A Symbolic Music Transformer for Real-Time Expressive Performance and Improvisation

SPROJ Midway Paper

Arnav Shirodkar
Dec 2023

Bard College

Computer Science

Chapter 1

Introduction

With the widespread proliferation of AI technology, deep architectures — many of which are based on neural networks — have advanced the state of the art in a variety of different research areas and applications. Within the relatively new domain of Music Information Retrieval (MIR), deep neural networks have also been successful for a variety of tasks, including tempo estimation, beat detection, genre classification, etc. CITE

Drawing inspiration from projects like George E. Lewis's *Voyager*[**Lewis:1**] and Al Biles's *GenJam*[**Biles:1**] two pioneering endeavors in human-computer interaction, this research proposes the use of transformer models to create a real-time musical improvisation companion and demonstrate the potential of AI to enhance the human experience of music through interdisciplinary exploration. The Transformer, a groundbreaking model introduced in the 2017 paper "Attention is All You Need" by Vaswani et. al, [**Vaswani:1**] is at the core of Large Language Models (LLMs) like ChatGPT that have exploded in popularity around the world. We hope to use a transformer architecture

This paper discusses the historical context for the rise of generative AI technology, popular model architectures for sequence-to-sequence tasks that are can be directly compared to our own and dives deep into the details of the transformer architecture. The paper also discusses the introduction of deep architectures into the budding field of MIR as well as prior deep-learning approaches in Music Generation.

The paper then evaluates personal attempts to build a transformer model for the task of musical improvisation, documenting various challenges and design decisions along the way as well as evaluating the model(s) through a variety of means. The design for the model's inference interface is also discussed, as well as opportunities for future improvement and work.

Chapter 2

Background

2.1 A Brief History of Generative AI

Generative AI refers to a subset of artificial intelligence techniques that focus on generating new content, be it text, images, music, or other forms of data. This technology has evolved significantly, from rule-based systems to advanced neural networks.

2.1.1 Early Systems

Initial generative models were simplistic, using rule-based systems for tasks like automatic report generation. However, they lacked the ability to create complex, human-like content. For references, look into early works on generative systems in AI, such as Buchanan et al.'s "A Model of Rule-Based Understanding in Reading" (1979). Rise of Neural Networks:

2.1.2 Deep Architectures

2.1.3 Generative AI in Image Processing

- Discuss models like stable diffusion, deepfake technology etc. leading to the creation of Dall-E and more

2.1.4 Generative AI in Sequence Modeling Tasks

Recurrent Neural Networks — The advent of neural networks brought a paradigm shift. Recurrent Neural Networks (RNNs) and later, Long Short-Term Memory networks (LSTMs), allowed for better handling of sequential data, crucial for tasks like language modeling.

Long Short-Term Memory Networks — Hochreiter Schmidhuber's "Long Short-Term Memory" (1997) is a seminal paper in this area. Transformers and Beyond:

Transformers — The introduction of transformer models, as seen in Vaswani et al.'s "Attention Is All You Need" (2017), marked a turning point. These models outperformed previous architectures in various tasks, leading to their widespread adoption. Vaswani et al.'s paper is a must-read for understanding the foundation of modern transformer models.

2.1.5 Impact of Generative AI in Various Fields

Generative AI has found applications in diverse fields, from creating realistic images and art to generating human-like text and aiding in scientific discovery. For examples of AI in art and text generation, see "A Neural Algorithm of Artistic Style" by Gatys et al. (2015) and Radford et al.'s "Language Models are Unsupervised Multitask Learners" (2019), respectively. Versatility of Generative Models:

These models are not just limited to creative tasks but also assist in data augmentation, simulation for autonomous vehicles, and more. Look into "Playing for Data: Ground Truth from Computer Games" by Richter et al. (2016) for insights into data augmentation for autonomous driving.

2.2 Music Information Retrieval

Chapter 3

Prior Work

3.1 Music Generation via Deep Learning

3.1.1 *ImprovRNN* and *GrooveVAE* by Magenta

TODO: Complete this section

3.1.2 *Learning Expressive Musical Performance* - LSTMs/RNNs for musical generation

In this paper, Oore et.al[**oore:1**] propose a novel approach to music generation, concentrating on direct performance generation. This involves jointly predicting the notes and their expressive attributes, such as timing and dynamics, rather than just creating or interpreting scores. This is in direct contrast to models like ImprovRNN or GrooveVAE that separate the generation of musical phrases from their expressive attributes. Specific to this task, the authors emphasizes the importance of using a dataset that is appropriate for the task of generating expressive musical performances. This implies the need for a dataset that captures not just the notes but also the nuances of timing and dynamics. To satisfy this requirement, the authors turn to the International E-Piano Dataset, which contains midi-recorded performances of piano works on a Disklavier, capturing musical notes as well as their expressive information with very high fidelity

The model employed is an LSTM-based network with three layers, each consisting of 512 cells. The network uses a temporally non-uniform representation of data, with a vocabulary of 413 different events, including NOTE-ON, NOTE-OFF, TIME-SHIFT, and VELOCITY events. The TIME-SHIFT events are particularly notable for allowing the model to capture expressive timing. They enable the network to move the time step forward by increments of 8 ms up to 1 second, thus maintaining expressiveness in note timings. The network operates on one-hot encoding over this event vocabulary, with each input to the RNN being a single one-hot 413-dimensional vector.

TODO: INCLUDE IMAGES FROM PAPER

The system was found to be effective in generating MIDI performances with expressive timing and dynamics. The feedback from professional composers and musicians suggested that the system's output resembled human performances in terms of local structure, like phrasing and dynamics, although it lacked long-term structural coherence. The conclusion highlights that the system sounds like a skilled pianist but lacks a clear long-term compositional plan, creating performances that are expressive but somewhat random in nature. This research marked a significant step in the field of generative music, especially in its focus on the expressiveness of musical performances. The use of LSTM networks for this purpose shows promising results in capturing the nuances that make music feel more human and less mechanically generated.

3.1.3 *Generating Music with Long-Term Structure* - A modified Transformer Architecture for Music Generation

TODO: Complete this section

3.2 Systems geared toward Musical Improvisation

3.2.1 *Voyager* — George E. Lewis

Voyager is a "non-hierarchical, interactive musical environment that privileges improvisation" [**Lewis:1**]. In this environment, an improvisor engages in dialogue with a computer-driven interactive virtual improvising orchestra. The program analyzes the human improvisers performance and uses the analysis to guide a complex, improvised response to the player, as well as an independent response from its own creative internal process

Voyager was conceived as a set of 64 asynchronously operating single-voice MIDI-controlled players, each of whom responds to and generates music in real-time. First, a low-level routine parses incoming MIDI data into separate streams, accommodating up to two human improvisers. The improvisers are either playing MIDI-equipped keyboard instruments or playing acoustic instruments that are interfaced with pitch detection devices that parse the sounds of acoustic instruments into MIDI data streams. A mid-level smoothing routine is also called to construct averages of pitch, velocity, probability of note activation, and note spacing.

In response, a global subroutine called *setPhraseBehaviour***??** (called in intervals of 5 - 7 seconds), continuously recombines the MIDI players into new ensemble combinations with defined behaviors. These behaviors define how players are in the new ensemble, choosing whether to let the new ensemble play alongside the older ensemble or to shut off the older ensemble entirely. Lewis created several varying sonic behavior groupings — some of which clash — such that they may be active simultaneously, or move in and out of a unified metric pulse. The *setPhraseBehaviour* routine also includes its own subroutines that determine the choice of timbre, melody generation

algorithm, pitch sets, volume range, transposition, tempo, note spacing, probability of playing a note, interval width range, MIDI-related ornamentation (Eg. reverb) and how these parameters may change over time.

```
:ap setphrasebehavior ( -- )
    ::ap" general phrasing " ( task recurs at intervals of 5000-7000 ms )
    5000 time-advance 11 irnd 200 * 5000 + to cycle

    begin
          ::ev
          bodymusic 0=              \ in this version this red light is always zero
                if calcork          \ set up new group of players, including number and position in space
                else allplayersoff  \ turn off all groups and start over with a new group.
                then
\ set up how system will follow input; set MIDI timbres
          setfollowbehavior         setreplies            setvoxbehavior

\ set melody algorithms, pitchsets, reverb and chorus type
          setwavebehavior           setscalebehavior      setreverbbehavior        setchorusbehavior

          computer-solo?            \ if no one is playing, I have a solo

\ set volume and velocity, microtonal tonic transposition
                if setvelbehavior   setvolbehavior       settonicbehavior

\ set octave, interval range, duration range
                setoctbehavior      setintbehavio        setwidbehavior   setlegatobehavior

\ set length of notes
                bodymusic 0=        \ in this version this red light is always zero
                         if setrestbehavior \ set up average degree of silence
                         then

\ set portamento, whether or not to follow tempo, and tempo ranges
                setportabehavior    settempofollow       setspdbehavior
                then

          ;;ev
    cycle time-advance
    again
    ;;ap
;ap
```

Fig. 1. *Voyager*'s top-level phrase behavior word, written as a FORMULA active process.

Figure 3.1: Voyager's setPhraseBehaviour Routine

Furthermore, each newly created ensemble chooses both a distinct group sonority, as well as a unique response to the input, deciding which improvisers — one, both, or none — influence its output behavior. The subroutine *setresponse***??** runs asynchronously to *setPhraseBehaviour*. This routine processes both the raw and the averages returned by the smoothing routine to decide how the ensemble responds to specific elements of the input, such as tempo, melodic intervals, etc.

A critical distinction between Voyager and other improvisatory systems is that Voyager is still able to function in the absence of any outside input. With no input, the specification of the system's behavior is entirely governed by *setPhraseBehaviour*. Given that the computer is able to spontaneously create music that is in no way influenced by the improviser, decisions made by Voyager have consequences that must

```
setresponse ( -- )
 setinputbasedur           \ set tempo ranges based on input note durations
 bodymusic 0=              \ in this version this red light is always zero
    if setinputplayprob    \ probability of note or rest, based on input
    then

\ set duration range and length of notes, interval range
 setinputlegato   setinputwid       setinputint

\ use pitchset based on last few input notes; set octave and microtonal tonic transposition
 setinputscale    setinputoct       setinputtonic

\ set MIDI volume and velocity
 setinputvol      setinputvel
;
```

Fig. 3. *Voyager*'s input response word, written in Forth, sets parameters based on analysis of MIDI input.

Figure 3.2: Voyager's setResponse Routine

be accounted for by the listening improviser, creating a situation where both the computer and the improviser are held equally accountable for the final output. This is an especially desirable quality for an improviser program, and something worth replicating within our proposed Transformer model.

3.2.2 *GenJam* by John A. Biles

GenJam[**Biles:1**], created by John A. Biles, is an interactive genetic algorithm that models a jazz improviser and performs as a featured soloist in the author's Virtual Quintet. Previous papers published by Biles have described GenJam's hierarchically related populations of melodic ideas, its chromosome representations for those ideas, its genetic operators for evolving new ideas, and the training of new soloists. This training is done under the guidance of a human mentor, who listens to GenJam improvise and indicates "good" or "bad" accordingly, as a kind of reinforcement loop. The mentor's feedback is used to increment or decrement the fitness of individual melodic ideas and serves as the environment in which musical ideas persist, or are removed from the stored hierarchy. New ideas evolve by 1) selecting the "better" ideas to be parents 2) breeding "children" ideas using single-point crossover and musically meaningful mutation, and 3) replacing the "worse" ideas in the population with these new children.

In their paper, Biles demonstrates how a four-bar phrase is mapped into "GenJam Normal Form (GJNF)" and discusses the various mutation operators available to GenJam to create a musical response. A key strength of GenJam is that it was later optimized to "trade-fours" with the player in a very traditional jazz style [**Biles:2**]. Many other improvising systems are unable to tell when a phrase ends (a massively subjective art in and of itself) which means they frequently interrupt another improvisor mid-phrase. GenJam instead hardcodes the knowledge of the four-bar phrase,

removing the problem entirely. The high-level algorithm it uses to accomplish this is described below:

1. GenJam first receives a chord progression for a specific tune, (read in from a data file) and constructs a chord-scale mapping for the entire tune. The table below provides an example of a chord scale-mapping used during the listening phase and in the playing phase of Jerome Kern's *All the Things You Are*

Bar	Chord	Scale	Pitches for new-note events 1-14
25	Fm7	Hexatonic Minor (avoid 6th)	C Eb F G Ab Bb C Eb F G Ab Bb C Eb
26	Bbm7	Hexatonic Minor (avoid 6th)	C Db Eb F Ab Bb C Db Eb F Ab Bb C Db
27	Eb7	Hexatonic Mixolydian (avoid 4th)	C Db Eb F G Bb C Db Eb F G Bb C Db
28	AbMaj7	Hexatonic Major (avoid 4th)	C Eb F Gb Ab Bb C Eb F Gb Ab Bb C Eb
29	DbMaj7	Hexatonic Major (avoid 4th)	C Db Eb F Ab Bb C Db Eb F Ab Bb C Db
30	Gb13	Hexatonic Mixolydian (avoid 4th)	Db Eb Fb Gb Ab Bb Db Eb Fb Gb Ab Bb Db Eb
31	Cm7	Hexatonic Minor (avoid 6th)	C D Eb F G Bb C D Eb F G Bb C D
32	Bdim	Whole/Half Diminished	D E F G Ab Bb B Db D E F G Ab Bb

Figure 3.3: An Example of chord-scale mappings for the tune *All the Things you are*, bars 25-32

2. The human performer plays four bars into a microphone plugged into a pitch-to-midi converter

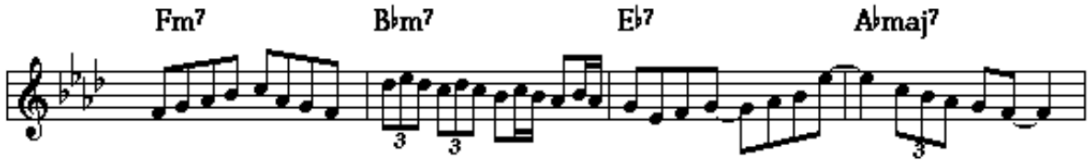

Figure 3.4: An example of a Charlie Parker quote played over measures 25-28 of *All the Things you are*

3. The pitch to MIDI converter sends MIDI events to GenJam running on the host computer

4. GenJam listens to MIDI events and quantizes them into 8th-note long windows. It then uses the chord-scale mappings provided to create measure and phrase chromosomes for the four-bar phrase in "GenJam Normal Form (GJNF)"

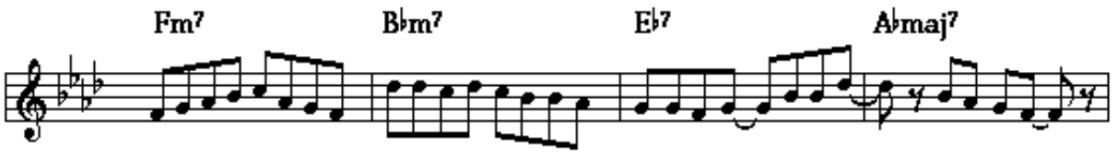

Figure 3.5: The melody from figure **??** in "GenJam Normal Form". Note that the triplet melodies cannot be preserved due to 8th note quantization

5. In the last 30 milliseconds of the human's four bar phrase, GenJam stops listening and performs musically meaningful mutations on some of the chromosomes in preparation to play them back as its next four. Since all the input has been converted to GJNF, the phrase may be mutated by using any of GenJam's musically meaningful mutation operators, which guarantees that the result is playable over the subsequent four-bar phrase. These mutations include reversing or inverting phrases, transposing phrases, repeating a phrase, and many other changes that can be combined to derive entirely new and musical four-bar phrases.

6. Mutated chromosomes are then used as GenJam's next four bars as if they were part of a stored soloist

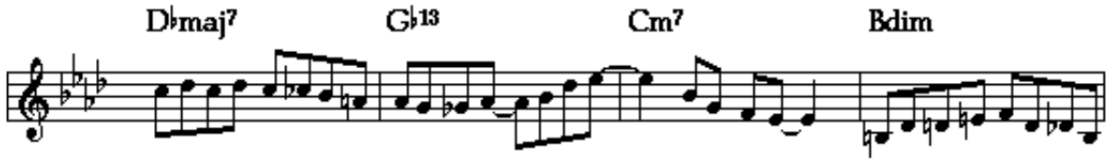

Figure 3.6: The four-bar phrase returned by GenJam to be played over bars 29-32

The system of trading fours as presented here in GenJam is extremely robust. Compared to Voyager which functions largely in the space of free improvisation, GenJam makes full use of hard-coded information to make the performance work (eg. the data file with chord progressions for a specific tune). In our transformer model, it may be useful to include an interface that can hold similar information and essentially set the "context" in which the improvised performance is happening.

Chapter 4

Project Architecture

4.1 Design Concerns and Choices

The task proposed in this project necessitated many early design choices that greatly affected the choice of software and libraries that would be used for the remainder of the project. Some of these concerns included:

- **Input Data Type** — For the task of Music Generation we can choose to work with 1) *real-time audio* or 2) *a symbolic representation of the music*, in a format such as MIDI. The choice of input greatly affects the model architecture, its output, as well the it's overall speed. Furthermore the choice of data format informs the choice of dataset, and it is critical to use data of a high quality
- **The Model Environment** — Given the project's aim to create a live, improvisation centric system, we first require an environment/software that is able to easily:
  1. Record live auditory information
  2. Transform it into a format acceptable to the model
  3. Trigger model inference
  4. Generate musical output given the model's output

  Additionally, for the system to be accessible to non-technical musicians, it would ideally support some form of visual GUI that makes it relatively easy to use and learn, while abstracting away much of the core logic in the program.
- **Separating model training from the performance environment** — Given the above, the model would likely have to live in an environment separate from the one it was trained in. Therefore the model must be flexible and support being exported and deployed to the environment of choice.
- **Model Size** — Our model had to balance between being large enough to extract

meaningful information from large amounts of sequential data, while also being small enough such that inference could still be relatively quickly.

Taking the above concerns and the problem statement into account, I made several design choices:

1. **Input Data Type** - To minimize the amount of processing required, I explicitly decided to use the midi encoding proposed by Oore et. al[**oore:1**] While this greatly lessens the processing requirement of the model itself, it places the onus of extracting a meaningful stream of MIDI data from the model environment.

2. **The Model Environment** - The model's performance environment is constructed in Max/MSP [**max/msp**]. Max/MSP — also known as Max — is a visual programming paradigm for music and multimedia developed by Cycling '74. It provides a graphical interface for creating and manipulating audio, video, and other multimedia content and is widely used in the fields of music composition, live performance, sound design, and interactive multimedia art. It is easy to use, highly optimized as an audio processing unit, and has an assortment of pre-built modules that directly address our needs, like a pitch-tracking module.

   Furthermore, Max/MSP features an internal Node.js runtime engine that can run and interact with custom Javascript scripts. With the full flexibility of Javascript at our disposal within Max/MSP, I have opted to use the Node.js runtime to create an Improvisor class that houses our transformer model and serves as the interface between our model and Max/MSP. All custom is currently being written in Typescript before being compiled into Javascript.

3. **Choice of Framework** - The most popular ML frameworks available today are PyTorch and Tensorflow, both of which leverage Python as the main language to train ML models. While PyTorch is reportedly easier to use and prototype models with, I have chose to use Tensorflow as my Machine learning library of choice, as it has a Javascript port, *Tensorflow.js*, that can run models built and compiled with the regular Tensorflow package and Python. This would effectively allow me to train and build my model in a separate Python environment, and then load it within the Node.js runtime housed within Max.

While the source code for the Music Transformer paper by Google has not yet been released, I found a 3rd party implementation of the paper by a PHD researcher INSERT on Github. While this implementation was helpful in understanding the various components of the Transformer, my own implementation differs significantly.

Chapter 5

Data

5.1 Choice of Dataset

For the purpose of this project, I decided to use the MAESTRO dataset[**hawthorne:1**] used by Huang et.al [**Huang:1**] during their attempt to build a modified transformer model for long-context music generation. Put together by the same team, MAESTRO (MIDI and Audio Edited for Synchronous TRacks and Organization) is a dataset composed of about 200 hours of virtuosic piano performances captured with fine alignment (3 ms) between note labels and audio waveforms. The dataset is made available by Google LLC under a Creative Commons Attribution Non-Commercial Share-Alike 4.0 (CC BY-NC-SA 4.0) license.

While there are several other datasets of paired piano audio and MIDI have been that published previously and have enabled significant advances in MIR tasks, I choose to use the MAESTRO dataset for several reasons. First, MAESTRO is around an order of magnitude larger than most other similar datasets, as shown making it a perfect choice for training transformer models that notoriously require a lot of data. Second, the quality of recorded midi performances in the MIDI dataset is extremely high, having been collected from several annual runs of the International Piano e-competition CITE. Virtuoso pianists perform on Yamaha Disklaviers: concert quality acoustic grand pianos that utilize an integrated high-precision MIDI capture and playback system. The recorded data is of sufficient fidelity to allow the audition stage of the competition to be judged remotely by listening to contestant performances reproduced over the wire on another Disklavier instrument.

Below is information collected about other similar datasets that were considered for this task by Hawthorne et.al[**hawthorne:1**]:

MusicNet (Thickstun et al., 2017) contains recordings of human performances, but separately sourced scores. As discussed in Hawthorne et al. (2018), the alignment between audio and score is not fully accurate. One advantage of MusicNet is that it contains instruments other than piano (not counted in table 2) and a wider variety of

recording environments, making it more flexible for ML tasks that deal directly with audio.

MAPS (Emiya et al., 2010) contains Disklavier recordings like MAESTRO and synthesized audio created from MIDI files that were originally entered via sequencer. The synthesized audio makes up a large fraction of the MAPS dataset, which implies that many of the performances are not as expressive as the MAESTRO tracks captured from live performances, making it less attractive for the task of musical improvisation.

Saarland Music Data (SMD) (Muller et al., 2011) is very similar to MAESTRO in that it contains recordings and aligned MIDI of human performances on a Disklavier, but is far smaller than the MAESTRO dataset

5.2 Dataset Analysis

The figure above gives us some brief statistics about the data set as well as details about a proposed train, validation and test split. Hawthorne et. al note that this split was proposed so that:

1. No composition should appear in more than one split
2. Train/validation/test should make up roughly 80/10/10 percent of the dataset (in time) respectively and that the validation and test splits should contain a variety of compositions
3. Extremely popular compositions performed by many performers should be placed in the training split.

They also note that these proportions should be true globally and also within each composer's corpus of performed work. However, they also note that maintaining these proportions however, is not always possible because some composers have too few compositions in the dataset.

TODO: Histograms of num pieces in each split by composer (can be viewed as a specific style) and by key (a specific pitch distribution)

TODO: Histograms of num pieces in each split by composer, weighted by duration

5.3 MIDI Encoding and Input Representation

The MIDI protocol covers a wide variety of message types, which raises a new challenge for our input representation. Hence, we adopt a sparse, MIDI-like representation from Oore et al. [**oore:1**] that specifically encodes information about pitch and time in an event-based manner. The vocabulary of this encoding consists of 128 NOTE-ON events, 128 NOTE-OFFs, 100 TIME-SHIFTs allowing for expressive timing at 10ms (specifically from the MAESTRO dataset) and 32 VELOCITY bins for expressive dynamics, as shown in Figure **??**.

Dataset	Performances	Compositions	Duration, hours	Notes, millions
SMD	50	50	4.7	0.15
MusicNet	156	60	15.3	0.58
MAPS	270	208	17.9	0.62
MAESTRO	1184	~430	172.3	6.18

Figure 5.1: Comparing Maestro to other datasets (Taken directly from Hawthorne et al.)

Split	Performances	Duration (hours)	Size (GB)	Notes (millions)
Train	962	159.2	96.3	5.66
Validation	137	19.4	11.8	0.64
Test	177	20.0	12.1	0.74
Total	**1276**	**198.7**	**120.2**	**7.04**

Figure 5.2: Split from the Maestro Dataset v3.0.0 (From `https://magenta.tensorflow.org/datasets/maestro`.)

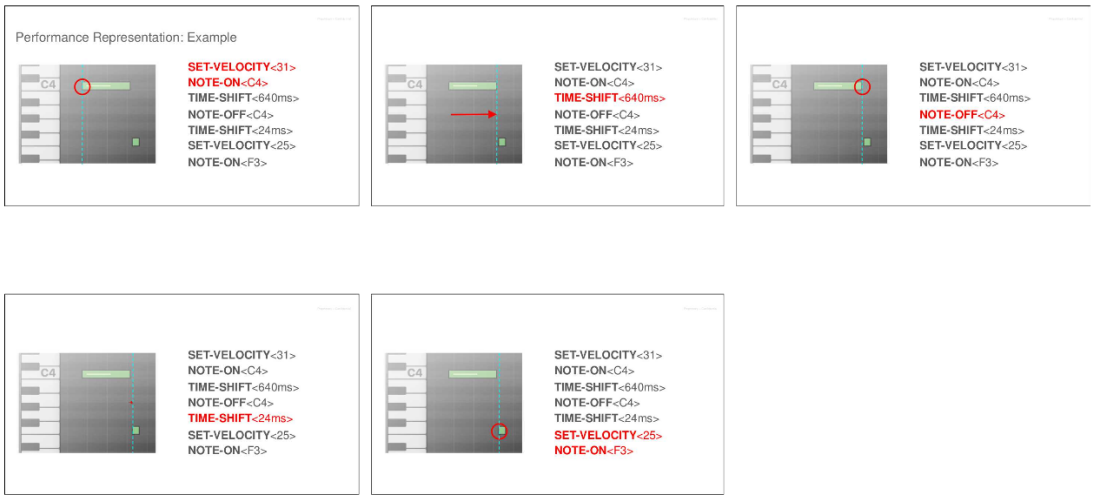

Figure 5.3: An illustration from Oore et al. of how a MIDI sequence is converted into a sequence of commands in our event vocabulary. Note: an arbitrary number of events can in principle occur between two time shifts.

The implementation of the MIDI encoding process was outsourced to a 3rd party library **midi-neural-preprocessor**, developed by INSERT NAME, the same researcher who attempted to implement the Google Paper on long context music-generation with Transofrmers. To process all of the MIDI files in the MAESTRO dataset, I wrote a Preprocessor class that employs multi-threading to encode each MIDI file and save it in a binary format.

I also conducted a statistical analysis of event sequences across all the MIDI files, gleaning some interesting results.

TODO: Discuss Statistics for event sequences by Split (Average, length, max, min, as well events that do not appear at all or appear overwhelmingly across sequences)

5.4 Building an input pipeline in Tensorflow

Given that I was dealing with a substantial amount of sequential data across thousands of files it was of paramount importance to experiment with different data pipelines and consider which was best suited for me given the limited compute resources on hand. While the raw encoded sequences themselves could all fit into memory, it is important to note that the training examples for the model are constructed by grabbing fixed length event sequence windows from the sequence, with a stride of 1. This results in an explosion of our data's memory footprint, illustrated below.

TODO: Provide brief diagram showing size of raw midi -¿ how many batches we actually extract given a window of size L and the resulting size of all the extracted training examples.

5.4.1 The BaseDataset class

- Parent class for various dataset implementations. Reads in and stores all the paths to the encoded midi binaries, while also filtering out some files based on minimum duration, or minimum event sequence length

5.4.2 The TestDataset class

- Subclasses BaseDataset. Built to first generate simple test sequences in memory to evaulate early models 1) Can generate major scale midi numbers across all keys, useful metric to see if the Transformer could first learn very basic data

2) Retrieves all the training examples from just one file, separating them into an 80/10/10 training , validation and test split that is shuffled around the file

3) Retrieves all files to attempt construction of an in-memory dataset

4) V2 of in-memory dataset that is optimized with numpy operations and integer types with a smaller memory footprint. Significant improvements, but not enough to

load all data into memory

5.4.3 The SampledDataset class

5.4.4 The SequenceDataset class

5.4.5 The MappedDataset class

5.4.6 Analyzing GPU Profiles

5.4.7 Thinking about the model

Given our desire to train a robust model that is able to function in real-time, the size of our model needed to simultaneously be small enough such that inference did not take extremely long, while also being large enough to encode the large amount of midi information being fed to it. For this purpose I decided to work with an architecture that is very similar to the baseline transformer architecture proposed in Vaswani et. al[**Vaswani:1**].

5.5 The Model

5.5.1 Background

In the landmark paper "Attention Is All You Need" [**Vaswani:1**], Vaswani et al. propose the Transformer model as an alternative to previous state-of-the-art sequence transduction models. These include Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTMs) [**Hochreiter:1**] and Gated RNNs [**Chung:1**] built with encoder-decoder architectures for sequence-to-sequence generation. Recurrent models typically handle computation based on the positions of symbols in input and output sequences. By aligning these positions with computational time steps, they generate a series of hidden states where the most recent hidden state, ht, depends on the previous hidden state, $ht - 1$, and the input for the current position (t).

This sequential nature inherently restricts parallelization and is especially true for longer sequences where memory limitations constrain the ability to batch multiple examples together. Recent research has made notable strides in improving computational efficiency through methods like factorization techniques [**Kuchaiev:1**] and conditional computation [**Shazeer:1**] while also enhancing model performance, particularly in the case of the latter approach. Many of these models also employ attention mechanisms, which allow some amount of modeling of input sequence dependencies without regard to their distance in the input/output sequence [**Kim:1**]. Nonetheless, the core constraint of sequential computation imposed by recurrent neural networks has remained.

The transformer model, however, does away with recurrence entirely and relies solely on attention mechanisms to draw global relationships between input and output. This makes the Transformer model highly parallelizable for faster, more efficient training.

5.5.2 Model Structure

Similar to prior state-of-the-art sequence transduction models, the transformer has an encoder-decoder structure where the encoder maps an input sequence of representations $X = (x_1, ...x_n)$ to some sequence of continuous feature representations $Z = (z_1, ...z_n)$. Provided with these continuous feature representations Z, the decoder can generate an output sequence one element at a time, using previously generated symbols as additional input to generate subsequent symbols. The Transformer uses a similar overall architecture but instead uses stacked self-attention layers and fully connected layers in both the encoder and decoder, as shown in Figure **??**.

5.5.3 Inputs and Embeddings

The input to a Transformer consists of a sequence of tokens. The tokens can be words, characters, numbers, or any other kind of symbol that inherently holds sequential logic. Before being fed into the model, these tokens are embedded into continuous vector representations — *feature vectors* — using an embedding layer such that each

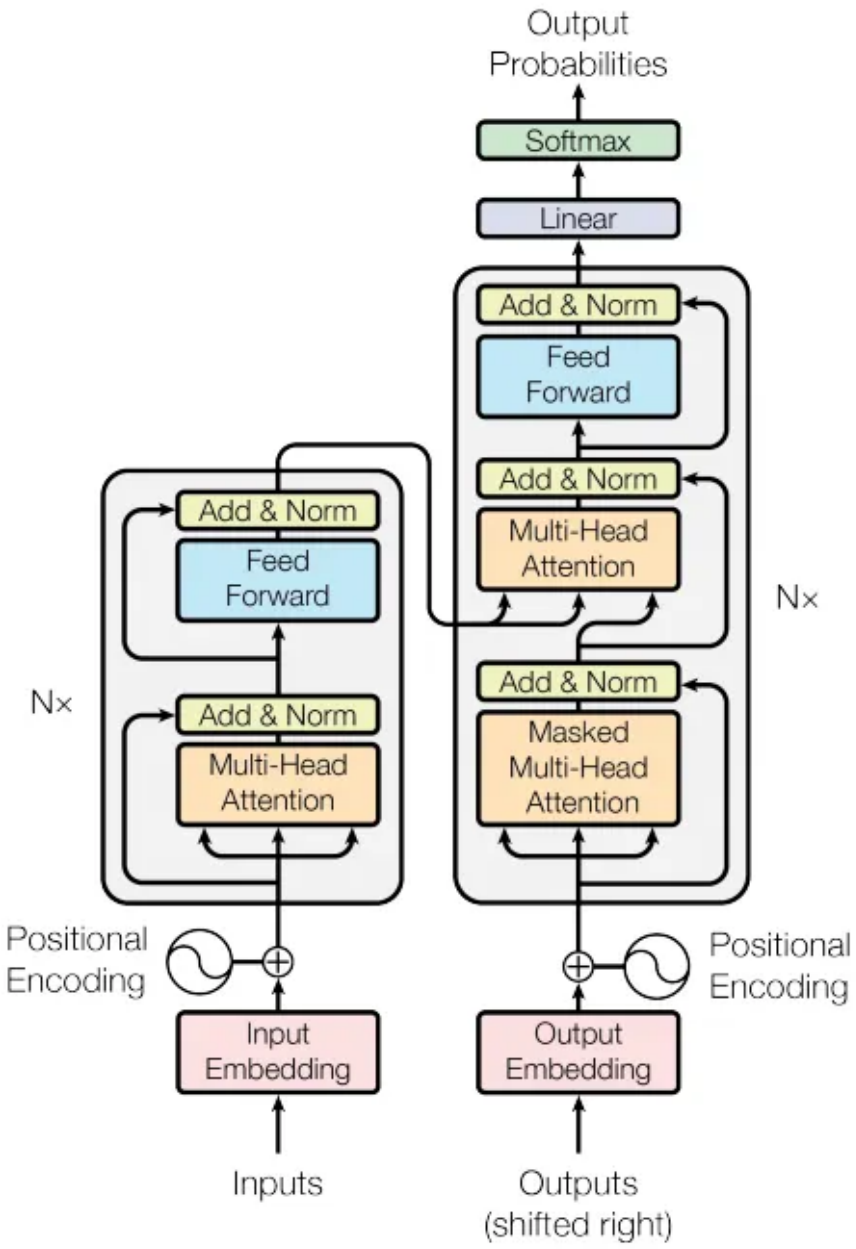

Figure 5.4: The Transformer Model Architecture (Taken directly from Vaswani et al.)

token is now represented by a unique vector. These input embeddings can be learned by the model during training. More commonly, we reuse learned input embeddings created by other models that may be trained on much larger data sets.

5.5.4 Positional Encoding

Unlike Recurrent Neural Networks, the self-attention mechanism of the Transformer creates a new problem: our input sequence no longer inherently holds positional information. To encode position into our input, Vaswani et al. propose a means to "inject" positional information about the tokens in our sequence. This is done by adding "positional encodings" to the input embeddings directly. Vaswani et al. use sinusoidal functions of different frequencies to encode position into each token's embedding:

$$\begin{aligned} PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{1.1}$$

where pos is the position in the token and i is the dimension of the positional encoding that corresponds to a sinusoidal function.

5.5.5 Encoder / Decoder Stacks

Encoder — The encoder is composed of a stack of 6 identical layers, each with 2 sub-layers. The first is a multi-head self-attention layer, followed by a fully connected feed-forward network. Each sub-layer has a residual connection, followed by layer normalization where the output of each sub-layer is $Normalize(x + sublayer(x))$. To facilitate this, all sub-layers in the model as well as the embedding layers output tensors of dimensions d_{model}, which is 512 for their implementation.

Decoder — The decoder is also composed of a stack of 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a second multi-head attention in the middle which acts on the output of the encoder stack. The multi-head attention sub-layer in the decoder stack is also modified to prevent positions from attending to subsequent positions with an input mask. This combined with the fact that the output embeddings are offset by one position, ensures that the predictions for any position pos_i can depend only on positions pos_k where $k < i$. Like the Encoder stack, we similarly employ residual connections and layer normalization around each sub-layer

5.5.6 Attention

Self-Attention

The self-attention function is at the core of the transformer and is the reason for the transformer's huge success in the field of NLP. The self-attention mechanism is what allows the model to be heavily parallelized, and also allows the model to draw relationships between tokens across the entire length of the input sequence in constant time. The auto-regressive nature of self-attention makes this mechanism particularly great for music, which is often composed of recurring structures that repeat and develop.

Similar to many modern-day search engines, an attention function can be described as mapping a query and a set of key-value pairs to some output, where each of these — the query, keys, values, and output — are vectors. The output is computed as the weighted sum of each matched value, where the weight assigned to each value is dependent on a "similarity" function that compares the query against the corresponding key.

- **Queries, Keys, and Values:** Each input token is associated with three vectors: *Query (Q), Key (K), and Value (V)*. These vectors are learned during training.
- **Attention Scores:** For each token in the sequence, the self-attention mechanism calculates an attention score by measuring the similarity between its Query vector and the Key vectors of all other tokens.
- **Weighted Sum:** The attention scores determine how much of every other token in the sequence contributes to the output for the current token. This is achieved with a weighted sum of each token's Value vectors, producing the context vector as output for the current token.

Multi-Head Attention

Instead of performing a single attention function over the entire Query, Key, and Value vectors of dimension d_{model}, Vaswani et al. propose a linear projection of the queries, keys, and values h times to d_{query}, d_{key}, d_{value} dimensions respectively. This is achieved by placing linear layers before the multi-head attention layer and projecting the queries, keys, and values in a smaller space. On each of these projected versions of queries, keys, and values we then perform the attention function in parallel, yielding d_v dimension output values. This allows us to capture multiple relationships between tokens as the model can attend to information between different representation subspaces at different positions. The output of each of these 'heads' is a vector of dimension d_k, which are concatenated and once again projected to get our final output.

$$MultiHead(Q, K, V) = Concat(head_1, ...head_h)W^O$$
$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{1.2}$$

where the projections are done via the following parameter matrices:

$$W_i^Q \in R^{\,d_{model} \times d_k}$$
$$W_i^K \in R^{\,d_{model} \times d_k}$$
$$W_i^V \in R^{\,d_{model} \times d_v}$$
$$W^O \in R^{\,hd_v \times d_{model}}$$

Scaled Dot-Product Attention

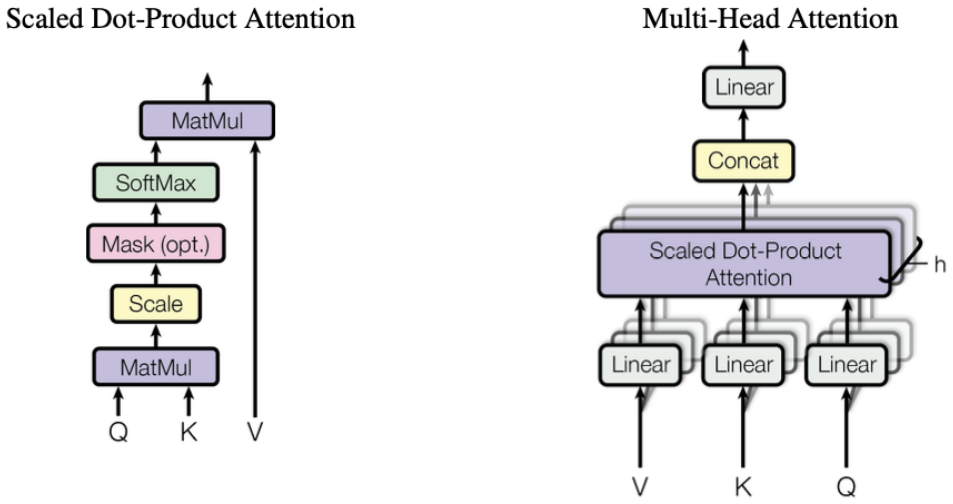

Figure 5.5: (left) Scaled Dot-product Attention. (Right) Multi-head Attention with several layers in parallel. Taken directly from Vaswani et al.

In particular, the form of attention implemented by Vaswani et al. is Scaled-Dot-Product Attention**??** First the dot products of the query with all the keys are calculated. It is then divided by $\sqrt{d_k}$ — to counter dot-product values growing too large, resulting in vanishing gradients — and converted to a probability distribution by applying a softmax function to obtain weights on the values. As described earlier, the attention function is computed on a set of multiple queries, keys, and values simultaneously, packed in matrices Q, K, and V respectively.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1.3}$$

5.5.7 Position-Wise Feed Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2(2)$$

where W_1, W_2 and b_1, b_2 are weights and biases of each linear layer respectively. In the case of Vaswani et al., the dimensionality of input and output is $d_{model} = 512$, and the inner layer has dimensionality $d_{ff} = 2048$.

5.6 Proposed Modifications to the Baseline Transformer Model

5.6.1 Considerations

Given the fact that we are applying the Transformer model to an entirely different domain of music, we must consider the differences between the domains of music and language to make modifications to our transformer model accordingly. One key issue is in the matter of positional encoding. In its original state, the Transformer model above relies on absolute position representations using positional sinusoids that are added to the input embeddings at each input position. This is very different from state-of-the-art models, usually RNNs and LSTMs, which instead model positions in relative terms. Furthermore, working with music dramatically alters the input our model might receive.

5.6.2 Relative Positional Self-Attention

Music has two dimensions along which relative differences matter more than absolute differences: Time and Pitch. This key information is captured in the event-based input representation proposed by Oore et al.

To capture such pairwise relationships between representations, Shaw et al. [**shaw:1**] introduce a version of the self-attention mechanism that is regulated by the distance between two points, which involves learning a separate relative position embedding E^r of shape (h, l, d_k), where h is the number of heads, l is the length of the input sequence, and d_k is the dimensionality of the projected keys, queries and values that are fed to each head in the multi-head attention layer. E^r holds an embedding of each possible pair-wise distance $r = i_q - j_k$ between a query and key in position i_q and j_k respectively.

These are ordered from $-L + 1$ to 0 and are learned separately for each head. As proposed in Shaw et al, the relative embeddings interact with queries to give rise to S^{rel}, and $L \times L$ dimensional logits matrix that can be used to modulate attention probabilities for each head.

$$AbsoluteAttention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

$$RelativeAttention(Q, K, V) = softmax(\frac{QK^T + S^{rel}}{\sqrt{d_k}})V \tag{1.4}$$

For each head, Shaw et al. create a temporary tensor R of shape (L, L, d_k), containing the embeddings that correspond to the relative distances between all keys and queries. Q is then reshaped to an $(L, 1, d_k)$ tensor, and $S^{rel} = QR^T$ This incurs a total space complexity of $O(L^2D)$, restricting its application to long sequences, and making the model less feasible for real-time generation tasks.

However, Huang et al. [**Huang:1**] propose a memory-efficient implementation of relative attention by reducing the intermediate memory requirement to $O(LD)$. Huang et al. observe that all of the terms we need from QR^T are already available when Q is directly multiplied with E^r, the relative positional embedding. After we compute QE^{rT}, its (i_q, r) entry contains the dot product of the query in position i_q with the embedding of relative distance r. Each relative logit (i_q, j_k) in the matrix S^{rel} from Equation **??** should be the dot product of the query in position i_q and the embedding of the relative distance $i_q - j_k$, to match up with the indexing in Q^{K^T}.

We therefore need to "skew" QE^{rT} to move the relative logits to their correct positions. The "skewing" procedure is illustrated in Figure **??** below and is detailed in Huang et al. [**Huang:1**] The time complexity for both methods is $O(L^2D)$, but Huang et al. demonstrates that in practice their method is 6x faster at input sequence length 650 since it bypasses the need to manipulate larger tensors.

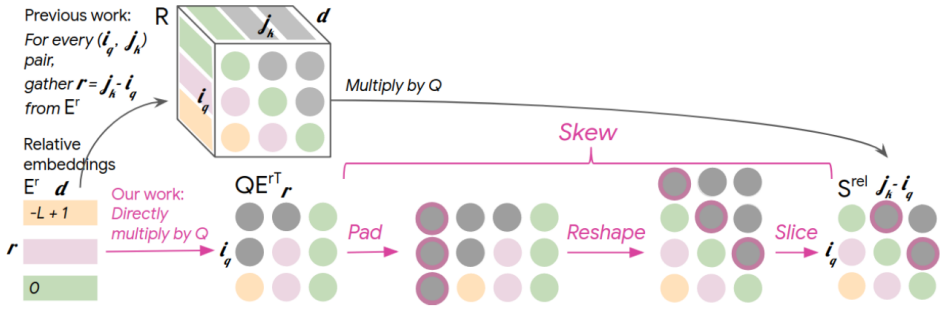

Figure 5.6: Skew Procedure from Huang et al.(2018) Gray indicates masked or padded positions, while each color corresponds to a different relative distance