An Attempt at Generating Beethoven's 33rd Piano Sonata with Music Transformer

Rafael Tamim
LAMSADE, Université Paris Dauphine - PSL, Paris, France.

Tristan Cazenave
LAMSADE, Université Paris Dauphine - PSL, Paris, France.

Abstract

Generating a new piano sonata in the style of Ludwig van Beethoven poses a significant challenge, particularly when attempting to emulate the distinct characteristics of his late compositional period. Beethoven's music, which evolved significantly over his lifetime, is renowned for its complexity, emotional depth, and innovative structures, especially in his later works. The task of creating a 33rd sonata that Beethoven might have composed had he lived longer involves not only capturing his unique style but also understanding the broader historical and stylistic contexts of classical music.

This research delves into the intricacies of Beethoven's musical evolution and explores how artificial intelligence can be employed to generate a composition that reflects the distinct qualities of his late period. The project addresses the challenge of limited data by carefully considering the influence of different composers and periods on Beethoven's style, aiming to produce a work that resonates with the profound expressiveness and complexity of his final compositions. By pushing the boundaries of AI in music, this study contributes to the ongoing dialogue between technology and creativity, offering new insights into the possibilities of machine-generated art in the classical music tradition. It also raises important questions about the role of human interpretation and intuition in the creative process, especially when replicating a style as nuanced and historically significant as Beethoven's. This intersection of AI and classical music serves as a thought-provoking

exploration of how technology can both mimic and enhance the creative practices of the past, suggesting new directions for the future of music composition.

A Question of Context

In classical music, as we define it today, there are several distinct periods. The most well-known, in chronological order, are: the Baroque period (1600-1750), the Classical period (1750-1820), the Romantic period (1820-1900), the Modern period (1900-1975), and finally the Contemporary period (1975 to the present). These periods differ, among other things, in their style, techniques, and forms of composition.

Within these periods, each composer develops their own musical universe. Beethoven's style is therefore unique, and it is essential to choose a model capable of learning to generate music in this style. However, we knew that training a deep learning model would require a large amount of data, and limiting ourselves to Beethoven's 32 sonatas would not be sufficient. Therefore, we chose to start with a more extensive and diverse dataset containing pieces from various periods, while also incorporating the sonatas later in the process.

Moreover, Beethoven’s style evolved throughout his lifetime. His piano sonatas illustrate this evolution well. Musicologists have classified them into three parts, corresponding to the three periods of Beethoven’s life. Therefore, to generate the 33rd sonata, not all the sonatas should be given the same importance. The sonatas from his late period should have a greater impact on the generation of the 33rd sonata than the earlier ones. We therefore considered implementing a weighting system to represent the importance of each sonata in the generation process. These weights will subsequently be used in the loss function calculation that the model aims to minimize. We also realized that this system could be applied to all pieces in the dataset. The idea was to assign a low weight to composers stylistically distant from Beethoven (e.g., Debussy, Rachmaninoff) and a high weight to his contemporaries (e.g., Mozart, Haydn).

A model like Museformer [1] already uses a weighting system, but this concerns how the model learns rather than what it learns. Museformer is based on the principle that some musical passages are more important than others, containing information about the structure of the work, such as the main theme. Ultimately, after an unsuccessful attempt to make this model work, we decided to switch to another model based on the same architecture, namely Music Transformer [2].

When AI Judges Music

To objectively evaluate the model, we chose to use perplexity as a metric, in addition to the calculation of accuracy and the loss function implemented in the base code.

Here is the definition of perplexity from the article *Decoding Perplexity and its Significance in LLMs* [7]:

"In brief, perplexity measures the model's confidence in its predictions. The concept of perplexity evaluates how confused the model is when predicting the next word in a sequence. Lower perplexity indicates that the model is more certain of its predictions. In contrast, higher perplexity suggests that the model is more uncertain. Perplexity is a crucial metric for assessing the performance of language models in tasks such as machine translation, speech recognition, and text generation.

The perplexity of a language model can be calculated using the average negative log-likelihood. The formula for perplexity is given by:

$$\text{Perplexity} = \exp(\text{Average NLL})$$

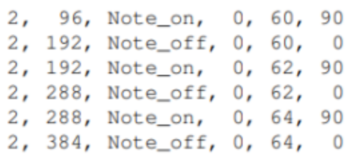
where the average negative log-likelihood (Average NLL) is defined as:

$$\text{Average NLL} = -\frac{1}{N}\sum_{i=1}^{N}\log p(w_i \mid w_{1:i-1})$$

Here, *N* is the number of words in the sequence, and $p(w_i \mid w_{1:i-1})$ is the predicted probability of the word w_i given the previous words $w_{1:i-1}$ .The exponential function is used to convert the average negative log-likelihood into perplexity, thus providing a measure of the model's confusion regarding the word sequence."

In the previous definition, we talk about predicting words. However, it can be adapted to the prediction of notes or other elements of musical vocabulary since our dataset consists of MIDI (Musical Instrument Digital Interface) files.

A MIDI sequence is a digital file that contains control information for music, such as the notes played, velocity, tempo changes, and channel control commands.



```
2,  96, Note_on,  0, 60, 90
2, 192, Note_off, 0, 60,  0
2, 192, Note_on,  0, 62, 90
2, 288, Note_off, 0, 62,  0
2, 288, Note_on,  0, 64, 90
2, 384, Note_off, 0, 64,  0
```

Figure 1 : excerpt from a MIDI file (turned into readable ascii)

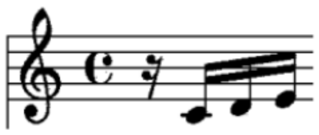


Figure 2 : Score corresponding to the MIDI excerpt

In figures 1 and 2, extracted from [3], we can visualize how a MIDI file works. In figure 1, the first line "2, 96, Note_on, 0, 60, 90" means "at 96 ticks (the moment the event occurs) on track 2, on channel 1, a middle C (C4) note is played with a velocity of 90." Below, we can clearly see the middle C on the sheet music.

This information is then tokenized during the preprocessing phase, and the model aims to predict the next token based on the previous tokens.

Training is a Sport

As mentioned earlier, training a model like Music Transformer requires a significant amount of data. Training the model directly on Beethoven's 32 sonatas would not have guaranteed a good result. Therefore, we split the training into two phases: a pre-training phase on a general classical music dataset of around 700 MIDI files, followed by a fine-tuning phase on Beethoven's 32 sonatas. During the first phase, the model adjusts its parameters to minimize a loss function, calculated based on the difference between the model's predictions and the actual data. In the second phase, the model builds on the parameters adjusted during the previous phase and fine-tunes them to capture the particular nuances of Beethoven's style, thereby improving the quality of the generated compositions.

We implemented a weighting system in the pre-training dataset by assigning each composer a weight relative to their similarity to Beethoven. The choice of weights is subjective, and we based them on our own musical knowledge. The following figures (3-6) show the different metric behaviors between training without weights and training with weights.

We can see that the accuracy decreased and the perplexity increased with the application of the weights. This did not positively influence the results, contrary to our expectations. As we will see later, the issue does not stem from the strategy implemented but rather from the dataset.

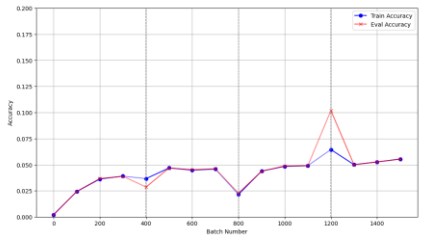


Figure 3 : Accuracy without weights

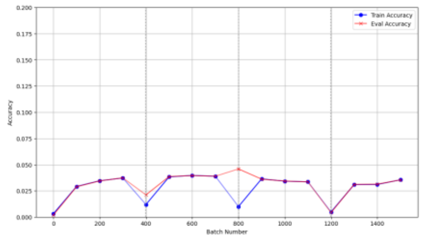


Figure 4 : Accuracy with weights

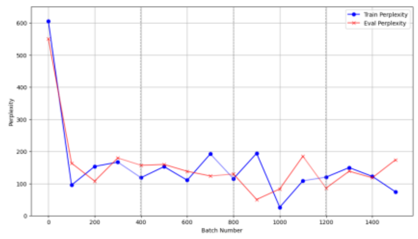


Figure 5 : Perplexity without weights

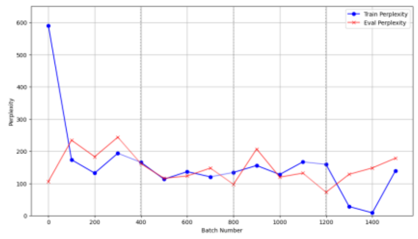


Figure 6 : Perplexity with weights

The Musical Parrot Syndrome

While observing the behavior of the loss function, we noticed signs of overfitting, indicating that the model does not generalize well to data it hasn't seen during training. In other words, it memorizes the pieces in the dataset and fails to generate a new piece effectively. To address this, we first considered adjusting the dropout rate. Depending on the severity of the overfitting, the dropout rate can be increased to mitigate the issue. Initially set at 20% in the base code, we increased it to 40%, which reduced the overfitting but did not eliminate it completely.

In truth, 700 pieces are not enough for a model as complex as Transformers. For instance, Museformer was trained on a dataset of nearly 30,000 pieces. Additionally, our dataset was not very homogeneous, with a majority of works by Bach and Chopin. We, therefore, sought a more extensive dataset, namely the Maestro dataset [4] developed by Google Magenta. With this dataset, the overfitting issue was resolved, even when keeping the dropout rate at its initial value. Moreover, the accuracy doubled, rising from 3.5% to 7%, and the perplexity decreased significantly from 178 to 61. However, a perplexity value of 61 is still too high; it means that during generation, the model has 61 possible tokens to choose from. To further expand the Maestro dataset, we transposed the pieces into all possible keys, like in [9], multiplying the number of pieces by 24 and resulting in a dataset of over 30,000 MIDI files. The following graphs (figures 7-8) show that the results improved dramatically with the transposition: accuracy reached 23%, and perplexity dropped to 14.

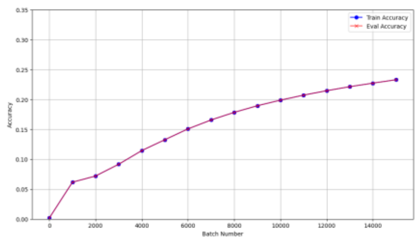


Figure 7 : Accuracy after transposition

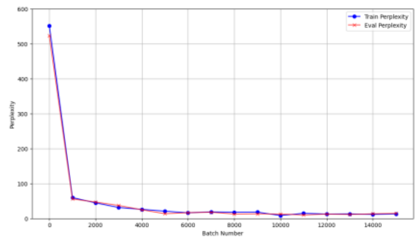


Figure 8 : Perplexity after transposition

The Cool New Tricks of AI

Let's start with the definition of a learning rate schedule from the site Towards Data Science [8]:

"A learning rate schedule is a crucial technique in training machine learning models because it allows for the adjustment of the learning rate throughout the training process. The primary utility of a learning rate schedule lies in the following aspects:

Improved Convergence: A well-adjusted learning rate can help converge more quickly to a minimum of the loss function. By gradually reducing the learning rate, the model’s weights can be refined with greater precision as the minimum is approached.

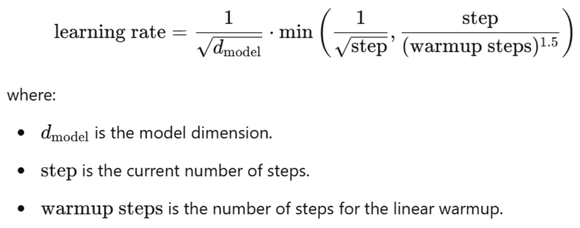
Prevention of Oscillations: Using a schedule prevents a too-high learning rate from causing oscillations around the local minimum. A rate that is too high may result in excessive jumps in the loss function values, hindering convergence.

Avoidance of Local Minima: Strategies such as gradually reducing the learning rate allow the model to overcome local minima by dynamically adjusting the rate. This helps in more effectively exploring the solution space.

Resource Optimization: By regulating the learning rate, computational resources can be optimized, ensuring that learning is neither too fast (which could lead to premature convergence) nor too slow (which might require more computation time).

In summary, the learning rate schedule plays a key role in improving the performance and efficiency of model training by allowing better management of the learning speed throughout the training process."

Let's now return to our code. The model's basic strategy combined several techniques: an inverse square root schedule to dynamically adjust the learning rate, a linear warmup to gradually increase the learning rate at the beginning of training, and a minimum schedule to ensure a minimum threshold for the learning rate. See figure 9 for the formula of the learning rate.



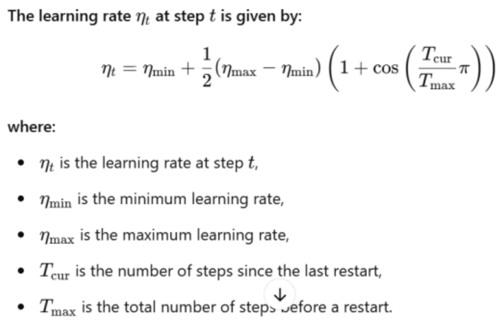
$$\text{learning rate} = \frac{1}{\sqrt{d_{\text{model}}}} \cdot \min\left(\frac{1}{\sqrt{\text{step}}}, \frac{\text{step}}{(\text{warmup steps})^{1.5}}\right)$$

where:

- d_{model} is the model dimension.
- step is the current number of steps.
- warmup steps is the number of steps for the linear warmup.

Figure 9 : Initial learning rate formula

This technique is quite commonly used, which led us to try another, more recent strategy: Cosine Annealing. This method first appeared in the paper "SGDR: Stochastic Gradient Descent with Warm Restarts," written by Ilya Loshchilov and Frank Hutter [5]. This paper, published in 2017, introduces a method for dynamically adjusting the learning rate by using a cosine function to periodically reduce the learning rate. See figure 10 for the formula.



The learning rate η_t at step t is given by:

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{T_{\text{cur}}}{T_{\max}}\pi\right)\right)$$

where:

- η_t is the learning rate at step t,
- $\eta_{\min}$ is the minimum learning rate,
- $\eta_{\max}$ is the maximum learning rate,
- T_{cur} is the number of steps since the last restart,
- $T_{\max}$ is the total number of steps before a restart.

Figure 10 : Learning rate in the Cosine Annealing strategy

During the testing phase, we compared the results of training with the baseline strategy to those of training with the Cosine Annealing strategy. As shown in the following figures (11-12), Cosine Annealing improves the metric values, particularly the perplexity. Indeed, we achieve a perplexity of 61 by the end of the fifth epoch compared to only 140 with the baseline strategy. Precision, on the other hand, increases by only 1.5 points, which is not a significant improvement. Here is the perplexity curve for both strategies:

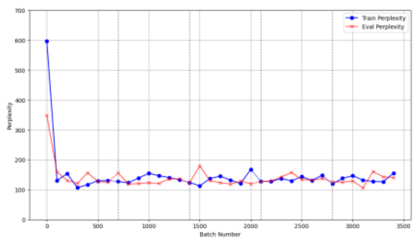


Figure 11 : Perplexity without Cosine Annealing strategy

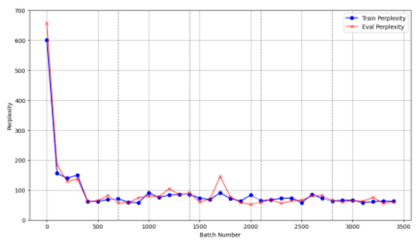


Figure 12 : Perplexity with Cosine Annealing strategy

With the Cosine Annealing strategy and the transposition of the Maestro dataset into all possible keys, we achieved very good results (for reference, 23% precision and a perplexity of 14). However, so far we have only discussed the pre-training phase. Let's see how this performs specifically on Beethoven's sonatas:

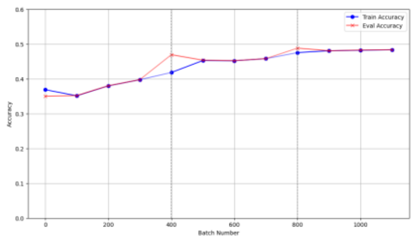


Figure 13 : Accuracy during fine-tune

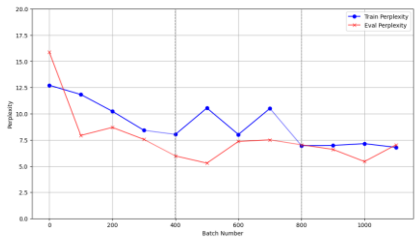


Figure 14 : Perplexity during fine-tune

The precision converges to 48% after 3 epochs, and the perplexity drops to 7! The model now has only 7 possible choices at each generation step.

When Numbers Lie

Finally, our last focus was to understand and improve the generation phase, i.e., the one that produces the final result. To generate a piece of music, the model takes as input a sequence containing the first *x* tokens from a MIDI file and uses what it learned during training to generate the following tokens, in other words, the continuation of the sequence.

We tested with several input sequences, particularly the beginning of Beethoven's 32nd sonata, and the result was rather surprising. There are many musical elements that do not contribute much to the structure of the piece, such as trills or notes repeated for a long time, which results in a somewhat incoherent output. Additionally, there is a noticeable lack of creativity as the same elements are frequently repeated. However, one might argue that the model has learned to use recurring motifs, treating them like a recurring theme. The beginning of the 32nd sonata contains few structural elements (very few harmonies or "melodies") and includes a trill, which helps explain why the result lacks creativity. Testing with other melodically more varied sonata openings shows better generation performance.

In generative models, different generation policies can be used, the main ones being argmax and softmax. Using argmax, the model systematically chooses the token with the highest probability at each generation step. With softmax, the model uses a probability distribution, allowing for more diversity in the choice of the token to be generated. In our code, the softmax policy was chosen. Therefore, the lack of creativity cannot really be justified by the policy used.

We think that the complexity of classical music explains why our model did not perform well. In classical music, there is a hierarchy of musical elements. Ornaments like trills, mordents, and appogiaturas enrich and express the music but remain secondary to the main melody. Similarly, passing notes, escape notes, appogiaturas, and suspensions add richness and harmonic tension but are generally less important than structural notes. However, our model treats all elements equally. In our opinion, learning should be decomposed into several tasks, such as one for the harmony of the piece, another for melody, rhythm, etc. Once these steps are completed, a basic structure of the piece can be filled with less important elements as mentioned previously. Such models already exist in the literature; for example, WuYun [6] proposes a two-step hierarchical architecture for melody generation guided by a skeleton. This paper focuses only on melody. In our case, a method to separate different musical layers (melody, harmony, etc.) would be needed, which is far from straightforward.

Conclusions

Artificial intelligence offers a wide range of applications, and in this context, we chose a field that is close to our heart: music. After considering several options, we ultimately decided to generate Beethoven's 33rd piano sonata. Our first task was to devise a method to generate a work in Beethoven's style, consistent with his final creative period. This led to the idea of a weighting system in the dataset to give more importance to works from Beethoven's later period. After selecting a suitable deep learning model for music generation, the next question was how to evaluate the results. Although subjective criteria are important, we needed objective criteria to assess the quality of the compositions generated by the AI.

The model training was carried out in two phases. This approach allowed the model to learn from a broad dataset before specializing in Beethoven. However, we encountered an overfitting issue, where the model memorized the specifics of the pieces, a phenomenon we tried to mitigate through various machine learning techniques, including our weighting system. As AI evolves rapidly, we tested a recent learning strategy, which significantly improved the results of both the pre-training and final phases compared to the baseline strategy. Despite good objective results, the generated sequences were not always pleasant to listen to. We then examined the model's generation policy, which provided some answers. Ultimately, this study helped us better understand the limitations of our model and consider future improvements.

From a technical perspective, the work accomplished reached a good level of completion while leaving room for improvements. The decision to use deep learning models, particularly Transformers, proved effective for handling the complexity of musical sequences. The two-phase approach provided a solid learning foundation, facilitating specialization.

However, the initial occurrence of overfitting highlighted a potential weakness in the model. While regularization techniques such as dropout and the weighting system helped mitigate this issue, it was the use of a larger and more diverse dataset that ultimately resolved the problem. This improvement allowed the model to produce more innovative and stylistically coherent works, suggesting that the quality and diversity of training data are crucial for enhancing the model's generative capabilities.

Integrating a weighting system to prioritize Beethoven's later sonatas was a creative and promising step. It influenced the musical generation by respecting the chronological and stylistic specifics of the composer. This concept could be expanded to other composers and styles, enriching research perspectives.

On the other hand, the project revealed some limitations. While effective, Transformers might not be the best tools for all musical generation tasks. The fine-tuning methodology on a limited number of sonatas may also not fully capture Beethoven's stylistic complexity. A more nuanced approach, combining machine learning techniques with elements of music theory, could offer more refined results.

References

[1] B. Yu, P. Lu, R. Wang, W. Hu, X. Tan, W. Ye, S. Zhang, T. Qin, and T. Liu. Museformer : Transformer with fine-and coarse-grained attention for music generation. In Advances in Neural Information Processing Systems (NeurIPS). NeurIPS, 2022.

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

Music transformer: Generating music with long-term structure. In Proceedings of the 7th International Conference on Learning Representations (ICLR). ICLR, 2018.

[3] Jean-Pierre Briot, Gaetan Hadjeres, and Francois-David Pachet. Deep Learning Techniques for Music Generation. Computational Synthesis and Creative Systems. Springer, 2019. ISBN 978-3-319-70162-2. Hardcover.

[4] Curtis Hawthorne, Andriy Stasyuk, Adam Roberts, Ian Simon, Cheng-Zhi Anna Huang, Sander Dieleman, Erich Elsen, Jesse Engel, and Douglas Eck. Enabling factorized piano music modeling and generation with the MAESTRO dataset. In International Conference on Learning Representations,2019. https://openreview.net/forum?id=r1lYRjC9F7.

[5] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. arXiv preprint arXiv:1608.03983, 2017.

[6] K. Zhang et al. Wuyun: Exploring hierarchical skeleton-guided melody generation using knowledge-enhanced deep learning. arXiv preprint arXiv:2301.04488, 2023.

[7]https://blog.uptrain.ai/decoding-perplexity-and-its-significance-in-llms/

[8]

https://towardsdatascience.com/a-visual-guide-to-learning-rate-schedulers-in-pytorch-24bbb262c863

[9] Huang, A., Wu, R. (2016). *A Transformer Based Pitch Sequence Autoencoder with MIDI Augmentation*. arXiv preprint arXiv:1611.03477.