Generating Classical Music with Transformers

Ethan Feng, Mariel Werner

Abstract

Models based on the Transformer Architecture (Vaswani et. al. 2017) have been a staple in the landscape of autoregressive generation for nearly a decade as of the publishing of this paper. While such models have proven to be proficient in areas such as Text Generation in GPTs (Generative Pre-Trained Transformers) and NLP (Natural Language Processing), their potential for autoregressive generation is yet unproven in other similar areas, especially in the field of Music Generation. With the goal of exploring the potential of the transformer architecture in more novel areas, we propose a small, 57M-parameter model built on a standard Transformer Architecture and trained on classical piano music in the MIDI format. We train on a dataset that primarily contains music from the Classical and Romantic periods, with music from the more modern periods also included, albeit at a smaller scale. To help our model better understand the nuances of classical music that are absent in conventional text applications, we introduce a novel Musical Loss Function that will work in tandem with standard loss functions, specifically Cross-Entropy Loss, to ensure coherent and melodious generation. Through our model, we aim to prove that ~50M-parameter models trained on smaller, meticulously selected datasets can compete with larger music generation models trained on larger datasets in terms of performance.

Introduction

For much of its history since its inception, the Transformer Architecture has mainly been used in applications that involve text data. Although areas such as NLP (Natural Language Processing) and Text Generation have been the chief applications of the Transformer Architecture, we believe that there are further, untapped potentials of the Transformer Architecture, specifically in the area of Music Generation. We recognize that generating music through sequences of notes is strikingly similar to generating text through sequences of letters; we feel that expanding the potential of this Architecture to the field of music should be the next logical step. In this paper, we examine the plausibility of applying a standard, proven transformer architecture to an autoregressive music generation. We train on our self-compiled custom dataset of piano works, containing over 200 classical piano pieces. With the goal of building a strong baseline of coherent, melodic music, we also introduce a novel loss function that penalizes the generation of factors such as uneven rhythms, unstable note densities, and long sequences of rests. Although there exist other models that incorporate large training datasets and complex architectures that have been proven to generate quality classical music to a considerable scale, we aim to prove that a smaller model of only 57M parameters can effectively compete with such large models. We find that our architecture, despite its small parameter count and training dataset, performs proficiently in autoregressive generation.

Literature Review

Our primary inspiration for utilizing the transformer architecture for the generation of classical music was the paper *Generating Piano Music with Transformers: A Comprehensive Study of*

Scale, Data and Metrics (Lehmkhul et. al. 2025), which demonstrated the potential for the generation of classical music using the transformer architecture, which we aimed to reproduce at scale. Music Transformers (Huang et. al. 2018), proved the potential of training transformers on MIDI, which we adapted for our paper. We owe our model architecture largely to *Attention is All You Need* (Vaswani et. al. 2017), which outlines the transformer architecture used in many AI models today, specifically in the field of Natural Language Processing (NLP) and in GPTs (Generative Pre-Trained Transformers). Additionally, we use Rotary Positional Embeddings (RoPE) derived from *RoFormer: Enhanced Transformer with Rotary Position Embedding* (Su et. al. 2021). Several inspirations were involved with the creation of our novel Musical Loss. With respect to our loss function, *This Time with Feeling: Learning Expressive Musical Performance* (Oore et al., 2018) and *Compound Word Transformer: Learning to Compose Full-Song Music over Dynamic Directed Hypergraphs* (Hsiao et al., 2021) were our most significant inspirations, with their introduction of custom losses directed towards theory and structure to guide the model towards cohesive output.

Results

We train our model for 25 epochs on the training set and validate it on a validation set, using both Cross Entropy and our novel Continuity Loss. During training, we used a training patience system that discontinues training if loss has not improved over the course of 5 epochs. This guards against overfitting. We found through previous iterations and loss curves that the model performed best at 25 epochs before beginning to overfit. At the end of the 25-epoch training run, the total training loss was 1.0775 and the validation loss was 1.5184. During testing, the model achieved a total loss of 1.3705. Our generation loop generates tokens autoregressively, token by token. First, the prompt sequence and previously generated music, if applicable, are given as input to the model. Then an additional music Continuity Penalty is applied, which serves as a final safeguard against incoherent generation. This penalty works similarly to the Continuity Loss component of the Musical Loss, with i and j being consecutive pitch tokens. Penalty P_g is calculated as follows:

$$P_g = (\max(|pitch(i) - pitch(j)|, 12) - 12) * 0.5.$$

This penalty function decreases logit values for tokens that are likely to create large melodic jumps. Next, Top p (Nucleus) sampling is applied. In particular, logits are first sorted in descending order, where their cumulative probability is calculated. All tokens below a 92% confidence threshold are then removed by setting their logits to -inf. Finally, the remaining logits are converted to probabilities via softmax, and the next token to be appended to the existing music is chosen stochastically (based on probabilities but with some randomness involved). The model ceases generation when either the maximum number of tokens is reached or an EOS (End of Sequence) token is generated. Loss vs. Epoch Graphs and Example Generations are shown in Fig.1 and Fig.2, respectively.

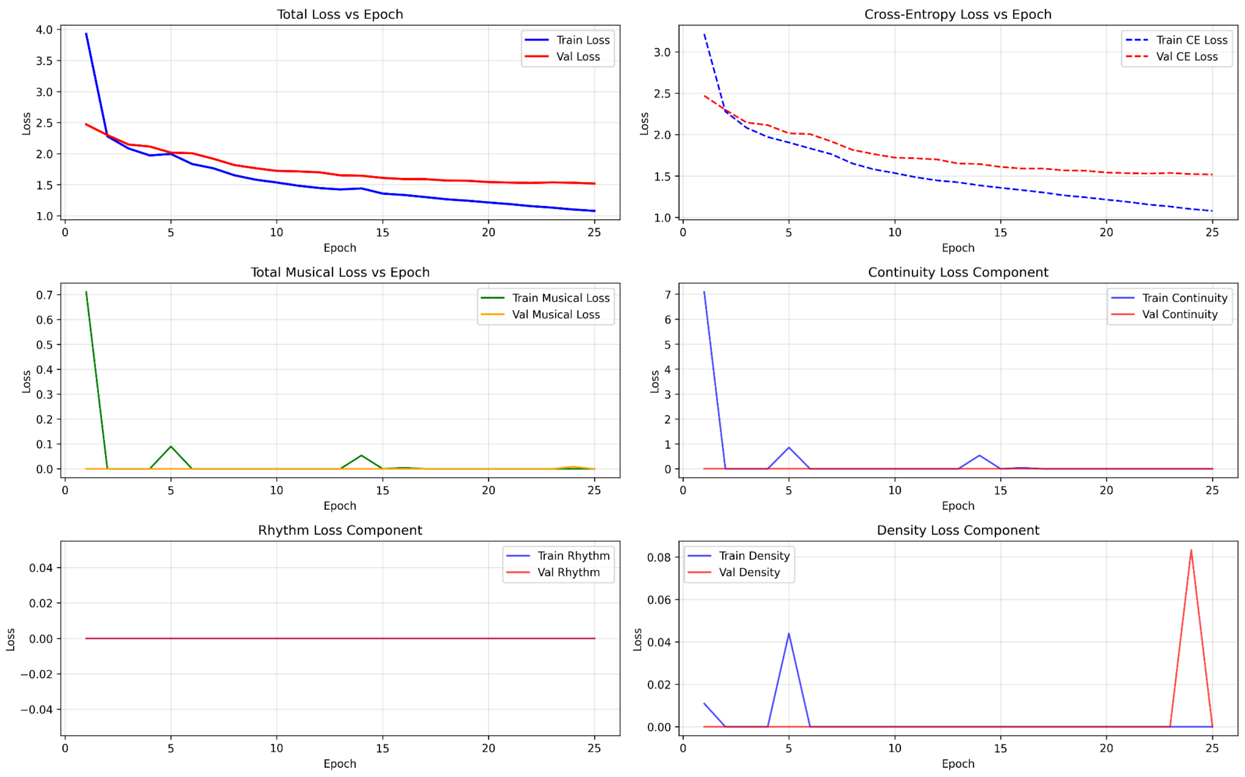

Fig.1 Loss vs. Epoch Graphs

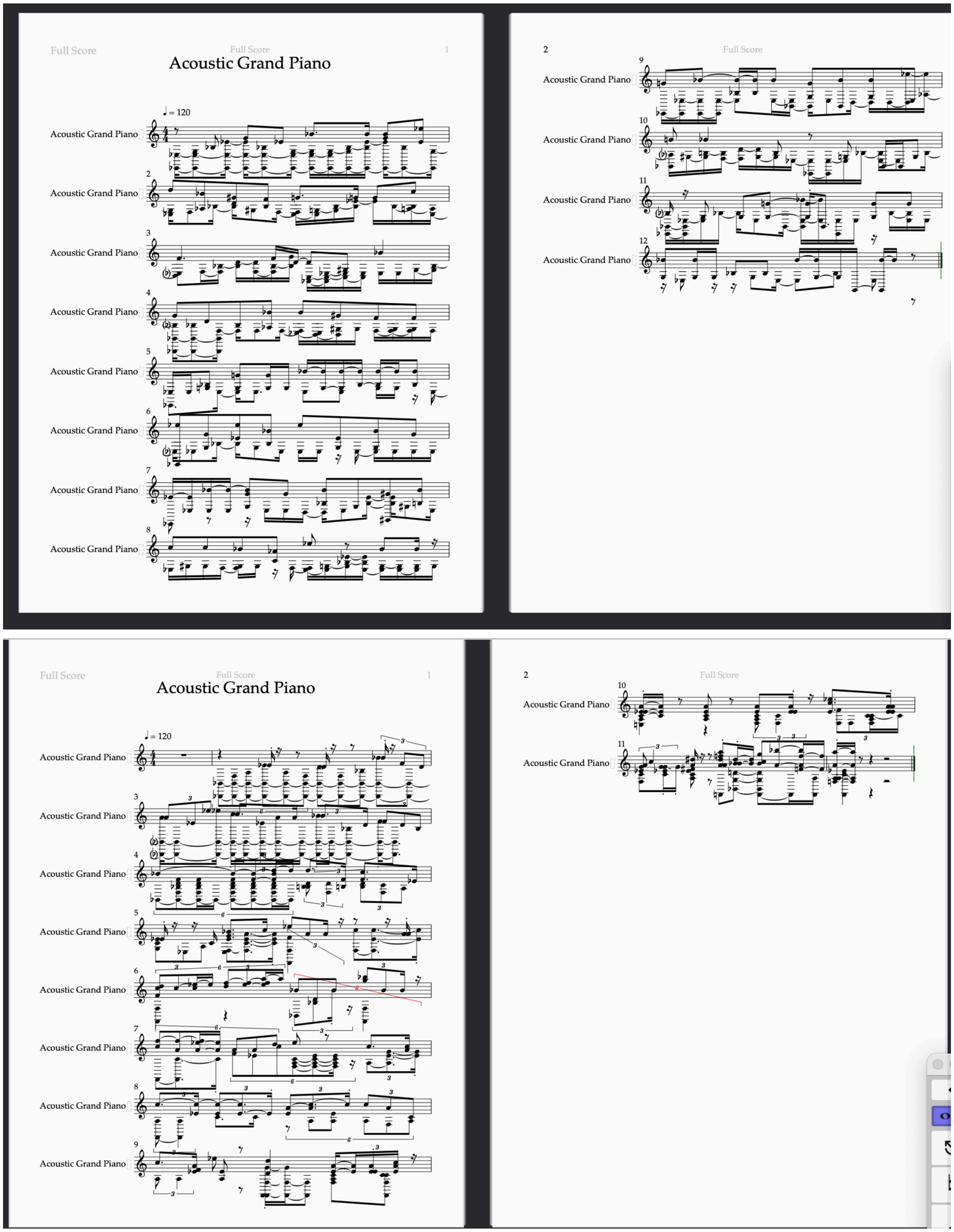

Fig.2: Example Generations. Prompt lasts for roughly 3 measures

Discussion

Our transformer-based music generation model demonstrated its capabilities effectively throughout training. Cross-Entropy Loss converged to 1.0775, while the final total musical loss was 0, indicating both effective token generation and proficient adherence to musical concepts essential to composition, such as rhythm, melody, and harmony. The extremely small values for each component of our Musical Loss (Continuity, Density, and Rhythm) for most of the training demonstrate that both the model's architecture and loss were integral to producing quality music. The Transformer Architecture has historically proven itself to be the state-of-the-art in other forms of autoregressive generation, shown by the recent rise of GPTs. Therefore, it should be no surprise that a proficient, coherent generation of music has been achieved through an architecture applied to similar tasks. Through its generated works, it is evident that the model has effectively learned musical patterns, motifs, melodies, and harmonies. Through its autoregressive generation, it demonstrates its understanding of broader classical music, as well as the style, motifs, and structure of its prompt tokens. The model also demonstrates some capability to extend and build on its prompt, with clear attempts to modify and develop melodies and harmonies. However, we observe that overaggressive penalties lead to repetitive melodies, mostly revolving around a single note, thus making the music grow repetitive and stale over time. We also observed that although the model does learn and produce coherent melodic and harmonic structure, it undergoes a kind of modal collapse as generation continues. This involves diminishing variation in note values in generated music as the piece goes on, with the melody diminishing and eventually collapsing into a single note value in extreme cases. Our attempts to mitigate this issue have so far been unsuccessful and we defer its resolution to future research.

Methods

Data Acquisition

We source our data from classical music databases online, primarily from The Classical Archives. We split our data into training, validation, and test sets at 80-10-10, with the chunks within each set randomized. Our data is in MIDI (Musical Instrument Digital Interface) format, which is the primary method of digital composition storage. MIDI functions by storing individual note values as timed events, specifying temporal occurrence, position, pitch, duration, volume, and articulation. We find MIDI as the ideal data due to its compatibility with packages that our model would utilize, and the ease and certainty with which it would allow us to use our model to generate music that could not be replicated by other file formats, such as .mp3 and .wav. Since MIDI files store music as a sequence of distinct note events, rather than in the form of sound waves, we find that using MIDI files to both train and generate allowed the model to produce structured and coherent musical content, which has been significantly harder to achieve using file formats such as .mp3 and .wav.

Our audio dataset, on which our model is trained, contains piano pieces by prolific composers ranging from Mozart to Rachmaninoff, with a majority of the dataset consisting of music from the Classical and Romantic periods of classical music history. Because music from these two periods abounds with lyrical and tonal depth as well as transparent theory, form, and structure, we felt it would provide a strong baseline for generation. The focus on piano music allows the model to generate music for a single instrument and blocks out training noise that could arise from the complexity of multi-instrument orchestral or chamber works.

We chose to split our training samples into 2048-token long chunks for many reasons. Not only does it adhere to GPU memory constraints in training, but it also allows us to glean more training examples from our existing data. Our fixed chunking size allows for efficient batching without wasting computation. Chunking also allows our model to focus on the structure of musical sentences and phrases, rather than the piece as a whole. This enables the model to learn intricate musical patterns, theory, and structure that would be lost if training took place over the course of unchunked pieces. Finally, we utilize a phrase augmentation function in our data structure aimed towards creating longer and more coherent musical sequences in training. For every 2048-token sequence in the training data, we randomly select a position within it. From that position, a 512-token context phase and a 512-token continuation phase. The context and continuation phases are combined to form a 1024-token training example. This method is applied randomly to 50% of batches. While the rest of the data is designed to help the model learn autoregressive, note-by-note generation, our Context-Continuation generation structure is intended to help the model learn musical phrases and ideas, which elevates both the coherence and quality of the generated music. Overall, the combination of chunking, music choice, Context-Continuation Data Sampling, and the MIDI format on which the model is trained plays a large role in ensuring coherent and musical generation.

Architecture

We utilize a decoder-only transformer architecture for our model. It takes in a MIDI file as input and generates tokens autoregressively. All MIDI data used for both training and autoregressive generation is tokenized through the PyTorch MIDITok package, using the REMI tokenizer with use_chords=False and num_velocities=16. The use_chords hyperparameter allows notes that occur simultaneously to be grouped as chords. We find that disabling this feature allows the model to generate each note independently, allowing for smoother voice leading and higher quality melodies. The num_velocities hyperparameter controls the granularity of note velocity encoding. We find that our model can provide the most coherent, melodious generation when being provided with 16 independent velocity levels. Our model takes in MIDI tokens as input and converts them into token embeddings. The embeddings then pass through the 8 transformer layers, the active component of the model. Every transformer layer begins with a Layer Normalization layer for pre-normalization. The embeddings are then passed through a 12-head Multi-Head Self-Attention layer with causal masking to ensure autoregressive generation. Within the attention layer, Rotary Positional Embeddings (RoPE) are applied. After passing through the Multi-Head Attention Layer and a residual connection, the embeddings then pass through a second Layer Normalization layer before being passed through a Feed-Forward Network consisting of Linear Expansion (768 -> 3072), GELU activation, and Linear Compression (3072 -> 768) layers. The embeddings are then passed through a final residual connection and used as input for the next transformer layer. After the embeddings are passed through the final transformer layer, they are fed into a final Layer Normalization layer and a linear output layer, which map the embeddings back to vocabulary-sized logits representing probabilities for each possible MIDI token. Model hyperparameters are listed in Table 1.

Model

Hyperparameter	Value	Description
d_model	768	Transformer embedding/hidden dimension
n_heads	12	Number of attention heads
n_layers	8	Number of transformer blocks
max_seq_len	2048	Maximum sequence length
dropout	0.1	Dropout rate throughout model
head_dim	64	Dimension per attention head (d_model / n_heads)
ff_expansion	4	Feedforward network expansion factor

Tokenizer Configuration

Hyperparameter	Value	Description
num_velocities	16	Number of velocity bins
use_chords	False	Whether to use chord tokens
use_programs	True	Whether to use program/instrument tokens

Training Configuration

Hyperparameter	Value	Description
batch_size	2	Training batch size
num_epochs	25	Maximum number of training epochs
learning_rate	1e-4	Initial learning rate for AdamW

weight_decay	0.01	L2 regularization weight
beta1	0.9	AdamW beta1 parameter
beta2	0.95	AdamW beta2 parameter
gradient_clip	1.0	Gradient clipping max norm
patience_limit	20	Early stopping patience (epochs)

Data Split

Split	Proportion	Description
train	80%	Training data
validation	10%	Validation data
test	10%	Test data
random_state	42	Random seed for reproducibility

Musical Loss Function

Hyperparameter	Value	Description
lambda_continuity	0.1	Weight for pitch continuity penalty
lambda_rhythm	0.1	Weight for rhythm regularity penalty
lambda_density	0.1	Weight for note density consistency
interval_threshold	12	Max semitones before continuity penalty
density_window	32	Window size for density analysis
density_threshold	5	Max density change before penalty

Data Augmentation

Hyperparameter	Value	Description
use_phrase_augmentation	True	Whether to use phrase continuation
phrase_length	512	Length of phrase segments (tokens)
augmentation_probability	0.5	Probability of applying augmentation

Generation/Inference

Hyperparameter	Value	Description
max_length	1000	Maximum tokens to generate
temperature	0.75	Sampling temperature
top_p	0.92	Nucleus sampling threshold
top_k	50	Top-k sampling (not actively used)
repetition_penalty	1.15	Penalty for repeating tokens
repetition_window	128	Window for repetition penalty
continuity_penalty_scale	0.5	Scale for inference continuity penalty
prompt_length	256	Length of generation prompt

Positional Encoding (RoPE)

Hyperparameter	Value	Description
base_frequency	10000	Base for inverse frequency calculation
max_seq_len	2048	Maximum sequence length for precomputation

Weight Initialization

Parameter Type	Initialization	Description
Linear weights	Normal(0, 0.02)	Gaussian initialization
Linear bias	Zeros	Zero initialization
Embedding weights	Normal(0, 0.02)	Gaussian initialization

Loss Function

Component	Type	Description
Primary loss	CrossEntropyLoss	Standard language modeling loss
Ignore index	pad_token_id	Ignore padding tokens in loss
Musical penalties	Custom	Continuity, rhythm, density penalties

Table 1: Hyperparameters

Loss Functions

We incorporate two types of loss: Cross-Entropy Loss, for general model accuracy and Musical Loss, which we introduce in this paper. We design the Musical Loss to reward the model for generating aesthetic, melodic, and smooth music. We define Musical Loss as a linear combination of three component losses, each prioritizing an important musical characteristic: Melodic Continuity, Rhythm Regularity, and Density Consistency.

Melodic Continuity

To compute loss for Melodic Continuity, we first use argmax to compute the most likely pair of tokens at timesteps t and $t + 1$. If both tokens correspond to pitch values that have an interval > 12 semitones, or one octave, we compute a quadratic penalty defined as

$$P = (|pitch(i) - pitch(j)| - 12)^2$$

where i and j are the pitch values of the consecutive MIDI tokens.

Continuity Loss

The total Continuity Loss is computed as $L_{continuity} = (\frac{1}{N}) \sum penalty$, with N as the number of pitch token pairs that exceed an octave. We then average all penalties of all qualifying note token pairs in the batch. The mathematical implementation is as follows:

$$L_{continuity} = \frac{1}{N} \sum_{b=1}^{B} \sum_{t=1}^{T} P(pitch_{b,t},\ pitch_{b,t+1})$$

Since continuous musical intervals greater than an octave can often make the generated music erratic and jarring, the model learns to avoid unreasonably large intervals through the continuity component of our musical loss. Therefore, this component provides a smoother melodic contour in generated music, which allows for an even, pleasant listening experience.

To compute loss for Rhythm Regularity, we first collect all duration values in each sequence of the batch. The variance of all durations for sequence b with durations $D_b = \{d_1,\ d_2,\ \dots,\ d_n\}$is as follows:

$$V_b = \frac{1}{n} \sum_i (d_i - mean(D_b))^2$$

where n is the number of duration tokens in sequence b. The variance is then applied to the Rhythm Loss, defined as follows:

$$L_{rhythm} = \frac{1}{B} \sum_b (0.5 \times V_b)$$

where B is defined as the number of sequences in batch B. The purpose of the Rhythmic Component of our Musical Loss is to foster the generation of smooth rhythmic contour. A piece that erratically incorporates various notes of radically different duration values will lead to the generated music being incoherent and hard to follow from the audience's perspective. By penalizing extreme variation in note value, we can allow the model to generate cohesive rhythmic structures, which are critical for a solid generation baseline.

The final component of our musical loss is the component pertaining to Musical Density. The density loss is implemented to prevent abrupt changes in chordal texture by retaining stable note density over time. We first define the penalty P as:

$$P = (|\rho_{w+1} - \rho_w| - 5)^2 \text{ if } |\rho_{w+1} - \rho_w| > 5, \text{ else } 0$$

Where ρ_w is the count of pitch tokens in pitch token w, and ρ_{w+1} is the count of pitch tokens in the token directly after w. We can now define Density L:

$$L = \frac{1}{M}\sum_{b}\sum_{w} P(\rho_{w}, \rho_{w+1})$$

for sequence b and w windows. While multiple layers and voices can be beneficial to the musical contour of a piece, erratic changes in musical density, such as sudden chords, can make the music seem incoherent, especially in the harmonic qualities of the generated pieces. By penalizing dynamically changing musical densities, such as the sudden and sporadic increases or decreases in melodic or harmonic layers, or the unprovoked addition of complex chords, the resulting music is more coherent and melodic.

The total loss is computed as the sum of components of the Musical Loss multiplied fixed weights λ_c, λ_r, and λ_d, corresponding to Continuity, Rhythm, and Density, respectively. The total Musical Loss is then added to the Cross Entropy Loss, L_ce. Thus, the total loss is computed as

$$L_{total} = L_{ce} + \lambda_c \times L_c + \lambda_r \times L_r + \lambda_d \times L_d.$$

Conclusion

We find that a standard, 57M-parameter transformer model trained on MIDI files of classical piano music is capable of generating coherent and melodic music when given a substantial prompt of existing music. To combat hurdles in music generation that do not appear in other uses of transformers, we implement a novel Musical Loss to ensure both coherence and melodiousness in generated music, while staying loyal to the prompt. There are many potential ethical real-world considerations for the model that we propose. We firmly believe that such technologies, like the one we propose in this paper, must be used carefully in order to avoid squandering human creativity, and should be used to assist humans, and not to replace them. Therefore, one potential ethical use of our model includes areas in music education. We envision our model being used in Music History and Theory lessons to help students understand fundamental theory concepts and the evolution of classical music. We also see the potential of our model as a teaching aid and practice tool in instrument lessons, creating a wider variety of repertoire for students that extends upon the works of great classical composers. There are many extensions to our model which we hope to see implemented in the future. By training our model on a larger dataset with a larger variety of composers from more diverse time periods, we anticipate that our model can be used for composer-conditioned generation on zero prompt tokens, as well as for other multimodal applications.

Acknowledgements

The author would like to thank Mariel Werner for her support and guidance for this project.

APPENDIX

This appendix includes an overview of deep learning concepts that are the technical foundation of this paper.

REMI Tokenizer

The REMI (REvamped MIDI) Tokenizer tokenizes MIDI data into various main token types. Each note in the music is represented in the form of several tokens. Bar tokens mark the beginnings of measures in the music. They are represented as Bar_1, Bar_2, etc. Position tokens represent the positioning of the note in the piece, usually in 16th note divisions, such as Position_0, Position_4, etc. Pitch tokens represent the note value of the note represented in MIDI. For example, the pitch value of Middle C, or C4 on the piano, is represented as Pitch_60. Velocity tokens represent the volume that a note should be played, an example of which is Velocity_80. Duration tokens represent the length of time which a note should be played. One example of such a token, representing an 8th note, is Duration_8. So, a note represented in tokenized format as Bar_0 Position_0 Pitch_60 Velocity_80 Duration_16 would be a 16th note Middle C (C4) played at mf (mezzoforte) volume, which is played at the beginning of bar 1.

Cross Entropy Loss

Cross Entropy Loss measures the difference between predicted probability distributions and ground truth probability distributions. It can be used in a wide variety of AI applications, such as classification, where the model's output consists of class probabilities. It is also especially useful in autoregressive generation, where the model attempts to predict the best next possible token by outputting probabilities for all possible tokens, where the most probable token is then appended to the existing sequence. The equation for cross-entropy loss is

$$L = -\sum_{i=1}^{C} y_i ln(p_i),$$

where y_i is the ground truth probability, with $y_i = 1$ if the predicted class is correct, else 0, and p_i is the predicted probability.

Transformer Model Architecture

Token Embeddings

Token Embeddings convert individual tokens into vector representations. They allow the model to learn representations of data such as text, and in the case of our model, MIDI. Their primary purpose in training is to help the model learn the semantic representation of individual tokens, and consist of the first layer in the architecture of virtually all NLP tasks, such as transformers, RNNs, and CNNs.

The implementation is as follows. For vocabulary of size V and embedding dimension d,

$$E : V \rightarrow R^d.$$

We can now define the embedding matrix W_E

$$W_E \in R^{V \times d},$$

where each row vector is an embedding for one token. Thus, the token embedding for a sequence of tokens $\left[t_1, t_2, t_3, \dots t_n\right]$ is defined as

$$H = \left[e_{t1}; e_{t2}; \dots; e_{tn}\right],$$

where e_i for all $i \in \left[1, 2, \dots t_n\right]$ is defined as

$$e_i = W_E[i, :] \in R^d$$

Positional Embeddings
Attention Mechanisms in transformers process all tokens in parallel, resulting in positional token data not being inherently learned. As a result, the results of Self-Attention will remain identical if order is shuffled. In areas of AI such as NLP (Natural Language Processing) and Music Generation, where order matters immensely, this can be a catastrophic problem, which Positional Embeddings seek out to alleviate. Positional Embeddings allow the model to learn not only semantic information on individual tokens, but also the way they should be arranged.

Implementation is as follows. For consecutive positions p and $p + 1$ and each dimension i in the embedding, define

$$E_p = sin(pos \div 10000^{\frac{2i}{d_{model}}}) \quad if\ i\ is\ even,$$
$$E_p = cos(pos \div 10000^{\frac{2i}{d_{model}}}) \quad else$$

where d_{model} is the embedding dimension in the model.

Rotary Positional Embeddings
Instead of simply encoding Positional Embeddings simply on the position of tokens in the sequence, Rotary Positional Embeddings (RoPE) rotates the token embedding by an angle depending on its position. RoPEs are especially popular among state-of-the-art LLMs, such as LLaMA, specifically in the Query and Key values of Self-Attention. RoPE allows the model to encode relative positions (relationship between tokens) instead of absolute positions (index of token inside sequence). This aids in preventing overfitting, ensuring generalization during training. Since RoPEs are purely deterministic (no weights involved), they also allow models to be smaller, and thus cheaper to train.

Implementation is as follows. For token m at position $x \in R^{d}$, we first split each element of embedding x into pairs:

$$\left[x_{0}, x_{1}\right], \left[x_{2}, x_{3}\right], ..., \left[x_{d-2}, x_{d-1}\right].$$

Multiply each pair by $\theta_{i} \cdot m$ so that for each dimension pair $i \in \left\{i_{0}, i_{1}, ..., i_{(d/2)-1}\right\}$:

$$\begin{bmatrix} x'(2i) \\ x'(2i+1) \end{bmatrix} = \begin{bmatrix} \cos(\theta_i \cdot m) & -\sin(\theta_i \cdot m) \\ \sin(\theta_i \cdot m) & \cos(\theta_i \cdot m) \end{bmatrix} \cdot \begin{bmatrix} x(2i) \\ x(2i+1) \end{bmatrix}$$

where θ_{i} is the rotation frequency for each dimension i, and x' is the output after applying the embedding.

Multi-Head Self-Attention

The main active component of most transformers, Multi-Head Self-Attention, contains a number of representation subspaces, or heads that each contain a number of tokens. Individual representation subspaces attend to, or create context for and draw relationships to all other tokens in other heads, including its own. It does this by determining the relevance and relationships that tokens have with one another, which allows the model to alter its weight matrices in order to learn information about all discrete tokens in training. Different heads will attend to different relationships between tokens. For example, one head might learn structural elements of tokens, while another may learn semantic information, or the meaning behind tokens. When an input sequence of token embeddings is given to the attention mechanism, they are first projected into Query (Q), Key (K), and value matrices using the learned weight matrices. For all heads, we now compute attention scores as

$$Attention(Q, K, V) = softmax(\frac{QK^{T}}{\sqrt{d_{k}}})V,$$

where d_{k} is the dimension of the key vector and the scaling factor $\sqrt{d_{k}}$ prevents dot products of matrices from growing too large. The attention mechanism will compute all attention scores in parallel with different Q, K, and V values for each head. The outputs of all heads are projected through a final linear layer,

$$MultiHead(Q, K, V) = Concat(head_{1}, ..., head_{h})W^{O},$$

where

$$head_{i} = Attention(QW_{i}^{Q}, KW_{i}^{K}, VW_{i}^{V}).$$

Linear Expansion/Compression

Linear Expansions are both types of Dense Layers, which project input vectors from one dimension into another via weight and bias matrices. Linear Expansions project from lower to higher dimensions, while Linear Compressions project from higher to lower dimensions. They are found in almost all Neural Networks, allowing them to learn complex features from data efficiently.

Implementation: We calculate output y from input x and learned weight matrices and bias

$$y = Wx + b.$$

Thus, to train the model, we calculate the gradients with respect to the input, weights and bias respectively:

$$\frac{\partial L}{\partial x} = W^T \cdot \frac{\partial L}{\partial y}$$
$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} \cdot x^T$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}.$$

Finally, we update the learnable weights and biases with learning rate α:

$$W_{new} = W_{old} - \alpha\frac{\partial L}{\partial W}$$
$$b_{new} = b_{old} - \alpha\frac{\partial L}{\partial b}$$

GELU Activation
GELU, or Gaussian Error, Linear Unit, is a type of activation function commonly used in Neural Networks that weights inputs according to their probability based on a standard normal distribution. Thus, GELU's outputs will converge towards zero as the input diverges in the negative direction, while the output will converge towards the input x as x diverges.Activation units allow neural networks to learn complex nonlinear patterns that could not be learned if only linear layers were used. It is especially popular in autoregressive generation models such as GPTs, as it does not instinctively negate negative values, like other activation functions such as ReLU, making it ideal for GPTs and other autoregressive models that need to learn fine details.

Approximation of Implementation:

$$GELU(x) \approx 0.5x(1 + tanh\left[(x + 0.044715x^3)\sqrt{2\pi}\right])$$

Layer Normalization
Layer Normalization is a type of normalization that normalizes based on each feature in an input example in a batch, rather than normalizing based on each batch as a whole. For each example, in the batch, for each example in the batch, Layer Normalization will first calculate the mean and variance of all features in the current layer. It will then subtract the mean and divide by the standard deviation of the features. Mathematically, this is implemented as

$$LayerNorm(x) \;=\; \gamma \cdot \frac{(x-\mu)}{\sqrt{\sigma^2+\epsilon}} + \;\beta,$$

where β and γ are learned weight matrices and ϵ is a small constant to prevent division by zero.

Works Cited

Ba, Jimmy Lei, et al. “Layer Normalization.” *ArXiv:1607.06450 [Cs, Stat]*, 21 July 2016, arxiv.org/abs/1607.06450.

Hendrycks, Dan, and Kevin Gimpel. “Gaussian Error Linear Units (GELUs).” *ArXiv:1606.08415 [Cs]*, 8 July 2020, arxiv.org/abs/1606.08415.

Holtzman, Ari, et al. “The Curious Case of Neural Text Degeneration.” *ArXiv:1904.09751 [Cs]*, 14 Feb. 2020, arxiv.org/abs/1904.09751.

Hsiao, Wen-Yi, et al. “Compound Word Transformer: Learning to Compose Full-Song Music over Dynamic Directed Hypergraphs.” *ArXiv:2101.02402 [Cs, Eess]*, 7 Jan. 2021, arxiv.org/abs/2101.02402.

Huang, Cheng-Zhi Anna, et al. “Music Transformer.” *ArXiv:1809.04281 [Cs, Eess, Stat]*, 12 Dec. 2018, arxiv.org/abs/1809.04281.

Lehmkuhl, Jonathan, et al. “Generating Piano Music with Transformers: A Comparative Study of Scale, Data, and Metrics.” *ArXiv.org*, 2025, arxiv.org/abs/2511.07268. Accessed 22 Jan. 2026.

Lin, Xufeng, et al. “On the Detection-To-Track Association for Online Multi-Object Tracking.” *Pattern Recognition Letters*, vol. 146, June 2021, pp. 200–207, arxiv.org/abs/2107.00500, https://doi.org/10.1016/j.patrec.2021.03.022. Accessed 22 Jan. 2026.

Oore, Sageev, et al. “This Time with Feeling: Learning Expressive Musical Performance.” *ArXiv.org*, 10 Aug. 2018, arxiv.org/abs/1808.03715.

Radford, Alec, et al. *Language Models Are Unsupervised Multitask Learners*.

“Somascape : MIDI Ways - Guide to the MIDI 1.0 Technical Specification.” *Somascape.org*, 2022, www.somascape.org/midi/tech/spec.html.

Su, Jianlin, et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 20 Apr. 2021, https://doi.org/10.48550/arxiv.2104.09864.

Vaswani, Ashish, et al. “Attention Is All You Need.” *Cornell University*, 12 June 2017, arxiv.org/abs/1706.03762.