Automatic Music Generation

ToTo Tokaeo
University of British Columbia
tokaeo@cs.ubc.ca

Simon Ningxiao Ren
University of British Columbia
ningxiao@student.ubc.ca

Jay ZheAn Mok
University of British Columbia
jm4304@student.ubc.ca

ABSTRACT

Automatic music generation is a growing area of research driven by advancements in deep neural networks. Though human composers remain the primary creators of music, automatically generated music could be useful in domains that require more simplified, repetitive music, such as in video games or study music. In this paper, we describe deep neural network architectures that have shown promise in automatic music generation. Then, we discuss our experiments implementing these models and apply them to our datasets. Lastly, we demonstrate that our simplified LSTM and transformer models, used on small datasets, can generate music with reasonable musicality as evaluated by certain musicality metrics. The major contributions of our project are the exploration of existing deep learning methods for generating music, simplifying the models for real-time performance, and evaluating the results on our own custom, smaller datasets.

INTRODUCTION

Music plays an important role in everyone's life, whether it is in the context of public ceremonies, personal enrichment, or cultural events. Recently, with the trend of using neural networks and machine learning in enhancing human activities, there have been attempts to use computers to generate, understand and play music. This has several applications, such as creating new musical compositions, preserving and analyzing ancient/traditional/classical types of music, creating interactive musical applications (video games [1], virtual reality, cultural events [2]), and using interactive-music for therapeutic applications [3] and so on.

Our objective thus is twofold: first, to investigate multiple network designs for music generation, and second, to playback the computer-generated music with expressiveness and human-like musicality by using genre-specific knowledge such as instrumentation, key transposition, and instrument range.

The objective will strive to address key challenges in music generation for computers include being able to generate music with interesting melodies and rhythms, correct harmonic choices in a given musical stylistic context, presence of thematic material in a piece especially for longer songs, and harmonious interplay between multiple voices. These issues are addressed in a number of ways and will be discussed in more depth in the related works and methodology section.

Our main contributions include: (1) the creation of a new and smaller dataset for training/evaluation based on Music from the Book of Poetry ShiJing and Bach's Flute Partitas, (2) the exploration of existing deep learning methods for generating music, (3) the simplification of previous neural network models into lightweight models, and (4) the investigation of music generation in real-time with expressive features.

RELATED WORKS

Music inherently itself is a language, therefore many techniques for music generation have adopted natural language processing (NLP) techniques to address this similar field of study. Bei Chen [4] used an RNN network that is able to generate polyphonic, or multi-track music. Dong [5] and Lee [6] both used GAN networks. However, in recent years trends have shifted towards the usage of transformers for generation tasks and language processing.

The transformer architecture was first introduced by Vasvani and researchers at Google in the "Attention Is All You Need" paper [7]. The paper demonstrated that a deep learning network that relies solely on the attention mechanism, and almost nothing else other than normalization and feed-forward networks, can produce state-of-the-art results for machine translation. This is in addition to reduced resource requirements for training. They called their breakthrough architecture the transformer. Since then, the transformer model has been improved on and adapted for more tasks, such as automatic text generation.

One notable example is GPT-2 [1], a language model that can generate long, coherent text. It uses the decoder component out of the decoder-encoder architecture in the original transformer. These decoders can be stacked on top of each other for increased model dimensionality, often to as high as 48 decoders. The paper demonstrated that the transformer decoder is very strongly scalable; with large datasets and higher stacks of decoders, the transformer can produce even more impressive results.

An example of using the transformer to generate music is the Music Transformer [8], as introduced by Huang et al. at Google Brain in the paper "Music Transformer." It adapts the transformer's decoder to music, utilizing their pitch and event-based representations. It also introduces algorithmic optimizations that reduce memory usage. Though the generated samples on the blog are impressive [2], this paper suffers from multiple reproducibility issues. The relevant code is not open-sourced, and many have tried and failed to reproduce the paper's alleged accuracy and quality of generated music [3] [4]. They cite the paper's lack of implementation details and model

[1] https://github.com/openai/gpt-2
[2] https://magenta.tensorflow.org/music-transformer
[3] https://github.com/COMP6248-Reproducability-Challenge/music-transformer-comp6248 and
[4] https://github.com/jason9693/MusicTransformer-pytorch

parameters as possible causes of the issue. We attempt to do something similar, but we also run into issues of our own, as we will describe in the later sections.

METHODOLOGY

In this section, we describe the different kinds of models in more detail that can be used to create music. We first show two architectures, VAE and GAN, which we have tested and have found out that they are improper for our project. Then, we introduce an RNN design with LSTM cells as our baseline. In the end, we present our network design that is based on the transformer architecture.

Variational Autoencoder (VAE)

Like regular VAEs, our VAE implementation is composed of an encoder network and a decoder network as shown in Figure 1. The input of the encoder network, x, is downsampled with 1D convolutional layers to a probability distribution over latent vectors. More specifically, the input is mapped to a mean vector, μ, and a standard deviation vector, σ. The two vectors parametrize a diagonal Gaussian distribution, $N(\mu, \sigma)$. In the decoder network, a latent space vector, z is sampled from the distribution and then upsampled with 1D de-convolutional layers into the original high-dimensional input space, where x comes from.

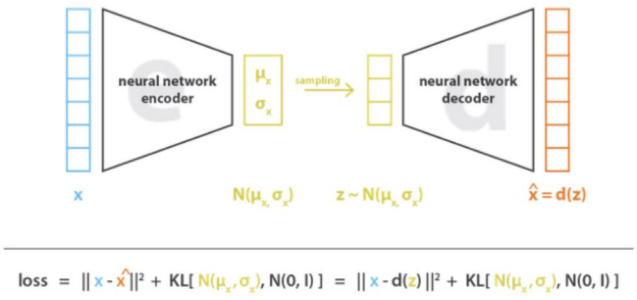


Figure 1. The architecture of VAE
5

The loss function for the VAE consists of two terms, a mean square error and a regularisation term as shown in Figure 1. During the training, the mean square error between the input and the reconstructed output tends to make the encode-decode scheme more efficient and effective, while the regularisation term can regularise the mean vector and standard deviation vector so that the derived distributions from the encoder network remain close to a standard normal distribution.

Generative Adversarial Network (GAN)

Our GAN is designed based on the architecture of LSGAN [9] shown in Figure 2. Similar to VAE, the network consists of two networks, a discriminator and a generator. The discriminator is a classifier that aims to recognize whether an input is from training data or from the generator. Similarly, we use 1D convolutional layers to aggregate features, then pass to a linear layer, followed by a sigmoid function to get a global classification probability. On the other hand, the generator starts from sampling input vector, z, from a Gaussian distribution, then upsample the input vector with 1D de-convolutional layers to a high-dimensional data space where the training data is from.

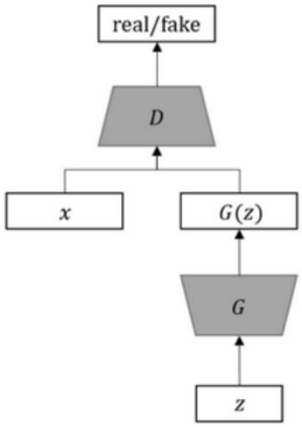


Figure 2. The architecture of LSGAN
[9]

The loss functions for both the discriminator and the generator are defined in Figure 3.

$$\mathcal{L}_D = \frac{1}{2}MSE[D(x),1] + \frac{1}{2}MSE[D(G(z)),0]$$

$$\mathcal{L}_G = \frac{1}{2}MSE[D(G(z),1]$$

Figure 3. Loss functions for LSGAN

Recurrent Neural Networks and LSTM

Recurrent Neural Networks (RNNs) are general-purpose networks that can be used to model relationships between sequential input data and potentially sequential output data. They build upon convolutional neural networks in that they also take into account sequential information, such as the position of the input, time information, and so on. Suppose there is an input sequence of symbols, which can be characters, notes, and so forth. They are X1, X2, ... XN, where N is the length of the symbolic sequence. Given an encoding function, the inputs can be encoded into a hidden sequence called H1, H2, .. HN where they are related by a recursive relationship, H1 = F(H0, X1), H2 = F(H1, X2), and so forth. The hidden outputs can then be decoded into the corresponding output depending on the length of the output sequence. If there is only one output for example, then the model can take any of the H1 to HN and process that node through some neural network such as a convolution network. An illustration is shown in Figure 4.

Because of the general architecture of RNNs, a number of modifications can be made to it to fit the application at hand. For example, the function F can be a simple convolution network, a long-short-term memory module (LSTM), a gated recurrent unit (GRU), and many other types of modules. These modules themselves are various in implementation and can be stacked,

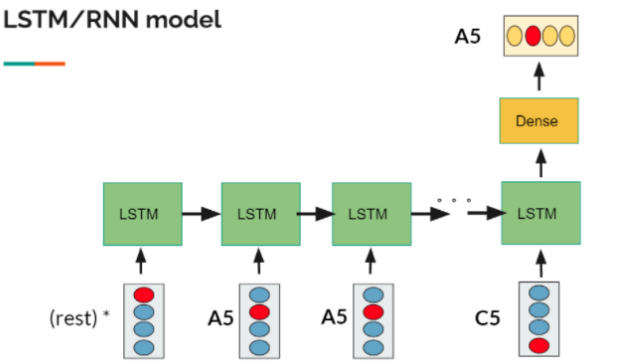


Figure 4. The general architecture of RNN with LSTM modules

bidirectional, and have various depths. In addition, the input data does not have to just be 1-D vectors: they can have a variety of so-called embeddings which can include one-hot, learned-embeddings, positional encodings offset with embedding, and many others. Furthermore, as mentioned earlier, the output does not have to be just one vector and can be multiple if more than one output is needed for the given application.

RNNs can also be interpreted as a type of decoder/encoder structure, that was described in the previous section in regards to VAE and GANs: the input sequence is first encoded into a series of hidden nodes, which can then be "decoded" into one or more outputs.

RNNs strength lies in their lightweights, ease of implementation, the flexibility of solving various types of sequential problems, and relatively low training time. However, RNNs cannot capture long-term information in the input sequence well and are not parallelizable due to the sequential inference of hidden nodes during a forward pass of the network.

Transformer and Attention Models

Due to the inability to capture long-term information, especially evident in music generation applications and translation, transformer architectures were introduced to improve upon RNN architectures. The transformer architecture performs major re-works of the RNN architecture and has three main important features: one, hidden nodes are represented as key, queries, and values inside the transformer architecture; two, connections between each key, query, and value can span the whole input sequence thereby capturing more long-term information; three, cross-attention and self-attention variations can further capture relationships between the input and output training data.

In regards to the first feature of the transformer, this part is captured by the "Attention" model. The attention model, unlike LSTMs which only have a hidden layer, will create more additional intermediate vectors. For example, previously, if H represents a hidden layer and X represents an input, then H1 = F(X0, H0) represents one hidden layer from the input X0 at position 0. In attention models, there are multiple hidden nodes, called keys, values, and queries, and can be represented as K0, V0, Q0 = F(X0). Figure 5 shows how through matrix multiplication the hidden vectors such as the keys, values, queries are generated by an input sequence.

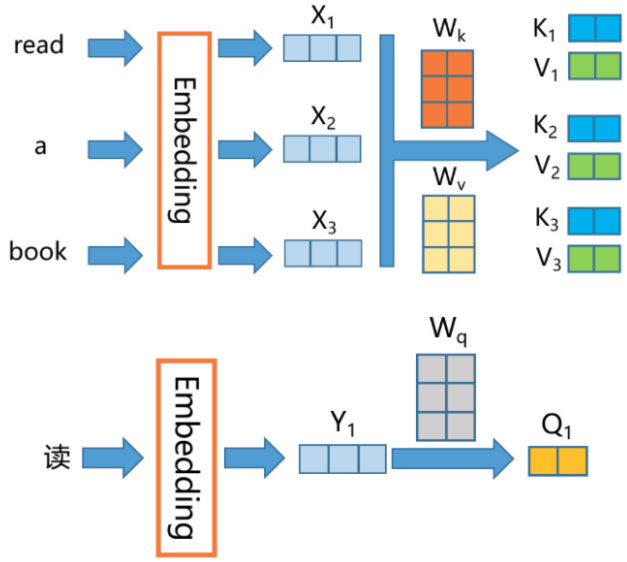
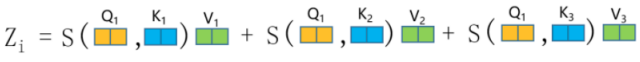


Figure 5. How Attention-model "hidden vectors" such as Key, Query, and Values are generated from an input sequence

These keys, values, and queries are collected for each input. The advantage here is that by having more "hidden layer" vectors, information is kept for each input time frame, which can then be directly used when relating information between relative input time frames. What this means is that the model can be parallelized because the program does not have to wait for the "kth" time frame hidden layer to exist in order to evaluate the next hidden layer as all the hidden layer-like "key", "queries", "values" are evaluated at the same time.

This leads to the second enhancement of the transformer: the parallelizable connections and more long-term relations that can be achieved by using key, query, and value representation rather than a single hidden layer representation. S in Figure 6 represents precisely a function that relates a query from a single time frame with keys from a range of time frames.



$$Z_i = Attention(Query, Source) = \Sigma_{i=1}^{L_x} S(Query, Key_i) Value_i$$

Figure 6. How attention models relate long-term information represented by coalescing queries, keys, and values from different time frames

The third advancement of the transformer is the use of cross and self-attention modules. Because keys, values, and queries can be collected from not only inputs, mixing keys, queries, and values between output and inputs can find relationships between the input and output normally not easily achievable using purely sequential RNNs. Self-attention is the other case

where there is no mixing of the input and output keys, values, and queries.

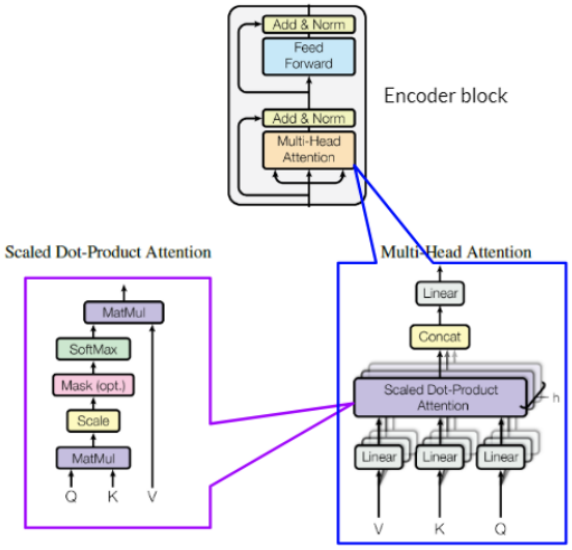


Figure 7. Attention block in the context of an encoder. Inputs are transformed into K, Q, V according to the previous figures, and then can be shaped into an output through matrix multiplication.

Transformer can also be thought of as a decoder/encoder module block: encoder makes the inputs into key, queries, and values which are then manipulated internally either by a fully connected network that can then be sent to a decoder. An illustration is shown in Figure 7.

Transformer in summary tries to improve upon the RNN in a number of ways, including using a more flexible internal hidden layer representation using keys, queries, and values can be parallelized and can get better long-term information relationships between not only inputs themselves but also with the output. However, the transformer is harder to train compared to RNNs and sometimes will fail for simpler kinds of models and training data.

EXPERIMENT

Dataset

There are four common representations of audio: pitch-based, piano-roll, event-based, and note-based. The pitch-based is a 1D array representing constant rhythmic pitches on the x-axis. The piano-roll and note-based methods are 2D arrays, representing rhythm and pitch on the XY axes. Finally, the event-based representation is a sparse 1D array that contains ordered events like noteon, noteoff, setvelocity, and timeshifts to model the changes that occur in audio. Since our models are designed for sequential data, the data representation has to be a 1D array.

Compared with the pitch-based representation, the event-based representation has a much shorter sequence length, which could be beneficial for transformer-based models in terms of words dependency. Despite this, we have experimented with both and found the pitch-based representation yields more promising results. This could be due to the expressiveness of the event-based representation being more than our simplified models can handle. With encodings specific to noteon, noteoff, setvelocity, and timeshifts, the domain of inputs is also not uniform. The number 128 can mean noteoff, while the next number, 129, sets a note velocity, which is completely different semantically. On the other hand, a pitch-based representation already restricts the domain of the inputs and outputs into the space of pitches, so by default, it biases the output to be more musically.

The raw data samples are in MIDI format. We used a variety of datasets from a variety of genres. Monophonic, or single-track, datasets include the music from the Book of Poetry (ShiJing) custom dataset with around 1-2 hours of music, bach clavichord music with around 1 hour of music. Polyphonic, or multi-track, datasets include the Bach Chorale, NES dataset, and the Liu De Hua music collection.

In terms of monophonic music, the piece is segmented into fixed sequences of pitches, and this input vector is then used as the basis for training. In terms of polyphonic music, each voice part is interleaved to generate an input vector, in a similar fashion to the music transformer paper.

Implementations

VAE and GAN

We implement a variety of neural networks and train them on a given music dataset.

For both the Variable AutoEncoder (VAE) and the Generative Adversial Network (GAN) models, the batch size of training data is set to 32. In terms of the optimizer, we use Adam optimizer with a default learning rate for the VAE model. For the GAN, we use Adam optimizer with learning rate and beta1 set to 0.0001 and 0.5 respectively for both the discriminator and the generator.

As mentioned in the Methodology section, both the VAE and the GAN are not suitable for music generation. Music is a sequence of notes with dependencies among each other, like languages. Training VAE on music representations is difficult to converge due to the KL vanishing problem, which typically happens when VAE only learns the local context but ignores global features. Our VAE uses convolutional layers to aggregate features. It is not able to learn long-term dependency, which causes generated samples to be disordered and noisy. On the other hand, the main problem with training the GAN is the loss imbalance between the discriminator and the generator. The discriminator always overpowers the generator and causes convergence failure. This is actually a common problem among many GANs. It can only be fixed by spending lots of time tuning up hyperparameters for the networks. Therefore, we decide to give up the implementations for both the VAE and the GAN.

RNN

We were able to use the RNN/LSTM and transformer models to generate computer music with sufficiently high quality. Figure 8 gives the essential hyperparameters. The LSTM model was implemented with 16 note input sequences, which were

embedded by 128 length encodings first before being sent into the LSTM modules. The hidden layer size was tested with 256 and 512 lengths and results will be shown later.

	LSTM (two sizes)	Transformer	Transformer with positional encoding
Note Sequence Length	16	16	16
Hidden Size	256 / 512	256	256
Embedding Length	128	128	128
Learning Rate, Batch Size	0.01, 512	0.01, 512	0.01, 512
Transformer Specific Parameters	N/A	2 heads, 2 layers	2 heads, 2 layers Positional Encoding
Model Size	~3MB / ~12MB	~7MB	~7MB
Training Time, 300 epoch, (Integrated Graphics GPU)	5 minutes	20 minutes	20 minutes
Inference Speed approximately	~3ms per note	~5ms per note	~5ms per note

Figure 8. Hyperparameters chosen for the LSTM and Transformer models for music generation

Transformer

Transformer models were also used in the music generation process. Instead of using LSTM modules, the input was fed into a transformer self-attention encoder module, and the output decoded straightforwardly via a feed forward neural network.

Positional embedding as mentioned in the original transformer paper was also implemented on the input embedding side.

For both models, the input sequence is fed into the model and a resulting output note is predicted. To generate more notes, the output note is appended to the input sequence thereby creating a recursive structure thereby to generate more notes until a stop token or maximum length is reached for the music generation. To introduce variation, we let the output vector be chosen not based on the absolute highest probability output note, but based on a relative probability. This variation enables the music to generate interesting melodies and surprises even when the same input sequence is passed to the system during generation.

In our system, we can both generate music from scratch by and from a given primer sequence, that is a short theme. To generate music from scratch, we feed the input a series of "pause" tokens, whereas for continuing a song, we feed the input the thematic material.

We can also customize playback by changing the pitch range, instrumentation type, and voices so as to create the best sounding arrangement of a given generated music. This helps with fitting the music generated with its original context (court music for example uses bells).

Finally, the experiments were trained on the Songs from the Book of Poetry (ShiJing) collection and from Bach's flute partita collection. We train the model via PaddlePaddle, PyTorch and use music21, muspy music signal processing toolkits to help with formatting and outputting the musical data to and from midi. We also train our models on a laptop with an on-board RTX GPU.

RESULTS AND EVALUATION

We evaluate the results of the project in two main ways. First, we use model metrics such as training loss to evaluate our deep learning models. Then, we evaluate the musicality of our model-generated samples in comparison to real music using a few musical metrics from the literature.

Though we did consider doing a subjective evaluation of music generation with surveys and user studies, we felt that we could not get adequately meaningful results within the time limit. This is perhaps something we could pursue further in future work.

Model Metrics

The results gathered during training are shown in the Figure 9 and in Figure 10. We can see the highlighted entries show the best accuracy or lowest loss for a given model. It is clear the transformer has the lowest entropy loss compared with the LSTM or the transformer with positional encoding model. It is interesting to observe that adding positional encoding in the manner of the original transformer helps a little bit with the loss but is not as good as the original transformer with no positional encoding. Positional encoding original purpose is to capture positional information of the musical note and use that to help predict future notes. Although sounding good in theory, in practice using absolute positional may actually hurt the results because it is not dynamic and does not align when pieces have changes, alterations that original pieces may not originally have had. The original authors of music transformer also mention the drawback of using absolute positional encoding and hence adopt relative positional encoding, which we may attempt in the future.

	LSTM (two versions)	Transformer	Transformer with positional encoding
Binary Cross Entropy Final Loss	0.271 / 0.261	0.245	0.251
Mean Accuracy	0.48 / 0.48	0.47	0.47

Figure 9. Training and Validation results for the different models used and their performance on different model metrics, including mean accuracy and entropy loss. We see that the transformer model has similar accuracy to the LSTM model but has lower cross entropy loss.

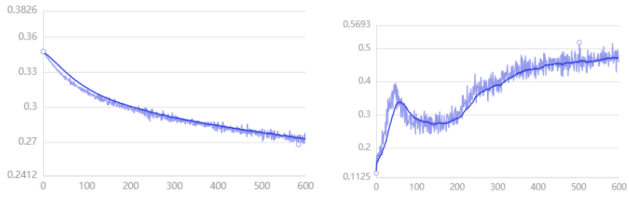


Figure 10. (left) is the loss for using the transformer model (right) is the accuracy of using the transformer model on the Book of Poetry dataset

Musicality Metrics

Although the metrics discussed above evaluate our models, they are done independently from any music domain knowledge. Using metrics that measure musicality may capture

more of the artistic variation in music. Still, objective musical evaluation is far from a solved problem; unlike the model evaluation previously discussed, there is no widely-accepted gold standard among musical evaluation. Different music generation papers tend to use many different metrics, often having to invent their own. Here, we use 5 relevant metrics, namely pitch entropy, pitch-class entropy, pitch-class histogram, pitch-class transition matrix, and scale consistency. For each one, we compare the metrics when applied to our generated sample to real music, as well as two controls: one audio generated with random notes, and one audio generated with only 1 repeating note.

We have compiled the numerical metrics into a table in Figure 11.

	Random	Same Note	Real Sample	Transformer Generated Sample
Pitch Entropy	4.9896696896	0.0	4.2362	4.043739
Pitch Class Entropy	3.55805879479	0.0	3.35164	3.221648
Scale Consistency	0.62518	1.0	0.8111	0.8814

Figure 11. Metrics for control group, generated sample, and real

Pitch entropy

Pitch entropy measures the Shannon entropy on the normalized note pitch histogram. The calculations were done using MusPy [6]. In musical pieces, certain pitches should be more frequent as part of the tonality, so the entropy should not be too high. A high entropy would mean high randomness, and a very low entropy would be no randomness at all, which is also not ideal. This metric is included in the displayed table in Figure 11.

As expected, the control example with random notes exhibits the highest pitch entropy, and the example with one note has the lowest. The pitch entropy for real music and generated music are of similar value, with generated music being somewhat lower. This may mean the generated music is slightly more predictable and repetitive.

Pitch class entropy

The same as pitch entropy, except instead of individual pitches, we have pitch classes instead, such as (C, C sharp, D, and so on). This may be more meaningful in the musical sense since different pitches are not always so distinct, i.e they may be in the same class. Again, the calculations are done in Muspy. This metric is included in the displayed table in Figure 11.

Similar to pitch entropy, the control example with random notes exhibits the highest pitch class entropy, and the example with one 1 note has the lowest. The pitch class entropy for real music and generated music are of similar value, again with generated music slightly lower. It appears that in our testing, the pitch class entropy and pitch entropy metrics seem to mirror each other.

[6]https://salu133445.github.io/muspy/metrics.html

Scale consistency

The scale consistency describes the percentage of pitches being in scale. For a certain scale, the pitches in a piece may be in-scale or out-scale. Then, in each scale, we can get a percentage of notes being in scale. Repeating the calculations for all scales and taking the max would give this evaluation metric. The calculations are done in Muspy and included in the displayed table in Figure 11.

The control audio with 1 note has a perfect scale consistency, which is to be expected, and the random audio has a low consistency of 0.62. The generated sample has a slightly higher consistency than the real sample, perhaps due to its repetitiveness.

Pitch Class Histogram

This is the histogram of pitch classes in a musical piece, on which the entropy statistics were calculated as mentioned earlier. This histogram may reveal the tonality of the music, showing which note classes the music focuses on. From our produced histograms in Figure 12, we observe that the peaks in the generated music's histogram closely resemble the peaks on the real music's histogram. This demonstrates that our model is capable of learning the pitch class patterns of the input music, and the output would mirror these patterns.

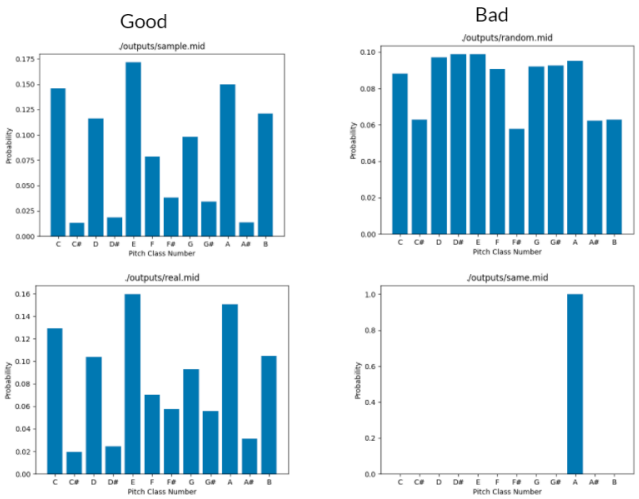


Figure 12. (Top Left) Generated audio sample from scratch (Top right) Random signal sample for comparison (Bottom Left) Real audio music from the training and evaluation Flute Partita dataset (Bottom right) Single note sample for comparison

Pitch class transition matrix

So far, all our metrics have been time and order independent, like a bag of pitches. However, it may also be useful to describe the transitions of pitches in a musical piece. One such way is the pitch class transition matrix, which shows the normalized frequency of pitch class transitions. These are calculated using the library prettymidi [7] and displayed in Figure 13.

[7]https://craffel.github.io/pretty-midi/

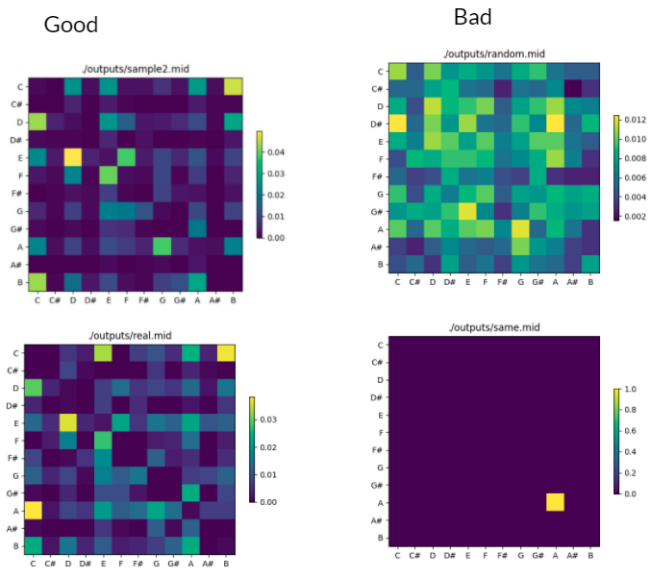


Figure 13. (Top Left) Generated audio sample from scratch (Top right) Random signal sample for comparison (Bottom Left) Real audio music from the training and evaluation Flute Partita dataset (Bottom right) Single note sample for comparison

The matrices for our control groups are as expected. One is almost completely random (for the random audio), while the other does only one kind of transition. The patterns for the real and generated music are more interesting. For one, there are certain columns and rows that are almost always 0, meaning that the transitions rarely occur from, or to, certain note classes. The diagonals are also completely empty, because notes of the same class are held rather than repeated. There are also some transitions that appear much more frequently, namely E to D, C to B, and B to C. These patterns appear in the matrices of both real and generated data, demonstrating that our model can also learn interesting transition patterns.

Lastly, one can also make interesting observations by looking at certain musical patterns in the piece, as demonstrated in Figure 14.

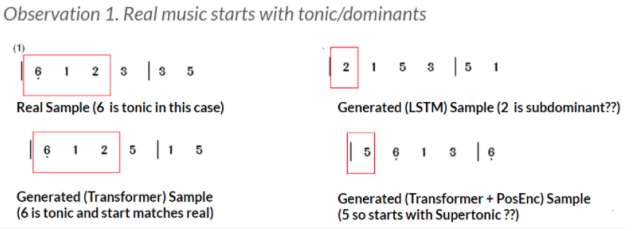


Figure 14. Different Models and the corresponding generated music cadence structures. Notice for real samples, the music begins with a "6 1 2" sequence, which can be emulated by the transformer models, but not so much by the LSTM and transformer with pure positional encoding models.

CONCLUSION

In this project, we have tested different network designs for music generation. Our implementations for LSTM and transformer have shown promising performance in music generation. At the same time, we also see the limitation of our implementations. Some improvements can be made in future work for a better music generation.

Limitation

The baseline LSTM model can generate plausible music but only at time scales of a few seconds. The model seems to forget the primer quickly and is unable to generate coherent continuations based on the primer. It is clear that the model can not capture a long-term dependency in the sequence, which is actually a common disadvantage among LSTM networks.

In contrast, our transformer model can reuse the primer and retain a better long-term structure. The generated music maintains some degree of consistency, which makes the music more plausible. However, due to the simplicity of our design and the lack of better positional encoding design, our transformer is not able to retain long-term dependency and consistency throughout the whole generated music. The generated music often deteriorates gradually as more notes are generated.

Future work

Our transformer network design can be improved in two aspects. First, we can increase the number of transformer encoder blocks to learn more complex music. Due to the computation limits, our model design is relatively small and can only learn from monophonic music. With more computing resources, we can scale up our model and use it to learn complex music that is played with multiple instruments. Second, a better implementation of positional encoding can be constructed for our transformer model. Currently, our model could not keep the generated music consistent in rhythms and styles throughout the whole music generation process. We can investigate more on better positional encoding methods that can allow our model to capture a better long-term dependency.

REFERENCES

[1] Dae Yeol Kim Soo Young Cho Chan Hyeong Park and Chae Bong Sohn. Action Game with Automatic Background Music Generation Using Genetic Algorithm. *Journal of The Korean Society for Computer Game*, 29:99–106, 2016.

[2] Joel Chadabe. Interactive music composition and performance system. *The Journal of the Acoustical Society of America*, 78(6):2159–2159, 1985.

[3] Jasti Vasudha, S. Iniya, G. Iyshwarya, and G. Jeyakumar. Computer Aided Music Generation Using Genetic Algorithm and Its Potential Applications, 2011.

[4] Wang Lei Miao Beichen, Guo Wei'an. A polyphony music generation system based on latent features and a recurrent neural network. *Journal of Intelligent Systems*, 14:99–106, 2019.

[5] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment, 2017.

[6] Sang gil Lee, Uiwon Hwang, Seonwoo Min, and Sungroh Yoon. Polyphonic music generation with sequence generative adversarial networks, 2018.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[8] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer, 2018.

[9] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2794–2802, 2017.