LSTM for Piano Music Generation

Hector Anadon
hectoral@kth.se

Albert Bou
albertbo@kth.se

Panteleimon Myriokefalitakis
pmy@kth.se

Talha Khan
talhak@kth.se

Abstract

In this project we try to generate piano music by using a Recurrent Neural Network architecture. More precisely, we feed the network with a dataset of piano music in the MIDI (Musical Instrument Digital Interface) file format, following a note-by-note approach. Our aim is to get a sound comparable to the original one from the dataset and in order to achieve this goal we made use of an architecture of Recurrent Neural Networks (RNNs) called Long Short-Term Memory (LSTM). The reason of choosing this architecture stems from the ability of these networks to remember states from the past, a property that is especially important when working with time sequences.

1 Introduction

Within deep learning, the RNN architecture is gaining wide interest. The essence of RNN and the interest in it is due to the fact that unlike traditional deep neural networks, RNNs maintain an internal memory. The ability to maintain a memory of relevant information from the past makes them well suited to learning sequential data such as speech processing, music, or handwriting. Our interest in this paper is exploring the effectiveness of a specific architecture of RNNs called LSTM in generating piano music given a dataset of piano music in the MIDI format.

The problem with the traditional RNN architecture is that they are difficult to train when dealing with learning long-range dependencies. Backpropagating errors over many time steps leads to either vanishing or exploding gradients [12]. LSTM handles the vanishing and exploding gradients by maintaining a long-term memory and short-term memory. Most of the recent progress using RNNs has been specifically using LSTM.

Before we go deeper about RNNs, it is good to mention that instead of Deep learning there are other ways to generate music as well. One of them was mentioned in [15] and it is about using Markov Models to generate music. The authors used HMMs (Hidden Markov Models) in order to make the music of two instruments consistent; for the rhythm and melody modeling prediction-suffix trees are used.

For music generation, there are six major projects both open and closed source [4]. The projects either use MIDI file note sequences or raw audio. Of particular note are Google's Magenta project which uses MIDI files but only generates a single stream of notes and DeepJazz which can generate chords as well but converts the file to a single pitch and single instrument. Both of these projects use LSTM for training. Another project of note is Google's WaveNet which processes raw audio and uses Convolutional Neural Networks (CNNs) for learning.

Researching for this paper, we found previous work of note that related to our project. Nayebi and Vitelli [14] attempted to model recurrent music sequences by operating on raw audio waveforms and using them as inputs. To achieve this goal, they used LSTM as well as the Gated Recurrent Unit (GRU) which is another RNN variant that is similar conceptually to LSTM.

In addition, Mozer in [13] used RNNs to synthesize music using a note-by-note approach. He inferred that RNNs are not such a good approach to finding long-term dependencies in data (vanishing gradient problem) and said "the compositions suffer from a lack of global coherence". The problem of vanishing gradients seems to be solved by the use of LSTM, as it can be seen in [8] where Eck and Schmidhuber used LSTM networks to analyze the structure of Blues songs. They used binary vectors to model individual notes which appear in a fixed sequence. As they mentioned, a feed-forward network would have no chance of composing music in this way. Without the ability to store past information, a network like this would be unable to keep track of where it is in a song. Finally, they inferred that LSTM networks were able to learn the global musical structure and use it to compose new pieces.

Later, Eck and Lapamle [7] described a method of using LSTM in combination with an autocorrelation-based predictor of metrical structure. The unconventionality here was in the addition of “time-delay connections that correspond to the metrical hierarchy of a particular piece of music”. They claimed that in this way the network was able to learn even more correlations in the input. Furthermore, Chung et al. in [10] used GRU networks to model polyphonic music sequences using MIDI datasets and found these networks’s performance to be similar to that of LSTM.

Our paper will use a note-by-note approach rather than processing raw audio due to its computational complexity [4]. Although raw audio can be used to create a wider range of sounds, its inherent complexity and computational expense made it unsuitable for this time constrained project. In addition, there are examples of good results having been produced using a note-by-note approach in DeepJazz and Magenta and so it is still a worthy task to explore. There are various LSTM architectures to choose from. The paper by Greff et. al [11] shows that none of the variants demonstrate significant improvement upon the standard LSTM architecture. Therefore, we decided upon using the standard version in order to gain a deeper insight into the architecture. For implementation we used the Keras library built upon TensorFlow and the Music21 library for extracting notes from MIDI files [3][5][6].

The rest of the paper will be divided into sections: Method, Experiments, Results, Discussion and Conclusions, and Future Work. In the Method section we explain our feature extraction, learning, and evaluation methods. In the Experiment section, we will detail our experimental setup and the various experiments we will attempt to obtain the best performance. After that, we will provide the obtained results from the experiments in the Results section and discuss the insight gained, areas of improvement and further future work in the Discussion and Conclusions and Future Work respectively.

2 Method

The present section provides specific information about our project implementation, as well as the reasoning behind all technical decisions made during the process. More specifically, the section is divided in two parts. First, in 2.1 data representation and feature extraction is explained. Second, Section 2.2 summarizes the network’s architecture used to tackle the problem of automatic music generation.

2.1 Data Representation and Feature Extraction

The dataset used for training is composed by files in MIDI format. The decision of working with this format stems from the fact that MIDI files offer a set of advantages with respect to other audio extensions such as WAV or MP3 that are very convenient for our specific problem.

MIDI can be understood like the digital alphabet for music. This format encodes a series of messages that tell an electronic device how to generate certain sounds. Thus, since no actual sounds are stored, MIDI files have a very small size. When working with big datasets, file size is obviously a critical aspect, especially if the data needs to be transferred to a remote server. Furthermore, with MIDI format all aspects of the sound can be edited. More specifically, with MIDI format features such as the pitch, the velocity, the volume, or the different instruments can be isolated and modified independently. That allowed us to define to a very high degree of precision to which type information our network had to be exposed (i.e. only chords; chords and notes; chords, notes and measures). Finally, MIDI guarantees no interference or background noise in the data.

With the aim to easily manipulate and create MIDI files, the Python-based package music21 was used as one of the main pillars of our implementation. Music21 contains a set of tools that has been around since 2008 and permits the study of large datasets of music and in our case, isolate sequences of piano chords and generate new melodies with individually generated chords.

Hence, with the help of the aforementioned Python package, the first steps in our implementations were to import the files from our dataset one by one and, for every timestep, obtain the chords and the notes to play, as well as the measures (key and time signature). This combination of metrics can be understood as one state in a time series. In the following step, by concatenating the information from all different piano pieces in our dataset, we obtained the total sequence of states we would use to train our LSTM network. Currently, different songs are concatenated without any type of transition.

Also, inputs and outputs in LSTM architectures are normally defined as a series of one-hot representations. Hence, it was necessary to generate a dictionary containing all unique combinations of metrics in the dataset to easily convert sequences of states into sequences of vectors of one-hot encodings. Another dictionary, where keys and values were interchanged with respect to the first one, was also defined in order to re-build state sequences given a series of network outputs in one-hot encoding.

2.2 Long short-term memory architecture

An LSTM network has been used to predict the following states of a piano song given a sequence of previous states. Previously, this LSTM network has been trained with a dataset of piano music. In this section, further details will be explained about the technical decisions made regarding LSTM.

As mentioned in earlier sections, an LSTM network follows an RNN architecture. RNNs are networks specialized in processing sequential data. Given an input and a state, a RNN generates an output which corresponds to the following state. With respect to other types of RNN, LSTM networks introduce a memory cell that allows to store old information to better capture long term dependencies. Their internal architecture permits to remove or add information to the memory cell to keep the most relevant information.

In this project a two layer LSTM has been used. Each of the layers contains 128 hidden nodes and use hyperbolic tangent as the activation function. Besides, dropout is applied after every layer to achieve a better generalization. Different architectures can be used. However, this is highly used in the state-of-the-art approaches [9]. The last layer is a fully connected layer with softmax activation function that outputs the probability of the predicted chord. This architecture can be seen in Figure 1.

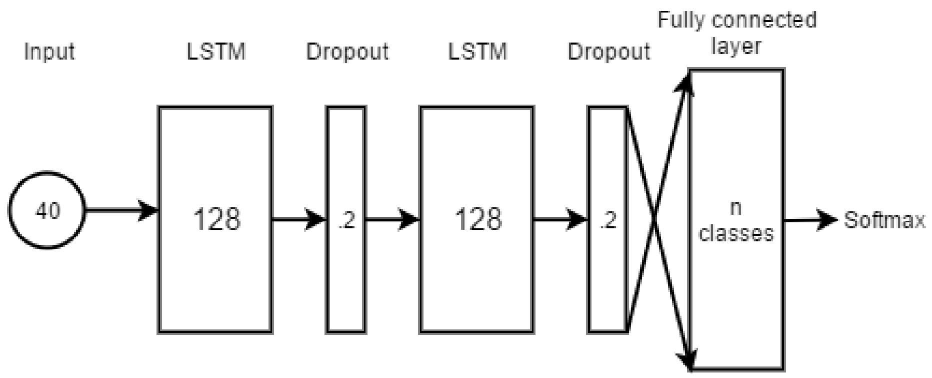


Figure 1: Model architecture

In order to train, sequences of 40 states with a 3 state shift step have been used as inputs to the network represented in one-hot encoding. The correspondent label in order to perform supervised learning is the state following the last state of the window. The loss function that the network is minimizing is *categorical crossentropy*. It is a multi-class logarithmic loss function that quantifies the accuracy of a classifier by penalizing false classifications of the following state. For optimizing it, RMSProp [16] has been used which is recommended for RNNs. The optimizer divides the gradient by a running average of its recent magnitude. For its parameters, the recommended [16] settings have been used, being η (learning rate) a fixed value (selected through validation in Section 3) with momentum and without decay.

After training, new sequences of states can be generated by the network. To that end, a seed of 40 states is randomly selected from the dataset. A seed is needed because RNN need a previous state to generate the next one. Then, by using a seed of length 40 we hope that the previous state to the first generated sample will have successfully summarized the important information along the whole seed sequence, allowing the network to suggest a suitable continuation.

The output of the network at each generation step is a set probabilities for each possible unique state that appear on our dataset. Given the set of probabilities of a state, the sample function selects a state depending on a parameter we called diversity. Variations on the value of this parameter result on selections of more or less risky states (less likely). In other words, diversity allows to have a wider variety of songs that would hopefully still seem human-made.

3 Experiments

In order to implement the method mentioned above, Keras high-level neural networks API on top of TensorFlow API has been used. The final experiments have be executed on K80 nodes of PDC Center of High Performance Computing.

3.1 Data

For our experiments we have used a dataset consisting of 324 piano songs in MIDI file format that we acquired from the Classical Piano MIDI page found in [1]. These pieces had been created from several artists such as Beethoven, Mozart as well as other great artists and come from different time periods, helping in the generalization of our approach.

From this data, we have generated two datasets, one small dataset of 6 songs corresponding to Mozart and Bach equally and one bigger of 150 songs correspondent to several authors. The reason of not using all of the dataset is because the preprocessing part takes really long.

An important reason that we use this specific dataset is that these songs only use one instrument - the piano - which simplifies the training. The features extracted from each MIDI song are chords, notes, and measures (key and time signature) for each time step. Typically, these songs had multiple voices playing various melodies which were then concatenated into one voice. This was to make the data monophonic which is the simplest type of music structure possible and is similar to what is done in Magenta which only accepts monophonic MIDI files.

There can be a potential problem with using many different artists from different time periods in that it introduces complexity that would not be present if trained with only one artist. An experiment will be done to compare the results in using the entire dataset with all the artists as compared to isolating a single artist for training.

Another potential problem with these experiments is that the dataset of 324 songs may be too small for training the network properly but will be done as we are constrained by our computational resources. A larger dataset would be computationally infeasible to complete within the time limit of this project.

3.2 Hyperparameters

A study done by Greff et. al [11] demonstrated that generally there was no improvement in performance using a variant LSTM architectures. The study did 5400 experimental runs various LSTM networks controlling various hyperparameters. The conclusion of the study was that by far the most important tuning parameter was that of learning rate accounting for more than two thirds of the variance in the results. The second most important parameter was the hidden layer size. Another important conclusion from the study was that there was no demonstrable dependence between the parameters and therefore the parameters could be tuned independently without expecting any adverse effects.

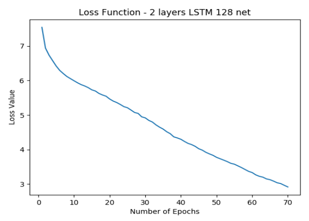
As part of the future work for our project, we would like to experiment with the two most important training parameters and systematically test the parameters to obtain the best results. To test the learning rate, we have to follow the method used in the study by Greff et. al : "For each dataset, there is a large basin (up to two orders of magnitude) of good learning rates inside of which the performance does not vary much. A related but unsurprising observation is that there is a sweet-spot for the learning rate at the high end of the basin." And so, to search for a good learning rate, it is planned to follow the method that "it is sufficient to do a coarse search by starting with a high value (e.g. 1.0) and dividing it by ten until performance stops increasing." Also due to the low demonstrated dependence between learning rate and hidden layer size, the best learning rate parameter can be tuned using a relatively low hidden layer size to save computational time.

For choosing a hidden layer size, the general rule that we are following is that larger networks perform better but also have a higher computational time and so we will restrict the max value of our hidden layer size to 1024 hidden units. The potential problem with overly large hidden layer size is the risk of potentially overfitting the data and so a careful balance must be observed. So far in our implementation, we manually searched for proper values for the learning rate and the number of hidden nodes due to time constraints.

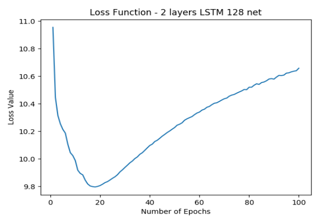
4 Results

There is inherent difficulty in evaluating the performance of generated music. There is no objective way to evaluate how well the LSTM performs. The paper by Allen Huang and Raymond Wu. [9] evaluated their generated music by asking 30 people to evaluate 3 different sources of piano music one of which was their own generated source. The results from that study demonstrated that their network generated what other evaluated as passable music. To evaluate the performance of our own network, we created a survey [2] in which 26 people rated 6 generated songs.

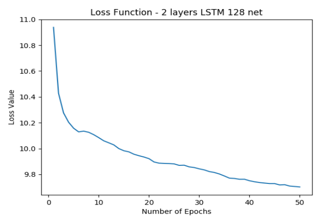
In Figure 2 we can observe the different training loss for the different experiments. In Figure 2a, the training has been made with the small Mozart-Bach dataset for 70 epochs. We can see that the loss is still decreasing. In Figure 2b it has been trained with the 150 songs dataset for 100 epochs. Keras default parameters have been used, however, as the loss is increasing we can deduce that the η parameter is too high thus overfitting the results. For that, in Figure 2c it has been trained with the same dataset with smaller learning rate, however, it would require longer training time resulting in further decreasing of the loss function value.



(a) Mozart-Bach dataset 70 epochs



(b) 150 songs for 100 epochs



(c) 150 songs for 50 epochs

Figure 2: Loss function

For evaluation purposes in the poll, a total of 6 songs are generated with the network parameters in different states of the loss function. In 3 we can observe the representation of the first song.

1. Lowest loss value of the small Mozart-Bach dataset, 50 epochs, Fig. 2a
2. Overfitted estate in epoch 100 of the 150 songs-dataset, Fig. 2b
3. Lowest loss with 150 songs-dataset, epoch 50, Fig. 2c
4. First epoch result with small Mozart-Bach dataset
5. Lowest loss with 150 songs-dataset, epoch 20, Fig. 2b
6. Lowest loss with 150 songs-dataset, epoch 50, Fig. 2c

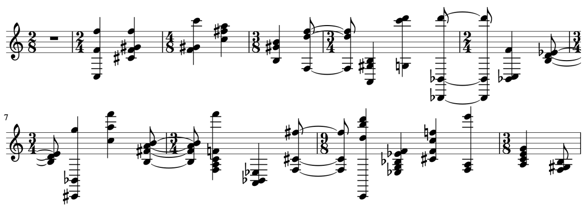


Figure 3: Pentagram with the first 10 measures of the song generated after training with 150 songs for 50 epoch.

The results from the survey can be seen in Figs. 4, 5, 6. From Fig. 4 we can observe the music education. There is a big percentage with music studies and the rest are interested in music however there are not a considerable number of experts.

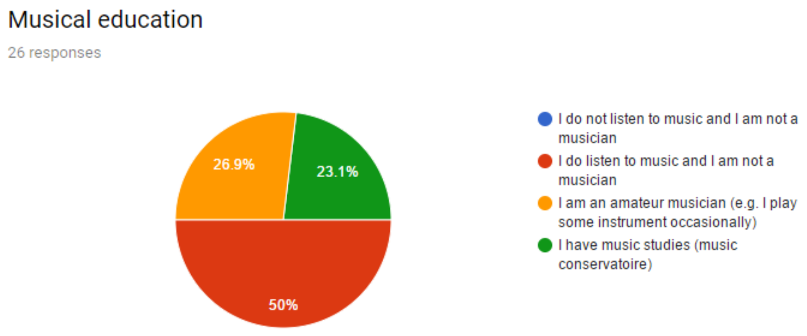


Figure 4: Music eduction

In order to measure the performance of the music generated, we asked to rank how much they like different songs in the left plot of Figure 5. The songs to be compared are, from left to right, (1), (2), (3), as previously mentioned. As shown in the graph, the general preference order is the first one followed by the second. We can then estate that it is easier to generate similar music to a smaller dataset than a bigger one. In addition, the second one (100 epochs, overfitted) is preferred over the third (50 epochs) meaning that it is required to train longer (more epochs) to obtain better music.

In the right hand plot in Figure 5, it is asked about how likely is the song generated by a computer. As expected, the poll suggests that it is very likely on average, but more likely in the first one (4) 1 epoch, over (5) 20 epochs reaffirming that longer training improve results. However, this music is not passable as real because only 4 people thought that it was low likelihood and 2 thought it was impossible, while the majority thought that it was very or most likely generated by a computer.

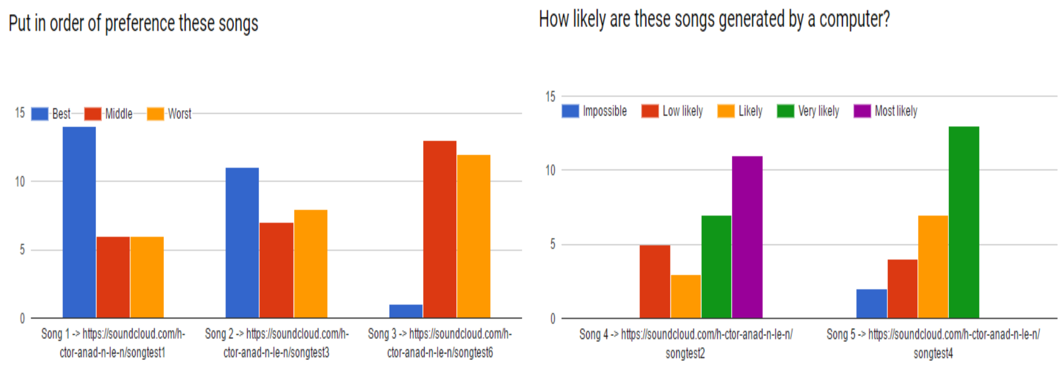


Figure 5: Song ranking

Finally, we asked what are the reasons they thought the music was computer generated. Among other things in Figure 6, the poll suggests that it is rhythmically monotonous (the length of the notes is the same), there is no harmony, nor melody and silences are missing. We talk about this improvements in Section 6

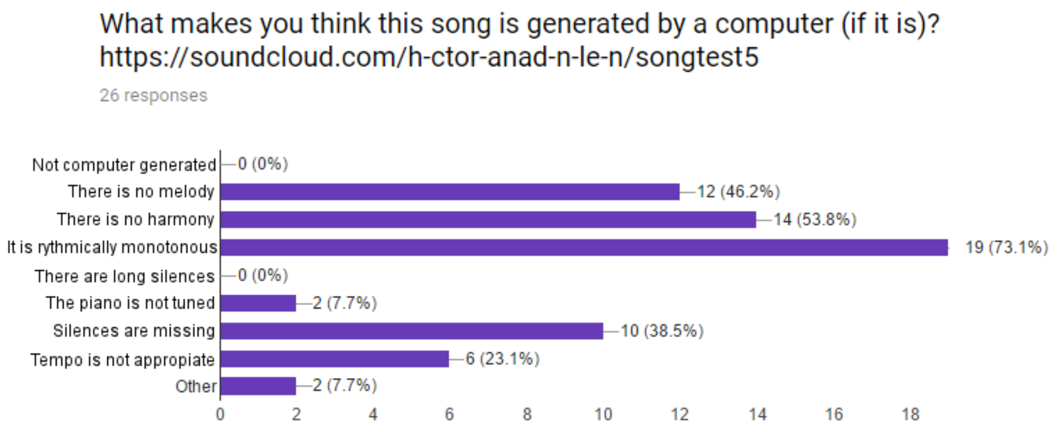


Figure 6: Reasons music is computer generated

Furthermore, the training songs and the generated songs have been visualized in a two dimensional plot representing how different they are between each other. In order to calculate that, the normalized number of appearance of each state is used as the feature vector. Then dimensionality reduction is applied with t-SNE. Despite not being a proper performance metric, it helps visualize that the generated songs lie, somehow, close to the original songs in the mentioned dimensional space.

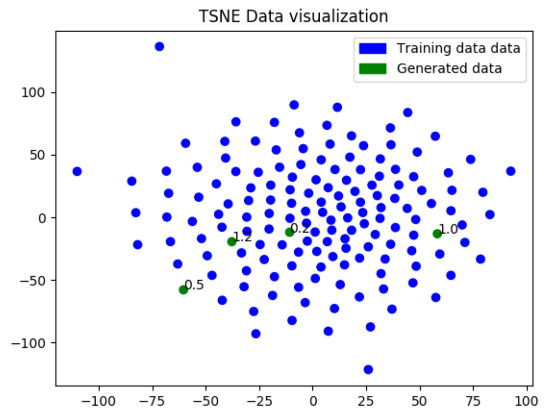


Figure 7: t-SNE songs representation resemblance

5 Discussion and Conclusions

In this project, we have implemented a LSTM-based program capable of generating piano music in an autonomous fashion. We fed our network with time series of music features extracted from MIDI files expecting to generate new songs with a certain resemblance to the original time series after training. Despite the fact that further experimentation should be done to reach the optimal performance of our implementation, results proved that this is a valid approach for music generation.

One of the conclusions that we extract from the evolution of the loss function during training is that longer training time is required to reach convergence, since the loss function value seems to keep decreasing after over 20 hours of training on a Tesla K80 GPU.

One of the consequences of this can be observed in the pentagram (Fig. 3), which displays a song generated by our network. In there, key and time signature changes occur every few chords/notes. That phenomenon does not exist in the original songs. It is true, however, that at early stages of the training process the aforementioned metrics change almost at every time step. By contrast, after several epochs the networks seems to learn at least to keep the metrics fixed during brief periods of time. Therefore, we consider that longer training times could help solving this issue. Summarizing, our assumption here is that further training would result in lower loss values and, consequently, the resulting network would be able to generate better songs.

In relation to the previous point, one of the problems that we have encountered during this project was the difficulty of finding a non-subjective evaluation metric to test the quality of the generated songs. In the present work, we have solved that issue with a survey, using a test group that included people with different degrees of relation with music.

Finally, time constrain impeded us to introduce several ideas and improvements initially planned. These improvements, and some others that stem from the results obtained, are explained in the following section.

6 Future Work

As mentioned in Section 3, the learning rate was obtained manually by trial and error. In the future, we would like to do a fine search in order to find optimal values for the learning rate η, as well as the number of nodes in each hidden layer (we agreed on having 2 hidden layers). The problem here is that due to time constraints we could not manage to complete the search and that is why we leave it as a work to be done. Initially, Our idea is to set the range of values for the number of nodes in each hidden layer, (128, 256, 512, 1024). Next, we will pick up randomly the size of each of the layers regarding the number of the layers we have (2 layers in the current experiment). In each of these architectures, a grid should be created (for the hyper-parameter η) by sampling 10 numbers from a normal distribution with mean being the best value of η that we found manually and standard deviation being the half of that η. Additionally, we will run the search for a few epochs for a subset of the total dataset (due to memory and time constraints). After this process the best architecture and the best value of η will be chosen with respect to the loss function. In short, we will keep the size of the layers and the η of the network which gave us the lowest loss.

Another aspect to be considered in the future is the length of the seed used to generate a new song. Right now, we work under the assumption that a seed of length 40 states is enough to capture the important information and the context of the song we want to generate. However, it may not be the best length choice. Therefore, seed length and the way the seed is generated is another hyper-parameter that could be optimized in a future implementation of our work.

Also as a future improvement, we will consider modeling transitions between piano pieces in the dataset with "silence states", since not doing it results in abrupt transitions at the end of the songs. These abrupt transitions could undermine the performance or the final network. Finally, evaluation results suggest that including other features such as the duration of the chords/notes in our states could help in the generation of more realistic piano songs. On the other side, key and time signature are features that the network should be able to learn autonomously without being specifically included in the state information. Therefore, further research on the most appropriate features to be considered should be done.

References

[1] Classical piano midi page. `http://www.piano-midi.de/midi_files.htm`.

[2] Survey for piano music generation with lstm. `https://docs.google.com/forms/d/1npdILmn8K3AB7HOulxyXlr_4CbIDwJUfpRFFa6zPNdw/prefill`, 2017.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul

Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Frank Brinkkemper. Analyzing 6 deep learning tools for music generation, October 2016.

[5] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[6] Michael Scott Cuthbert and Christopher Ariza. music21: A toolkit for computer-aided musicology and symbolic music data.

[7] D. Eck and J. Laplme. Learning musical structure directly from sequences of music. *In Artificial Neural Networks—ICANN*, 2008.

[8] Douglas Eck and Jurgen Schmidhuber. Learning the long-term structure of the blues. *In Artificial Neural Networks—ICANN*, page 284–289, 2002.

[9] Allen Huang and Raymond Wu. Deep learning for music. 2016.

[10] K. Cho J. Chung, C. Gulcehre and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *NIPS 2014 Deep Learning and Representation Learning Workshop*, 2014.

[11] Jan Koutnik Bas R. Steunebrink Jurgen Schmidhuber Klaus Greff, Rupesh Kumar Srivastava. Lstm: A search space odyssey. *arXiv:1503.04069v1*, 2015.

[12] Elkan Lipton, Berkowitz. A critical review of recurrent neural networks for sequence learning. *arXiv:1506.00019v4*, 2015.

[13] Michael C. Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science*, 5(6):247–280, 1994.

[14] Vitelli Nayebi. Gruv: Algorithmic music generation using recurrent neural networks. 2016.

[15] Walter Schulze and Brink van der Merwe. Music generation with markov models, 2011.

[16] Geoffrey Hinton with Nitish Srivastava Kevin Swersky. Lecture 6a, overview of mini-batch gradient descent. 2016.