to an exact beat, the play state will inherit the previous note, and the articulate state will be set to 0. We derived the note matrix as our final input to the model after traversing all the notes in the MIDI files.

$$Note\ State\ Matrix = \begin{bmatrix} [p_0, a_0]_{t=1} & \cdots & [p_{127}, a_{127}]_{t=1} \\ & \vdots & \\ [p_0, a_0]_{t=T} & \cdots & [p_{127}, a_{127}]_{t=T} \end{bmatrix}$$

Fig 6 Note State Matrix

The training and validating process share a similar structure. In this structure, we used a generator to generate batches consistently and feed them into the model. The model would then run through the time-axis and note-axis, and the output is the state matrix of the notes. Then we take the output matrix to compute the value of the loss function with regard to the original input batch, and optimize the loss function with the optimizer we chose. The loss function computes the binary cross-entropy between the output matrix and the input batch, the pseudo-code of the function is presented in Fig 7. After running the model with several optimizers, we chose to use RMSProp since it has the best performance in minimizing the loss function value.

```
Algorithm 1 Loss function derivation
Input: Output note state matrix, $O_{B*N*T*2}$;
       Input note state matrix, $I_{B*N*T*2}$;
Output: The value of loss function on all batches, $Loss(O, I)$;
 1: Initialization
 2: for b from 1 to Batch_size do
 3:     for n from 1 to N do
 4:         for t from 1 to T do
 5:             Compute binary_crossentropy(O[b,n,t], I[b,n,t])
 6:             Loss += binary_crossentropy
 7:         end for
 8:     end for
 9: end for
10: Loss = Loss/(B*T*N)
11: return Loss
```

Fig 7 Loss function derivation

In the model part, there are eight layers implemented with TensorFlow functions. Other than the LSTM layers and Dense layer used for computing the output, we also implemented several layers to integrate data inputs and the operations of changing the axis on the bi-axial LSTM model. The layers are shown in Fig 8.

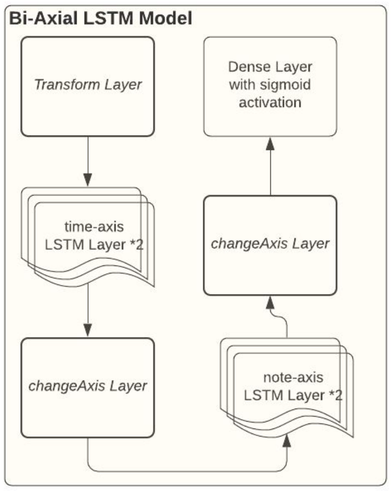

Fig 8 Flowcharts of the bi-axial model

The first layer was developed by ourselves, which takes the 4-D state arrays of each note and transforms them into expanded lists that contain the pitch, pitch class, previous vicinity, context, and beat of each note. The workflow for this layer is presented in Fig 9.

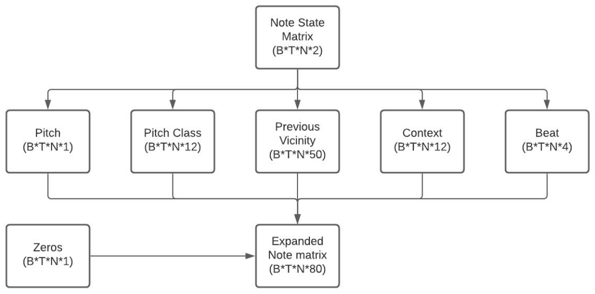
Note State Matrix (B*T*N*2)
Pitch (B*T*N*1)
Pitch Class (B*T*N*12)
Previous Vicinity (B*T*N*50)
Context (B*T*N*12)
Beat (B*T*N*4)
Zeros (B*T*N*1)
Expanded Note matrix (B*T*N*80)

Fig 9 Flowcharts of the TransformLayer

The second and third layers are the LSTM layers on the time-axis, and they both contain 300 cells with 0.5 of dropout. The operations on the expanded input lists in these layers are independent on the note-level. Therefore we can train the model parallelly on each note and update each parameter's weight in the sequence of time. Then the ChangeAxisLayer1 will change the permutation of time and note in order to fit the form in the note-axis LSTM layers. The note-wise LSTM layers have 100 and 50 cells respectively, with a 0.5 dropout ratio. These LSTM layers can be trained parallelly on the time-level but in the sequence of notes similarly. Finally, the Dense layer with sigmoid activation would give us the predicted state of each note.

The final step of the program is to generate new music pieces with the trained model. Using the "model.predict()" method in TensorFlow, we can generate new state matrices with input batches. Then, with the help of the Mido python package, we can construct new MIDI files. We set some default values in the code when writing MIDI messages, but they are subjective to change. The logic of this function is just like the reverse of the initial step. The pseudo-code of this function is shown in Fig 10.

```
Algorithm 2 Note state matrix to MIDI
Input: Note state matrix, $N_{B*N*T*2}$;
Output: MIDI file;
 1: Initialize MIDI file, default parameters, previous state matrix
 2: for tick, state in (N_timesteps, N) do
 3:     Initialize onNotes and offNotes lists
 4:     for note in range(128) do
 5:         Transform state[note] from probabilistic form to binary form
 6:         if note was played in previous state then
 7:             if note is not played in current state then
 8:                 Append current note to offNotes list
 9:             else if note is articulated in current state then
10:                 Append current note to offNotes list
11:                 Append current note to onNotes list
12:             end if
13:         end if
14:     end for
15:     Write onNotes and offNotes list to MIDI file with default parameters
16: end for
17: return MIDI file
```

Fig 10 Note state matrix to MIDI file

5. Results

5.1. Project Results

Fig 11 illustrates the loss of our model after 30 epochs. As we can see in the graph, the loss quickly converged to nearly 0 after 15 epochs for both the training and validation sets. Since there is no gap between these two sets of loss and they are both low enough, our model seems to perform correctly, without apparent signs of over-fitting or under-fitting.

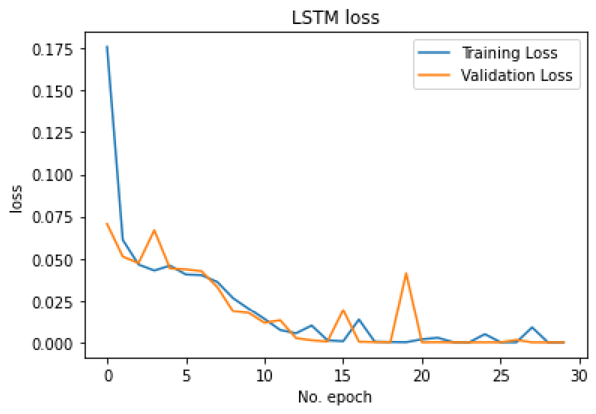

Fig 11 Training and Validation Binary-entropy Loss

To compare our model results with the original work's, we multiplied the cross-entropy by -128 and generated the log likelihood trend over epochs in Fig 12. In this plot, we saw the metric surpass the best results of the original work's and quickly increased to zero. This may raise some doubts as we used relatively fewer epochs and less training time. However, note that we also used fewer data and four musicians' work for our model. And we will discuss this in more detail in the discussion part later.

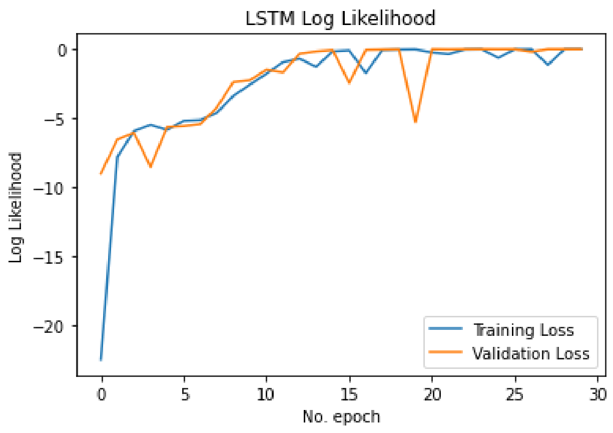

Fig 12 Training and Validation Log Likelihood Trend

In terms of metrics, we also chose both accuracy and AUC for comparison, in case the biased set of 0 and 1 in our target will affect the precision of accuracy. Fig 13, the trend of accuracy throughout the 30 epochs, has shown a bizarre "U" shape: the accuracy started at nearly 1 in the first epoch, and quickly dropped to about 0.5 after one epoch, and fluctuated a bit before soared back to almost one again in around 13th epoch. This situation might result from false negatives as we have nearly 75% of 0s in one batch of state matrices. Therefore, we introduce AUC to solve the problem of biased datasets.

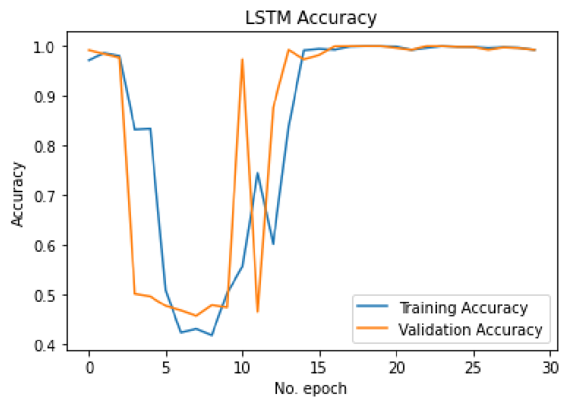

Fig 13 Training and Validation Accuracy

As Fig 14 shows, the AUC graph has an overall increasing trend, reaching nearly one after 10 epochs, yet it is with a clear temptation to decrease to a slightly lower position, at around 0.98. Although this declining trend is confusing, we can tell our model's promising results based on the almost 100% correct prediction rate returned by AUC. And we will further discuss the potential causes of our accuracy and AUC's abnormal behavior in the discussion section of insights gained.

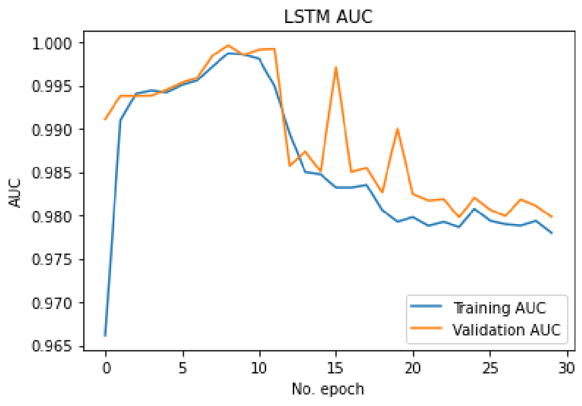

Fig 14 Training and Validation AUC Trend

Pertaining to the verification part, the above four graphs all show some level of identicality between the trend of training set metrics and one of validation set metrics. These inconspicuous gaps between the training metrics and validation metrics proved that our model is at the right capacity.

5.2. Comparison of the Results Between the Original Paper and Students' Project

One of the most significant discrepancies between ours and the author is the log-likelihood of measuring the loss between the ground truth and the predicted probability.

The log-likelihood was missing in the original blog, but we found the author later reported his best and median log-likelihood in his published version of this work [3]. They are -4.92 and -5.01, respectively, while our model's log-likelihood is somewhat close to 0 for both the training set and the validation set. This can be doubtable, especially for our number of epochs and duration of training.

We also noticed the differences between the music generated by our model and the author's model because of the deliberate deployment of the range of notes. We did not cut off the lower and upper bound of the notes as the author did, and therefore, the notes in the range 0 to 23 and 103 to 127 are likely to be played or held in our model-composed pieces, whereas in the original results, these notes will forever remain silent.

5.3. Discussion of Insights Gained

First of all, for the problem of log-likelihood, we argued that we used a relatively smaller amount of songs (87 pieces) compared to the amount of data the author used for the original work (332 pieces, all the songs on the site). This small amount of training set can be relatively easier to learn, and since we intentionally chose musicians with similar composing styles, the model is presumably performing better than the one with a more extensive and more diverse data set. Also, the fact that we used only a quarter of the amount the original model used and 20 more pieces for validation will also result in our much shorter training time and the precision of its prediction.

Secondly, for the range of notes, we cannot really tell how far our model's results will be different from the original one in terms of the generated pieces for sure, but we made the decision for reasons, and we do want to minimize the human interference for the model learning and generating music process.

In addition, we also found out that the validation accuracy is always higher than the training accuracy in each step and each epoch. This is usually because of the existing overlap between the validation set and the training set. However, we stored our training set and validation set in different folders and are sure there is no duplication of them. Therefore, we speculate that since we separated the songs of these composers into these two folders, the overlapped part is actually due to the strong personal composing style of these individual musicians. In terms of why this did not happen to the original model training (as the author did not mention any problems like this), we assume it was a result of our limited amount of dataset, which magnified the problem. As the datasets contain more musicians, diversity will be introduced and hence eliminate the issue's magnitude. We did not intend to avoid this problem by making these two data sets' musicians exclusive to each other just because it would not make sense to train models based on Bach and use it to predict Beethoven.

We also mentioned the U-shaped accuracy and a tendency of decreasing in AUC. We cannot figure out the root of these problems quickly for now, but we conjecture that since accuracy and AUC both calculate the input and output one to one correspondence, but we need to compare the note generated from the current input with the output with one timestep lag, therefore, these two metrics behaved weirdly. However, one may argue that this high accuracy or AUC is still achieved eventually. Can this be allowed if our assumption (that the calculation is one time step ahead) were correct? We could not say for sure, but with the high dependency between adjacent notes or time in each song, this is likely to happen.

6. Conclusion

This paper is a replication work of a music composing model, namely the bi-axial RNN Music Composition Model, created by Daniel Johnson in 2015. We intended to rebuild this bi-axial model with time-axis and note-axis to preserve the nature of music notes -- being both time-dependent and note-dependent.

As a result, we successfully trained our model to generate euphonic pieces, with reserved time-invariant and note-invariant patterns in our model, and at the same time achieved chords and articulation. Therefore, we would like to announce the accomplishment of the objectives and goals for this paper. As a part of our quantitative results, we trained our model with a significantly small loss and higher accuracy with almost 100% correctness in prediction, without any tendency of under-fitting or over-fitting.

Through this journal toward the above achievement, we have encountered troubles and conquered challenges, and learned numerous lessons. First of all, we have gained a more profound understanding of LSTM, how it works under the hood, and how to creatively transfer the time axis to the note axis to retain dependency.

Secondly, we once blindly believed in the choice of the optimizer (AdaDelta) in the original work and neglected the loss changes while paying more attention to accuracy (which was steadily climbing through the training). Later, after realizing the problem, we tried out several optimizers of choices and settled on the one that performed the quickest convergence.

Thirdly, we studied the customized layer in TensorFlow and how we can set it to an untrainable one, and therefore successfully deployed data transformation within the model sequence.

With these highlights, admittedly, our model still needs improvement. The first one on the list is to enlarge our training dataset, but at the same time still consider the styles of different composers. Besides, we have to find out why our weird accuracy and AUC rule out any possible errors. Thirdly, we would like to tune the hyper-parameters more, like the number of layers, batch size, and other metrics.

7. Acknowledgement

We would like to express our special thanks of gratitude to our Prof. Zoran Kostic as well as our TAs who gave us the golden opportunity to do this wonderful project on the topic, and offered us help in this project.

8. References

[1] D. Eck and J. Schmidhuber, “A First Look at Music Composition using LSTM Recurrent Neural Networks,” 2002.
[2] D. Johnson, “Composing Music With Recurrent Neural Networks,” *Daniel D. Johnson Personal Page*, Aug. 03, 2015. https://www.danieldjohnson.com/2015/08/03/composing-music-with-recurrent-neural-networks/ (accessed Dec. 20, 2020).
[3] D. Johnson, “Generating Polyphonic Music Using Tied Parallel Networks,” in *Computational Intelligence in Music, Sound, Art and Design*, vol. 10198, J. Correia, V. Ciesielski, and A. Liapis, Eds. Cham: Springer International Publishing, 2017, pp. 128–143.
[4]“MIDI,” *Wikipedia*. Dec. 11, 2020, Accessed: Dec. 20, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=MIDI&oldid=993681755.

[5] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, “Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription,” *arXiv preprint*, 2012.

9. Appendix

9.1 Individual Student Contributions in Fractions

	yl4318	yl4319	cw3210
Last Name	Liu	Li	Wang
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Defined customized layers for connecting the two LSTM models	Defined the model training code with TensorFlow package and save model output	Defined the input and output data’s transformation functions
What I did 2	Wrote paper abstract, part1-3	Wrote paper part 4, 7, and overall composing	Wrote paper part 5-6, 8