Project Report

PopH

PopH Team: Ivan Anokhin, Egor Nuzhin

Introduction

In the recent year there are attempts to generate music using neural networks. Neural network models are particularly flexible because they can be trained based on the complex patterns in an existing musical dataset. One particularly interesting approach to music composition is training a probabilistic model of polyphonic music. Such an approach attempts to model music as a probability distribution, where individual sequences are assigned probabilities based on how likely they are to occur in a musical piece. Importantly, instead of specifying particular composition rules, we can train such a model based on a large corpus of music, and allow it to discover patterns from that dataset, similarly to someone learning to compose music by studying existing pieces. Once trained, the model can be used to generate new music based on the training dataset by sampling from the resulting probability distribution.

Related works

- DeepJ: Style-Specific Music Generation (2018 - **arXiv:1801.00887**);
- Generating Polyphonic Music Using Tied Parallel Networks (2017 - Daniel D. Johnson);
- Attention Is All You Need (2017 - **arXiv:1706.03762)**

The DeepJ paper develops some ideas about losses and NN architecture on top of the Generating Polyphonic Music paper.

We take initial architecture and approach for the music generation from the first two papers and take ideas about attention mechanism to improve the initial nn architecture from the last one.

Dataset description

We used open source dataset: The Lakh MIDI Dataset

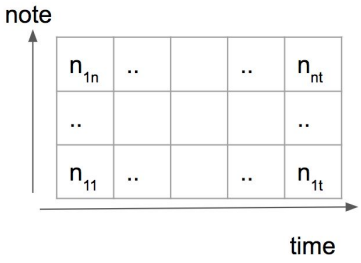
Transformed structure of midi file to use in model is:

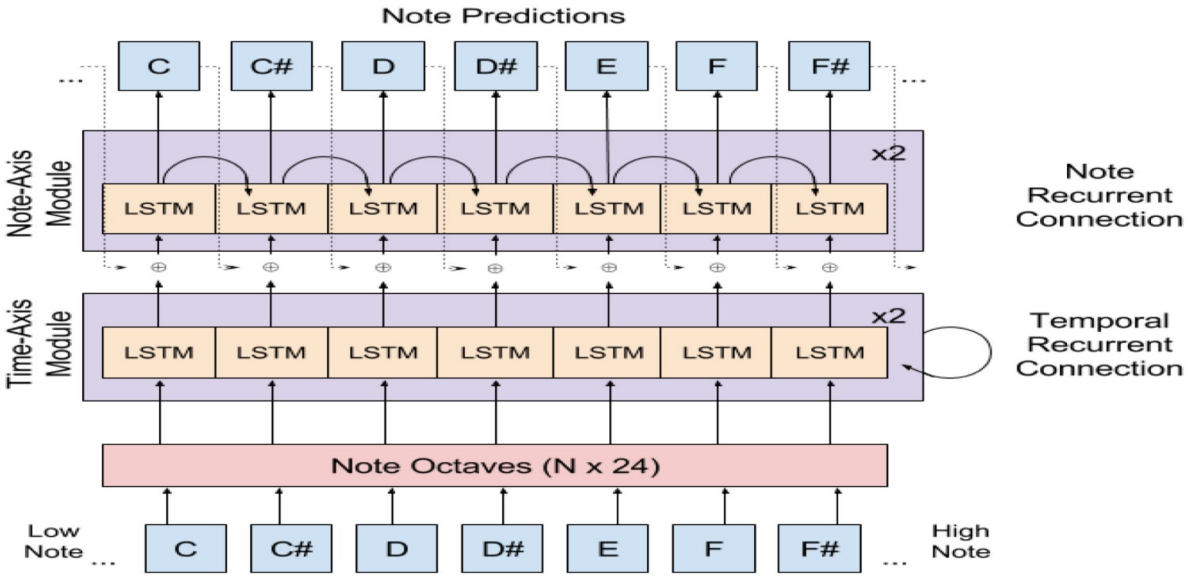
- Play matrix - whether the particular note in particular time is played or not
- Replay matrix - -//- is replayed or not (i.e. key on piano is pressed)
- Volume matrix

ML Methods

Biaxial LSTM Architecture in a nutshell

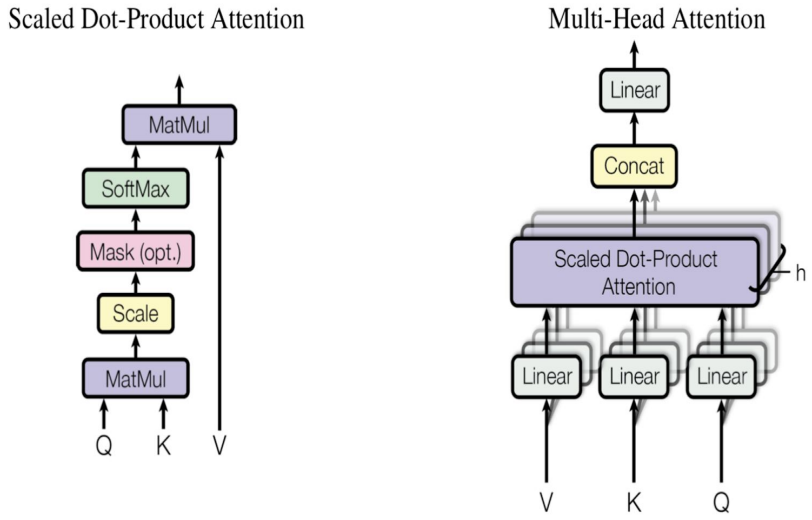
The biaxial lstm model takes as input 3d tensors with dimensions that correspond to time, notes, and notes features (volume, play or not, replay or not).
Note octaves in the picture correspond to feature extraction that increases note feature space. After that the 3d tensor is fed to LSTM along time axis (and for each note we share weights of the LSTM - because of time invariance for each note). LSTM-time-output has the same dimension as input and it can be fed to the note-LSTM (along the note axis - the note-LSTM also share weights for each time steps). After that model make predictions of the notes 3d tensor in the next time step (using dense layers).
It is worth mention that this model trying to predict next note not only using information about notes in previous time step but also about note that are already generated by model in a current time step as we generate note from the highest one to the lowest.
(see more details in the origin papers)





Attention Mechanism

In original paper self-attention and multi-head attention mechanism is used for translation task. The authors described it in next words: The input consists of queries (Q) and keys (K) of dimension dk, and values (V) of dimension dv. We compute the dot products of the query with all keys and apply a softmax function to obtain the weights on the values.



Instead of performing a single attention function the authors found it beneficial to linearly project the queries, keys and values several times with different, learned linear projections. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

where W's are projection matrices.

Self Attention Mechanism

In self attention is the specific case of attention when Q=K=V. In our model we tackle with particular this mechanism.

Experiments/Discussion

NN Architecture

From high level perspective we want to combine efficiently three main blocks: LSTM-time block, LSTM-notes block, self-attention block.
Motivation behind implementation self attention block:

1. it seems that there is lack of information for the lstm-time module because it does not know any information about relative position of other note through it propagation in time.
2. It seems strange to generate notes from the highest to lowest, maybe relative position is all that matters and self-attention can replace LSTM-notes block.

Self-attention block was implemented as multi-head attention block.

Several experiments were done to achieve it permuting and adding and replacing blocks. By Self-attention-note block we mean self-attention along note axis.

Experiments, stacking blocks from left to rights in the model:

1. Self-attention notes block, LSTM-time block, LSTM-notes block
2. LSTM-time block, LSTM-notes block, Self-attention-note block
3. Self-attention notes block, LSTM-time block, LSTM-notes block, Self-attention-note block
4. Self-attention notes block, LSTM-time block, Self-attention-note block

We also have done experiments with implementing self-attention not along particular axis but along the whole timeXnotes matrix (with proper padding). But it was inefficient because of memory error. So we were able to test it only on small input tensors.

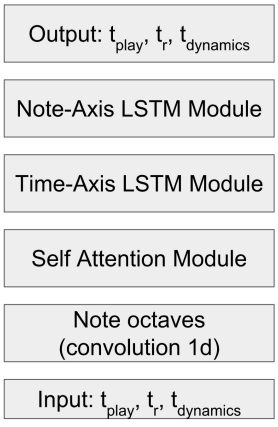
There also were done experiments to choose the best attention model (usage of normalization and residual connection)

Final Architecture

Some models that seems superior in the small amount of data but are not working so good in big amount as it was in our case with “LSTM-time block, LSTM-notes block, Self-attention-note block” model.

The best model that we got in our experiment is “Self-attention notes block, LSTM-time block, LSTM-notes block” model. The model speeds up loss convergens (our model converges to the same loss within 72 epochs instead of 100 in DeepJ model) and seems to produce the same comparable result as the initial model from the paper (in our taste).

Final Model Architecture:



Losses

Initial losses that is used in DeepJ paper:

$$L_{play} = \sum t_{play} \log(y_{play}) + (1 - t_{play}) \log(1 - y_{play})$$

$$L_r = \sum t_{play} (t_r \log(y_r) + (1 - t_r) \log(1 - y_r))$$

$$L_{dynamics} = \sum t_{play} (t_{dynamics} - y_{dynamics})^2$$

$$L = L_{play} + L_r + L_{dynamic}$$

First is cross-entropy of play matrix, second is cross entropy of replay matrix and last is sum of squared errors of volume matrix. Last two errors computes only for positions where note are playing (because otherwise nothing to repeat or change volume)

Experiments with losses:

We tried different types of losses:

- Used various types of harmony matrices.
- Used softmax normalization and non normalized values of notes when we multiplied them on Harmony matrix
- Used through time integration over number of disharmonious notes to find average tonality key in time span.
- For computing losses over the butch we used mean value and mean square value

And over all configurations best quality of generated music was shown by model below with mean value through butch.

Final Losses:

Our loss we contributed by extra component: harmony loss. It based on idea, that all chords in music usually have definite tonality key.

To compute harmony loss we translate note vector y_{play} intro one octave (since harmony rules are invariant for octave translations).

Then we define tonality vectors c_{maj} and c_{min}. Multiplication by these vectors tell as how many notes does not get into first tonality. Then we build circular matrices C_{maj} and C_{min} to find number of notes that got into one of 12 possible tonality types in each classes: major-key and minor-key. After that we define tonality key for chords as tonality with minimal number of misused notes. And our loss is proportional to this of misused notes.

$$L = L_{play} + L_r + L_{dynamic} + L_{harm}$$

$$L_{harm} = \min\left[A_{disharm} SoftMax\left(\tilde{y}_{play}\right)\right]$$

$$\tilde{y}_{play} = y_{play}[:12] + y_{play}[12:24] + y_{play}[24:36] + y_{play}[36:48]$$

$$A_{disharm} = \begin{pmatrix} C_{maj} \\ C_{\min} \end{pmatrix}$$

$$c_{maj} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$c_{\min} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Conclusion

We implement our model with extra harmony loss that produces results as good as the initial model (in our taste) and converges faster. Results comparison of music generated by models with similar time of training and on the same corpus of data can be found in our git repository or in Canvas.

The main comparison of models have been done in two datasets:

- On the one Bach Fugue 16G Minor with many epochs - trying to prove that our models can overfit the data
- On the large corpus of Mozart - to compare the model generalization ability.

Contributions

Anokhin Ivan: implemented different version of attention mechanism for our model, adopted and supplemented code from DeepJ, implemented harmony losses, made experiments comparing models with different architectures, different version of attention module and losses.

Egor Nuzhin: developed idea of harmony losses and its implementation, made experiments comparing models with different variants of harmony losses, experiments comparing DeepJ model and our final model, set up the server in MIcrosoft Azure with three gpu, dealt with many problems that occurs during usage of the server.

Git Repository: https://github.com/avecplezir/PopH