Scaling Laws for Language Models on Symbolic Music Data

Prabhjeet Singh

All code and experiments are available in the **project repository**.

1 Introduction

Natural language tasks have become a central focus in the field of artificial intelligence, attracting significant attention from a large research community. As this area has grown rapidly over the past decade, substantial effort has been devoted to improving the performance of neural network language models. It has long been observed that larger models tend to outperform smaller ones, yet for many years there was no clear quantitative understanding of how increasing model size affects performance. A major step toward addressing this question was made by Kaplan et al. (2020) [1] in their work on scaling laws for neural language models. Through large-scale empirical experiments, they demonstrated that the validation loss of autoregressive language models follows smooth power-law relationships with respect to model size, dataset size, and training compute. These findings were instrumental in establishing that performance improvements from scaling occur in a regular and predictable manner as models increase in size. Consequently, scaling laws have become an important framework for understanding language model behavior and for guiding decisions about model design, data usage, and computational resources.

Prior literature suggests that most of the work on scaling laws has focused on language models trained on textual data. This domain has been a major focus of researchers around the world. In this project, however, we explore scaling laws for language models trained on symbolic music data instead of textual data. The motivation behind this project is to examine whether scaling laws that are applicable to text-based language models extend to the domain of symbolic music data. This is a suitable area of study because, just like text, symbolic music can be represented as a sequence of tokens and modeled using standard language modeling techniques. On the other hand, it differs from natural language in many ways, such as following stricter syntactic rules, having strong temporal structure, and being organized hierarchically across notes, measures, and musical phrases.

To study scaling laws in the music domain, we conduct a systematic empirical study of scaling behavior for language models trained on symbolic music data. We first build an end-to-end preprocessing pipeline that converts raw MIDI files into a text-based symbolic music representation (ABC notation), applies data cleaning and tokenization, performs vocabulary generation and token encoding, and produces training, validation, and test splits at scale. Using this dataset, we train a family of decoder-only transformer language models of varying sizes under a controlled training setup, where the number of model parameters is varied while all other factors are kept consistent. We measure validation performance after training and analyze how loss changes with model capacity. In addition, we train recurrent neural network models (LSTMs) with comparable parameter counts to transformer models in order to directly compare their scaling behavior with that of transformers. Finally, we train the best-performing model more extensively and evaluate its generative capabilities through both quantitative metrics and qualitative analysis of generated symbolic music samples.

2 Data

2.1 Dataset Description

For this project, We use the Lakh MIDI Dataset[1], which is a large collection of MIDI files covering a wide range of musical styles and genres. The dataset contains over 100,000 MIDI files. These files represent music in a structured, event-based format, encoding information such as note pitches, durations, and timing. We choose this dataset because it is widely used in music information retrieval and symbolic music modeling research and provides broad stylistic diversity. Its large scale enables controlled scaling experiments, while its diversity makes it a suitable choice for studying the behavior of language models in the symbolic music domain.

2.2 Preprocessing Pipeline

MIDI files represent music in a structured format but it is not directly suitable for language modeling because it is stored in a binary format. For studying scaling laws, we convert the MIDI files into ABC notation which is a human-readable symbolic music representation. ABC notation provides a textual encoding of musical structure of MIDI, making it well suited for sequence modeling and large-scale token-based training. In the data preprocessing pipeline, each MIDI file is first converted into ABC notation using the midi2abc command-line tool. Files that fail conversion or produce invalid or corrupted ABC output are discarded, ensuring that only syntactically valid symbolic music sequences are included in the dataset. Due to computational constraints, we extracted a total of 50K ABC files. We then applied a custom music-aware tokenization scheme to process these files. More details on the tokenization are provided in the next subsection. After tokenizing the data, we constructed a vocabulary by counting token frequencies across all sequences. Finally, all token sequences were encoded using the generated vocabulary. Then, the encoded sequences are split into training, validation, and test sets using a 98% / 1% / 1% split. The majority of tokens are assigned to the training set to support large-scale scaling experiments, while the validation and test sets are used exclusively for evaluation.

Table 2.2 summarizes the dataset statistics after preprocessing.

2.3 Tokenization Strategy

The tokenization approach used in this project is not purely character-level or note-level. Instead, we use a custom, music-aware tokenization scheme. The main reason for this choice is to give the model musical data in a format that better matches how music is naturally structured.

This tokenization includes musically meaningful elements such as meter, key signature, bar lines, chord brackets, rests, and other structural symbols. Because of this, the model can learn both small musical events, like notes and durations, and larger musical patterns and structure.

We also tried character-level tokenization during the project, which produced a vocabulary of 96 characters. However, we decided not to use it because character-level tokens do not represent musical ideas well and often break meaningful musical units into pieces. As a result, this makes it harder for the model to learn useful musical patterns compared to the music-aware tokenization.

Below example shows the tokenization of a valid ABC sequence.

ABC Sequence:

```
X: 1
T: Cooley's
M: 4/4
L: 1/8
```

[1] https://colinraffel.com/projects/lmd/

Vocabulary Size	27,224
Total Tokens	
Train	1,167,894,118
Validation	11,907,471
Test	11,808,149
Total	1,191,609,738
Sequence Count	
Train	257,964
Validation	2,632
Test	2,633
Sequence Length Distribution	
Min	30
Max	5,000
Mean	4,524.89
Median	5,000.00
Std	1,163.53
Conversion Success Rate	
Total MIDI Files	116,189
Successful	115,296
Failed	893
Success Rate	99.23%
Quality Filters Applied	
Min Tokens	30
Max Tokens	5,000
Filtered (too short)	0
Sequences from splits	257,141

Table 1: Statistics of the data preprocessing pipeline

```
R: reel
K: Emin
|:D2|EB{c}BA B2 EB|~B2 AB dBAG|FDAD BDAD|FDAD dAFD|
```

Tokenized ABC Sequence:

```
['X:', 'M:4/4', 'L:1/8', 'K:E', '|', 'D2', '|', 'E', 'B',
 'c', 'B', 'A', 'B2', 'E', 'B', '|', 'B2', 'A', 'B', 'd',
 'B', 'A', 'G', '|', 'F', 'D', 'A', 'D', 'B', 'D', 'A', 'D',
 '|', 'F', 'D', 'A', 'D', 'd', 'A', 'F', 'D', '|']
```

Each ABC file is first tokenized into a sequence of musically meaningful symbols. Sequences shorter than 30 tokens are discarded. Discarding them is a preprocessing decision as they may not contain sufficient musical content to help us with training. Sequences longer than 5,000 tokens are split into contiguous chunks to keep the stored data manageable. During training, random 256-token windows are sampled from the flattened token array, so the 5,000 token limit does not affect the model's context length. A vocabulary is constructed by counting token frequencies across all sequences, with special tokens added for padding, unknown symbols, and sequence boundaries. Finally, all token sequences are encoded as integer indices resulting in a dataset ready for training autoregressive language models.

2.4 Dataset Statistics and Visualizations

During the conversion of MIDI files to ABC notation, a total of 893 MIDI files failed to convert out of 116,189 input files. From the remaining set of 115,296 converted ABC files, we select a subset of 50,000 files for all subsequent experiments. While using a larger portion of the dataset would likely improve model performance, this choice was

primarily driven by computational resource constraints. The selected subset still provides sufficient scale and diversity to support meaningful scaling experiments. After tokenization and chunking, the dataset contains 263,229 sequences in total. No ABC files were discarded due to the minimum sequence length requirement (30 tokens), as all files met this threshold. Out of the 50,000 ABC files, 6,088 were short enough to remain as single sequences, while the remaining files were split into 257,141 chunks due to exceeding the maximum length of 5,000 tokens. After splitting the dataset into training, validation, and test sets using a 98% / 1% / 1% split, the training set contains approximately 1.16 billion tokens, while the validation and test sets each contain around 11 million tokens.

3 Methods

We study scaling behavior using a family of autoregressive transformer models trained on symbolic music token sequences. In the following subsection, we describe the model architectures used.

3.1 Model Architectures

3.1.1 Transformer

We implemented a decoder-only transformer language model from scratch. The architecture has following components:

- **Token Embedding:** Maps each token in the vocabulary to a dense vector of dimension d_{model}.
- **Positional Embedding:** Learned positional embeddings for each position in the context window (256 positions).
- **Transformer Blocks:** Each block contains:
  - Multi-head self-attention with causal masking to prevent attending to future tokens
  - Feed-forward network with GELU activation and expansion ratio of 4 ($d_{ff} = 4 \times d_{model}$)
  - Pre-norm design with LayerNorm applied before attention and feed-forward layers
  - Residual connections around both sub-layers
  - Dropout of 0.1 after attention and feed-forward layers
- **Final LayerNorm:** Applied after the last transformer block.
- **Output Projection:** Linear layer mapping from d_{model} to vocabulary size for next-token prediction.

We trained five transformer models of varying sizes as shown below in table 2.

Model	**Layers**	d_{model}	**Heads**	d_{ff}	**Parameters**
Tiny	4	64	4	256	3.7M
Small	6	128	4	512	8.2M
Medium	8	256	8	1024	20.3M
Large	8	512	8	2048	53.2M
XL	8	1024	16	4096	156.8M

Table 2: Transformer model configurations. All models use a context length of 256 tokens.

We varied both depth (number of layers) and width (d_{model}) to cover a range of parameter counts from 3.7M to 156.8M. The number of attention heads scales with d_{model} to keep the head dimension reasonable (16-64 dimensions per head).

3.1.2 LSTM

In addition to transformers, we implemented LSTM-based language models with comparable parameter counts. The architecture consists of:

- **Token Embedding:** Same vocabulary and embedding approach as the transformer.
- **LSTM Layers:** Stacked LSTM cells with dropout between layers.
- **Output Projection:** Linear layer mapping from hidden size to vocabulary size.

Table 3 shows LSTM model configurations.

Model	Hidden Size	Layers	Parameters
Tiny	64	1	3.5M
Small	128	2	7.3M
Medium	320	3	19.9M
Large	680	4	51.9M

Table 3: LSTM model configurations. All models use the same context length and vocabulary as the transformers.

We varied both hidden state size and number of layers rather than just the size of hidden state alone. This gives a fairer comparison with transformers, where depth plays an important role in learning hierarchical patterns. The parameter counts were chosen to roughly match the transformer models (Tiny, Small, Medium, Large) so that any performance differences reflect architectural capabilities rather than model capacity.

3.2 Training Setup

All models were trained with the same hyperparameters to ensure fair comparison:

- **Context Length:** 256 tokens. This is long enough to capture several bars of music while keeping training efficient.
- **Batch Size:** 4,096 tokens (16 sequences of 256 tokens each).
- **Optimizer:** AdamW with weight decay of 0.1.
- **Learning Rate:** Peak learning rate of 3×10^{-4} with linear warmup for the first 5% of training, followed by cosine decay to zero.
- **Training Data:** 100 million tokens per model for the scaling study, which corresponds to 390,625 samples and 24,414 training steps.
- **Regularization:** Dropout of 0.1 applied in attention and feed-forward layers.

To ensure all models trained on exactly the same data, We pre-generated fixed sampling indices with a random seed of 42. Instead of randomly sampling positions during training, We saved 390,625 starting positions beforehand. Both transformer and LSTM models used these same indices, so any performance differences come from the architecture, not from seeing different data. For the best model, We continued training the XL transformer for an additional 100 million tokens (200 million total) using a new set of indices generated with seed 123.

3.3 Experimental Design for Scaling Studies

The goal of the scaling study is to measure how validation loss decreases as model size increases. We trained each model for exactly 1 epoch on 100 million tokens and recorded the final validation loss. To quantify the scaling behavior, we fit a power law of the form:

$$L = a \cdot N^{-\alpha} + c \tag{1}$$

where L is the validation loss, N is the number of parameters, a is a constant, α is the scaling exponent, and c represents the irreducible loss. A higher α means the model improves faster as size increases. We ran this experiment separately for transformers (5 models) and LSTMs (4 models). Both architectures used the same training data, same tokenization, same batch size, and same learning rate schedule. This ensures that any difference in scaling behavior comes from the architecture itself. For comparing the two architectures, we also tracked training time and GPU memory usage for each model.

3.4 Evaluation Metrics

3.4.1 Scaling Study

For the scaling study, we used validation loss as the primary metric. This measures how well the model predicts the next token on unseen data. Lower validation loss indicates better generalization.

3.4.2 Best Model Evaluation

For evaluating the best model (XL transformer trained on 200M tokens), we used:

- **Perplexity:** Calculated as $\exp(\text{loss})$ on 10,000 randomly sampled sequences from the test set. Evaluating on the full test set would require significantly more compute, so we used this subset as a reasonable approximation. A perplexity of 1.42 means the model narrows down to roughly 1.4 equally likely choices per token on average.
- **Syntactic Validity:** Percentage of generated samples that have valid ABC structure. We checked for presence of notes, bar lines, and balanced brackets. This measures whether the model learned the basic format of ABC notation.
- **MIDI Conversion Rate:** Percentage of syntactically valid samples that successfully convert to MIDI using the music21 library.

4 Results

4.1 Transformer Scaling

Fitting a power law $L = a \cdot N^{-\alpha} + c$ to the validation losses gives:

$$L = 274643.68 \cdot N^{-0.8717} + 0.353 \tag{2}$$

The scaling exponent $\alpha = 0.8717$ indicates that validation loss decreases significantly as model size increases. Table 4 shows the results for all five transformer models trained on 100 million tokens.

Model	Parameters	Train Loss	Val Loss	Time (min)	GPU (GB)
Tiny	3.7M	0.9264	0.8684	5.7	2.08
Small	8.2M	0.6485	0.6150	8.7	2.36
Medium	20.3M	0.4902	0.4662	16.0	3.25
Large	53.2M	0.4205	0.4021	32.5	4.24
XL	156.8M	0.4073	0.3751	88.5	7.50

Table 4: Transformer scaling results. All models trained for 1 epoch on 100M tokens.

Figure 1 shows the training loss curves for all transformer models. Larger models converge to lower loss values.

Figure 1: Training loss curves for transformer models. Larger models achieve lower final loss.

Figure 2 shows the scaling plot with the power law fit.

4.2 LSTM Scaling

The power law fit for LSTM gives:

$$L = 312622.62 \cdot N^{-0.8813} + 0.514 \tag{3}$$

The LSTM scaling exponent $\alpha = 0.8813$ is slightly higher than the transformer's 0.8717.

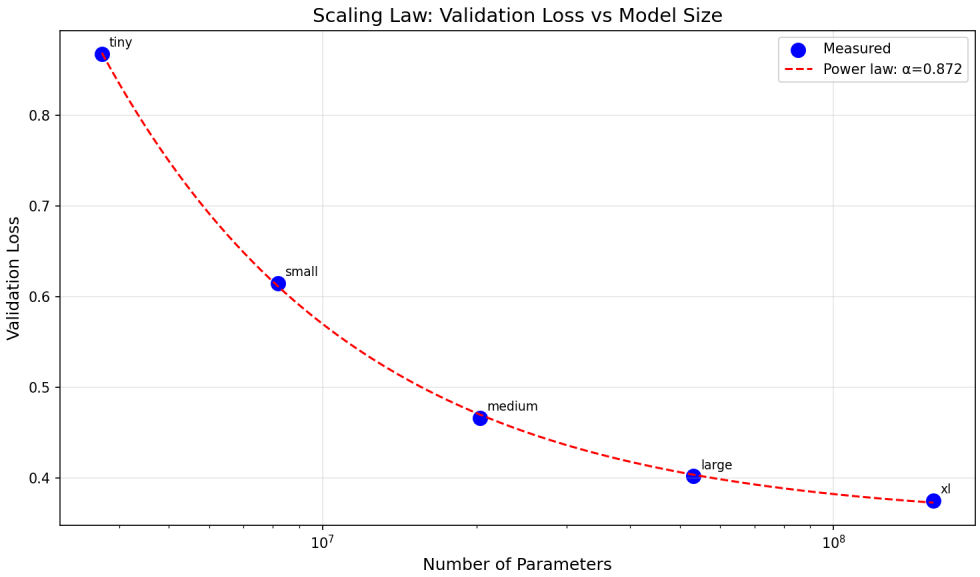

Figure 2: Transformer scaling: Validation loss vs Number of parameters. The dashed line shows the power law fit with $\alpha = 0.872$.

Table 5 shows the results for the four LSTM models.

Model	Parameters	Train Loss	Val Loss	Time (min)	GPU (GB)
Tiny	3.5M	1.0250	1.0413	10.3	1.86
Small	7.3M	0.8085	0.7972	12.8	1.92
Medium	19.9M	0.6850	0.6251	26.5	2.18
Large	51.9M	0.5776	0.5656	57.0	2.80

Table 5: LSTM scaling results. All models trained for 1 epoch on 100M tokens using the same data as transformers.

Figure 3 shows the training loss curves for all LSTM models while figure 4 shows the LSTM scaling plot.

4.3 Architecture Comparison

Figure 5 shows both architectures on the same plot. Although LSTM has a marginally higher scaling exponent, transformers consistently achieve lower validation loss at every model size. The irreducible loss c is also lower for transformers (0.353 vs 0.514), suggesting that transformers would continue to outperform LSTMs even with much larger models.
Table 6 shows the scaling exponents values for transformers and LSTMs

Metric	Transformer	LSTM
Scaling exponent (α)	0.8717	0.8813
Irreducible loss (c)	0.353	0.514
Best val loss (largest model)	0.3751	0.5656

Table 6: Scaling exponent comparison. While LSTM has a slightly steeper exponent, transformers achieve lower absolute loss.

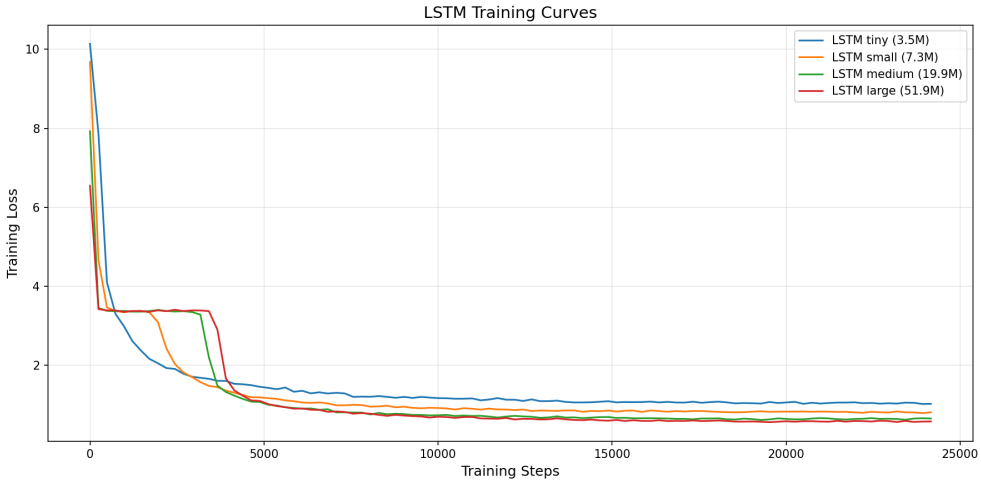

Figure 3: Training loss curves for LSTM models. Similar to transformers, larger models achieve lower final loss.

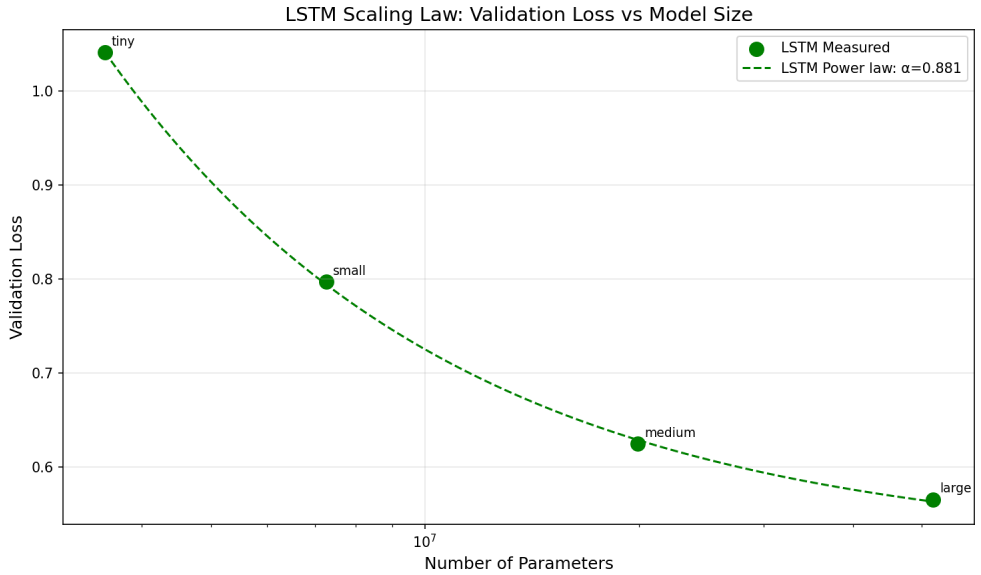

Figure 4: LSTM scaling: Validation loss vs Number of parameters. The dashed line shows the power law fit with $\alpha = 0.881$.

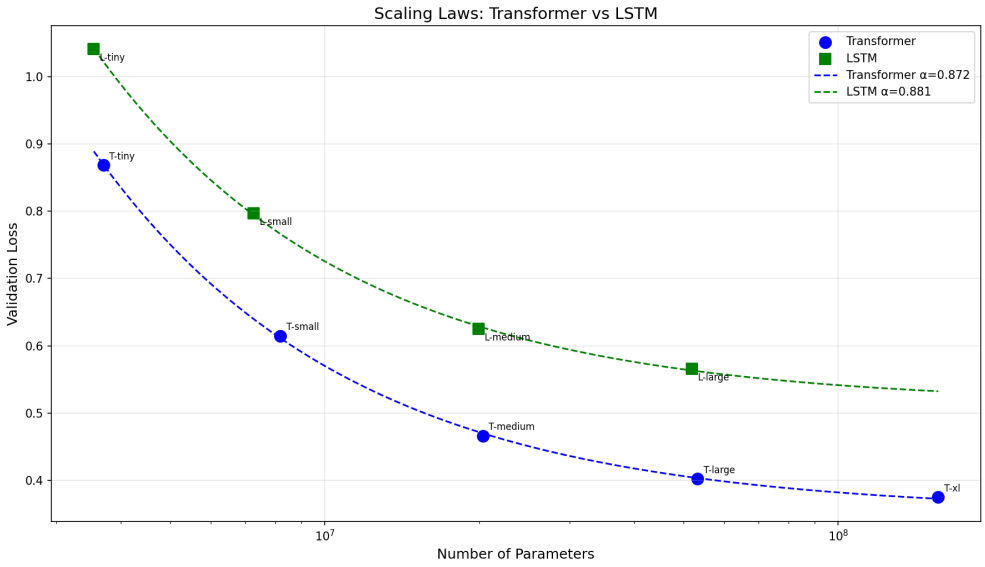

Figure 5: Scaling comparison between transformer and LSTM. Both architectures show similar scaling exponents, but transformers achieve lower validation loss at every model size.

4.4 Computational Costs

Table 7 compares training time and memory usage.

Model	Architecture	Parameters	Val Loss	Time (min)	GPU (GB)
Tiny	Transformer	3.7M	0.8684	5.7	2.08
Small	Transformer	8.2M	0.6150	8.7	2.36
Medium	Transformer	20.3M	0.4662	16.0	3.25
Large	Transformer	53.2M	0.4021	32.5	4.24
XL	Transformer	156.8M	0.3751	88.5	7.50
Tiny	LSTM	3.5M	1.0413	10.3	1.86
Small	LSTM	7.3M	0.7972	12.8	1.92
Medium	LSTM	19.9M	0.6251	26.5	2.18
Large	LSTM	51.9M	0.5656	57.0	2.80

Table 7: Computational costs comparison. LSTMs use less GPU memory but transformers achieve lower loss.

LSTMs use less GPU memory because they do not store attention matrices. However, transformers achieve significantly lower validation loss at comparable parameter counts, making them more sample-efficient for this task.

4.5 Best Model Results

We selected the XL transformer as the best model based on having the lowest validation loss. We continued training it for an additional 100 million tokens (200 million total). Table 8 summarizes the results.

4.6 Generation Results

We generated 30 samples using three approaches:

1. **Pure unconditional:** Starting with only the beginning token. The model produced invalid outputs like repeated rests or notes without ABC headers. This happens because training used random 256-token windows, which

Metric	Value
Parameters	156.8M
Total tokens trained	200M
Final train loss	0.3733
Final val loss	0.3487
Test loss	0.3495
Test perplexity	1.42

Table 8: Best model (XL transformer) results after training on 200M tokens.

mostly captured mid-song content rather than file beginnings.

2. **Header-prompted:** Providing header tokens and then letting the model generate the melody freely. This produced valid, playable music.

3. **Conditional:** Providing headers plus a few starting notes. This also produced valid music that continued naturally from the given notes.

Table 9 summarizes the generation evaluation results.

Metric	Value
Total samples generated	30
Valid ABC syntax	19/30 (63.3%)
MIDI conversion (of valid samples)	17/19 (89.5%)

Table 9: Generation evaluation results.

Most of the invalid samples came from pure unconditional generation, which failed to produce proper ABC headers. Other than 2, all other valid ABC sequences gets converted to MIDI. Sample outputs are included in Appendix A. The generated music can be played using online ABC players such as `https://abcjs.net/abcjs-editor.html`.

5 Discussion

5.1 Key Insights from Scaling Experiments

Both transformer and LSTM architectures follow power law scaling, with similar exponents ($\alpha = 0.87$ for transformers, $\alpha = 0.88$ for LSTMs). However, transformers achieve lower absolute loss at every model size. The irreducible loss c is also lower for transformers (0.353 vs 0.514), meaning transformers would continue to outperform LSTMs even with much larger models. More importantly, our scaling exponents are much higher than those reported for natural language. Kaplan et al. (2020) [1] found $\alpha \approx 0.076$ for text language models, while we observe $\alpha \approx 0.87$ for music. This suggests that music modeling benefits more steeply from increased model size. The possible explanation for this is that music has a more regular structure than text. For example, music follows time signatures that enforce regular bar lengths, and it is written in keys that restrict which notes naturally fit together. Rhythms and melodic patterns also tend to repeat in predictable ways. As model capacity increases, larger models can learn and exploit these structured patterns more effectively.

5.2 Music-Specific Patterns

Examining the generated samples, the model learned several musical concepts:

- **Note names and octaves:** The model produces valid note names (C, D, E, F, G, A, B). Commas indicate lower octaves (e.g., A,,, and B,,, in Unconditional Sample 1 are three octaves below). Lowercase letters indicate higher octaves (e.g., c, d, e in Conditional Sample 2, and c', d', e' in Conditional Sample 5 for even higher octaves).

- **Rhythm and durations:** Generated music includes note durations such as D2 (half note), D/2 (eighth note), C8 (very long note), and C3/2 (dotted note). Rests appear with durations like z8, z4, z/2.
- **Bar lines:** The model places bar lines (|) to separate measures throughout the outputs.
- **Accidentals:** Sharps appear as ^A, ^D, ^G, ^c, ^f, ^d (Unconditional Sample 1, Conditional Samples 1, 8, 9). Flats appear as _A, _G, _g, _B (Conditional Samples 3, 6). Natural signs appear as =C, =E, =F, =G, =A (Unconditional Sample 14, Conditional Samples 5, 7, 8).
- **Chords:** The model generates chord notation with brackets such as [A,,,-G,,,], [GG,], [B,E,], [^fF], [eE] (Unconditional Samples 1, 9, 11, 13) and [a_g], [bg], [GC], [EB,] (Conditional Samples 6, 10).
- **Ties:** Notes connected across bars appear with dashes, such as E,,,6- and d8- (Unconditional Sample 1, Conditional Sample 5).
- **Time signatures:** The model generates music in different meters including 4/4, 3/4 (Conditional Samples 3, 9), 6/8 (Conditional Samples 5, 10), 2/4 (Conditional Sample 7), and 2/2 (Unconditional Sample 1).
- **Triplets:** The model produces triplet notation using (3, such as (3CC^D, and (3E,G,G, (Unconditional Sample 8) and (3cccccccc (Conditional Sample 1).

5.2.1 Musical Coherence and Structure

The conditional samples show reasonable musical coherence. Conditional Sample 2 in G major (`GABcdzB,E|FGFABz3|`) demonstrates stepwise motion and stays within the key. The melody moves up (G-A-B-c-d) then resolves down, which is a common melodic pattern. Conditional Sample 3 in D major with 3/4 time (`DEFEF2FF-|FGFEGGAA-|`) shows waltz-like rhythm with longer notes on beats. The model respects key signatures to some extent. Conditional Sample 9 in G major uses ^c and ^f consistently, which are leading tones common in G major harmony.

5.2.2 Failure Modes

Pure unconditional generation often failed. Samples 4, 6, and 7 got stuck in repetitive loops like `z-496|z-496|...` and `z-428|z-428|....` Samples 12, 18, and 19 produced mostly rests (`z8|z8|...` or `z4z4|z4z4|...`). Sample 16 repeated the same low note pattern (`B,,,/2z/2B,,,z...`). These failures happen because the model was trained on random 256-token windows, which mostly contained mid-song content rather than headers which are only present in the beginning.

Even successful samples sometimes degrade. Conditional Sample 4 starts well (`ABce4-e3/2z/2|ABce3-e/2z3/2|`) but then produces long stretches of rests. Conditional Sample 10 begins with a melody (`EAcec'ec'e|bafg'e'c'B|`) but ends with repeated held chords (`[GB,]8|z8|z8|...`).

5.2.3 What the Model Learned

From the generated outputs, we can see that the model has learned several important musical patterns. It understands octave relationships, using commas for lower notes and apostrophes for higher ones. It also captures local melodic structure, such as stepwise motion and short arpeggios. In addition, the model shows rhythmic variety through different note lengths and occasional triplets, and it can produce basic chord voicings by stacking notes inside brackets.

However, the model still struggles with higher-level musical structure. It often cannot start a piece naturally without a prompt, and it does not clearly know when to end. As a result, many samples fade out into long rests or fall into repetitive loops instead of reaching a clear conclusion.

5.3 Challenges Encountered

Before discussing the challenges in detail, we note that many of the major design and methodological decisions made during this project have already been discussed across different sections of this report.

One of the main challenges we faced was our limited prior knowledge of music theory and ABC notation. Before starting any experiments, we had to spend a significant amount of time understanding how ABC notation is structured and what different symbols mean. To help with this, we relied on AI-based tools to interpret the notation and to assist in analyzing the outputs produced by our models. These tools helped speed up our learning and made it easier to reason about the results.

However, I want to clearly state that I carefully reviewed and attempted to verify all interpretations suggested by these AI systems myself. I made an effort to understand the generated music on my own and only included observations that I could reasonably justify within the limits of my time and knowledge. That said, it is still possible that I do not fully understand all aspects of the musical structure.

Another major challenge was limited computational resources. Even though we used paid online GPU services, the available compute was still not sufficient to run large-scale or extensive experiments. This constraint limited the size of the models we could train and the number of experiments we were able to conduct.

From an experimental standpoint, tokenization and vocabulary construction were particularly time-consuming due to the iterative nature of understanding the musical domain and refining tokenization strategies accordingly. Additionally, we encountered several issues during the conversion of ABC representations to MIDI format. While we were able to identify and resolve some of these issues, we were unable to fully stabilize the conversion pipeline within the project timeframe. Another big challenge is the generation with pure conditional approach. When starting with only the beginning token, the model produced invalid outputs like repeated rests or notes without headers.

5.4 Limitations and Future Work

The biggest limitation of our project is that we were not able to train the models for long enough. Although the full training dataset contains more than one billion tokens, our best model was trained on only about 200 million tokens. This was mainly due to limited computational resources. With more training on a larger portion of the dataset, the models would likely learn stronger musical structure and produce better outputs.

Another important limitation comes from our tokenization design. We used a custom, music-aware tokenization scheme where musical elements were manually defined and encoded. While this helped us inject domain knowledge into the model, this approach is static in nature and lack flexibility. Because the tokenization rules are fixed, the model cannot adapt or learn more efficient representations from the data itself.

For future work, it would be interesting to explore more flexible and data-driven tokenization methods. For example, techniques such as Byte Pair Encoding (BPE) or other subword-based approaches could allow the tokenizer to be learned automatically from the training data, rather than relying on hand-crafted rules. This may help the model discover better musical abstractions on its own.

Another factor to consider is that our scaling analysis is based on a small number of models. We trained only five models on symbolic music data, with parameter counts ranging from 3.7M to 157M. As a result, the estimated scaling exponents should be interpreted with caution. The power-law fit is based on limited data and may not generalize to larger model sizes. Understanding the true scaling behavior of music language models at larger scales remains an open question.

Overall, increasing the amount of training, experimenting with more dynamic tokenization methods, and training a larger number of models are clear directions for future work that could significantly improve the results of this project.

6 Conclusion

This project explored whether neural scaling laws, originally developed with keeping textual data in mind apply to music modeling. We trained decoder-only transformers and LSTMs of varying sizes on ABC notation converted from MIDI files and measured how validation loss decreases with model size.

Our results suggest that scaling laws do hold for music. Both architectures follow power law relationships between parameter count and loss. The transformer achieved $\alpha = 0.87$ and LSTM achieved $\alpha = 0.88$. This means that larger models consistently achieve lower loss following a power law relationship. What is surprising is how strongly music follows scaling laws compared to text. Kaplan et al. [1] found $\alpha \approx 0.076$ for language models on text, roughly 10 times smaller than our values. This suggests that music benefits more from larger models than text does. The reason could be that music has structure and patterns that text lacks. Text is different because the next word depends on world knowledge, context, and meaning, which are harder to capture just by adding more parameters.

Overall, our results suggest that scaling laws may provide a useful framework for music modeling, but further investigation at larger scales is needed.

References

[1] Tom Henighan Tom B. Brown Benjamin Chess Rewon Child Scott Gray Alec Radford Jeffrey Wu Dario Amodei Jared Kaplan, Sam McCandlish. Scaling laws for neural language models. *https://arxiv.org/pdf/2001.08361*, 2020.

A Generated Samples

Below are five example outputs generated by our best-performing model (XL Transformer, 156.8M parameters). The ABC notation can be rendered and played using online tools such as `https://abcjs.net/abcjs-editor.html`. All generated MIDI files are available in the results directory of the project repository along with other outputs. Please read the Readme file for more details about the project.

Sample Output 1

```
X:1
M:4/4
L:1/8
K:G
GABcdzB,E|FGFABz3|G/2F/2GA/2B/2GcdcB|
GABcdzB,G|FGABcdzB,/2z/2|GABcdzB,E|
FGFABzA,G|FGFABz3|G/2F/2GABcdzB,|
GABcdzB,E|FGFABzA,G|FGFABzG,A|GABcdzB,E|
FGFABzG,A|GABcdzB,E|FGFABzF,A|GABcdzB,E|
```

Sample Output 2

```
X:1
M:3/4
```

```
L:1/8
K:G
DGBdBGˆc|GBdBGBˆc|Gˆcgcgcˆf|dAdˆgdGˆc|
GBd4z|D3ˆFˆcdf|d2eddˆc|z8|d6-d/2z3/2|ˆCDADADB|ˆcAdADDB|
ˆcA-[e-A]2e/2Bzˆdgb|[dˆF-]3[ˆc-F]c2-c/2z3/2|EˆFAFAFB|ˆc8|z3ABD2z|
ddddˆfdfe|z8|d3ˆFˆcdf|d3Aˆdad|ˆcAdADDB|
```

Sample Output 3

```
X:1
M:6/8
L:1/8
K:E
EFGcBGA|=E8-|E/2z6z3/2|ABcedcdA|B8-|B/2z6z3/2|abc'd'c'd'c'b2|afe4-ec2|dcdbad'2z|abc'd'c'ba
d'2z2b2=g2|f8-|f4=E,,2c'2|=f6b2|
a6e2|_e3e2a2|=gfefe2zB|c'8-|
[c'-c]2c'6-c'3/2|d'8-|d'6a2|
=abc'd'c'b2b|[c'c]6z2|z8|z8|z8|z8|z8|z8|
z8|z8|z8|z8|z8|
```

Sample Output 4

```
X:1
M:4/4
L:1/8
K:C
CDEFGAcB,|ˆA,B,DGGDCA,|CDEFGAcB,|
C3/2z2z/2EFzG-|GF/2z/2F/2F/2AGCzD|
EFEDzEFz|(3ccccccccc|GG3/2z/2GGA/2z/
2A/2z/2A/2A/2|c/2z/2c/2z/2cc/2z/2edc/
2z/2ˆd/2c/2|G/2-[GE-]2E3/2zE/2E/2EEˆD|
CˆA,3/2z/2G,/2G,/2G,F,E,/2F,/2G,/2A,/2|
CCCC/2z/2C/2C/2CCz|E/2E/2E/2z/2C/2DCC3/2z3/2|
```

Sample Output 5

```
X:1
M:3/4
L:1/8
K:D
DEFEF2FF-|FGFEGGAA-|AGc_A2z2|EFGFG2FF-|
FE2FEFFG-|G_GFEEz2E|GAAAA2GG|GAGAA2GG|
GFGFE2G_A-|_A2GGGFFG-|G_GFEDEEE|
_GAA2AGGG|GF2GGGAA|AGAGGGFG-|
G_G2F2G3|z8|z8|z8|M:5/4GA|
GFGFE2GG-|G_A=AGAG2A|
```