Polyphonic Music Generation with Recurrent Neural Network

Yuanhao Geng
geng29@wisc.edu

Haoran Teng
hteng22@wisc.edu

Zixiang Xu
zxu475@wisc.edu

Abstract

With the boosting application of deep learning methods used in artistic creation, people started paying increasing attention to art-related tasks gradually, especially for musical information retrieval. In this project, our group uses an LSTM-based neural network to deal with the polyphonic music generation task. More specifically, we will generate three voice parts for one given theme or melody with the self-designed encoder and decoder. For background data, Johann Sebastian Bach's chorales are selected to be our dataset. One-hot encoding, data padding, normalization process and basic musical knowledge for counterpoint and harmonics are involved in our project. We also design and conduct sufficient experiments for hyperparameter selection and manual evaluation for the model outputs. Source codes are available at `https://github.com/GYHHAHA/STAT-453-Demo`. *For future work, with rapidly developing NLP methods these days, such as BERT, seq2seq, ELMO, etc., we may mix these effective techniques with the music generation task to achieve better performance.*

1. Introduction

1.1. Motivation

Nowadays, machine learning methods for artistic creation are getting more attractive and feasible, especially with the rise of deep learning. Our group plans to develop a model to generate the polyphonic music for four voice parts based on the Johann Sebastian Bach's chorales dataset.

1.2. Background Preparation

There are amount of music works in Bach's life, which can be divided into vocal music, organ music, keyboard music, and chamber music. Among them, vocal music works account for the largest proportion of the total works. Here, 371 Four-Part Chorales are selected as the training data set for the following reasons. Firstly, these pieces have been important sources of musical examples for conservatory music and acoustic teaching materials for a long time, which are regarded as the standard templates for counterpoint composition exercises. Thus they are rare excellent materials in acoustics teaching. Second, the dataset is large enough and its form is unified. The basic format for a chorale piece is the classic four-part STAB structure while other vocal music works often feature much more complex structures, such as passion, cantata, mass and so on. It is difficult to implement a unified coding for them. Lastly, the music has a good effect, and it is easy to operate with audio sources. We could distinguish the style easily, which is helpful for users to test the model effect.

The Bach-Chorale dataset could be extracted from music21 Python package [6]. Figure 1 is a demonstration of a single chorale piece, which contains four voice parts, soprano, alto, tenor and bass. Usually, the piece from the official dataset only has finite kinds of note duration, which is easier for us to deal with. But the chorales share different tonality. Thus the normalization is needed to transfer all of them into C Major. The whole dataset has about 300 chorales with the same format as shown in the figure. **music21** is a toolkit for computational musicology, it provides massive convenient functions to process with original sheet music files for many musical information retrieval tasks.

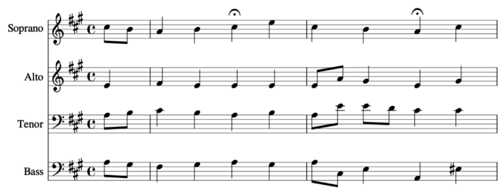

Figure 1: An example from Bach chorale dataset

We take Pytorch [20] to implement the network. Models are trained with a single RTX2070 from our own.

2. Related Work

There is a history of AI music composition[7]. [17],[1],[4] told us the early history of related algorithms. [16] explains evolutionary algorithms and traditional neural networks utilized in music composition, including ART, RNN and LSTM. Compared to computer vision, natural language processing and recommendation system, MIR (Musical Information Retrieval) is a relatively new area about pattern recognition and musical computing. Music generation, serving as an important subtask for MIR, has gained increasing attention in recent years [3].

In the past, probabilistic models were the most widely-used methods, such as hidden markov model [23] and probabilistic inference model [19], for music generation. Moreover, some researches compose music automatically with evolutionary algorithm, intelligent agent[18], and random field. Another very effective method is to use a neural network to learn music features, and then automatically learn to generate new music clips with the features.

Back in 1989, [22]RNNs were used to predict the pitch and duration of notes to generate mono music. Nevertheless, it is hard for RNNs to memorize historical sections with the gradient vanishing problem. Hochreiter [14] came up with a distinct RNN LSTM to help memorize and obtain contents of the sequence. After that, Eck [10] created music with LSTM in 2002. He could compose blues with a short recording. In 2012, Boulanger designed the RNN-RBM model[2]. Superior models showed up in recent years as well, such as Melody RNN[24](promoted ability of long-term structures learning), Anticipation-RNN[12](user-defined positional constraints), TP-LSTM-NADE and BALSTM[15](translation invariance). At present, deep learning algorithm has become the mainstream method for music generation. Now with boosting deep learning techniques, it's possible to implement these new methods in the MIR field.

Here are also some related works that we refer to: [8], [9], [27] and [11] introduced generative adversarial networks based methods. And [21], [25], [26] gave extra strategies for polyphonic music generation.

Our group tends to use a new encoding strategy, which is different from the method in [13], and the RNN structure like LSTM or GRU [5] to generate Bach style chorales for four voice parts. This is an exciting attempt for our group to use these methods for artistic creation.

3. Proposed Method

Apart from the encoder-decoder process, our main algorithm has two parts. First, we want to generate a monophonic melody. Then, we need to generate 3 other voice parts based on the melody from the former step. Here, RNN structure is introduced into our network since this is clearly a sequential problem. In the following part, we will dive into the details of our algorithm.

3.1. Encoder

The encoder consists of 9 attributes and 8 method functions, including one initialization method and four external callable functions. When initializing a class instance, the three parameters that can be provided are: the label of an appointed voice part (S-soprano-0; T-tenor-1; A-Mezzo-soprano-2; B-bass-3; default-0, which is the highest voice part), the minimum read-in step size (default set to be 0.25 beats), and the key mode (Major/Minor, default is Major). The main purpose of the encoder is to read in the dataset in the original music format, and after encoding twice, output the encoded information, which is supposed to be the input to the network.

The initial state of `song_list` attribute is an empty list. After explicitly calling the `get_all_chorals()` method, the first encoding results of all datasets would be stored in it. `song_list` is a three-dimensional list, in which the outermost layer presents the total number of tracks, the middle layer is the total step length of a single song, and the inner layer contains the single-step encoded result. The first encoding process is described in detail as below: Due to the particularity of the note duration of the chorale, the note duration of any single note of all tracks can only be an integer multiple of 0.25 beats, so it is only necessary to read the music in steps of 0.25 beats. Also, note that the note of a certain voice part at a certain moment is determined by and only by two factors, the pitch, and whether it is the continuation of the previous note (if it is a new note, it will be recorded as 1, otherwise as 0). Hence, a 9-dimensional list is sufficient to completely store the information of a certain time. Its construction is like:

```
[time, the pitch of an appointed voice
part, the note duration of an appointed
voice part, the pitch difference of
the first remaining voice part, the
note duration of the first remaining
voice part, the pitch difference of the
second remaining voice part, the note
duration of the second remaining voice
part, the pitch difference of the third
remaining voice part, the note duration
of the third remaining voice part],
```

where the pitch difference refers to the number of semitones between the difference of the note and the note of the appointed voice part at that moment. Meanwhile, it is stipulated that in `music21.pitch`, the midi value corresponding to the C-4 pitch is 60, and the midi value corresponding to the rest is 0.

In order to prepare the `data` set and `target` set, we need to call the `get_dataset()` method to encode the `song_list` in the class instance for the second time. For the `data` set, it is essentially a three-dimensional list. The difference from `song_list` is that the third dimension of `data` set is set by a new encoding method. While for the `target` set, it is essentially a four-dimensional list, and the innermost layer becomes the number of voice parts. The second encoding process is described in detail as below: In the first encoding process, the class instances `data_max`, `data_min`, `target_max`, `target_min` are used to record the range of pitch differences for the appointed voice part and other voice parts. Thus we could perform OneHot encoding for the appointed voice part based on those variables, that is, a certain pitch to be set to 1, and other pitches to be 0, and the last bit stores the info of whether it is a continuation. For example, the pitch range of the soprano part is 59-81, if the pitch at that moment is 63 and the note continues, then the length of the list should be 23, the 5th and last bits should be 1, and other bits should be 0. Below in Figure 2 is an example of note embedding. For the rest of the voice parts, we perform the same encoding process with two differences: First, we need an independent list for the state of each voice part at a certain moment. And the second is that the length of the list should correspond to the pitch difference range instead of the pitch range. The reason for this approach is that the pitch of a single note does not have absolute meaning, while the pitch difference relations are universally applicable. This encoding method of replacing intervals with pitch difference is also one of the innovations in the construction of this encoder.

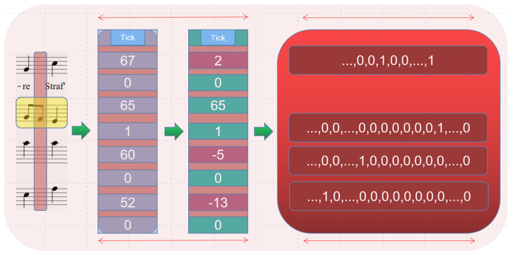

Figure 2: How note embedding works

In addition, in order to meet the requirements of the network, it is necessary to perform padding operations on the `data` set and the target set, that is, the length of the second dimension of all lists should be unified to the maximum sequence length of this dimension, and the insufficient part is filled with a list of all zeros. Finally, after two encoding processes, the returned dataset should be in dictionary format, which is divided into three parts: `data`, `target` and `seq_length`. The value of key `seq_length` is a one-dimensional list, where we store the actual length of tracks before padding.

3.2. Network

In Figure 3, basic network structure is demonstrated. It has the following components: three one-way LSTM layers + three linear layers + one activation layer + one normalization layer.

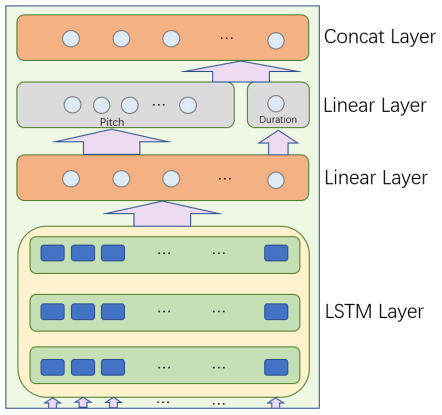

Figure 3: Network Structure

The basic LSTM cell structure is shown in Figure 4. The last two linear layers are parallel to each other while connecting to the activation layer and the normalization layer with `batch_size` set to 5. At the beginning of the network, the dimension of the input is set to exactly the innermost dimension of the `data` set. And the number of final output layers is the same as the number of hidden layers, which is set to 300. Since it is a one-way LSTM, the final output dimension should still be 300. The reason for choosing a one-way LSTM here is that: First, the computational cost would be small and it could reduce training time. Secondly, in actual experiments, it is found that the use of two-way LSTM does not significantly improve the network.

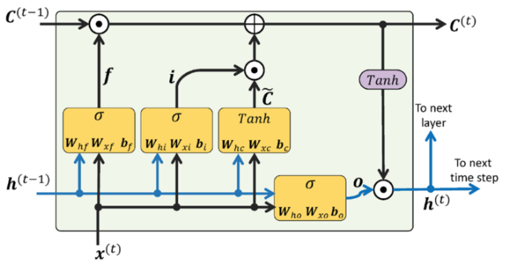

Figure 4: The cell structure of LSTM
Source: Machine Learning(3rd Edition)

Then, we come to the first fully-connected linear layer, the input is consistent with the output of the previous part with dimension set to 300. Afterwards, the separation of the pitch layer and the note duration layer is performed immediately, where the pitch layer is composed of a linear layer with an output dimension of `3*(target_size-1)` and a normalization layer, and the note duration layer is composed of a linear layer with an output dimension of `3*1` and an activation layer. Finally, re-splice to ensure that the result dimension is consistent with the target set dimension to proceed to the loss function calculation step.

Since the discrimination of pitch is a classification problem, cross-entropy is introduced as the loss function. In order to avoid the situation where loss becomes `inf` or `nan`, a truncation function is implemented outside the logarithm function. In this way, the calculated value after logarithm function would be controlled within `[-10,0]`, which could prevent from slow convergence.

In addition, since a padding operation was performed before the `data` set enters network, dynamic RNN input should be performed according to `seq_length` before entering the LSTM layer. And when calculating the loss function, the loss that does not belong to the sequence should not be included in the result. For the first problem, the `pack_padded_sequence` function and `pad_packed_sequence` function in PyTorch should be called to compress the long sequence into one dimension, then expand the one-dimensional sequence into a multi-dimensional, and finally fill in the list with zeros according to the maximum `seq_length` value to restore the initial sequence length (This refers to the length of the second layer, and the length of the third layer has been increased to 300 when outputting the LSTM layer). For the second problem, with masking, the whole sequence omits the padding part by multiplying zero onto corresponding positions.

When initializing the network, we need to explicitly pass in the input layer size (determined by the pitch range of the appointed voice part), hidden layer size (for debugging), output layer size (determined by the pitch range of the remaining voice parts), and the number of LSTM layers, the maximum sequence length (i.e. `max{seq_length}`). Also, though the `length` attribute has an empty list as the default value, in practice, a list of length 5 (`batch_size` value) needs to be passed in during network training for dynamic RNN input.

Generally speaking, there are three core steps for training neural network in PyTorch, namely, gradient clearing (`optimizer.zero_grad()`), back propagation (`loss.backward()`), and optimizer iteration (`optimizer.step()`). When it comes to the test mode, if we would like to predict for the remaining three voice parts, just load the track to be predicted and call the network input. Thus at this time, we don't need to perform the previous three steps. Instead, we get the probability distribution directly and start decoding, that is, unlike the self-encoding algorithm, the decoder is not used during training. Besides, the optimizer we use here is RMSprop. Compared with AdaGrad, due to the increase of the coefficient that controls the historical gradient decreasing, and due to the cumulative square gradient appearing at the denominator when the parameters are updated, it ensures the parameter space to gain a larger gradient in smoother direction while the opposite in steeper directions, which would in the end speed up training. Later experiments did prove that the RMSprop method performs better.

3.3. Decoder

The decoder consists of two attributes and four methods, including one initialization method and two external callable functions. When initializing and creating decoder class instances, the previous decoder instance must be passed in as one of the attributes, and the function of the second attribute is to store the information after the first decoding. After initialization, the appointed melody information and the prediction result should be decoded for the first time by `decode_oneHot_to_short_code()` method. The parameters introduced here are the probability distribution of the new melody (the appointed voice part) input and network output in the previous test mode.

When calling the first decoding function, we used two probability threshold parameters, duration probability and note changing probability. The idea here came from traditional and acoustic writing principles: In the process of harmony, if the latter chord has a sound in the former chord, it should continue in the same part in most cases. For the OneHot coding of a voice part at a certain time, there are two situations that the sound will not last. Firstly, the end value of coding is less than the threshold value of note-changing probability. Secondly, The value of the voice part at the previous time has probability at the current time exceeds the threshold value of duration probability. If both cases pass the test, the note with the maximum pitch probability in the current code can be selected as the new note. After the first decoding, the length of the innermost vector generated is exactly the same as that of the first encoding, which is still 9. Specifically,

```
[time, the pitch of an appointed voice
part, the note duration of an appointed
voice part, the subscript of the output
note list of the first remaining voice
part, the note duration of the first
remaining voice part, the subscript
of the output note list of the second
remaining voice part, the note duration
of the second remaining voice part,
the subscript of the output note list
of the third remaining voice part, the
note duration of the third remaining
voice part].
```
Notice that the 5th, 7th and 9th stored are array subscripts, not sound differences, they will be converted at the second decoding.

Next, with `get_score()` method, the information stored in the `new_short_code` attribute is decoded for the second time, output to `Score` instance, and saved as a .xml file. Firstly, create a four-part empty score for filling. Secondly, the voice part information was read in turn for single time steps. If it is the specified voice part, the MIDI pitch information is filled directly. If it is the other parts, it needs to be converted: the actual pitch is the sum of the specified part pitch at the current time, the array subscript and the minimum value of the range of pitch difference. Thirdly, according to the coding rule, whether the original sound is sustained or not is determined by the 0-1 value with odd subscripts (Python counts from 0). Fourthly, after this method, our model returns the result, including midi and spectrum files.

4. Experiments

4.1. Dataset Details

We conducted our experiments on the Bach Chorales Dataset from Music21 third-party package[6]. It could be directly reached from the build-in dataset iterator of Music21. The official guide link is at `https://web.mit.edu/music21/doc/moduleReference/moduleCorpusChorales.html`. In detail, it contains 361 chorales from Bach's sacred chorale pieces from his Cantatas, Passions and Mass with different bar lengths, varying from 8 to 107. Usually, one bar contains four beats. In our experiment, 70% of the dataset were regarded as the training set while the remaining 20% and 10% dataset served as validation set and test set respectively.

4.2. Experiment Details

Here, we tried many combinations among the optimizer, the network structure, and the learning rate. Details of the combination setups for the experiment are shown in Table 1. We took Adam, SGD and RMSprop as candidate optimizers and set the potential learning rate to 0.003, 0.01 and 0.03. In order to get the optimal LSTM hyperparameters setup, we restricted its hidden size to 200, 300 and 400 and its layer number to 2, 3 and 4 respectively. Then we conducted our experiment with the help of cross entropy loss to observe the best hyperparameter combination. The best combination result can be reached from `https://github.com/GYHHAHA/STAT-453-Demo`. We ran the experiment on a single RTX 2070 card. Normally, the training process would finish in around an hour.

Table 1: Options for Hyperparameters

	$Option_1$	$Option_2$	$Option_3$
Optimizer	Adam	SGD	RMSprop
Learning Rate	0.003	0.01	0.03
LSTM Hidden Size	200	300	400
LSTM Layer	2	3	4

4.3. Manual Evaluation

In addition to the statistical metrics, due to the fact that a criterion for the evaluation on art outputs could be vague since the similarity between Bach's real pieces and AI pieces is difficult to emotionally judge and may vary among people, we planned to introduce a survey about the identification for real Bach chorales and generated pieces. In detail, we let 20 people make judgments for 50 pieces on whether each piece is generated or composed. Then we made a hypothesis test to show whether the people are able to tell the difference between the fake one and the real one, which could partly measure the success of our model.

5. Results and Discussion

5.1. Experiment Results

From Table 2, we could figure out the best model hyperparameter combination as lr=0.003, optimizer=RMSprop, LSTM hidden layer size=300 ,and LSTM layer number=3. We have provided the pre-trained model for the best result in our GitHub demo example.

5.2. Survey Results

In the hypothesis test, the null hypothesis is that people are able to tell the difference between the real piece and the fake piece. To simplify the survey analysis process, we assume our work is successful when people get an accuracy lower than 50%. Since the response is currently binary, now we only need to get the p-value from the success count over total of 50 trials with the Bernoulli test.

Table 2: Performance comparison for difference hyperparameter combinations

		lr: 0.003			lr: 0.01			lr: 0.03		
Hidden Size	Layer	Adam	SGD	RMSprop	Adam	SGD	RMSprop	Adam	SGD	RMSprop
	2	0.9186	0.9452	0.7362	0.7262	0.7521	0.7407	0.9188	0.8091	0.8997
200	3	0.6958	0.9597	0.6951	0.8808	0.8325	0.9206	0.7376	0.9432	0.8228
	4	0.7365	0.715	0.9809	0.8019	0.9796	0.8928	0.9613	0.8402	0.6939
	2	0.6936	0.7547	0.7822	0.74	0.8788	0.8879	0.6976	0.7264	0.7848
300	3	0.7487	0.771	**0.6895**	0.9053	0.9785	0.8973	0.7222	0.8639	0.8375
	4	0.8342	0.813	0.8084	0.8346	0.7091	0.866	0.9478	0.7006	0.8089
	2	0.8349	0.985	0.7005	0.8132	0.9767	0.7008	0.7422	0.9484	0.9369
400	3	0.8636	0.7084	0.7557	0.7204	0.8509	0.9045	0.7945	0.8729	0.9207
	4	0.8699	0.7957	0.8445	0.7472	0.7407	0.7875	0.9494	0.8016	0.8238

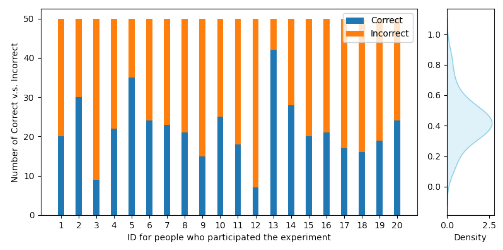

Figure 5: The Survey Results with Bar Plot and Density Plot

In Figure 5, we could see the visualized result for our survey experiment. Qualitatively speaking, since there are 5 trials regarded as success out of 20, we got our

$$p - value = 0.020695 < 0.05,$$

which means people are not able to distinguish the model outputs and the real pieces. Namely, the model is able to produce some deceptive artistic fractions.

5.3. Weakness and Improvement

Though the model has achieved an impressive effect on the generation task, there still exists some possible weaknesses. First, the chorales are highly structurally harmonious, which means some tough situations in the regular counterpoint composition process are not included. Thus we are not sure if this simple LSTM-based model has the ability to deal with more complex datasets. Second, currently we are using cross-entropy as our loss function. But according to basic harmony theory, there are many solutions to accompany a fixed melody or theme. Thus over-fitting may be a serious problem, especially in artistic-related tasks. A looser loss may be required for future musical information retrieval problems.

Data argumentation is also an important technique in Computer Vision field. Similarly, in polyphonic chorale generation, the order of the voice parts can be unfixed. More specifically, we can change some sections or bars with the other voice. According to musical theory, they harmony with each other. This kind of operation may add robustness to the model, especially when the dataset is not very large.

Besides, as we know that attention mechanism is powerful in modern deep learning. Recalling our process of dealing with the duration and pitch, it seems to be natural to implement the attention method on this two-level tensor, which may get significant improvements in our results.

6. Conclusions

In this paper, we successfully introduced a neural network with a recurrent structure for polyphonic music generation. We inventively designed the encoder and decoder structure to transfer the sheet music score format into the tensor format. In the previous works, MIR methods pay more attention to classical machine learning model, especially for the polyphonic sequence generation task, while here we try the deep learning method to gain a satisfactory result, which widens the diversity of the potential artistic forms. In future work, with the boosting of various Natural Language Processing methods, such as BERT, seq2seq, ELMO, GPT and so on, it can be very exciting to introduce these new deep learning tools into different Musical Information Retrieval tasks, and namely, more complex block structures instead of the single LSTM are worth a try.

7. Contributions

Haoran Teng and Zixiang Xu were responsible for the encoder and decoder design and the completion of the survey, while Yuanhao Geng took charge of the network design, the connection for both encoder and decoder and the analysis of the survey results. Also, we would like to express our gratitude to Prof. Raschka for his patient and detailed advice all the way through the project.

References

[1] C. Ames. Automated composition in retrospect: 1956-1986. 20(2):169–185, 1987.

[2] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent. Modeling temporal dependencies in high dimensional sequences: Application to polyphonic music generation and transcriptio. *ICML*, pages 1881–1888, 2012.

[3] J.-P. Briot, G. Hadjeres, and F. Pachet. *Deep learning techniques for music generation*. Springer, 2020.

[4] K. H. Burns. The history and development of algorithms in music composition. *PhD thesis, Ball State University*, 1995.

[5] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[6] M. S. Cuthbert and C. Ariza. music21: A toolkit for computer-aided musicology and symbolic music data. 2010.

[7] S. Dai, Z. Zhang, and G. G. Xia. Music style transfer issues: A position paper. *arXiv preprint arXiv:1803.06841*, 2018.

[8] 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. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[9] H.-W. Dong and Y.-H. Yang. Convolutional generative adversarial networks with binary neurons for polyphonic music generation. *arXiv preprint arXiv:1804.09399*, 2018.

[10] D. Eck and J. Schmidhuber. Finding temporal structure in music: Blues improvisation with lstm recurrent networks. *Proceedings of the 12th IEEE workshop on neural networks for signal processing*, pages 747–756, 2002.

[11] G. L. Guimaraes, B. Sanchez-Lengeling, C. Outeiral, P. L. C. Farias, and A. Aspuru-Guzik. Objective-reinforced generative adversarial networks (organ) for sequence generation models. *arXiv preprint arXiv:1705.10843*, 2017.

[12] G. Hadjeres and F. Nielsen. Interactive music generation with positional constraints using anticipation-rnns. *arXiv preprint arXiv:1709.06404*, 2017.

[13] G. Hadjeres, F. Pachet, and F. Nielsen. Deepbach: a steerable model for bach chorales generation. In *International Conference on Machine Learning*, pages 1362–1371. PMLR, 2017.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] D. D. Johnson. Generating polyphonic music using tied parallel networks. *International conference on evolutionary and biologically inspired music and art*, page 128–143, 2017.

[16] C.-H. Liu and C.-K. Ting. Computational intelligence in music composition: A survey. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(1):2–15, 2016.

[17] G. Loy and C. Abbott. Programming languages for computer music synthesis, performance, and composition. *ACM Comput. Surv*, 17(2):235–265, 1985.

[18] M. Minsky. Music, mind, and meaning. pages 1–19, 1982.

[19] J.-F. Paiement, D. Eck, and S. Bengio. A probabilistic model for chord progressions. In *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR)*, number CONF, 2005.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[21] O. Peracha. Improving polyphonic music models with feature-rich encoding. *arXiv preprint arXiv:1911.11775*, 2019.

[22] P. M. Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.

[23] A. Van Der Merwe and W. Schulze. Music generation with markov models. *IEEE MultiMedia*, 18(3):78–85, 2010.

[24] E. Waite. Generating long-term structure in songs and stories. *Magenta Bolg: https://magenta.tensorflow.org/blog/2016/07/15/lookback-rnn-attention-rnn/*, 2016.

[25] Z. Wang, D. Wang, Y. Zhang, and G. Xia. Learning interpretable representation for controllable polyphonic music generation. *arXiv preprint arXiv:2008.07122*, 2020.

[26] Z. Wang, Y. Zhang, Y. Zhang, J. Jiang, R. Yang, J. Zhao, and G. Xia. Pianotree vae: Structured representation learning for polyphonic music. *arXiv preprint arXiv:2008.07118*, 2020.

[27] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847*, 2017.