RiskOptimix Music Transformer

Thomas van der Hulst

November 2025

1 Introduction

Music generation has been a challenge for some time, with fast enhancements in performance since the introduction of attention mechanisms (Vaswani et al., 2017). In 2018, researchers at Google introduced the Music Transformer (Huang et al., 2018), which demonstrated the ability to generate expressive piano performances. Built on the Transformer architecture (Vaswani et al., 2017), the Music Transformer adapted self-attention mechanisms to handle long sequences of notes, precise timing relationships, and the need to maintain musical coherence over extended periods. By introducing relative positional encodings, the model could better capture the temporal relationships, important to musical structure (Huang et al., 2018).

In this paper, I discuss my recreation of the Music Transformer. I tried to keep as close to the implementation done by the Google Researchers, but sometimes used more modern techniques for better training. The final trained model obtained a perplexity score of 6.91, an accuracy of 0.4299, and samples of the generated music by this model can be found on `https://music-transformer-portfolio.vercel.app/research`.

In the following sections, I will walk through the different components of this implementation: the data preprocessing pipeline that transforms MIDI files into learnable representations, the architectural decisions that make the Transformer suitable for music generation, the training process, and finally, the results and what they show about the model's musical understanding. This whole project was done out of my passion for data science and large machine learning models and my passion for piano music. I hope this document shows that this paper gives a glimpse into the amazing things I learned throughout this process.

A last note must be made that I in no shape or form claim this work to be my own. Firstly, the project is purely a recreation of the Google Music Transformer, with no newly added critical components. Secondly, as the code needs to be as efficient as possible to be able to train on home-used laptops, and this material is quite heavy, I worked together a lot with AI to implement the theory into code. For me this project was all about creating some unique and cool while learning through the process, and now I want to share what I learned with others. I am always open to feedback, ideas or critical notes. Feel free to contact me at t.van.der.hulst@student.vu.nl. Thank you!

2 Data Representation and Preprocessing

Before we can train the Music Transformer, we must address the challenge: how do we represent musical performances in a way that a neural network can process? Unlike text, where we have discrete words, or images, where we have pixels, music is continuous in time and can have multiple notes sounding simultaneously. The choices we make in representing music will profoundly affect what the model can learn and generate.

In this section, I discuss the event-based representation adopted from the original Music Transformer paper (Huang et al., 2018), and discuss the encoding and quantization strategies that transform MIDI files into learnable sequences.

2.1 Why event-based representation?

There are several ways to represent music for machine learning. The most straightforward is the pianoroll representation, where time is divided into fixed steps (e.g., every 10ms) and each step records which notes are active. While intuitive, this approach has a flaw for expressive piano music: it generates extremely long sequences.

Consider one minute of piano music with 10ms resolution. A pianoroll would require 6,000 time steps, and at each step we must represent up to 88 piano keys. For a piece with subtle timing variations and held notes, this becomes prohibitively long—far beyond what Transformers could handle in 2018, and still challenging today.

The event-based representation solves this by only encoding *when something changes*. Instead of representing every time step, we represent discrete events: a note starting, a note ending, time passing, or velocity changing. For the same minute of music, this might produce only 2,000–3,000 events, making it tractable for the Transformer to process while preserving the expressive timing information that makes human performances musical.

2.2 The token vocabulary

Following the Music Transformer paper, I implement a vocabulary of 388 tokens organized into four categories, plus three special tokens for training:

NOTE_ON events (tokens 0–127) Each of the 128 MIDI pitches has its own NOTE_ON token. When the model generates NOTE_ON(60), it starts playing middle C with the current velocity. This separate representation for each pitch allows the model to learn pitch-specific patterns.

NOTE_OFF events (tokens 128–255) Similarly, each pitch has a corresponding NOTE_OFF token that ends a sounding note. The separation of note starts and ends is crucial for representing polyphony—multiple notes can be active simultaneously, and they can end in any order.

TIME_SHIFT events (tokens 256–355) These 100 tokens represent forward time movements in 10ms increments, from 10ms (token 256) up to 1,000ms (token 355). When the model needs to advance time by more than 1 second, it emits multiple TIME_SHIFT tokens. This step of 10ms is sufficient to capture the subtle timing variations in expressive performances while keeping the vocabulary compact.

SET_VELOCITY events (tokens 356–387) The 32 velocity tokens set the dynamic level for subsequent NOTE_ON events. MIDI velocity ranges from 0–127, which we quantize into 32 bins. This quantization sacrifices some dynamic precision but dramatically reduces the vocabulary size while keeping most of the valuable information.

Special tokens Three additional tokens facilitate training: PAD_TOKEN (388) for padding sequences to uniform length, SOS_TOKEN (389) to mark sequence starts, and EOS_TOKEN (390) to mark sequence ends. The total vocabulary size is therefore 391 tokens.

2.3 From MIDI to tokens

The encoding process transforms a MIDI file into this token sequence through several steps:

2.3.1 Event extraction

First, we extract all note events from the MIDI file, filtering out drum tracks and notes shorter than 10ms (which are often performance artifacts or grace notes that would complicate learning). Each note becomes two events: a NOTE_ON at its start time and a NOTE_OFF at its end time.

These events are then sorted by time, with an important detail: when multiple events occur at the same time, NOTE_OFF events are placed before NOTE_ON events, and events of the same type are sorted by pitch. This ordering ensures consistent tokenization and avoids ambiguities in polyphonic passages.

2.3.2 Time quantization

An important aspect of the encoding is time quantization. We cannot represent arbitrary floating-point times. We must round them to multiples of our 10ms resolution. But if we are not careful, small rounding errors can accumulate, causing the decoded timing to drift away from the encoded timing.

To prevent this, I track a `quantized_time` variable that accumulates the actual quantized time shifts, rather than using the original event times directly. When encoding a time shift, I:

1. Calculate the desired shift: `shift_ms = (event_time - quantized_time) * 1000`
2. Quantize to the nearest 10ms: `shift_ms = round(shift_ms / 10) * 10`
3. Emit TIME_SHIFT token(s) for this quantized shift
4. Update: `quantized_time += shift_ms / 1000`

This ensures that encoding followed by decoding preserves timing exactly. Without this careful tracking, rounding errors would compound over a minute-long piece, potentially shifting events by hundreds of milliseconds.

2.3.3 Velocity quantization

MIDI velocity values (0–127) are quantized into 32 bins using the formula:

$$\text{bin} = \left\lfloor \frac{\text{velocity}}{128} \times 32 \right\rfloor$$

During decoding, we reconstruct the velocity as the bin center:

$$\text{velocity} = (\text{bin} + 0.5) \times \frac{128}{32}$$

This means bin 0 represents velocities 0–3 and decodes to velocity 2, bin 1 represents 4–7 and decodes to 6, and so on. The quantization introduces some loss of dynamic nuance, but 32 levels prove sufficient for capturing expressive variation.

2.4 Sequence generation example

To illustrate the encoding process, consider a simple C major chord (C4, E4, G4) played at time 0 and held for 1 second, all with velocity 80:

1. **SET_VELOCITY(80)**: Velocity 80 $\rightarrow$ bin 20 $\rightarrow$ token 376
2. **NOTE_ON(60)**: Middle C $\rightarrow$ token 60
3. **NOTE_ON(64)**: E4 $\rightarrow$ token 64
4. **NOTE_ON(67)**: G4 $\rightarrow$ token 67
5. **TIME_SHIFT(1000ms)**: Token 355 (the maximum single shift)
6. **NOTE_OFF(60)**: Token 188
7. **NOTE_OFF(64)**: Token 192
8. **NOTE_OFF(67)**: Token 195

This 8-token sequence encodes a full second of music.

2.5 Data augmentation

While the MAESTRO dataset provides over 200 hours of piano performances, this is nothing in comparison to the billions of tokens used to train modern language models. To maximize the effective size of our training data, we use data augmentation—applying transformations to existing examples that preserve their musical essence while creating variation.

The Music Transformer paper specifies two forms of augmentation: pitch transposition and time stretching. These are applied stochastically during training, meaning each time we load a MIDI file, we randomly apply one of several possible transformations. This simple technique can increase the diversity of training examples without requiring explicit storage of augmented versions.

2.5.1 Pitch transposition

Pitch transposition shifts all notes up or down by a fixed number of semitones. A melody often remains recognizable when played in a different key. This property makes transposition an ideal augmentation: we create genuinely different performances while maintaining the musical structure the model should learn.

Following the original paper, I transpose by values uniformly sampled from the set $\{-3, -2, -1, 0, 1, 2, 3\}$ semitones. This range allows transposition up to a minor third in either direction—enough to create meaningful variation without straying too far from the original key. A value of 0 means no transposition, which occurs $\frac{1}{7} \approx 14\%$ of the time.

Implementation details Transposition is straightforward: for each note, we add the transposition value to its MIDI pitch number. However, MIDI pitch is constrained to the range [0, 127], representing about seven octaves. If transposition would push a note outside this range, we clip it to the boundary:

$$\text{new\_pitch} = \text{clip}(\text{pitch} + \text{semitones}, 0, 127)$$

This clipping primarily affects extreme registers. If the original performance contains notes near the top of the piano's range (around MIDI pitch 108, the highest note on an 88-key piano), transposing up might clip some notes to the maximum. In practice, such notes are rare in the MAESTRO dataset, so clipping occurs infrequently.

2.5.2 Time stretching

Time stretching uniformly scales all time values by a constant factor. A factor of 1.05 means everything happens 5% slower; 0.95 means 5% faster. Unlike transposition, time stretching does not have an obvious musical analog—we cannot simply play a piece faster and expect it to sound identical. However, modest time stretches preserve the relative timing that defines musical structure: a sixteenth note followed by an eighth note maintains that 1:2 ratio regardless of the overall tempo.

The paper specifies time stretch factors uniformly sampled from $\{0.95, 0.975, 1.0, 1.025, 1.05\}$. This provides a $\pm 5\%$ range around the original tempo, which is subtle enough that the stretched version still sounds like a reasonable performance of the same piece, but different enough to prevent the model from memorizing exact timings.

What gets stretched? Time stretching affects every temporal element in the MIDI file:

- **Note start times**: If a note began at 1.0 seconds, with factor 1.05 it now begins at 1.05 seconds
- **Note end times**: Durations stretch proportionally with start times
- **Control changes**: This includes sustain pedal events, which must stretch to remain synchronized with the notes they affect
- **Pitch bends**: Though rare in piano music, these are also stretched for completeness

The implementation applies the stretch factor as a simple multiplication: $\text{time}_{\text{new}} = \text{time}_{\text{old}} \times \text{factor}$. This preserves all relative timing relationships—if two notes were 100ms apart, they remain 100ms apart after stretching (measured in the original time units).

2.5.3 Why augmentation works

The effectiveness of data augmentation rests on a subtle insight: we want the model to learn musical structure, not memorize specific performances. If every C major scale the model sees is in exactly the same key at exactly the same tempo, it might overfit to those specific characteristics rather than learning the general pattern of scales.

By training on transposed and time-stretched versions, we force the model to learn invariant representations. It cannot simply memorize "middle C followed by D," because sometimes that pattern appears as D followed by E (after transposition), and sometimes the timing between them varies (after stretching). The model must learn the abstract relationship—an ascending major second interval—rather than the surface-level details.

This type of augmentation is particularly powerful for music because musical structure is fundamentally relational. Melodies are defined by intervals, not absolute pitches. Rhythms are defined by proportions, not absolute durations. Our augmentation exploits this by varying the surface details while preserving the underlying relationships.

2.5.4 Stochastic application during training

Importantly, augmentation is applied *stochastically during training*, not as a preprocessing step. When the data loader fetches a MIDI file, it:

1. Loads the original file from disk
2. Randomly samples a transposition from $\{-3, -2, -1, 0, 1, 2, 3\}$
3. Randomly samples a time stretch from $\{0.95, 0.975, 1.0, 1.025, 1.05\}$
4. Applies both transformations
5. Encodes the augmented MIDI into tokens

This means the model sees a different version of each piece every epoch. A particular training example might appear as the original in epoch 1, transposed up 2 semitones in epoch 2, transposed down 1 semitone and stretched to 1.05x in epoch 3, and so on. With 7 transposition options and 5 stretch options, we have $7 \times 5 = 35$ possible augmented versions of each original file.

The stochastic approach has two advantages over preprocessing all augmentations upfront. First, it requires no additional storage as we keep only the original files. Second, it provides effectively infinite variation: since augmentation is sampled randomly, the model never sees exactly the same combination twice (or at least, not until it has seen all 35 possibilities).

2.5.5 Augmentation and sustain pedal interaction

Augmentation must be applied *after* sustain pedal preprocessing. The time stretching affects control change times, so we need the sustain periods to stretch along with the notes they modify. If we applied augmentation before sustain preprocessing, the stretched pedal timings would be out of sync with the stretched notes.

The preprocessing pipeline therefore follows this order:

1. Load MIDI file
2. Apply sustain pedal preprocessing (extend note durations)
3. Apply random augmentation (transposition and time stretching)
4. Encode to tokens

This ensures that all temporal relationships remain consistent throughout the transformations.

2.6 The MAESTRO dataset

Having discussed the encoding scheme, and augmentation strategies, we now need actual data to train on. For this project, I use the MAESTRO dataset (MIDI and Audio Edited for Synchronous TRacks and Organization), a dataset of approximately 200 hours of virtuosic piano performances captured from the International Piano-e-Competition.

2.6.1 Two encoding strategies

The dataset supports two different approaches to augmentation and encoding, controlled by the `preencode_train` parameter:

Pre-encoded mode (default) In this mode, all MIDI files are loaded, preprocessed, and encoded once during dataset initialization. The encoded token sequences are stored in memory. During training, each call to `__getitem__` simply indexes into these pre-encoded sequences and extracts a window of tokens.

This approach is fast—fetching a training example requires only array indexing and slicing. However, it cannot apply the full MIDI-level augmentation (transposition and time stretching) because those transformations must happen before encoding.

To still provide augmentation benefit, I implement *token-level pitch shifting*: after extracting a sequence of tokens, we shift all NOTE_ON and NOTE_OFF tokens by a random number of semitones. This is much cheaper than re-encoding the MIDI file and provides similar benefits for pitch variation (though not for timing variation).

Token-level shifting works by adding an offset to the token IDs:

$$\begin{gathered}\text{NOTE\_ON}(60) + 2 \to \text{NOTE\_ON}(62) \\ \text{token } 60 + 2 \to \text{token } 62\end{gathered}$$

Before applying the shift, we verify that all resulting pitches would remain in [0, 127]. If any note would go out of bounds, we skip the shift for that sequence rather than clipping, preventing the distribution of pitches from becoming skewed toward the boundaries.

On-the-fly mode When `preencode_train=False`, the dataset stores only the list of MIDI filenames. Each call to `__getitem__` loads the MIDI file from disk, applies random augmentation (both transposition and time stretching), preprocesses it, and encodes it to tokens—all on the fly.

This approach provides maximum augmentation variety: every single training example is genuinely different, with both pitch and timing variations. However, it is significantly slower. Loading and parsing MIDI files takes time, and encoding thousands of notes per file adds overhead.

The choice between these modes involves the tradeoff: speed vs. diversity. For this project, I default to pre-encoded mode with token-level pitch shifting, which provides a good balance: we get pitch variation (the more important of the two augmentations) while maintaining fast training iteration.

2.6.2 Random cropping vs. sequential chunks

Another decision is how to extract sequences from pieces. I implement two strategies:

Random cropping (training) During training, we sample random windows from each piece. If a piece has 10,000 tokens and we need 2,048-token sequences, we randomly choose a start index in the range [0, 10000 - 2048 - 1] and extract tokens from that position.

The random cropping serves two purposes. First, it provides additional variation as the model sees different sections of pieces in different orders across epochs. Second, it efficiently utilizes all the data: even a 3,000-token piece can provide multiple diverse training examples by sampling different windows.

Sequential chunks (validation) For validation and testing, we use non-overlapping sequential windows. A 10,000-token piece is divided into chunks: [0:2048], [2048:4096], [4096:6144], and so on. This ensures consistent evaluation where the same piece always produces the same validation sequences, allowing us to meaningfully compare perplexity across training runs.

2.6.3 Creating autoregressive training pairs

The Transformer is trained autoregressively: given tokens $[x_0, x_1, \ldots, x_{t-1}]$, predict x_t. To enable this, each training example must provide both inputs and labels.

The implementation extracts sequences of length `seq_length + 1` tokens, then splits them:

- **Input**: tokens [0, 1, 2, ..., seq_length - 1]
- **Label**: tokens [1, 2, 3, ..., seq_length]

This creates the proper alignment: the model receives position i as input and must predict position $i+1$ as output. During training, the loss is computed between the model's predictions and the label sequence.

The causal masking in the Transformer ensures that when predicting position i, the model can only attend to positions $[0, \ldots, i-1]$, not to positions $\geq i$. This makes training consistent with generation: at generation time, the model only has access to previous tokens.

2.6.4 The complete data pipeline

Putting everything together, here is the full pipeline for a training example:

1. **Initialization** (once per training run):
    - Load MAESTRO metadata
    - Filter by split (train/validation/test) and optionally by year
    - If pre-encoding: load all MIDI files, apply sustain preprocessing, encode to tokens
    - Create sequence indices (mappings from sequence number to piece and position)
2. **Per training example**:
    - Look up which piece and position this sequence comes from
    - If pre-encoded: extract tokens from stored array
    - If on-the-fly: load MIDI, apply augmentation, preprocess, encode
    - Optionally apply token-level pitch shift (pre-encoded mode only)
    - Extract window of `seq_length + 1` tokens
    - Split into input [0:seq_length] and labels [1:seq_length+1]
    - Return as PyTorch tensors

This pipeline transforms raw MIDI files into the tokenized sequences needed for training, while providing flexibility to balance speed and augmentation diversity based on available compute resources.

With this foundation in place, we now discuss the model architecture and training process that will learn to generate music from this prepared data.

3 The Transformer model

The Music Transformer builds upon the standard Transformer decoder architecture, but introduces a modification: relative positional self-attention. In this section, I will explain how this mechanism works and why it is helpful for generating music.

3.1 Why relative attention matters for music

The original Transformer architecture uses absolute positional encoding, either learned embeddings or sinusoidal functions that tell the model the position. While this works well for many tasks, in music relative positions are more important: musical patterns are defined by intervals and distances, not absolute positions. The Music Transformer recognizes this by learning to pay attention to *how far apart* things are, rather than *where* they are absolutely.

3.2 Relative position self-attention

In the Music Transformer, self-attention is extended to incorporate this relative position information. In standard self-attention, we compute the attention scores between queries and keys using a dot product:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where Q, K and V are the queries, keys and values derived from the input sequence. This captures the context-based relationships and thus how similar query i is to key j based on their features.

In the Music Transformer, a relative position term is added:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T + S^{rel}}{\sqrt{d_k}}\right)V.$$

The term S^{rel} is a matrix where entry (i, j) represents how much attention query i should pay to position j based on their relative distance $(j - i)$. This is computed by learning a separate set of embeddings E^r for each possible relative distance, then combining them with the queries.

For each head of attention, we learn embeddings for relative distances ranging from $-(L - 1)$ to $+(L - 1)$, where L is the sequence length. Following the original Music Transformer, we clip this to a maximum distance of 1024 to keep the number of parameters manageable. Each relative distance gets its own d_k-dimensional embedding, learned separately for each attention head. This allows different heads to specialize in different types of temporal relationships.

3.3 The memory-efficient skewing algorithm

The approach above is extremely useful, but there is a computational challenge. The straightforward way to compute S^{rel} would be to create a tensor R of shape (L, L, d_k) that contains the relative position embedding for every possible query-key pair. Then we would compute $Q \cdot R$ to get S^{rel}. But this requires $O(L^2 d_k)$ memory, which is prohibitive for a sequence length of 2048 and embedding dimension of 64.

The Music Transformer paper introduced a skewing algorithm that reduces this to $O(L \cdot d_k)$ memory while computing the same result. The insight here is that we do not actually need to construct the full R tensor. Instead we can directly compute $Q \cdot E^r$, giving us a matrix of shape $(L, 2L - 1)$ where entry (i, r) contains the dot product of query i with relative embedding r. We then rearrange this matrix so that entry (i, j) contains the dot product of query i with the embedding for relative distance $(j - i)$. This is exactly what we need for S^{rel}. The skewing algorithm, discussed in Huang et al. (2018), uses padding, reshaping and slicing operations to achieve this rearranging.

3.4 Multi-head attention with relative positions

The full attention mechanism uses multiple attention heads in parallel, just like the standard Transformer (Vaswani et al., 2017). Each head has its own query, key and value projections and its own set of relative position embeddings. This allows the model to attend to different aspects simultaneously.

3.5 Causal masking for autoregressive generation

The Music Transformer is an autoregressive model, meaning it generates music one event at a time, conditioning on all previous events. To enforce this during training, we apply a causal mask that prevents position i from attending to any position $j > i$.

The causal mask is efficiently created as an upper triangular boolean matrix and used across batches and heads. When applied, masked positions get attention logits of $-\infty$, which become zero after the softmax (as $\exp(-\infty) = 0$), removing those positions from consideration.

3.6 Feed-Forward Networks

To give the model additional representational power, each attention layer is followed by a position-wise feed-forward network (FFN). The function of the FFN is given by:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2.$$

This consists of two linear transformations with a ReLU activation in between. The same network is applied independently to each position in the sequence. While attention mixes information across positions, the FFN processes each position's representation individually, allowing the model to perform complex non-linear transformations on the attended information.

In my implementation, I follow the standard Transformer practice of setting the inner dimension d_{ff} to four times the model dimension. With $d_{model} = 512$, this means $d_{ff} = 2048$. This expansion and contraction $(512 \to 2048 \to 512)$ gives the model a high-dimensional space to work with in the middle.

The FFN also includes dropout after the activation, which is set to 0.1. This regularization helps prevent overfitting, which is particularly important when training on a relatively small dataset such as MAESTRO.

3.7 The decoder block

A single layer of the Music Transformer consists of two sub-layers, each wrapped with a residual connection and layer normalization:

1. Relative position self-attention (described in Subsections 3.1–3.5)
2. Position-wise feed-forward network (described in Subsection 3.6)

The original Transformer paper used Post-LN normalization where layer normalization is applied *after* the residual connection:

$$x = \text{LayerNorm}(x + \text{Sublayer}(x))$$

However, more recent research has shown that Pre-LN normalization where layer normalization is applied *before* the sub-layer can lead to more stable training:

$$x = x + \text{Sublayer}(\text{LayerNorm}(x)),$$

which is why this type of normalization is chosen for this model.

4 The complete architecture

With all the individual components defined, we can now assemble the complete Music Transformer. The full model is essentially a stack of decoder blocks, bookended by input and output layers that handle the conversion between discrete tokens and continuous representations.

4.1 From tokens to embeddings

Music, in our representation, is a sequence of discrete events: `NOTE_ON(60)`, `TIME_SHIFT(50)`, `NOTE_OFF(60)`, and so on. The first step is converting these discrete tokens into continuous vectors that the neural network can process.

This is accomplished with the embedding layer, a lookup table that maps each token in our vocabulary to a learned d_{model}-dimensional vector. With a vocabulary size of 388 tokens and $d_{model} = 512$, this embedding matrix has shape $(388, 512)$.

After looking up the embeddings, they are scaled by $\sqrt{d_{model}}$ following the original Transformer paper.

4.2 Stacking the decoder blocks

After embedding, the representation passes through a stack of decoder blocks. Each block applies:

1. Relative position self-attention
2. Position-wise feed-forward network

Both with residual connections and layer normalization. Matching the original Music Transformer configuration, 6 layers are used.

The output of layer i becomes the input to layer $i + 1$, allowing the model to build increasingly abstract representations.

4.3 Output projection and weight tying

The final step is converting the continuous representations back to discrete token predictions. This is done with a linear projection from d_{model} to $vocab\_size$, giving us logits for each token in the vocabulary. Applying softmax to these logits yields a probability distribution over the next token.

To reduce the number of parameters and improve generalization, I employ weight tying between the input embedding matrix and this output projection matrix. This technique, introduced in language modeling, leverages the intuition that tokens that are similar in the embedding space should also be similarly likely as outputs.

4.4 Autoregressive generation

During inference, we generate music autoregressively. This means one token at a time, conditioning on all previously generated tokens. The process is given by:

1. Start with a primer sequence (either empty or user-provided)
2. Run the model forward to get logits for the next token
3. Sample from the probability distribution (with optional temperature/top-k/top-p adjustments)
4. Append the sampled token to the sequence
5. Repeat until we reach the desired length

The sampling strategies of step 3 are given by:

- Temperature scaling: Dividing logits by temperature T before softmax. Lower temperature $(T < 1)$ makes the distribution sharper, preferring high-probability tokens (more conservative). Higher temperature $(T > 1)$ flattens the distribution, giving lower-probability tokens more chance (more exploratory).
- Top-k sampling: Only sample from the k most likely tokens. This prevents the model from selecting very unlikely tokens that might break musical coherence.
- Nucleus (top-p) sampling: Dynamically determine the sampling pool by taking the smallest set of tokens whose cumulative probability exceeds p. This adapts to the model's confidence. When it is certain, the pool is small; when uncertain, the pool is larger.

4.5 Model size and implementation details

The final configuration uses:

- Model dimension: $d_{model} = 512$
- Layers: 6 decoder blocks
- Attention heads: 8 per block
- Feed-forward dimension: $d_{ff} = 2048$ (4 times model dimension)
- Maximum relative distance: 1024
- Vocabulary size: 388 tokens

Xavier uniform initialization is used for all weight matrices. This helps ensure activations and gradients have reasonable magnitudes at the start of training. Dropout is applied at 0.1 throughout—in attention weights, feed-forward networks, and residual connections. This helps prevent overfitting.

With this complete architecture, we have a powerful model capable of learning long-range dependencies in music. The combination of relative attention (for capturing musical structure), multi-head attention (for learning different aspects of music), deep stacking (for hierarchical representations), and careful normalization (for training stability) creates a system that can generate coherent and expressive piano performances.

In the next section I discuss the training process that enables this model to learn from data.

5 Training the Music Transformer

With the architecture in place, we now face the challenge of actually training the model. Training a Transformer with approximately 25 million parameters on musical sequences requires careful attention to optimization, learning rate scheduling and regularization. In this section I discuss the decisions and techniques that were used for this model.

5.1 Optimization and learning rate scheduling

The challenge is that training these types of models is sensitive to the learning rate. Too high, and the optimization diverges or gets stuck in poor local minima. Too low, and training takes prohibitively long or stalls entirely. Rather than using a fixed learning rate throughout training, we therefore use a learning rate schedule that adapts over time. The original Transformer paper introduced a schedule that has become standard:

$$\text{lr} = d_{model}^{-0.5} \cdot \min(\text{step}^{-0.5}, \text{step} \cdot \text{warmup\_steps}^{-1.5}).$$

This schedule has two distinct phases. In the warmup phase ($\text{step} < \text{warmup\_steps}$), the learning rate increases linearly from near-zero to its peak value. During the first few thousand steps, the model's parameters are randomly initialized and the gradients can be unstable—too large in some directions, too small in others. A small learning rate during warmup lets the model find a reasonable region of parameter space before we accelerate training.

During the decay phase ($\text{step} \geq \text{warmup\_steps}$), the learning rate decreases proportionally to $1/\sqrt{\text{step}}$. This inverse square root decay is gentler than exponential decay and allows the model to continue learning for longer.

5.2 Optimizer choice: Adam and AdamW

For the optimizer itself, there are two commonly used choices for Transformers: Adam and AdamW.

Adam (Adaptive Moment Estimation) is the standard choice for Transformers. It adapts the learning rate for each parameter based on estimates of first and second moments of the gradients. The formula for the first moment is:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t,$$

and the formula for the second moment is:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2,$$

where g_t is the gradient at step t, m_t tracks the moving average of the gradients (momentum), and v_t tracks the moving average of the squared gradients (variance). This means parameters that receive consistent gradient signals get larger updates, while noisy parameters get smaller updates.

To correct for the initialization bias toward zero, we apply bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t},$$

and update the parameters by:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

The hyperparameters used are:

- $\beta_1 = 0.9$ (momentum for first moment)
- $\beta_2 = 0.999$ (momentum for second moment)
- $\varepsilon = 10^{-8}$ (for numerical stability)

AdamW (Loshchilov and Hutter, 2017) modifies this by decoupling weight decay from the gradient-based updates:

$$\theta_t = \theta_{t-1} - \alpha \cdot \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} + \lambda \cdot \theta_{t-1} \right),$$

where λ is the weight decay coefficient. The weight decay is applied directly to the parameters, not mixed with the adaptive learning rate adjustments. This leads to better generalization in practice. While the original Music Transformer used the Adam optimizer, I used the AdamW optimizer with a weight decay of $\lambda = 0.01$, as this provided better performance during training.

5.3 Training implementation

With the optimizer and learning rate schedule defined, we can now discuss the practical implementation of the training loop. Training a large model requires several techniques to ensure stability, efficiency, and reproducibility.

5.3.1 Gradient accumulation

GPU memory limits the batch size we can use. For example, with a sequence length of 2048 and model dimension of 512, a single training example requires substantial memory. Gradient accumulation allows us to simulate larger batch sizes by accumulating gradients over multiple forward-backward passes before updating the weights.

If we want an effective batch size of 32 but can only fit 8 examples in GPU memory, we perform 4 forward-backward passes (accumulating gradients each time) before calling the optimizer step. The loss is divided by the number of accumulation steps to ensure the gradient magnitudes remain consistent with true batch training. This technique allows training with batch sizes that would otherwise be impossible on available hardware.

5.3.2 Gradient clipping

Deep networks can sometimes experience exploding gradients, where gradients become extremely large and cause the optimization to diverge. To prevent this, we apply gradient clipping, which scales down gradients if their global norm exceeds a threshold. The clipped gradient is given by:

$$\mathbf{g}_{\text{clipped}} = \min\left(1, \frac{\tau}{\|\mathbf{g}\|}\right) \cdot \mathbf{g},$$

where τ is the clipping threshold and $\|\mathbf{g}\|$ is the L2 norm of all gradients. A typical value is $\tau = 1.0$, which prevents any single update from being too large while allowing normal gradients to pass through unchanged.

5.3.3 Validation and early stopping

To monitor the model's generalization, we periodically evaluate on a held-out validation set. This serves two purposes: tracking learning progress and implementing early stopping to prevent overfitting. Early stopping monitors the validation loss and stops training if it fails to improve for a specified number of evaluations (the patience parameter). This prevents wasting computational resources on training that no longer improves the model.

The validation loop computes the same metrics as training (loss, perplexity, and accuracy) but without updating model parameters. These metrics provide insight into whether the model is learning meaningful patterns or simply memorizing the training data.

5.3.4 Checkpointing

During long training runs, it is important to save checkpoints regularly. Each checkpoint contains the model weights, optimizer state, learning rate scheduler state, current training step, and best validation loss. This allows us to:

- Resume training if interrupted
- Roll back to earlier checkpoints if training diverges
- Keep the best model based on validation performance
- Analyze how the model evolved during training

Checkpoints are saved at regular intervals (e.g., every 1000 steps) and whenever the model achieves a new best validation loss. The best checkpoint is maintained separately to ensure we can always recover the highest-quality model.

5.3.5 Sample generation during training

To qualitatively assess the model's progress, we periodically generate music samples during training. Samples are generated using a primer sequence from the validation set, allowing us to see how the model continues existing musical patterns.

These samples are particularly valuable for debugging. If the model achieves low loss but generates incoherent music, it may indicate issues with the token representation or generation strategy. Conversely, musically pleasing samples with moderate loss suggest the model has learned meaningful patterns.

5.3.6 Metrics and logging

Throughout training, we track three key metrics:

- **Loss**: The cross-entropy loss measuring prediction error

- **Perplexity**: Computed as $\exp(\text{loss})$, representing the model's uncertainty per token
- **Accuracy**: The fraction of correctly predicted tokens

These metrics are logged to both TensorBoard (for local visualization) and Weights & Biases (for cloud-based tracking and collaboration). TensorBoard provides real-time plots during training, while Weights & Biases enables comparing multiple experiments and sharing results.

Lower perplexity indicates the model is more confident and accurate in its predictions. For reference, the original Music Transformer achieved a perplexity of around 6.4 on the Piano-e-Competition dataset. Tracking these metrics over time reveals whether the model is learning, overfitting, or has converged.

5.4 Model specifications and configurations

To facilitate experimentation and accommodate different hardware constraints, I implemented three distinct model configurations: a tiny configuration for fast prototyping and debugging, a medium configuration for training on home-type GPUs, and a full configuration that exactly replicates the original Music Transformer architecture from Huang et al. (2018).

5.4.1 Configuration philosophy

The three configurations maintain architectural proportions while scaling model capacity. Each configuration preserves the relationship $d_{ff} = 4 \times d_{model}$ and sets max_rel_dist to half the sequence length, following the paper's design. The configurations differ primarily in model dimension, number of layers, and sequence length, which are the main factors affecting both model capacity and memory requirements.

5.4.2 Configuration comparison

Table 1 presents a detailed comparison of the three configurations. The full configuration matches the original paper exactly, achieving approximately 25 million parameters. For this project, I trained the full configuration to replicate the paper's results and demonstrate the model's capability to generate expressive piano music.

6 Results

After implementing the complete pipeline—from data preprocessing through model architecture to training—we can now examine how well the Music Transformer learns to generate piano music. In this section, I present the training dynamics, final evaluation metrics, and provide access to generated samples through an interactive web demo.

6.1 Training dynamics

The model was trained for 7,000 steps on the MAESTRO dataset, with training taking approximately 18.4 hours on a single GPU. Figure 1 shows the evolution of key metrics throughout training.

Loss and perplexity The training loss exhibits the characteristic steep descent of neural network optimization, dropping from an initial value around 0.6–0.7 to approximately 0.25 by step 7,000. The corresponding perplexity—computed as $\exp(\text{loss})$—falls dramatically from around 1,000 (indicating near-random predictions initially) to below 10, suggesting the model has learned meaningful patterns in the music.

The perplexity value is particularly interpretable: a perplexity of 7 means that, on average, the model is about as uncertain as if it were choosing uniformly among 7 tokens at each position. Given our vocabulary of 391 tokens, this represents substantial learning—the model has gone from being completely confused to having strong preferences about what should come next.

Table 1: Model configuration comparison. The full configuration exactly matches the original Music Transformer paper (Huang et al., 2018).

Parameter	**Tiny**	**Medium**	**Full (Paper)**
Architecture			
Model dimension (d_{model})	128	256	512
Number of layers	2	4	6
Attention heads	4	8	8
Feed-forward dimension (d_{ff})	512	1024	2048
Max relative distance	256	512	1024
Training			
Sequence length	256	1024	2048
Batch size	4	4	2
Gradient accumulation	1	4	8
Effective batch size	4	16	16
Warmup steps	100	2000	4000
Data			
MAESTRO years	2018 only	2015–2018	All years
Max pieces	5	All	All
Augmentation	No	Yes	Yes
Optimization			
Optimizer	Adam	AdamW	AdamW
Learning rate schedule	Constant	Transformer	Transformer
Peak learning rate	0.001	1.0	1.0
Weight decay	0.01	0.01	0.01
β_1 / β_2	0.9 / 0.999	0.9 / 0.98	0.9 / 0.98
Gradient clipping	1.0	1.0	1.0

Accuracy Token-level accuracy starts near 5–10% (slightly better than random chance, as some tokens like TIME_SHIFT are more common than others) and climbs steadily to approximately 42% by the end of training. This means the model correctly predicts the exact next token nearly half the time.

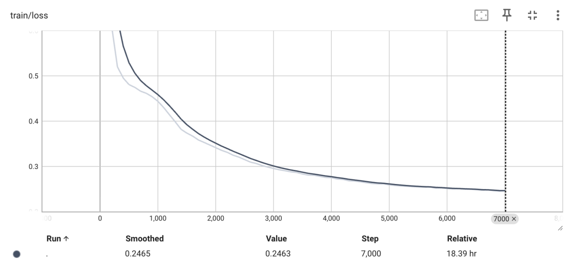
It is important to note that 42% accuracy, while it might seem modest, is actually quite strong for music generation. Music is inherently multi-modal—there are often many valid continuations at any given point. A melody might continue upward or downward, a harmony might resolve or extend, and expressive timing variations mean the exact duration is somewhat flexible. The model need not match the training data exactly to produce musically convincing output.

Learning rate schedule The learning rate curve clearly shows the warmup and decay schedule discussed in Section 4.1. The rate increases linearly from near-zero to its peak of approximately 5×10^{-4} around step 4,000 (the warmup phase), then gradually decreases according to the inverse square root schedule. This shape is characteristic of Transformer training and helps ensure stable optimization—the warmup prevents early instability, while the decay allows the model to fine-tune its predictions as training progresses.

6.2 Final evaluation metrics

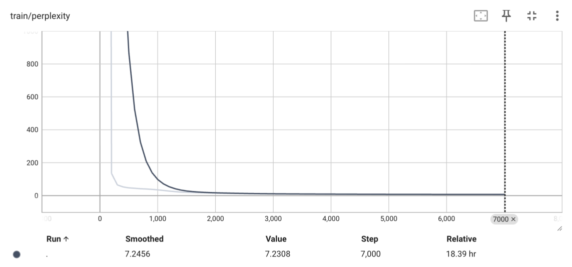
After training, I evaluated the model on the validation set to measure its generalization performance. The validation metrics are given in Table 2.

The validation perplexity of 6.91 is remarkably close to the training perplexity of 7.23, indicating that the model generalizes well without significant overfitting. This shows the effectiveness of the data augmentation



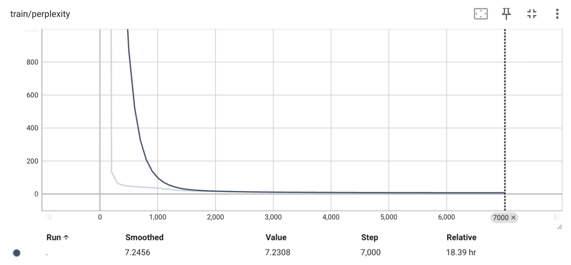
Run ↑	Smoothed	Value	Step	Relative
● .	0.2465	0.2463	7,000	18.39 hr

(a) Training loss



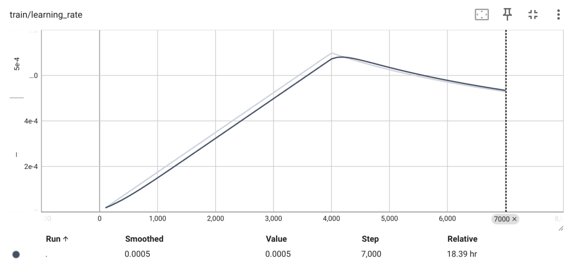
Run ↑	Smoothed	Value	Step	Relative
● .	7.2456	7.2308	7,000	18.39 hr

(b) Training perplexity



Run ↑	Smoothed	Value	Step	Relative
● .	7.2456	7.2308	7,000	18.39 hr

(c) Training accuracy



Run ↑	Smoothed	Value	Step	Relative
● .	0.0005	0.0005	7,000	18.39 hr

(d) Learning rate schedule

Figure 1: Training dynamics over 7,000 steps. All curves show smoothed values with the raw values shown in lighter colors.

Metric	Value
Loss	1.9232
Perplexity	6.91
Accuracy	42.99%

Table 2: Validation set performance after 7,000 training steps.

strategy and the regularization techniques (dropout, weight decay) employed during training.

For context, the original Music Transformer paper reported a validation perplexity of approximately 6.4 on the Piano-e-Competition dataset. While direct comparison is difficult due to different datasets (MAESTRO vs. Piano-e-Competition) and potentially different training durations, achieving a perplexity in the same ballpark suggests that this implementation successfully replicates the paper's approach.

6.3 Qualitative assessment: generated samples

Ultimately, the quality of a music generation model must be judged by listening to its output. Quantitative metrics like perplexity tell us whether the model has learned statistical patterns, but they cannot capture musical coherence, expressiveness, or aesthetic appeal.

I have created an interactive web demo where you can listen to samples generated by this model:

`https://music-transformer-portfolio.vercel.app/`

When listening to the samples, several qualities emerge that the metrics alone cannot capture.

Limitations Of course, the model is not perfect. Occasionally, samples may wander aimlessly without clear direction, or repeat short patterns excessively. Very long generations sometimes lose coherence, suggesting that even with relative attention, modeling structure at the largest scales remains challenging.

6.4 Summary

The trained Music Transformer achieves strong quantitative results (validation perplexity of 6.91, accuracy of 43%) and generates musically coherent piano performances. The training curves show healthy learning dynamics with good generalization, and the generated samples sound reasonable for AI-generated music.

These results showcase the ideas from the original paper: that self-attention with relative positional encodings is well-suited for music generation, that the event-based representation efficiently captures expressive performance, and that Transformers can learn to generate minute-long compositions with compelling musical structure.

I encourage readers to visit the demo website and listen to the samples themselves. After all, the true measure of a music generation system is how it sounds, not just what the metrics say. Thank you for reading this paper. If you have any questions, comments or suggestions, always feel free to reach out.