


Fall 2023

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

Arnav Shirodkar
Bard College

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_f2023

 Part of the [Artificial Intelligence and Robotics Commons](#), [Graphics and Human Computer Interfaces Commons](#), [Music Practice Commons](#), and the [Software Engineering Commons](#)



This work is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License](#).

Recommended Citation

Shirodkar, Arnav, "A Symbolic Music Transformer for Real-Time Expressive Performance and Improvisation" (2023). *Senior Projects Fall 2023*. 54.
https://digitalcommons.bard.edu/senproj_f2023/54

This Open Access is brought to you for free and open access by the Bard Undergraduate Senior Projects at Bard Digital Commons. It has been accepted for inclusion in Senior Projects Fall 2023 by an authorized administrator of Bard Digital Commons. For more information, please contact digitalcommons@bard.edu.

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

Senior Project Final Paper

Arnav Shirodkar

Dec 2023



Bard College

Computer Science

Acknowledgements

I would like to thank my Senior Project advisor Sven Anderson for all your guidance while taking on this fairly colossal project. I'm eternally grateful that you were constantly there to reel me back in when I was filled to the brim with one idea after another and helped me make the scope of my work concrete. You helped crystallize the vision of my work, turning abstract thoughts into concrete reality. Each of our senior project discussions was not just enlightening but also a source of great joy, blending intellectual rigor and genuine fun in a way I never thought possible.

To Matt Sargent, thank you for always reminding me that you don't have to pick between science and music; you can do both. Your encouragement has been a constant reminder that my passions for computer science and music need not be mutually exclusive. You have inspired me to keep walking the path where these two worlds converge, whether as a musician, a composer, or a researcher.

To my friends; most of all the Bard Percussion Studio and the Ultimate Frisbee Team, the laughter and kinship I have had with all of you has been a wellspring motivation to keep going forward, Your presence in my life has been a source of strength, often more than most of you might realize.

To my cherished family — I would not be here without you. You are the pillar of everything I have achieved, and everything I aspire to achieve. Thank you for always believing in me, providing for me, and inspiring me to be the best version of myself that I can be.

To Hominy - Every step of this journey has been possible because of your steadfast presence in my life. Your love, care, and unyielding support have been my sanctuary. Thank you, from the bottom of my heart, for being my partner-in-crime.

Contents

1	Introduction	4
2	Background - A Brief History of Generative AI	5
2.1	Early Generative AI - Expert Systems	5
2.2	Generative AI Today - Deep Architectures	7
2.2.1	Examples of Generative AI models	7
2.3	Impact of Generative AI in Various Fields	8
3	Relevant Work	9
3.1	Music generation systems geared toward Musical Improvisation	9
3.1.1	<i>Voyager</i> — George E. Lewis	9
3.1.2	<i>GenJam</i> by John A. Biles	11
3.2	Music Generation via Deep Learning	13
3.2.1	<i>Learning Expressive Musical Performance</i>	14
3.2.2	<i>Generating Music with Long Term Structure</i>	16
3.3	Concerns and Choices	18
3.3.1	Input Data Type	18
3.3.2	Choice of Performance Environment	18
3.3.3	Choice of ML Framework and Model Size	19
4	The Transformer Model	21
4.1	Background	21
4.2	Model Structure	22
4.3	Inputs and Embeddings	22
4.4	Positional Encoding	22
4.5	Encoder / Decoder Stacks	23
4.6	Attention	25
4.6.1	Self-Attention	25
4.6.2	Multi-Head Attention	25
4.6.3	Scaled Dot-Product Attention	26
4.7	Position-Wise Feed-Forward Networks	27
4.8	Summing up	28
5	The Performance Environment: A prototype	29

5.1	For the non-technical performer	29
5.2	Details of the Patch	31
5.3	Pitch Tracking	33
5.3.1	Custom code within the Javascript Runtime	33
6	Project Design	35
6.1	Choice of Dataset	35
6.2	Dataset Analysis	36
6.3	MIDI Encoding and Input Representation	40
6.4	Building an input pipeline in Tensorflow	41
6.4.1	The BaseDataset class	41
6.4.2	The TestDataset class	42
6.4.3	The RandomCropDataset class	44
6.4.4	The SequenceDataset class	46
6.5	Streamlining training experiments	46
6.6	Model Details	48
7	Results and Analysis	51
7.1	Metrics	51
7.2	Custom Transformer	52
7.3	Keras Transformer	55
7.3.1	Experimenting with Hyper-parameters	55
7.4	Final Attempts	56
7.5	Conclusions	58
8	Reflections	60
8.1	Avenues for different implementations	60
8.2	Future Work	61
9	Appendix	63
9.1	Detailed Directory Structure	63
9.2	PyModel	64
9.2.1	Dataset Implementations	64
9.2.2	Custom Transformer Implementation	80
9.2.3	Keras Transformer Implementation	92
9.3	Analysis	97
9.4	TSImprovisor	99
	Bibliography	104

Chapter 1

Introduction

With the widespread proliferation of AI technology, deep architectures — many of which are based on neural networks — have been incredibly successful in a variety of different research areas and applications. Within the relatively new domain of Music Information Retrieval (MIR), deep neural networks have slowly gained prominence and have been successful for a variety of tasks, including tempo estimation, beat detection, and genre classification (Humphrey, Bello, and LeCun 2012).

Drawing inspiration from projects like George E. Lewis’s *Voyager* (Lewis 2000) and Al Biles’s *GenJam* (Biles 1994) two pioneering endeavors in human-computer interaction, this project attempts to tackle the problem of expressive music generation and seeks to create a Symbolic Music Transformer as a real-time musical improvisation companion, exploring the potential of AI to enhance the human experience of music. The Transformer, a groundbreaking model introduced in the 2017 paper “Attention is All You Need” (Vaswani et al. 2023) is at the core of Large Language Models (LLMs) like ChatGPT that have exploded in popularity around the world.

This paper discusses the historical context for the rise of generative AI technology, popular model architectures for sequence-to-sequence tasks that can be directly compared to our own, and dives deep into the details of the transformer architecture.

I successfully manage to implement the first iteration of a Transformer model that can generate musical output. While the model struggles to generalize to a variety of inputs — likely due to limited training resources and data used while training — it can learn the structure of the proposed MIDI encoding scheme (Oore et al. 2018) and can generate expressive MIDI performances. I also briefly present a working prototype of a performance environment built with Max/MSP which can parse auditory information in real-time and serve as the interface between the model and the musician. I document and evaluate various challenges and design decisions through a variety of means, as well as opportunities for future improvement and work.

Chapter 2

Background - A Brief History of Generative AI

Before diving into the history of Generative AI, it is important to provide a simple definition for our understanding of the field: Generative AI refers to a subset of artificial intelligence techniques that focus on generating new content in various forms — text, images, music, or other forms of data — given some or no amount of input. This technology has evolved significantly, from rule-based systems to advanced neural networks.

2.1 Early Generative AI - Expert Systems

Early attempts at generative AI models were based on deterministic, rule-based systems — also known as *expert systems* — that dominated the early 1960s. An early example of such a model from this era is Weizenbaum’s program ELIZA, a computer program built for the study of natural language communication between man and machine (Weizenbaum 1966). For its time the model was incredibly innovative in its use of rules to simulate the process of reading, understanding, and responding to textual input.

Such systems became tangible demonstrations of the power of small amounts of domain-specific knowledge to enable an intelligent decision-making process within programs in numerous areas of importance. However, such systems suffer from some key limitations:

- **Domain-Specific Knowledge** — Rule-based systems often require large amounts of domain-specific expertise to develop accurate rules. This is a time-consuming process of converting human-defined rules into machine logic and is also prone to human error and bias.
- **Scalability and Maintainability** — Expert systems can become exceedingly difficult to manage as the set of domain rules expands and changes. Given that

Men are all alike.

IN WHAT WAY

They're always bugging us about something or other.

CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.

YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.

I AM SORRY TO HEAR YOU ARE DEPRESSED

It's true. I am unhappy.

DO YOU THINK COMING HERE WILL HELP YOU NOT TO
BE UNHAPPY

I need some help, that much seems certain.

WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP

Perhaps I could learn to get along with my mother.

TELL ME MORE ABOUT YOUR FAMILY

My mother takes care of me.

WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU

My father.

YOUR FATHER

You are like my father in some ways.

WHAT RESEMBLANCE DO YOU SEE

You are not very aggressive but I think you don't want me to
notice that.

WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE

You don't argue with me.

WHY DO YOU THINK I DON'T ARGUE WITH YOU

You are afraid of me.

DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU

My father is afraid of everybody.

WHAT ELSE COMES TO MIND WHEN YOU THINK OF
YOUR FATHER

Bullies.

DOES THAT HAVE ANYTHING TO DO WITH THE FACT
THAT YOUR BOYFRIEND MADE YOU COME HERE

Figure 2.1: An example of a conversation with Eliza from Weizenbaum et.al.

such systems are unable to learn from new data, rules must be manually added to changed to ensure that a system can perform its task well. For example, MYCIN (Buchanan and Shortliffe 1984) was a system for diagnosing bacterial infections and recommending medical treatment, consisting of around 600 manually derived rules.

- **Complexity** — Due to the manually defined nature of rule-based systems, they may oversimplify complex problems and may fail to capture nuances in their target task, resulting in overly simple or inaccurate output.

2.2 Generative AI Today - Deep Architectures

Expert systems were instrumental in laying the groundwork for future developments in AI, establishing that "knowledge is power" (Buchanan 2005). Recognizing the limitations of expert systems, there was a gradual shift toward data-driven learning rather than rule-based learning, building the foundation for systems that would be able to adapt to new information and grow better at handling complex, non-linear relationships in data. The dominance of neural network architectures was cemented by the creation of the back-propagation algorithm (Rumelhart, Hinton, and Williams 1986), which allowed deep neural network architectures to learn and model representations in data very effectively. Nearly all of the generative models we see today incorporate elements of feed-forward neural networks which are integral to processing input data, transforming and learning data representations, and generating output.

2.2.1 Examples of Generative AI models

Generative Adversarial Networks (GANs) — Introduced by Ian Goodfellow and his colleagues in 2014, GANs represented a breakthrough in generative AI for image processing (Goodfellow et al. 2020). They consist of two neural networks, a generator, and a discriminator, that are trained simultaneously. The generator creates images, while the discriminator evaluates them. This competition drives the generator model to produce increasingly realistic images.

Recurrent Neural Networks — While neural networks can effectively model single or neighboring input entities, long-term relationships are important when dealing with sequential data, such as time series data. Recurrent Neural Networks (RNNs) were created specifically for such sequential data, where feed-forward connections from lower to higher layers are complemented by feedback connections from higher to lower layers. These connections can model delays in the signal and thus represent memory-like sequence modeling units. RNNs can therefore model temporal sequences. Several recurrent network models have been introduced, such as the long-short-term memory (LSTM) (Hochreiter and Schmidhuber 1997). Oore et. al uses such networks for expressive music generation, discussed in a subsequent chapter.

Variational Autoencoders (VAEs) — VAEs are based on the architecture of autoencoders, which consist of two main components: an encoder and a decoder. The

encoder compresses input data into a smaller, dense representation or *latent space*, and the decoder reconstructs the input data from this compressed form. Unlike standard autoencoders, VAEs introduce a probabilistic approach to the encoding process. (Kingma and Welling 2022). The Magenta team at Google has used VAEs to develop a variety of generative models in the music domain, such as *GrooveVAE*, which adds expressive dynamics to MIDI-sequenced drum beats to create a more human sense groove.

Transformers — Another kind of autoencoder model, the introduction of transformer models (Vaswani et al. 2023) marked a significant turning point in sequence modeling tasks by relying on self-attention mechanisms alone to compute representations of input and output. These models outperformed previous architectures by a wide margin, leading to their widespread adoption. Vaswani et al.’s paper is a must-read for understanding the foundation of modern transformer models and will be covered in greater detail in a subsequent chapter.

2.3 Impact of Generative AI in Various Fields

The rise of Generative AI marks a transformative era in the field of artificial intelligence, characterized by the ability of computers to create new content. These technologies have evolved into very sophisticated models, including the likes of OpenAI’s DALL-E, capable of generating highly realistic and creative visual content from textual descriptions, and ChatGPT-4, the leading large language model that can process multimedia content to interact with a user in a general context. These models are not just limited to creative tasks but also assist in data augmentation, simulation tasks, and more. (Richter et al. 2016). Generative AI is reshaping industries, posing significant ethical and societal questions about authorship, and redefining the boundaries of machine-assisted creativity and innovation.

Chapter 3

Relevant Work

3.1 Music generation systems geared toward Musical Improvisation

The subsequent two projects (while less technically relevant) were extremely inspirational to me as systems that privileged and made real the spirit of musical improvisation, in real-time. They provided me with ideas about how I might structure my model and the long-term goals I might hope to achieve in pursuing this work past the senior project.

In both cases, I was left rather mesmerized when I watched performances operated with these systems.

3.1.1 *Voyager* — George E. Lewis

Voyager is a “non-hierarchical, interactive musical environment that privileges improvisation” (Lewis 2000). In this environment, an improviser engages in dialogue with a computer-driven interactive virtual improvising orchestra. The program analyzes the human improviser’s performance and uses the analysis to guide a complex, improvised response to the player, as well as an independent response from its 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* ([Figure 3.1](#)) is called in

intervals of 5 - 7 seconds and 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 internal 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 (e.g. 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 allplayeroff   \ 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 *setresponse* subroutine in [Figure 3.2](#) runs completely independent of *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.

```

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

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 can spontaneously create music that is in no way influenced by the improviser, decisions made by Voyager have consequences that must 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.

3.1.2 *GenJam* by John A. Biles

GenJam (Biles 1994) 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 can provide feedback as to whether the output was "good" or "bad". The mentor's feedback is used to increment or decrement the fitness¹ of individual melodic ideas and serves as the environment in which musical ideas either persist or are removed from the stored hierarchy. Mimicking biological evolutionary processes, new ideas evolve by selecting the "better" ideas to be parents, breeding "children" ideas using musically meaningful mutations, and replacing the "worse" ideas in the population with these new children.

¹**Fitness** in genetic algorithms refers to a measure of how well a solution solves the problem at hand or how "fit" it is in the environment defined by the problem.

In their paper, Biles demonstrates how a four-bar phrase is mapped into the “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 1998). 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 improviser 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 Eb 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)”.
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



Figure 3.5: The melody from Figure 3.4 in “GenJam Normal Form”. Note that the triplet melodies cannot be preserved due to 8th note quantization

in preparation to play them back as its next four. Since all the input has been converted to GJNF, the phrase may be mutated 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 building my system, 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, helping the model adjust to different scenarios like trading fours, taking a solo, or comping over another musician’s solo.

3.2 Music Generation via Deep Learning

The following two papers are much less in the ideological domain but are projects put forth by the Magenta project at Google, an open-source research project that explores the role of machine learning in creative endeavors. If *Voyager* and *GenJam* are my inspiration to make machines that can improvise, these papers showed me how to make that possible.

3.2.1 Learning Expressive Musical Performance

In this paper, Oore et.al propose a novel approach to music generation, concentrating on direct performance generation (Oore et al. 2018). 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 other models created by Magenta like MelodyRNN or GrooveVAE, which separate the generation of musical phrases from their expressive attributes. Specific to this task, the authors emphasize the importance of using a dataset that is appropriate for the task of generating expressive musical performances, one 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 Competition Dataset, which contains MIDI-aligned performances of piano works on a Yamaha Disklavier, capturing musical notes as well as their expressive information with very high fidelity.



Figure 3.7: MIDI Encoding example proposed by Oore et.al.

The model employed is an LSTM-based network with three layers, each consisting of 512 cells. Oore et. al introduces a novel means of encoding MIDI data to an event-based representation of integers, encoding NOTE-ON, NOTE-OFF, TIME-SHIFT, and VELOCITY events, all of which can be retrieved from the MIDI message protocol. 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. An example of this is in Figure 3.7, which displays a snippet of a piano-roll with two notes and a gap between them. The encoding creates the following events in order: the SET-VELOCITY and NOTE-ON events for the first note, a TIME-SHIFT event covering the duration of the event, the NOTE-OFF event for the first note followed by the SET-VELOCITY and NOTE-ON events for the next note.

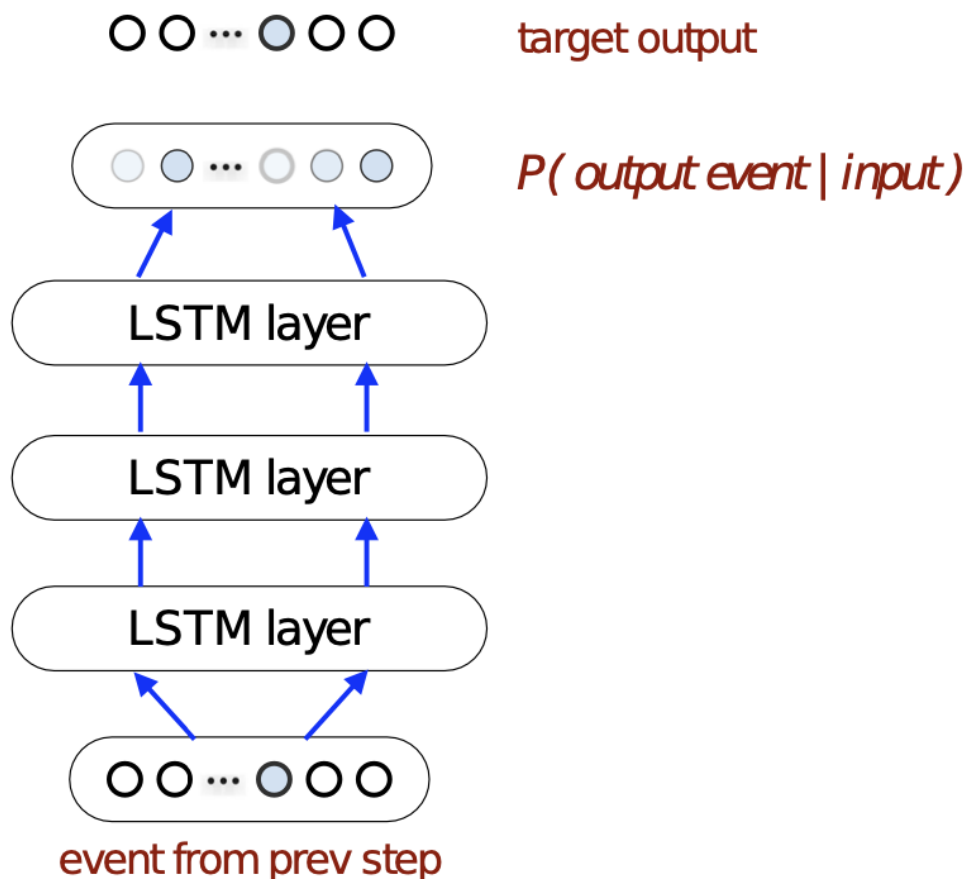


Figure 3.8: LSTM architecture used by Oore et. al

The network operates on one-hot encoding² over this event vocabulary, with each input to the RNN being a single one-hot 413-dimensional vector.

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 while system can generate output that sounds like a skilled pianist, the lack of long-term structure creates performances that are expressive but somewhat random. This research marked a significant step in the field of generative music, especially in its focus on the expressiveness of musical

²**One-hot encoding** is a method used to convert categorical data into a numerical format that can be understood and processed by machine learning algorithms. For example, in a color classification problem with a categorical variable representing the colors “Red”, “Green”, and “Blue”, we could represent them as the vectors [1,0,0], [0,1,0], and [0,0,1] respectively.

performances. The use of LSTM networks for this purpose shows promising results in capturing the nuances that make the music feel more human and less mechanically generated.

Key Takeaways — For myself, the MIDI encoding proposed by Oore et. al. was the most fascinating part of this paper, and the generated expressive performances are a testament to its success. However, this event representation also comes with certain caveats:

- While the event representation is far less memory intensive than raw audio or a multi-attribute grid-based representation like a piano roll, it does involve significantly longer data sequences when compared to similar symbolic data that lacks the same amount of fine precision for dynamics and timing.
- The event representation also compromises the relationship between position and time in the sequence, as TIME-SHIFT events are encoded into the sequence itself (e.g. a single note’s NOTE-ON and NOTE-OFF events can be very far away), compared to the position of an element being indicative of the time it was recorded.

3.2.2 *Generating Music with Long Term Structure*

In this paper, a Transformer with a modified attention mechanism is proposed for the task of expressive music generation with long-term structure (Huang et al. 2018). They argue that music has two dimensions along which relative differences matter more than absolute differences: Time and Pitch. This key information is already captured in the event-based input representation proposed by Oore et.al., which they use to encode the entirety of the MAESTRO dataset (Hawthorne et al. 2019) for training purposes.

The paper discusses and introduces an alternative version of the self-attention mechanism that is regulated by the distance between two points (Shaw, Uszkoreit, and Vaswani 2018), by constructing S^{rel} , an $L \times L$ dimensional logits³ matrix that can be used to modulate the attention probabilities for each head

$$\begin{aligned}
 AbsoluteAttention(Q, K, V) &= softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \\
 RelativeAttention(Q, K, V) &= softmax\left(\frac{QK^T + S^{rel}}{\sqrt{d_k}}\right)V \tag{1.4}
 \end{aligned}$$

Sparing the intricate details, the implementation by Shaw et.al. incurs an additional space complexity of $O(L^2D)$, where L is the length of the sequence and D is the

³In machine learning a **logit** often refers to the output of the model before it is passed through the softmax activation function to produce a probability.]

dimensionality of the input to the model because each attention head requires multiple intermediate tensors to create S^{rel} . This restricts its application to long sequences and makes the model less feasible for long sequences and real-time generation tasks.

Huang et.al. instead propose a memory-efficient implementation of relative attention by reducing the intermediate memory requirement to $O(LD)$ (Huang et al. 2018). They find an alternative means to construct S^{rel} with fewer intermediate tensors, by first constructing an alternate version of S^{rel} with all the logits in the wrong position, and then propose an in-place “skew” procedure to move the relative logits to their correct positions. The “skewing” procedure is illustrated in Figure 3.9 below and is detailed in Huang et al. The time complexity for both methods is still $O(L^2D)$, but they demonstrate that in practice, their method is 6x faster at input sequences of length 650, since it bypasses the need to manipulate larger tensors.

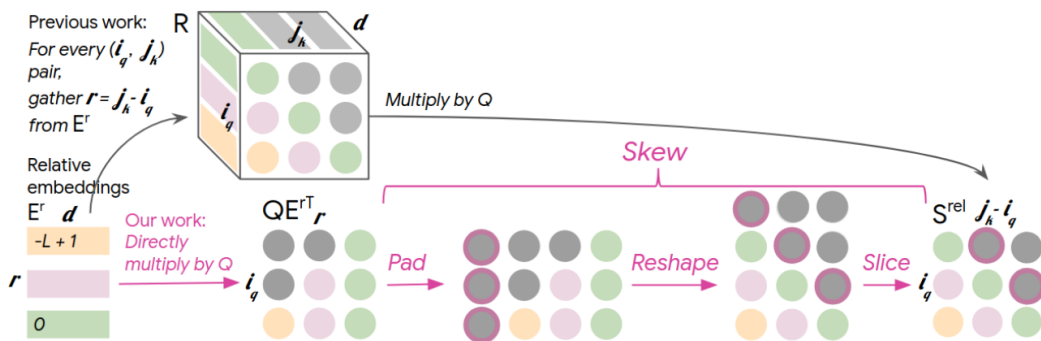


Figure 3.9: Skew Procedure from Huang et al.(2018)

Another incredible innovation from this paper was their ability to visualize the attention distribution from the generated input, showing how generated notes attend to previous notes in the sequence. This provides huge insight into the model’s inner workings.

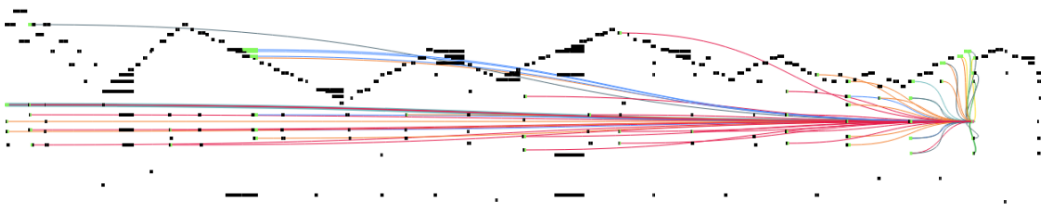


Figure 3.10: An example of a sequence generated by the Relative Attention Transformer. The query note is in the left-hand and attends to its immediate past neighbors to the earlier left-hand chords, with most attention lines distributed in the lower half of the piano roll (Huang et al. 2018)

Key Takeaways — The paper above was the first documented use of a Transformer architecture for music generation and confirmed my idea that Transformer architec-

tures could be well-suited to the task of Music Generation, even without the implementation of the relative attention mechanisms. It provided me with key insights into the implementation details of the Transformer as well as with hyper-parameter values I used in my attempt to train such a Transformer model.

3.3 Concerns and Choices

Given the scope of the project and the number of different works involved, I made many early design choices that greatly affected the choice of software and libraries that would be used throughout development.

3.3.1 Input Data Type

- **Concern** — 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 its overall speed. Furthermore, the choice of data format informs the choice of the dataset, and as discussed by Oore et. al, it is critical to use data of a high quality that matches the problem description to a high degree.
- **Choice** — I explicitly decided to use the MIDI encoding proposed by Oore et.al., given the success of expressive performances demonstrated by both the RNN and Transformer models described earlier. As mentioned previously, while this choice greatly lessens the processing requirement of the model compared to using raw audio, the length of the MIDI-encoded sequences extends far beyond the original MIDI file due to TIME-SHIFT events being encoded in the sequence, forcing us to grapple with excessively long sequences. This also compromises the sequence’s relationship between position and time, while placing the responsibility of extracting a meaningful stream of MIDI data from the model environment.

3.3.2 Choice of Performance Environment

- **Concern** — Given the project’s aim to create a live, improvisation-centric system, we first require an environment/software that can perform multiple tasks:
 1. Record live auditory information
 2. Parse 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. Considering the unique constraints of the performance environment, the model would likely be trained and evaluated within a separate environment that privileges tasks such as manipulating the training data, setting up data processing pipelines, and training the model with an ML framework of choice.

- **Choice** — I chose to construct the model’s performance environment in Max/MSP (Cycling ’74 2023). 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 objects that directly address our needs, like a pitch-tracking module that can be used to help parse live audio to midi. Max programs are known as ”patches” and involve connecting a graph of various Max objects. Max also supports composability and reusability, with the ability to define and reuse custom sub-patches across multiple projects.

Furthermore, Max features an internal Node.js runtime engine that can communicate directly with Max patches via the Max-For-Node API. Leveraging this, we can exploit the full flexibility of Javascript/Typescript in our patch in a manner that is completely abstracted from the performer, housing our machine-learning model within the Node.js runtime and doing any additional processing as required.

3.3.3 Choice of ML Framework and Model Size

- **Concern** - The most popular ML frameworks available today are PyTorch and Tensorflow/Keras, both of which leverage Python as the main language to train ML models. The ML framework of choice must be flexible enough to support models being exported from the training environment easily, and the performance environment must be able to host the model in its exported form. Our model also has 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 run relatively quickly.
- **Choice** — While PyTorch is reportedly easier to use and prototype models with, I have chosen to use Tensorflow/Keras as my Machine learning library of choice specifically because of my familiarity with it and because has a pre-existing Javascript port, *Tensorflow.js*, which supports run models trained with the regular Tensorflow backend. 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.

Given that I am trying to keep the inference time of the model to a minimum, I choose to work with smaller models of only 1-2 encoder-decoder stacks, rather

than the usual practice of using 4-8 encoder-decoder stacks. This helps to keep the model small and may force it to extrapolate and generalize with limited parameters.

Chapter 4

The Transformer Model

4.1 Background

In the landmark paper “Attention Is All You Need” (Vaswani et al. 2023), 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 and Schmidhuber 1997) and Gated RNNs (Chung et al. 2014) built with encoder-decoder architectures for sequence-to-sequence generation. Recurrent models typically handle computation on a sequence based on the positions of symbols in the input and output. At each step, the RNN generates a series of hidden states such that the most recent hidden state, h_t , depends on the previous hidden state, h_{t-1} , and the input for the current position (t). The task performed at each element in the sequence is the same, just with different “memory” of what came prior, hence the name “recurrent”

This sequential nature of processing means that an RNN only looks at one piece of the input data at a given time, rather than looking at larger chunks of the sequence. When processing a very long sentence or a large set of sentences, these models struggle because they have to process each part of the sequence in order. This makes parallelization ¹ difficult and requires a lot of memory, as the model has to keep track of the information through each step of the sequence. Recent research has made notable strides in improving computational efficiency through methods like factorization techniques (Kuchaiev and Ginsburg 2017) and conditional computation (Shazeer et al. 2017) while enhancing model performance. 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 et al. 2017). Nonetheless, the core constraint of sequential computation has remained as a bottleneck to the ability to train RNNs on large amounts of data.

¹**Parallelization** in computing refers to the process of dividing a task into smaller parts that can be processed simultaneously on a computer, compared to one after the other in sequence. When done correctly, this can greatly improve the overall speed of a process

The Transformer model, however, does away with the recurrent aspect of RNNs entirely and relies solely on dot-product attention mechanisms to draw global relationships between input and output. While this intensifies the Transformer’s memory requirement due to the $O(n^2)$ dot-product computations, it makes the model highly parallelizable for training on large datasets.

4.2 Model Structure

Similar to many autoencoder models, the Transformer has an encoder-decoder structure where the encoder maps an input sequence of representations $X = (x_1, \dots, x_n)$ to some sequence of continuous feature representations $Z = (z_1, \dots, z_n)$. 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 with stacked self-attention layers and fully connected layers in both the encoder and decoder, as shown in Figure [Figure 4.1](#).

4.3 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 often embedded into continuous vector representations — *feature vectors* — using an embedding layer such that each 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.

4.4 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:

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

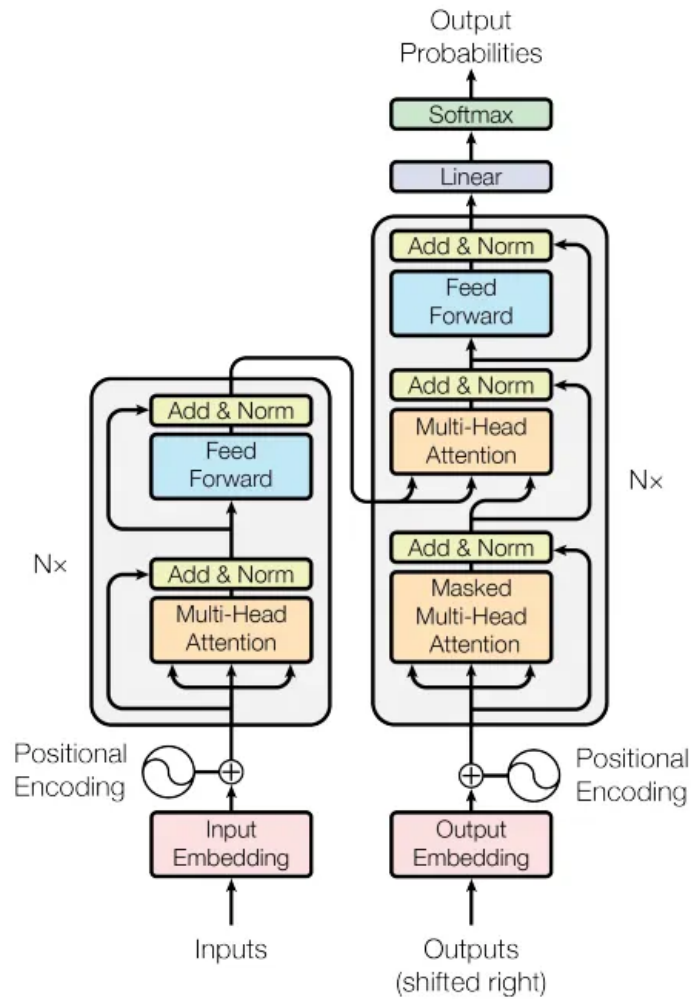


Figure 4.1: The Transformer Model Architecture. Left side is the Encoder, Right side is the Decoder (Vaswani et al. 2023)

In Formula 1.1, pos is the position in the token and i is the dimension of the positional encoding that corresponds to a sinusoidal function. While it may seem intuitive to use simple integer values, the benefit of using sine and cosine is that the output of sine and cosine is normalized with $[-1, 1]$, unlike integer numbers which can grow very large. This removes the need for additional training to determine position since a unique value is generated for each position index.

4.5 Encoder / Decoder Stacks

Encoder — The Encoder is composed of a stack of N identical layers, each with 2 sub-layers, as shown in the left block in Figure 4.1. The first is a multi-head self-attention layer, followed by a fully connected feed-forward network. Each sub-layer

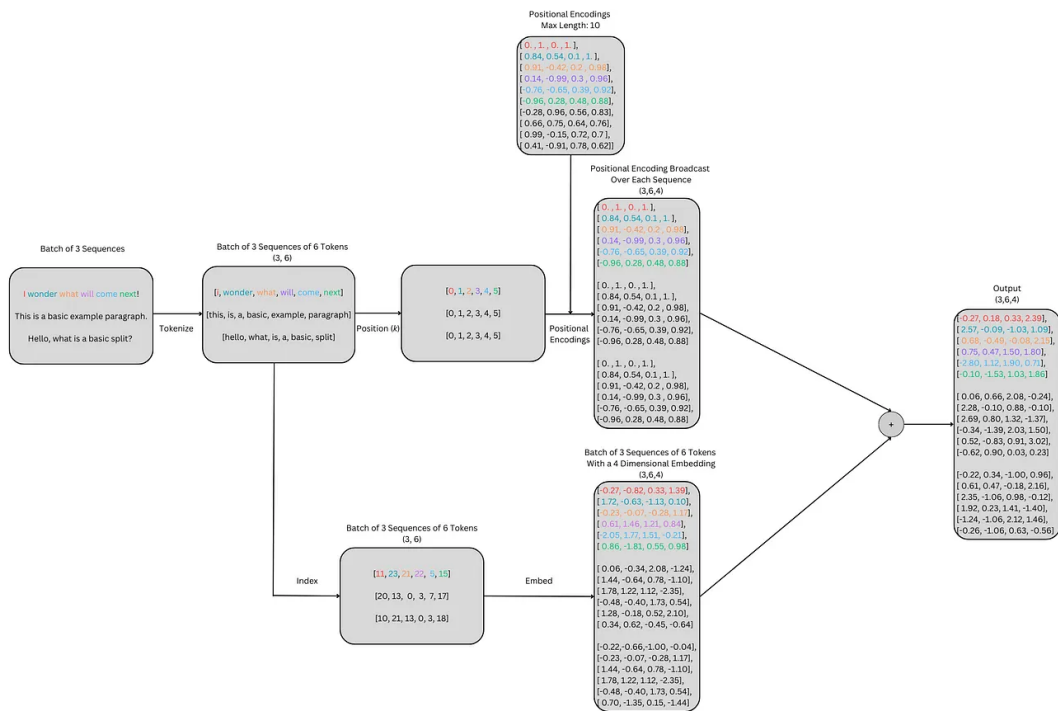


Figure 4.2: Visual Example of Positional Encoding being added to an embedded vector representation (Taken from Hunter Phillips <https://medium.com/@hunter-j-phillips/positional-encoding-7a93db4109e6>)

has a residual connection², followed by layer normalization³ where the output of each sub-layer is equivalent to $Normalize(x + sublayer(x))$. In the simplest terms, our Encoder receives a sequence and converts it into a form of contextual input for the Decoder.

Decoder — The Decoder is also composed of a stack of N identical layers, but each with 3 sub-layers, as shown in the right block in Figure 4.1. From the bottom up, the Decoder uses 2 multi-head attention layers. The bottom-most or first layer is completely auto-regressive. The second multi-head attention layer receives the encoder outputs and is often called the **Cross-Attention layer**, where attention values are computed not auto-regressive but, across the encoder output as well as the data flowing through the Decoder. In simple terms, it is in this layer that the data flowing through the Decoder is influenced by the encoded context received from the Encoder. Like the Encoder stack, we similarly employ residual connections and layer

²**Residual connections** is a technique used in deep learning architectures, to allow the flow of information and gradients through layers of a network more directly, by skipping one or more layers

³**Layer Normalization** is a technique in machine learning used to stabilize the distribution of the activations in a neural network, performing normalization for every individual data point across all features

normalization around each sub-layer.

4.6 Attention

4.6.1 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. In an auto-regressive scenario, these are the same vectors!
- **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.

4.6.2 Multi-Head Attention

Instead of performing a single attention function over the entire Q, K, and V vectors of dimension d_{model} , which is the dimensionality of the input to the Multi-Head Attention Layer, Vaswani et.al. propose a linear projection of the queries, keys, and values h times to d_{query} , d_{key} , d_{value} dimensions respectively, where each of these corresponds to d_{model}/h . This is achieved by placing linear layers before the multi-head attention layer and projecting the queries, keys, and values into a smaller space. From the collection of h projected versions of the queries, keys, and values, the attention function is applied in parallel to each of these projected versions of Q, K, and V. The output of each of these ‘heads’ is a which are concatenated and once again projected to get our final output back in the d_{model} space, as shown in [Figure 4.3](#). This parallelism allows

the model to focus on different parts of the input sequence differently for each head and capture different relationships within the sequence.

$$\begin{aligned} \text{MultiHeadAttention}(Q, K, V) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \\ \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \quad (1.2)$$

where the projections are done via the following parameter matrices:

$$\begin{aligned} W_i^Q &\in \mathbb{R}^{d_{\text{model}} \times d_k} \\ W_i^K &\in \mathbb{R}^{d_{\text{model}} \times d_k} \\ W_i^V &\in \mathbb{R}^{d_{\text{model}} \times d_v} \\ W^O &\in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}} \end{aligned} \quad (1.3)$$

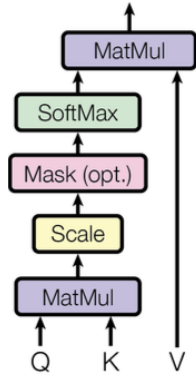
As described by Vaswani et. al, the Transformer uses multi-head attention in three different ways:

- In the "Encoder-Decoder attention" layers, the *Queries* come from the previous Decoder layer, while *Keys* and *Values* come from the output of the Encoder. This allows every position in the decoder to attend to all positions in the input sequence. This mimics the typical Encoder-Decoder attention mechanisms in other sequence-to-sequence models and is often referred to as **Cross Attention**, and is the middle attention block of the Decoder in [Figure 4.1](#).
- In the Encoder self-attention layers, all of the keys, values, and queries come from the output of the previous layer and can attend to all positions in the Sequence. This is often called **Global Attention**
- Finally, the lower attention block of the Decoder in [Figure 4.1](#) uses autoregressive self-attention, attending to all positions in the Decoder up to and including the current position. Think of this as making sure that the Decoder cannot predict the next event based on future events. To ensure this, an input mask is added within the scaled dot-product attention (setting to $-\infty$), masking out values in the input of the softmax that correspond to illegal connections. This is often called **Causal Self-Attention**.

4.6.3 Scaled Dot-Product Attention

In particular, the form of attention implemented by Vaswani et al. is Scaled-Dot-Product Attention. First, the dot products of the query vector with all of the key

Scaled Dot-Product Attention



Multi-Head Attention

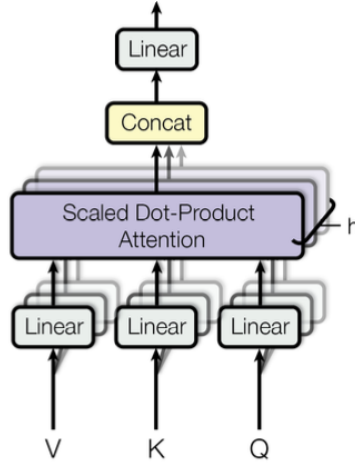


Figure 4.3: (Left) Scaled Dot-product Attention. (Right) Multi-head Attention with several layers in parallel. (from Vaswani et.al.)

vectors are calculated. It is then divided by $\sqrt{d_k}$ and converted to a probability distribution by applying a softmax⁴ function. We divide by $\sqrt{d_k}$ to counter dot-product values from growing too large which can cause vanishing gradients.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1.4)$$

4.7 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 ReLU⁵ activation in between:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \quad (1.5)$$

⁴The **softmax** function is a mathematical function commonly used in machine learning, particularly in the context of classification tasks. It transforms an output vector of real-valued scores (often called “logits” in machine learning contexts) into a vector of values that sum up to one, effectively representing probabilities. This is often used at the end of a neural network to determine what is the most likely output class for a classification task given a certain input

⁵**ReLU** stands for Rectified Linear Unit, and it is a type of activation function commonly used in neural networks. The ReLU function is defined mathematically as $f(x) = max(0, x)$. (i.e the function outputs x if it is greater than 0, or 0 otherwise.)

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 $d_{ff} = 2048$.

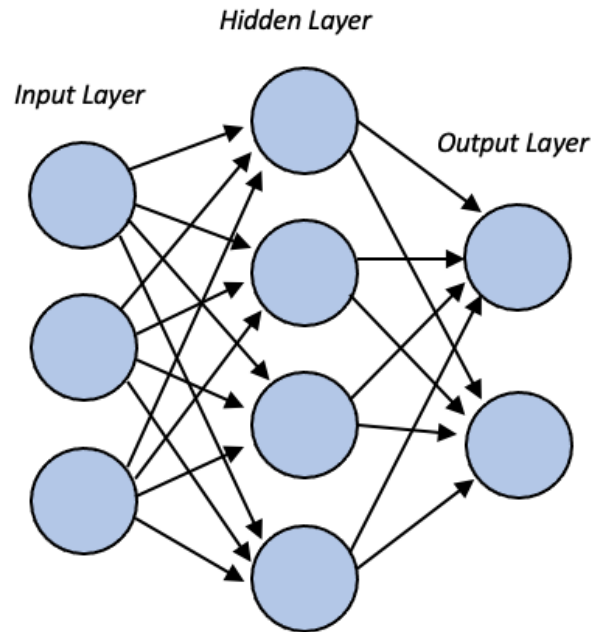


Figure 4.4: Example of a simplified 3-layer feed-forward network

4.8 Summing up

Overall, the Transformer is an extremely dense and complicated architecture with many different moving parts. It marks a significant advancement in the field of deep learning and natural language processing. Its unique approach to handling sequences through attention mechanisms alone allows it to capture long-range dependencies and contextual relationships in data more effectively than traditional sequence-based models like RNNs and LSTMs. This quality makes Transformers especially powerful for tasks like machine translation, text generation, and semantic analysis, where understanding context is crucial.

The parallelization capabilities of Transformers have also set a new standard, enabling the use of much larger datasets. The adaptability of the Transformer model has led to its rapid adoption across different domains beyond NLP, including computer vision and in my case, even music generation.

To understand this architecture that has revolutionized the AI world as deeply as possible, I implemented my own version of it. All code is referenced in the [chapter 9](#), under [subsection 9.2.2](#).

Chapter 5

The Performance Environment: A prototype

Before working on training of the model itself, I created a prototype for our model environment and tested it against a pre-built music generation model, ImprovRNN, taken from Google’s Magenta Library. ImprovRNN takes in a quantized note sequence as well as an optionally provided chord progression and returns a candidate continuation of the sequence. The description of this performance patch is brief because I could not test it with our trained Transformer models due to time constraints. What I do want to show, however, is the visual nature of the program. I often feel that Machine Learning research in the music domain is too far removed from musicians, and it was important to me that my interface was approachable for a non-technical musician.

5.1 For the non-technical performer

Within Max, I have created a patch called `interface.maxpat` that serves as the interface between the performer and the entire system. To abstract and hide the intricate details of the Max conveniently features a “presentation mode”, making it particularly approachable for a non-technical performer who might use the system in the future.

The patch has 4 distinct control sections:

1. **setTimeSettings** — Here the performer can pre-set settings based on the kind of music they would like to play. This includes setting a time signature using *numbeats* and *beatvalue* (eg. 9/8 would be 9 beats, each with an 8th note value). The performer can also alter the quarter note tempo and as well as `stepsPerQuarter` variable, which represents how many times a quarter note should be evenly split, which is useful for any kind of note quantization.
2. **Script Messaging** — After the time settings have been set, the performer can

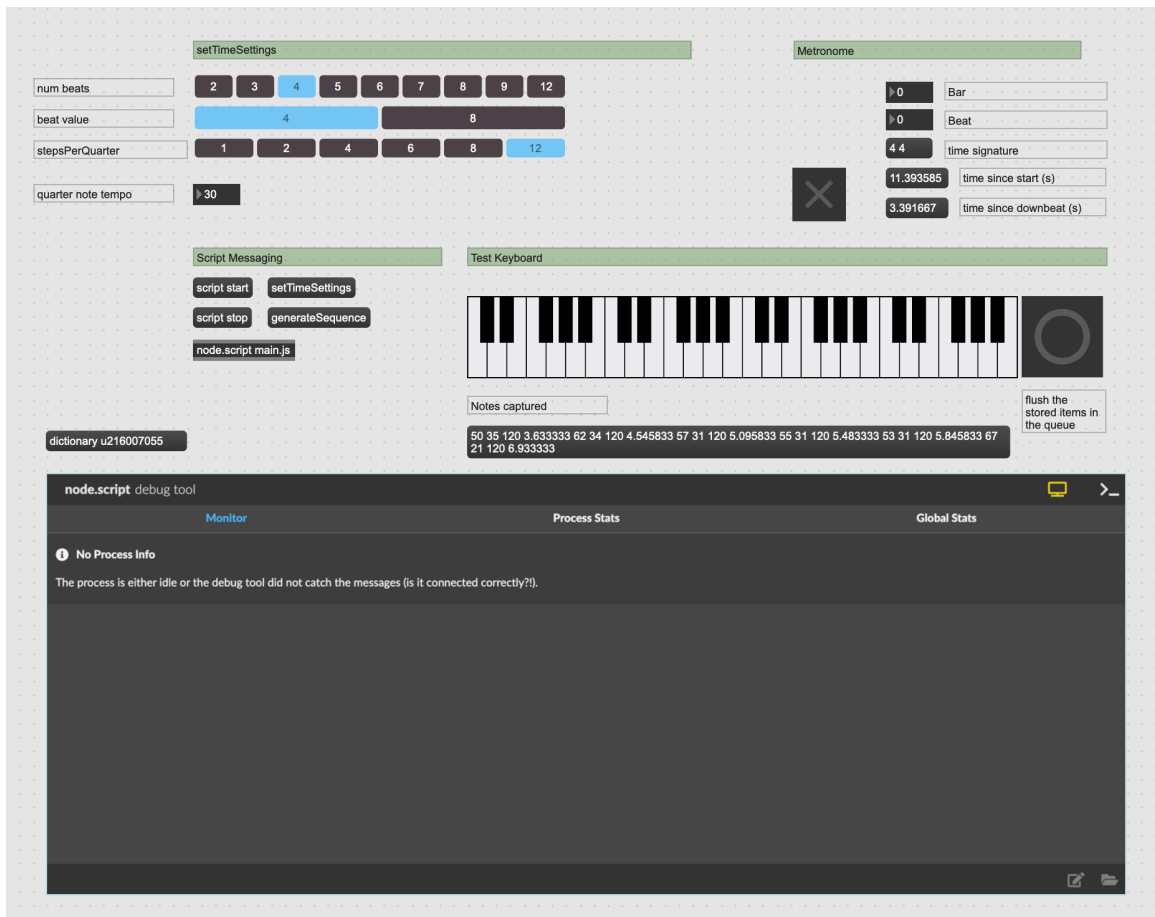


Figure 5.1: Performer interface using Presentation mode in Max, which lets us selectively show Max objects whilst hiding all of the necessary “cabling”

start the script “main.js” the entry point file for our Javascript interface, by clicking on `script start` They can also manually call `setTimeSettings` to reset the established settings if they change it once the script has already started

3. **Metronome** — Once the script has started, the model has loaded and the time settings have been set, the performer can click the toggle within this section to start the metronome for the piece. The metronome here is created using Max’s *global transport* object, which is a crucial component for tracking the input midi notes.
4. **Test Keyboard and Pitch Tracking Module** — In its current state, the environment works with the test keyboard rendered within Max itself which can send out note onsets as well as note velocities. This can be readily swapped out with a pitch-tracking module. Next to the keyboard is also a “bang” object that will manually flush our note queue for testing purposes. In reality, the note queue will automatically flush its contents to the Javascript runtime at set intervals based on the time settings set by the performer.

With a working Pitch-Tracking Module, a performer need only click start and the model would automatically be loaded and listening, ready to generate output sequences. These output sequences can then be sent to any available digital audio workstation via the `noteout` object in Max to play the note back.

5.2 Details of the Patch

To send all the required information to our Javascript Runtime, I first instantiated a metronome that ticks alongside the host computer’s internal clock. After the metronome has started, the *global transport* object mentioned earlier is simultaneously able to keep track of the current bar and beat (based on the time settings provided) as well as the total ticks elapsed since the metronome started.

Given that we are working with MIDI in real-time, we have to manually calculate the time data that is naturally present in a MIDI file. Every time a note is triggered via the test keyboard, we can immediately retrieve a note value as well as a velocity value. It also then uses the values from the global transport object to determine how long the note was played for, as well as the delta time between the start of the note, and the start of the bar. These four variables are packed into a queue every time a note is sounded. Every time a new bar starts, Max then flushes the note queue entirely into our Javascript Runtime.

Our Javascript process then parses these notes into a stream of MIDI note objects, which are then quantized and passed to the model using methods from the `Magenta.js` library.

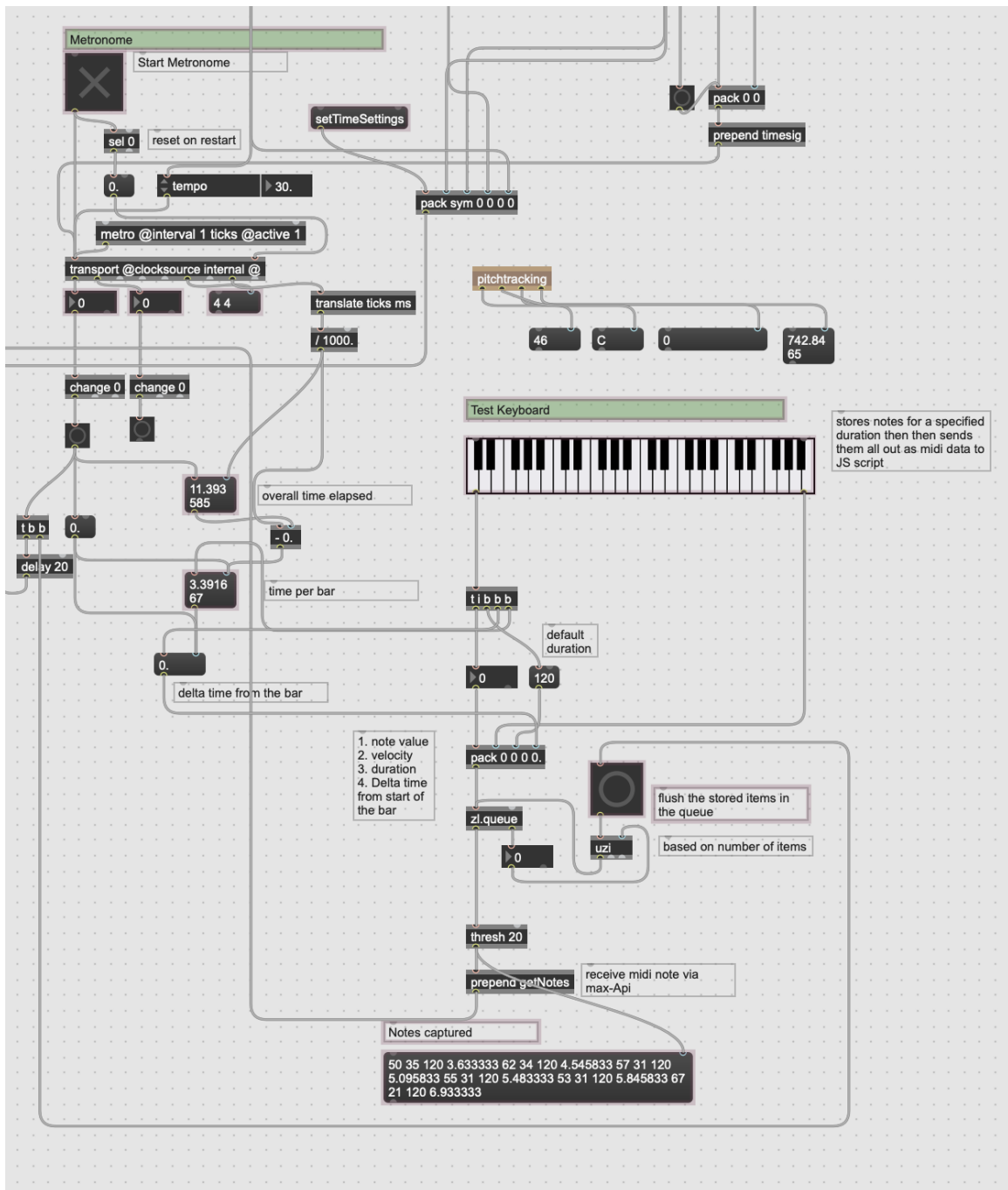


Figure 5.2: The Max patch collecting note events into a queue

5.3 Pitch Tracking

As it currently stands, the pitch-tracking module has already been included in the patch but has not yet been connected to the patch to replace the test keyboard. The pitch-tracking module is currently built to track monophonic pitch with the *Sigmund* object created by Max Miller Puckette. While polyphonic pitch tracking is notoriously difficult and not within the scope of this paper, one possible method is to use harmonic partial subtraction alongside the *Fiddle* object in Max (Robertson and Plumbley 2009).

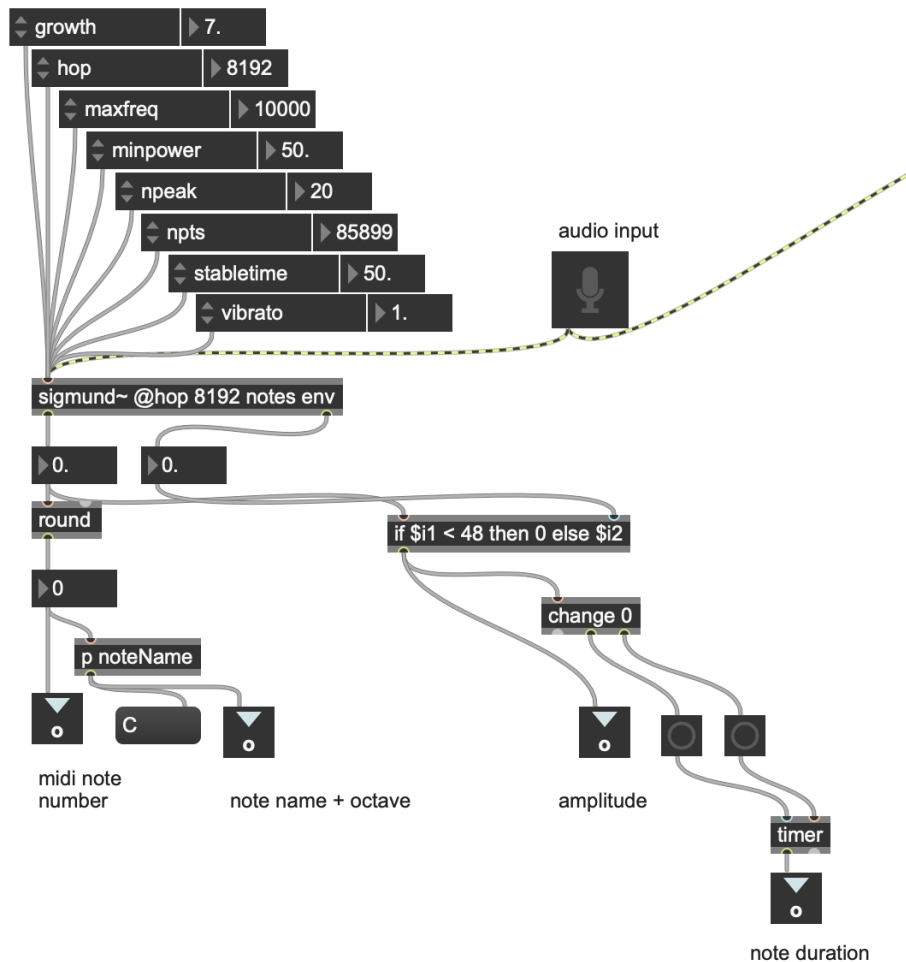


Figure 5.3: Simple Pitch-tracking with Sigmund

5.3.1 Custom code within the Javascript Runtime

I also wrote custom Typescript code that uses the `Max-for-Node` API to communicate and send messages between the patch and its Node.js runtime that houses the model. The most important part of this code is the `Improvisor` class, which handles model initialization/loading, and a bulk of the communication. All Typescript code can be

found in [chapter 9](#) under 9.4.

Chapter 6

Project Design

6.1 Choice of Dataset

For this project, I decided to use the MAESTRO dataset (Hawthorne et al. 2019) used by Huang et.al 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 3-millisecond fine alignment 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 tracks that have been published previously and have enabled significant advances in MIR tasks, I chose to use the MAESTRO dataset for several reasons. First, MAESTRO is much larger than most other similar datasets, making it a perfect choice for training transformer models that notoriously require a large amount of data. Second, the quality of recorded MIDI performances in the MIDI dataset is extremely high, collected from several annual runs of the International Piano e-competition. Virtuoso pianists perform on Yamaha Disklaviers which utilize an integrated high-precision MIDI capture and playback system, while simultaneously having an acoustic quality comparable to the upper echelon of concert grand pianos. The recorded data is granular enough that the audition stage of the competition is remotely judged by listening to contestant performances reproduced on another Disklavier instrument (hence the name 'E-Competition').

Below is information about other similar datasets that were considered for the task of expressive music generation (Hawthorne et al. 2019):

1. **MusicNet** (Thickstun, Harchaoui, and Kakade 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.

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 6.1: Comparing MAESTRO to other datasets (from Hawthorne et al.)

One advantage of MusicNet is that it contains instruments other than piano (not counted in the table above) and a wider variety of acoustic environments, making it more flexible for ML tasks that deal directly with audio.

2. **MAPS** (Emiya et al. 2010) contains Disklavier recordings like MAESTRO and synthesized audio created from MIDI files that were entered via a sequencer. Unfortunately, the synthesized audio makes up a large fraction of the MAPS dataset, which is not as expressive as the MAESTRO tracks captured from live performances, making it less attractive for the task of musical improvisation.
3. **Saarland Music Data (SMD)** (Müller et al. 2011) is very similar to MAESTRO in that it contains raw audio recordings and aligned MIDI of human performances on a Disklavier, but is far smaller than the MAESTRO dataset.

6.2 Dataset Analysis

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 6.2: Maestro Dataset Split (From <https://magenta.tensorflow.org/datasets/maestro>)

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 notes 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 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 as well as within each composer's corpus of performed work. However, Hawthorne et al. also note that maintaining these proportions is not always possible because some composers have too few compositions in the dataset. The graphs below show that while the dataset has a large amount of data, there is an especially huge disparity in the amount of music for each composer in the training set. While this is mirrored within the validation and test sets as well, this fundamentally implies that any model trained on the entire dataset will be biased toward the musical style of a specific composer.

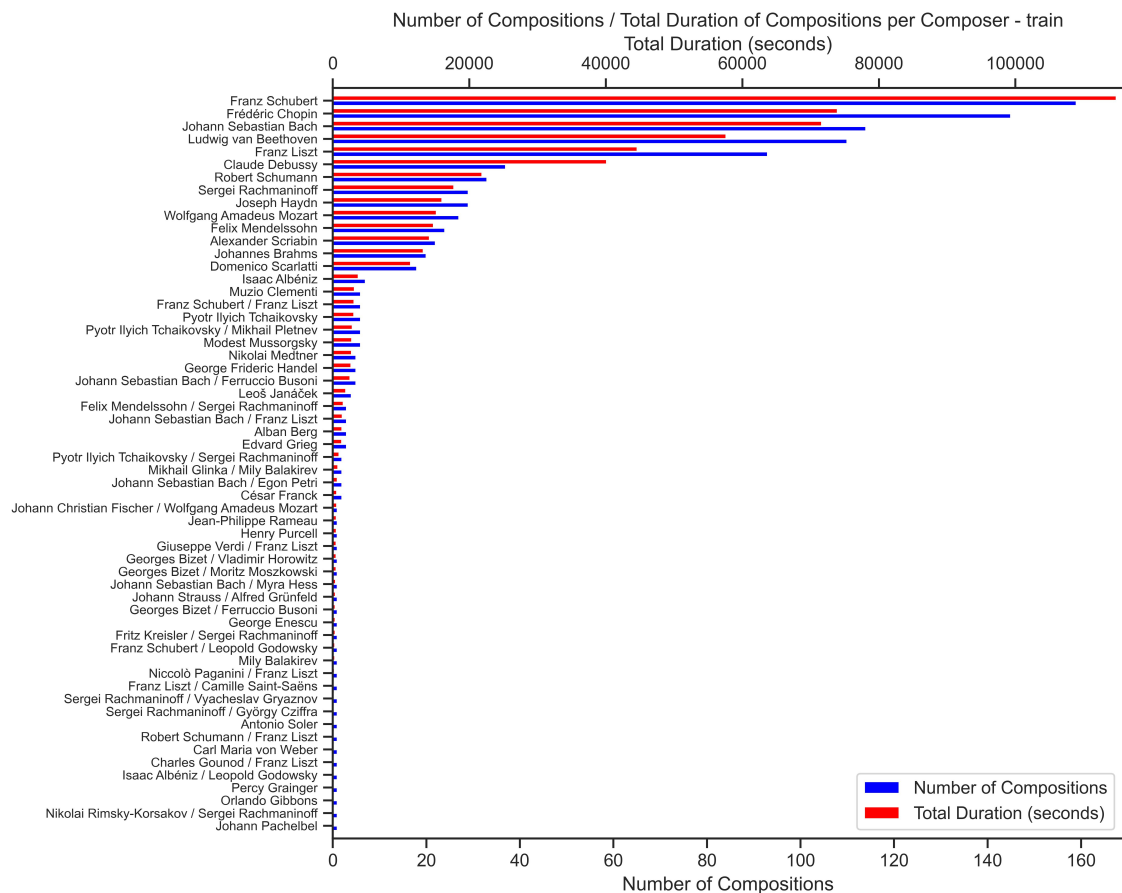


Figure 6.3: Histograms of Counted Compositions / Total Duration by Composer in the MAESTRO train split

Furthermore, given that the MAESTRO dataset is constructed from multiple years

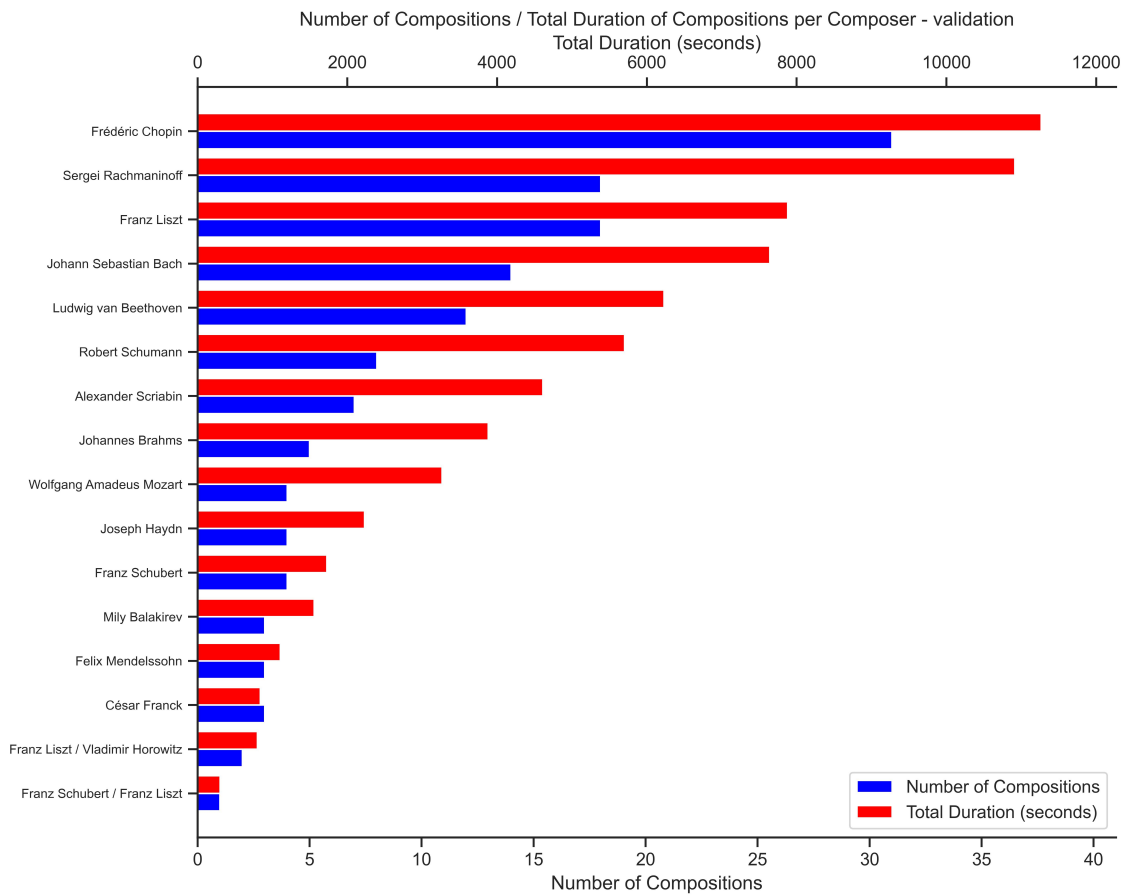


Figure 6.4: Histograms of Counted Compositions / Total Duration by Composer in the MAESTRO validation split

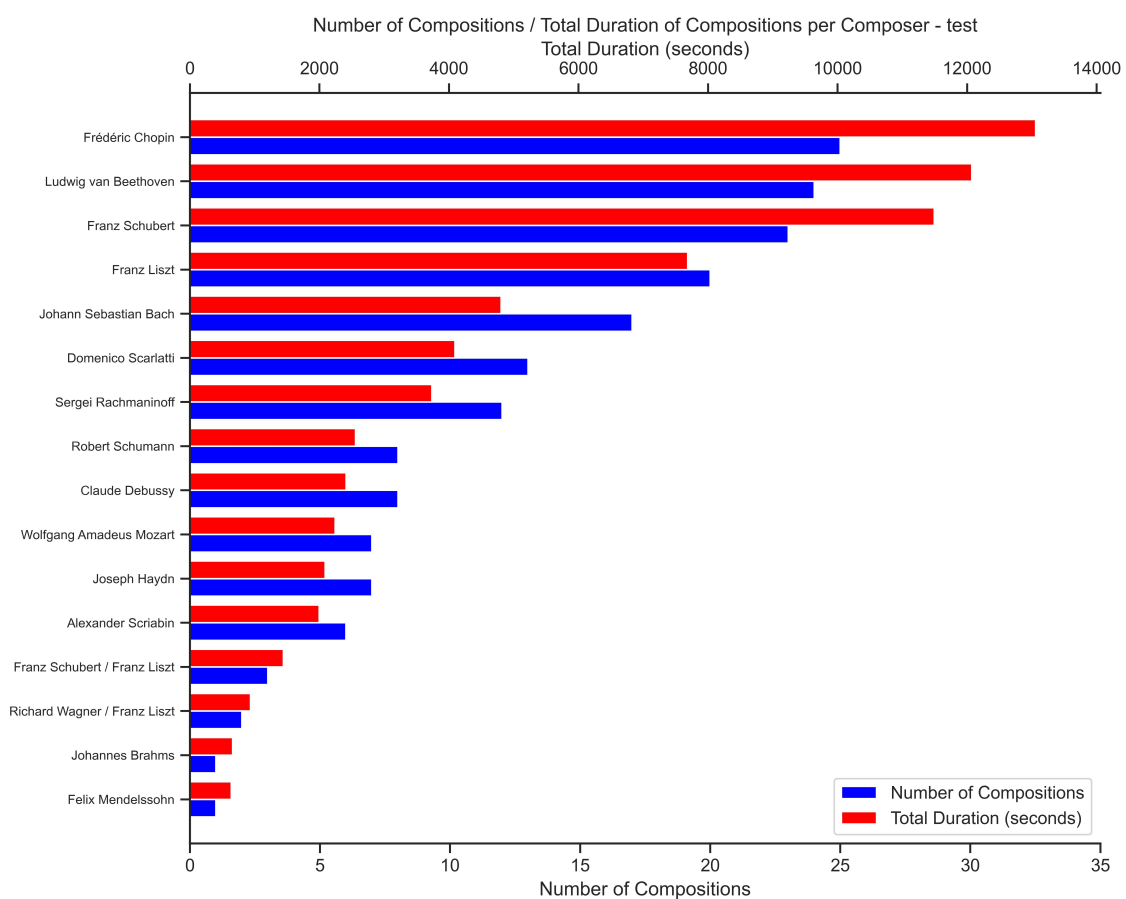


Figure 6.5: Histograms of Counted Compositions / Total Duration by Composer in the MAESTRO test split

of piano competition entries, some pieces show up far more often than others in the training. While different interpretations of a piece from different performers may allow the model to have more variation in its expressive output, this may also overexpose one piece of music to the model during training, biasing a specific style and possibly even a specific tonal center(s). Hawthorne et. al discusses this in their paper but does not disclose any data to show how severe this problem is. Using a simple regular expression, we find 451 duplicate files or 46.89% of the original training set. The validation and test sets remain the same due to the conditions of the original split mentioned above.

6.3 MIDI Encoding and Input Representation

I adopt the sparse, MIDI-like event representation from Oore et al discussed earlier. The vocabulary of this encoding consists of 128 NOTE-ON events, 128 NOTE-OFFs, 100 TIME-SHIFTS allowing for expressive timing at 10ms, and 32 VELOCITY bins for expressive dynamics, as shown in [Figure 3.7](#).

Rather than implement the full MIDI encoding process myself, I found and used a working implementation online: [Midi-neural-preprocessor](#) developed by [Kevin-Yang](#). I examined the code to ensure the implementation matched the description by Oore et. al and tested it by writing unit tests and comparing MIDI files with their original counterparts after they had been encoded and decoded. My unit tests counted every recorded event across all files, ensuring that the registered note events fell within the range of an actual grand piano (MIDI note range 21 - 108) and that there were an equal number of note-on and note-off events for each note recorded. Since the encoding process for each MIDI file was quite long, I wrote a preprocessor script that employs multi-threading to speed up the encoding process of each MIDI file before saving it to a raw binary format. This made the encoding process across the dataset about 10x faster.

Below are some interesting statistics about the collected MIDI event sequences.

Event Statistics by split				
Split	Train	Validation	Test	All
Average	24166	19112	17140	22648
Maximum	100467	99866	67859	100467
Minimum	5	2258	1837	5
Std Deviation	17942.411	16304.913	12660.855	17342.889

Table 6.1: Event statistics captured by counting every encoded event across all files

The table above provides insight into the huge variance in sequence length across examples, especially within the training set, and is dealt with accordingly in my dataset implementations.

6.4 Building an input pipeline in Tensorflow

Given that I was dealing with a substantial amount of sequential data across thousands of files and constrained by limited memory/compute resources, it was of paramount importance to experiment with different data pipelines and consider which was best suited for me given the limited computing 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 as elaborated below.

Raw Sequence: [12, 35, 16, 237, 238, 14, 18, 19, 78, 113, 114] ~ 11 events

For windows of size 5:

Window 1 : [12, 35, 16, 237, 238]

Window 2 : [35, 16, 237, 238, 14]

Window 3 : [16, 237, 238, 14, 18]

Window 4 : [237, 238, 14, 18, 19]

Window 5 : [238, 14, 18, 19, 78]

Window 6 : [14, 18, 19, 78, 113]

Window 7 : [18, 19, 78, 113, 114]

Total number of events: 5 * 7 = 35

Figure 6.6: Generating examples from large sequence using a rolling window

- Assuming a fixed overhead for each list O and a fixed integer size I , For each sequence of length N , the memory footprint is equal to $I * N + O$
- For a stride of S , the number of rolling windows of size L we can extract from the sequence (where $N > L$) = $\lfloor (N - L) / S \rfloor$
- Total number of events across all examples = $\lfloor (N - L) / S \rfloor * L * I + (O * L)$. Therefore **memory requirement grows quadratically as L increases**
- e.g. For a sequence of 32-bit integers of size 20,000 (80KB), a window size of 1024, and stride 1, the number of events generated across all examples = 19432448 (77.73 MB). **That's nearly 1000 times larger, and only for a single file!**

6.4.1 The BaseDataset class

The BaseDataset class serves as the parent class for all subsequent various dataset implementations and accomplishes two key functions. It first loads metadata from the JSON file provided with the MAESTRO dataset, which tags each MIDI file to its respective composer and title, year of creation, whether it should belong to the

training, validation, or test split, and more. A separate variable, the `fileDict`, stores key-value pairs of the MIDI file's key within the JSON file, as well as the path to its encoded MIDI representation when `get_encoded_files()` is called.

The `get_encoded_files()` function also implements additional logic that filters out files that do not meet a minimum duration or a minimum number of encoded events (see code 6.7). Based on the recorded event statistics above, some files can be considered outliers in terms of their duration and the length of their encoded event sequence, the latter being a more useful metric since the length of a piece's encoded event sequence is not necessarily tied to its duration, but to the number of individual midi events generated during a full midi performance. The filtering removes pieces whose overall length is less than our training window as they are less relevant to the case of "improvisation" where the goal is primarily to generate more music within a certain context. Training on such shorter training examples might pre-empt the model to stop early or generate no response at all, thinking that a piece has ended, because the teacher-forced sequence passed to our decoder might be significantly padded with 0 values.

```
1 #Remove midi files that are too short in length by number of events
2 if min_event_length is not None:
3     i = 0
4     while i < len(midi_filenames_from_json):
5         #open the file and get the length
6         with open(midi_filenames_from_json[i], 'rb') as f:
7             if self.data_format == "pickle":
8                 data = pickle.load(f)
9             else:
10                data = np.load(f,allow_pickle=True)
11        if len(data) < min_event_length:
12            del midi_filenames_from_json[i]
13        i+=1
```

Figure 6.7: Code from `BaseDataset.py` to filter files by event sequence length

6.4.2 The TestDataset class

The `TestDataset` class inherits from the `BaseDataset` class to accomplish a few specific goals. First, the `TestDataset` class implements `mockTFDataset_from_scale()`, which generates test sequences based on major/minor scales and arpeggios. These are simple sequences of integers representing MIDI note numbers with no encoding of velocity or time, serving only as a baseline test to verify that our model architecture could learn and overfit to a basic training set sequence.

The second purpose of the `TestDataset` class was to explore how much data might realistically be used with an in-memory dataset. To experiment with this, the `TestDataset` class implements a method `retrieve_files_by_maestro_split()` to first separate encoded MIDI paths by their intended split indicated in the MAESTRO metadata JSON file, while specifying the number of files to keep from each split as

```

1 import tensorflow as tf
2
3 #scales in C
4 MAJOR_SCALE = [24, 26, 28, 29, 31, 33, 35]
5 MINOR_SCALE = [24, 26, 27, 29, 31, 32, 34]
6 MAJOR_ARPEGGIO_7 = [24, 28, 31, 35]
7 MINOR_ARPEGGIO_7 = [24, 27, 31, 34]
8 MAX_VAL = 127
9
10 def constructScales(self,scale):
11     #Only iterate within the max midi note range
12     num_iterations = (MAX_VAL - scale[-1]) // 12
13     single_scale = []
14
15     #get degrees in all octaves for C root
16     for i in range(num_iterations):
17         for note in scale:
18             single_scale.append(note + i*12)
19
20     #get scale degrees for all roots (any note in the octave)
21     all_scales = [[note + i for note in single_scale] for i in range(12)]
22     return single_scale, all_scales
23

```

Figure 6.8: Code from TestDataset.py to generate simple test sequences from major/minor scales/ arpeggios.

```

1 def memory_limit(percent):
2     #Limit max memory usage to half
3     soft, hard = resource.getrlimit(resource.RLIMIT_AS)
4     # Convert KiB to bytes, and divide in two to half
5     resource.setrlimit(resource.RLIMIT_AS, (int(get_memory() * 1024 * percent),
6     ↪ hard))
7
8 def get_memory():
9     with open('/proc/meminfo', 'r') as mem:
10         free_memory = 0
11         for i in mem:
12             sline = i.split()
13             if str(sline[0]) in ('MemFree:', 'Buffers:', 'Cached:'):
14                 free_memory += int(sline[1])
15     return free_memory # KiB

```

Figure 6.9: Code from TestDataset.py to put a limit on memory allocation from within a python process

an argument. We also implement two functions `memory_limit()` & `get_memory()` to safely test this by limiting the total memory that can be allocated by the process, preventing the system from becoming non-responsive.

The naive implementation of the TestDataset uses Python lists to hold encoded MIDI sequences and extract fixed-length rolling windows to construct a TensorFlow dataset

instance that existed entirely in memory. However, this quickly causes runtime and memory to become a bottleneck. For example, the system takes 194.18 seconds just to create 3 Tensorflow datasets (train/validation/test) with only 8/2/2 number of files respectively. To optimize this, I use Numpy arrays instead of Python lists and set our integer datatype to `uint16` (given that our event sequence only spans 0 - 388), greatly speeding up all processing while minimizing memory usage (see code 6.10). However, even after greatly optimizing the in-memory dataset, we find that the upper bound on our in-memory dataset is ~ 160 files. While this severely limits our dataset, training with an in-memory dataset was the most straightforward training method and still managed to yield a viable model, discussed in a later chapter.

```

1 import time as time
2
3 def rolling_window(self,sequence, seq_len, stride=1):
4     ls = []
5     #append windows of length seq_len to outer list
6     for i in range(0,len(sequence) - seq_len + 1,stride):
7         ls.append(sequence[i: i + seq_len])
8     return ls
9 ...
10 ...
11 def rolling_window_np(self,seq,seq_len, stride=1):
12     # Ensure the sequence is a numpy array
13     if type(seq) == list:
14         seq = np.array(seq, dtype=np.uint16)
15
16     # Compute the shape of the resulting 2D array after applying the rolling
17     ↪ window
18     shape = seq.shape[:-1] + (seq.shape[-1] - seq_len + 1, seq_len)
19
20     # Compute the strides to be used for creating the rolling window view
21     strides = seq.strides + (seq.strides[-1],)
22
23     # Create the rolling window view using as_strided
24     return np.lib.stride_tricks.as_strided(seq, shape=shape, strides=strides)
25 '''
26 >>> File has 18647 encoded midi events - taking rolling windows of size 1000
27 >>> Time taken for rolling window with encoded midi data:0.061524391174316406
28 >>> Time taken for rolling window np with encoded midi data:0.00029015541076660156
29 '''

```

Figure 6.10: Examples of code from `TestDataset.py` optimized with Numpy for improved processing

6.4.3 The `RandomCropDataset` class

In their paper on the long-context music transformer, Huang et. al. state that they use random crops of event sequences in their training process, leading me to experiment with a similar dataset implementation. For this class, I wrote a Python

generator that retrieves a batch of sequence examples to train the model, where each example is retrieved from a randomly selected file. Tensorflow allows us to create a `tf.data.Dataset` instance from a Python generator via the `from_generator` method, which accepts a callable generator as well as a tensor output specification.

```

1 def _batch_generator(self,length, num_tokens_to_predict=None):
2     if num_tokens_to_predict is None:
3         num_tokens_to_predict = length
4     while True:
5         #get batch randomly selects n files and calls extract_random_crop on them
6         data = self.get_batch(length+num_tokens_to_predict)
7
8         x = data[:, 0:length]
9         y = data[:, length:length+num_tokens_to_predict]
10
11         #attach start-of-sequences and end-of-sequences tokens to y examples
12         y = np.array([[token_sos] + list(seq) + [token_eos] for seq in y])
13         yield x,y
14
15 def extract_random_crop(self, file_data,length):
16     with open(file_data, 'rb') as f:
17         data = pickle.load(f)
18
19     start_index = random.randint(0, len(data))
20     end_index = start_index + length
21
22     #If the end index is greater than the length of the file and the pad length is
23     ↪ less than half the window size, pad the sequence with 0s
24     #Otherwise, recursively find a new random crop
25     if end_index > len(data):
26         pad_length = end_index - len(data)
27         if pad_length <= length // 2:
28             sequence = np.pad(data[start_index:], (0, pad_length), 'constant',
29                 ↪ constant_values=0)
30         else:
31             return self.extract_random_crop(file_data)
32     else:
33         sequence = data[start_index:end_index]
34
35     return sequence

```

Figure 6.11: Code from `RandomDataset.py` which shows how random crops are returned from a python generator

While this dataset implementation was straightforward, I remained doubtful as to how efficient it was. From a probabilistic perspective, our generator would select an even distribution of sequences across the entire dataset, especially if training was allowed to run for a long time. However, with shorter training times, it was completely possible that certain sequences could be revisited multiple times and be overexposed to the model.

6.4.4 The SequenceDataset class

To directly address my concerns with the RandomCropDataset class, the final dataset class I implemented was the SequenceDataset class. The SequenceDataset class inherits from both my own BaseDataset class, as well as the `tf.keras.utils.Sequence` class, a Keras helper class that can be used instead of a traditional Tensorflow dataset. The SequenceDataset generates examples in a structured manner:

1. We retrieve all paths for encoded MIDI files associated with a specific split in class the variable `self.data`
2. We convert them to a list of `FileData` objects. These objects store the path to the encoded MIDI file, as well as the next starting index a sequence should be retrieved from (initialized at 0).
3. Every time a batch is generated, n `FileData` objects are selected, a sequence is extracted from the recorded starting index, and the starting index is incremented by 1. If we detect that our extracted sequence has hit the end of the full encoded sequence, that `FileData` object is removed from `self.data` and instead added to a separate list variable `self.complete_files`.
4. As more and more batches are generated, all of our `FileData` objects slowly migrate `self.complete_files`.
5. When `self.data` is found to be empty we assign `self.data` to `self.complete_files` and `self.complete_files` to a new empty list, resetting the dataset so that training can resume.

This ensures that every example sequence is visited at least once before looping over the entire dataset. To implement the interface required by `tf.keras.utils.Sequence`, we also add the `__len__()` and `__getitem__()` methods which calculate the total number of batches in the data and return a batch respectively. Like the RandomCropDataset, an added benefit of using a generator-style implementation is that Tensorflow supports multi-processing. While this ensures we visit each example at most once over each pass of the dataset, it also unintentionally forces examples from each file to be extracted sequentially. This may have unintended effects on training, especially toward the end of a single pass when a batch may contain several sequential examples from just a handful of files.

For the full dataset implementations, please refer to [chapter 9](#).

6.5 Streamlining training experiments

Given the scope of the project, it was crucial to be able to run multiple experiments with different model architectures and slightly altered hyperparameters, while also storing data from each experimental run. To facilitate this, I created a `Params` class that would take a snapshot of all the parameters involved in the experiment, as well

```

1 class FileData():
2     def __init__(self, file_path):
3         self.path = file_path
4         self.current_note_index = 0
5     ...
6     ...
7 def extract_sequence_v2(self, file_data, length):
8     #=grabs a sequence of size 'length' from file, starting at the start index 0.
9     ↪ Then, shifts current_note_index by 1
10    with open(file_data.path, 'rb') as f:
11        data = pickle.load(f)
12
13    #pick up from the last recorded start_index
14    start_index = file_data.current_note_index
15
16    #if the start index + length is less than the file length, we can grab a
17    ↪ sequence of length
18    if start_index + length < len(data):
19        #extract a sequence of length len from the file
20        data = data[start_index:start_index + length]
21
22        #update the start index for the next sequence
23        file_data.current_note_index += 1
24    else:
25        #if there is not enough data left in the file (only possible with stride >
26        ↪ 1) then start from padding the remaining sequence with zeros
27        data = data[start_index:]
28        while len(data) < length:
29            data = np.append(data, self.params.pad_token)
30        self.move_to_complete_list(file_data)
31    return data
32 ...
33 ...
34 #Reset all indices
35 def reset(self):
36     self.data += self.complete_files
37     self.complete_files = []
38     file.current_note_index = 0 for file in self.data
39     random.shuffle(self.data)

```

Figure 6.12: Code examples from SequenceDataset.py showing the FileData object, sequence extraction, and how the dataset is reset

as several helper functions such as `setup_experiment()` in `train_utils.py` that would assist in executing a single experiment and saving all data about it.

All trained models are stored in the `models` directory, where each model has the following:

- Saved model checkpoints
- Tensorboard logs to inspect the model during and after training

```

1 class Params:
2     def __init__(self, param_dict):
3         #set dict values class attributes
4         for key, value in param_dict.items():
5             setattr(self, key, value)
6
7     def print_params(self):
8         #Print all the parameters
9         all_attrs = vars(self)
10        for key, value in all_attrs.items():
11            print(key, ":", value)
12
13    def get_params(self):
14        #Return all the parameters
15        return vars(self)
16 ...
17 ...
18 def save_params(p,base_path,logger,args):
19     #Save parameters that will be used to load
20     ↪ the model
21     try:
22         logger.info("Saving Params...")
23         with open(base_path+'params.json', 'w')
24             ↪ as file:
25             param_dict = p.get_params()
26             param_dict['name'] = args.name
27             param_dict['training_date'] =
28                 ↪ datetime.now()
29             json.dump(param_dict, file)
30         logger.info("Params Saved!")
31     except Exception as e:
32         logger.error(e)

```

```

1 {
2     "num_heads": 8,
3     "key_dim": 64,
4     "value_dim": 64,
5     "model_dim": 384,
6     "batch_size": 8,
7     "l_r": 0.001,
8     "feed_forward_dim": 1042,
9     "dropout_rate": 0.2,
10    "encoder_vocab_size": 388,
11    "num_encoder_layers": 1,
12    "decoder_vocab_size": 388,
13    "num_decoder_layers": 1,
14    "epochs": 200,
15    "beta_1": 0.9,
16    "beta_2": 0.98,
17    "epsilon": 1e-08,
18    "encoder_seq_len": 512,
19    "decoder_seq_len": 514,
20    "max_seq_len": 512,
21    "pad_token": 0,
22    "token_sos": 1,
23    "token_eos": 2,
24    "debug": true,
25    "steps_per_epoch": 500,
26    "save_freq": 10,
27    "seed": 236,
28    "name":
29    ↪ "baseline_80_files_np",
30    "training_date": "28/11/2023
31    ↪ 03:54:13"

```

Figure 6.13: (Left) The Params class and the save_params() method
(Right) Example of params.json from a training run

- history.json file with the training history returned by model.fit()
- Parameters stored in params.json
- An output log/nohup.out log

This makes it much simpler to run inference for a specific model, as we can easily find and load the parameters of a model, and its trained weights, and generate MIDI outputs accordingly.

6.6 Model Details

To understand the Transformer architecture as deeply as possible, I first implemented a Transformer model in Keras by subclassing the Keras Layer and Model class, build-

ing layers such as the `PositionalEncodingLayer` and the `MultiHeadAttention` layer myself connecting all pieces of the architecture manually. As a baseline to compare against, I also implemented a second version of the Transformer using pre-built layers from Keras and Keras_NLP and the TensorFlow functional API to connect all the different layers. All the code for both transformer implementations is included in [chapter 9](#). under [9.2.2](#) and [9.2.3](#).

Across either implementation, we use hyper-parameters largely informed by the music transformer paper (Huang et al. 2018). We train on sequences of the length of 512 for the encoder and 514 for the decoder (inclusive of start and end tokens). This is smaller than what is used by Huang et.al. and is largely due to memory constraints in our training setup. Similarly, we use a batch size of 8, as larger batch sizes quickly cause the GPU to run out of memory when training. We experiment with an embedding dimension of 256/384, 10%/20% dropout after each sublayer, and use a dimension of 1024 in the transformer feedforward network. We also experiment with an attention size of 512 and 4/8 heads, such that the dimension of the queries, keys, and values are equal to att_size/num_heads . While training, we make use of early stopping on our validation set with a patience of 3 epochs. When training with the RandomCrop-Dataset or SequenceDataset which may have infinitely looping data, we set steps per epoch to 1000. Similarly, our validation set is not fixed and the loss computation is averaged over 1000 steps on the validation data in between each epoch.

For our optimizer, we make use of an Adam optimizer with default values (0.9 for `Beta-1`, 0.98 for `Beta-2`, and 10^{-8} for `Epsilon`). Instead of providing the optimizer with a fixed learning rate of 0.1 as done by Huang et. al., we use a learning rate scheduler based on the formula proposed in "Attention is All you need" (Vaswani et al. 2023).

$$lrate = d_{model}^{-0.5} * \min(step_num^{-0.5}, step_num \cdot warmup_steps) \quad (1.6)$$

This increases the learning rate linearly for the first N warmup training steps and decreases it thereafter proportionally to the inverse square root of the step number. I use the default value of 4000 for the `warmup_steps`.

For our loss function, I make use of the `sparse-categorical-cross-entropy` loss object provided by TensorFlow, which is meant for multi-label classification problems. At each step, our model essentially predicts what 'class' the next MIDI event will be and acts on the final linear layer outputs of the model, which uses softmax activation to retrieve a probability distribution over our event vocabulary. Training over the Decoder was done in a teacher-forced¹ manner.

¹**Teacher forcing** is a training strategy used in the context of machine learning, particularly in the training of Recurrent Neural Networks (RNNs) and similar models that generate sequences, such as Long Short-Term Memory (LSTM) networks. In a standard sequence generation task without teacher forcing, the model generates an output at each step of the sequence and then feeds its

output from the previous step as the input for the next step. However, this can lead to problems during training if the model starts generating incorrect outputs; these errors can accumulate and lead the model further astray. Teacher forcing addresses this issue by using a different approach during training. Instead of feeding the model's output from the previous step into the next step, it uses the actual or "true" output from the training dataset at each step. This ensures that the model is always trained with the correct sequence up to that point, helping it learn more effectively.

Chapter 7

Results and Analysis

7.1 Metrics

While there are a large number of metrics to evaluate NLP tasks, such as *Perplexity*, *Bilingual Evaluation Understudy score (BLEU)* (Papineni et al. 2002) and *Recall-Oriented Understudy for Gisting Evaluation (ROUGE)* (Lin 2004) etc., metrics for evaluating generated models — and generated music specifically — are very limited. In the visual domain, where generative models have been experimented with on a larger scale with output that can be immediately perceived, there are already several arguments that raise doubts regarding the efficacy of standard metrics for evaluating model quality (Theis, Oord, and Bethge 2016). These explanations extend naturally to the case of musically generative models. In particular, they point out that ultimately, “models need to be evaluated directly with respect to the application(s) they were intended for”. In the case of the model we hope to create, this involves human listening. One such subjective study was carried out in the creation of a Jazz transformer, where listeners were asked to blindly rate the model’s output against real data in a five-point Likert scale on four aspects (Wu and Yang 2020) :

- **Overall Quality** (does it sound good?)
- **Impression** (Can you remember a certain part of the melody?)
- **Structure** (does it sound musically coherent with recurring ideas etc.)
- **Richness** (is it diverse and interesting?)

Unfortunately, any research involving human subjects must be reviewed by an Institutional Review Board (IRB), which could not be completed for this project.

Given the above limitations, I evaluate trained models to on the standard evaluation metrics criteria supplemented by additional metrics inspired by Wu et.al for a set of examples selected from our test set to generate full-length responses.

- **Negative Logarithmic Loss (NLL)** — This is a standard loss metric used

to measure the performance of probabilistic models, including those in natural language processing, machine learning, and other fields. In Tensorflow, we can compute this by using the `sparse_categorical_crossentropy` loss metric over our entire test set.

- **Pitch Class Histogram Entropy** — Proposed by Wu et.al., this metric gains insight into the model’s ability to use different pitches. I collect the notes that appear over sequential output and construct a 12-dimensional pitch class histogram, h , normalized by the total note count such that $\sum_i h_i = 1$. If the piece has clear tonality and learns well, specific pitch classes should dominate the pitch histogram (e.g., the tonic and the dominant), resulting in a low entropy that is close to the input. If the tonality is unstable, the usage of pitch classes may be scattered, giving rise to a high entropy. On the other end of the spectrum, if our entropy is suspiciously low, the model may have learned even less and may be simply repeating the same note over and over. The entropy value is calculated as:

$$Entropy(h) = - \sum_{i=0}^{11} h_i * \log_2(h_i) \tag{1.7}$$

- **Time-Shift and Velocity Histogram Entropy** — I propose a similar metric for TIME-SHIFT events, grouping them by the time-shift event vocabulary of size 100. I assume that if the entropy of the output is high, the model has an unstable perception of time. If the entropy of the model is low, it likely uses specific time-shift events much more often than others, which might be indicative of the model learning some kind of specified rhythm. The same is applied to velocity events, which are a measure of coherent musical expressiveness.
- **Subjective Listening and Visualized Piano rolls/Chroma** — Finally, I evaluate the model output with examples generated from our test set, generating piano rolls and pitch chroma vectors to be visually examined.

7.2 Custom Transformer

While the implementation of the Custom Transformer was syntactically sound, early attempts to train our Custom Transformer revealed that it was not learning effectively when compared to our model built with pre-built Keras layers, either due to implementation errors or uncaught bugs. The figure below includes sample output from one of the earliest trained models, trained using the `RandomCropDataset`.

Examining [Figure 7.1](#) above, we can inspect the generated piano rolls and see that the model struggles to learn the MIDI encoding structure and instead plays notes

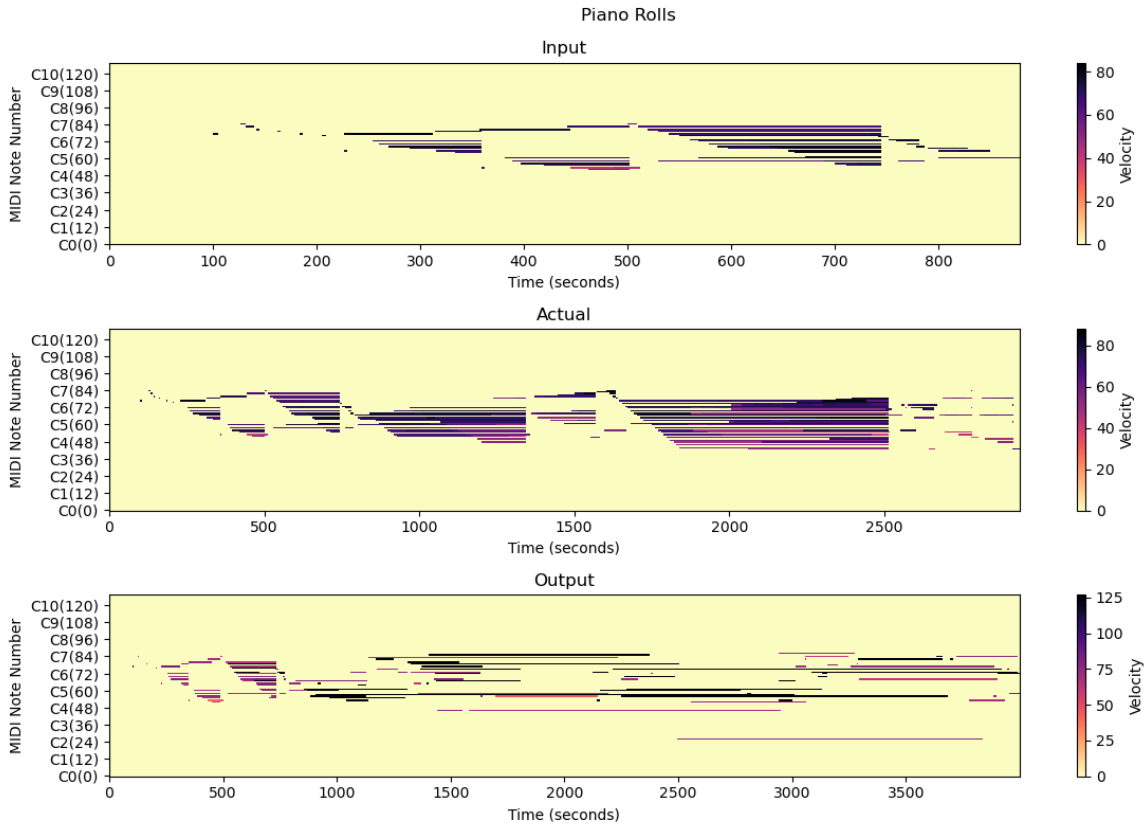


Figure 7.1: Examples Piano Rolls from training the Custom Transformer: (Top) The input MIDI sequence. (Middle) The actual MIDI sequence in its entirety (Bottom) Input sequence (till ~ 700 ms) followed by the output generated by the Transformer to the front

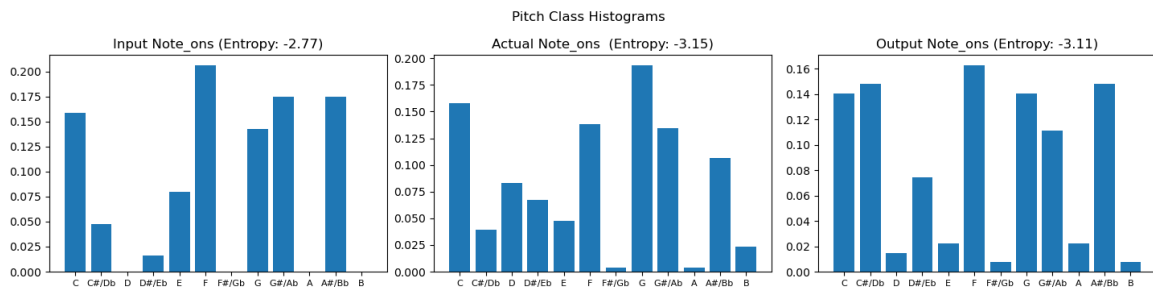


Figure 7.2: Pitch Class histogram of Note-on events from piano rolls, annotated with Entropy values and the input sequence removed from the model output

seemingly randomly, with many notes held for extremely long periods and the sense of time lost completely. Interestingly, [Figure 7.2](#) suggests that the model does learn relevant pitch-based relationships given that the segment is in the tonality of C. While the actual MIDI files feature a prominent tonic-dominant relationship with C and G popping up most frequently, the Custom Transformer has seemingly learned the tonic-

subdominant relationship, with C and F popping up very frequently. Furthermore, it avoids the Tritone¹ relationship which is between C and F#.

Inspecting [Figure 7.3](#), we can see a much higher entropy value and a much larger scattering across the time-shift even histogram bins, suggesting that the model struggled to learn rhythmic relationships in the encoded sequence.

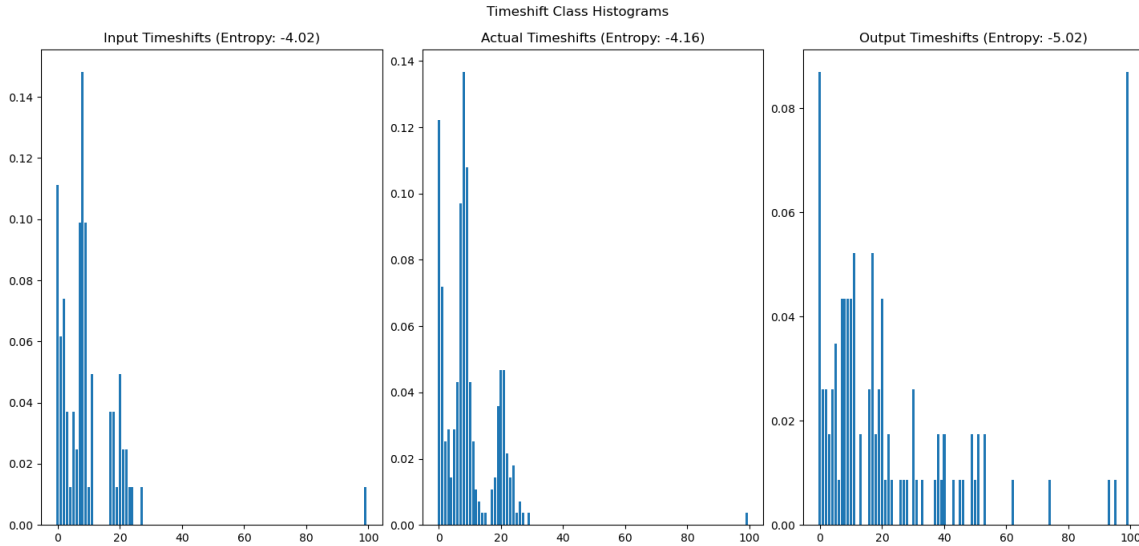


Figure 7.3: Time-shift event histogram from piano rolls, annotated with Entropy values and the input sequence removed from the model output

The failure of the Custom Transformer was further corroborated by its inability to overfit to much smaller test sequences. The table below shows the results of our Custom transformer against the Keras Transformer trained with the same hyperparameters for 50 epochs, on the Major scale sequences generated by our `TestDataset`. It is important to note that in this experiment, the test set was randomly sampled from the rolling windows extracted from the generated scalaric sequences. As demonstrated in [Figure 6.6](#) from the previous chapter, nearby sequences have a large degree of overlap, implying that our Custom Transformer performs poorly even when it has seen a large amount of the test set. This hints at an uncaught implementation error.

This revelation was especially disappointing as I had spent a large amount of time developing the Custom Transformer to eventually implement the relative attention mechanism described in Huang et.al, which is not yet supported in Keras. As a result, all subsequent evaluation is conducted on the Transformer implementation using Keras_NLP layers. (See [subsection 9.2.3](#))

¹A **Tritone** is a musical interval that spans three whole tones or six semitones. It has a unique, dissonant sound, which historically led to its nickname “diabolus in musica” or “the devil in music in the context of early Western Classical Music

Model	Custom Transformer	Keras Transformer
Loss Values	4.344	0.045
Accuracy Values	0.081	0.992

Table 7.1: Loss/Accuracy values for both models when trained on test integer sequences

7.3 Keras Transformer

7.3.1 Experimenting with Hyper-parameters

	layers	d_model	h	dropout	size(10^6)	ValidationNLL	TestNLL
Base	2	384	8	0.1	32.4	2.319	2.263
a)	1	⋮	⋮	⋮	16.6	2.575	2.504
b)	⋮	256	⋮	⋮	15.3	2.290	2.233
c)	⋮	⋮	4	⋮	18.2	2.381	2.306
d)	⋮	⋮	⋮	0.2	32.4	2.791	2.677

Table 7.2: Results from hyper-parameter tuning experiments with the `RandomCropDataset` for the base model and variants a) - d). All model parameters are the same as the Base model, except where explicitly indicated

During training, the `RandomCropDataset` was far more successful than our `SequenceDataset` implementation, which frequently caused training to abruptly halt due to errors when shifting `FileData` objects. Therefore I use this dataset implementation to experiment with different hyperparameter values to optimize the model. I successfully managed to train multiple model variants with slightly altered hyper-parameter values, err-ing on the side of a smaller model. I find that a smaller hidden size than the one used in Huang et.al. works slightly better in our implementation while halving the size of the model for much faster training and inference. Fewer encoder/decoder stacks, in-creasing dropout, and using fewer attention heads seem to worsen our log loss values. It is also important to note that all models were terminated via early stopping, and only ran for 38k - 42k steps of training.

However, the actual generated output from these models is extremely poor, in some ways even poorer than those generated by the Custom Transformer, a testament to the sentiment provided by Oore et.al. that traditional loss metrics may be far from good indicators of model performance.

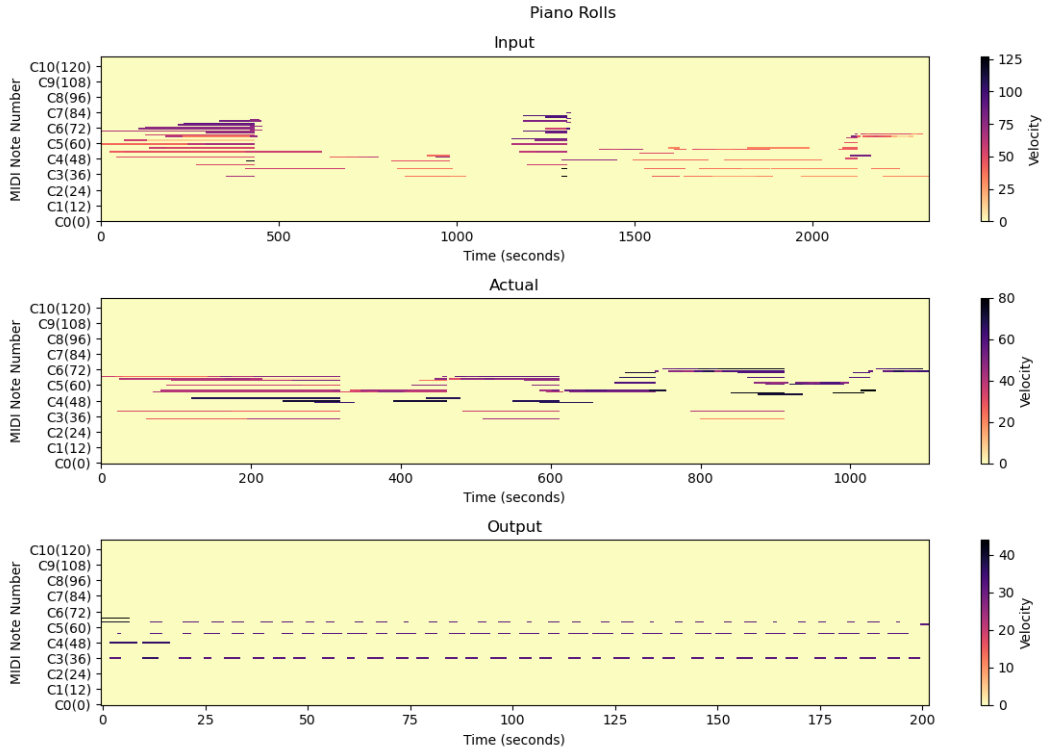


Figure 7.4: Piano Rolls generated from model b) trained with the `RandomCropDataset`. The model most just plays a C major chord very quickly

7.4 Final Attempts

Given these findings I then attempted to train a separate model with the optimal hyperparameters from our `RandomCropDataset` experiments, but while using the `SequenceDataset` implementation. Unfortunately, the training time for a full epoch to cover the entire training set was far too long, coming in at about ~ 317 hours, even with training distributed across 2 GPUs. Even when I tried reducing the training time by eliminating duplicate works from the training set based on earlier analysis, our training time per epoch was still ~ 196 hours per epoch, which was far too long to complete multiple rounds of training. Rudimentary analysis of the GPU performance of the `SequenceDataset` with the NVIDIA tool `nvprof`, revealed that the GPU was suffering from low shared memory. Shared memory on a GPU acts as a user-managed cache and allows threads within the same block to share data efficiently. It's much faster than accessing main GPU memory, but it's also much smaller. In our case, a low amount of shared memory implies that the demands of shared memory from the running processes exceed its availability, which is common in complex machine learning models, where many threads are working with a large amount of data that they need to share quickly.

Instead, I manage to train a much smaller model with an in-memory dataset using only 80/20 files, in a training/validation split which yields interesting results. While

the Log-Loss metric on the test set is significantly higher for this model, coming in at **6.574**, I find that the model is able to generate output that resembles some form of expressive generation.

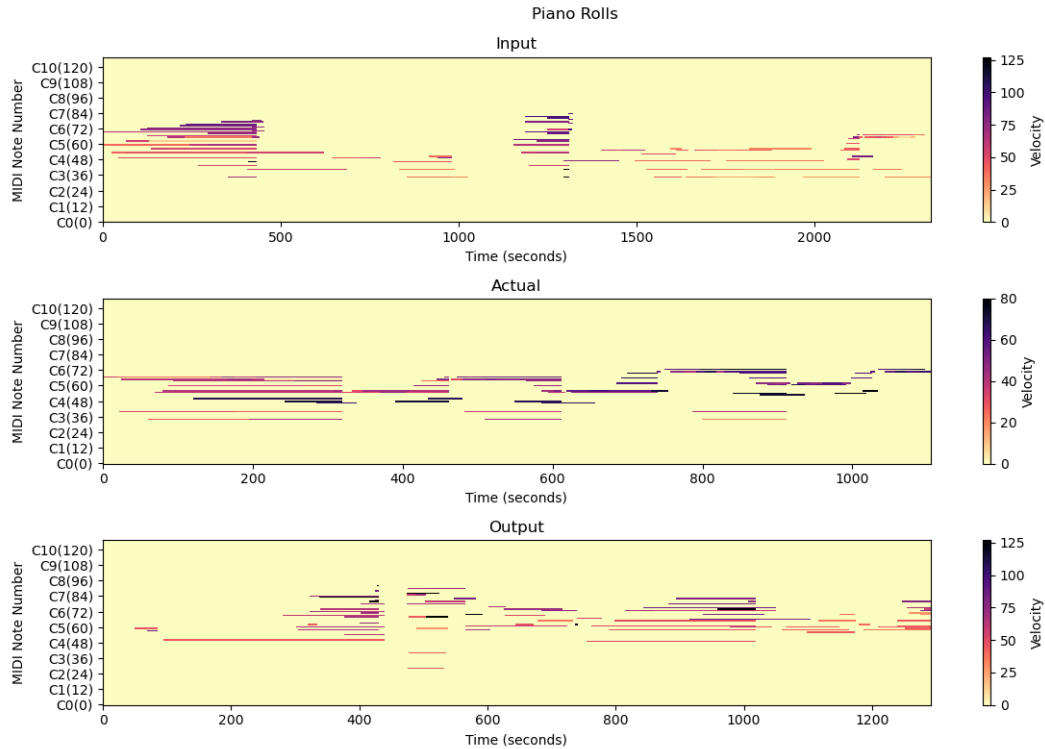


Figure 7.5: Examples Piano Rolls from training model trained with an in-memory dataset : (Top) The actual input MIDI sequence (Middle) The actual MIDI sequence in its entirety (Bottom) The output generated by the Transformer. The input sequence is the same is [Figure 7.4](#)

The response generated from this last model actually captures elements valid elements of expressive performance. It is able to capture rhythmic phrases and interesting pitch relationships. Unfortunately, by inspecting the individual pitch class histogram bins, I find that it is still quite poor in terms of being able to adequately respond to the pitch context provided in the input. This is unsurprising, given that this model is trained on a mere fraction of the dataset and with no data augmentation. It is likely overexposed to tonalities present from the randomly sampled training set.

The Velocity and Time-Shift Histograms (Figures [7.7,7.8](#)) however are much more promising, with lower entropy values and a much tighter distribution compared to previous results, indicating that the model has learned some semblance of rhythm, as well as expressive performance.

MIDI samples from this last model are available [here](#), and the code used to construct piano rolls/histograms can be found in [chapter 9](#) under [section 9.3](#)

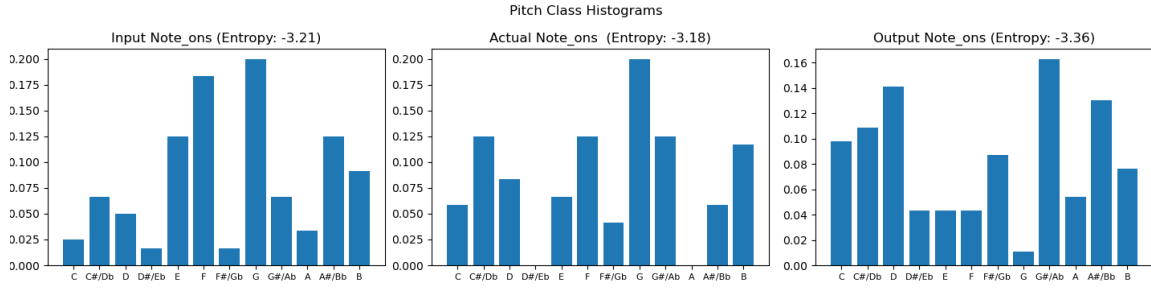


Figure 7.6: Pitch Class histogram of Note-on events from the model trained with an in-memory dataset

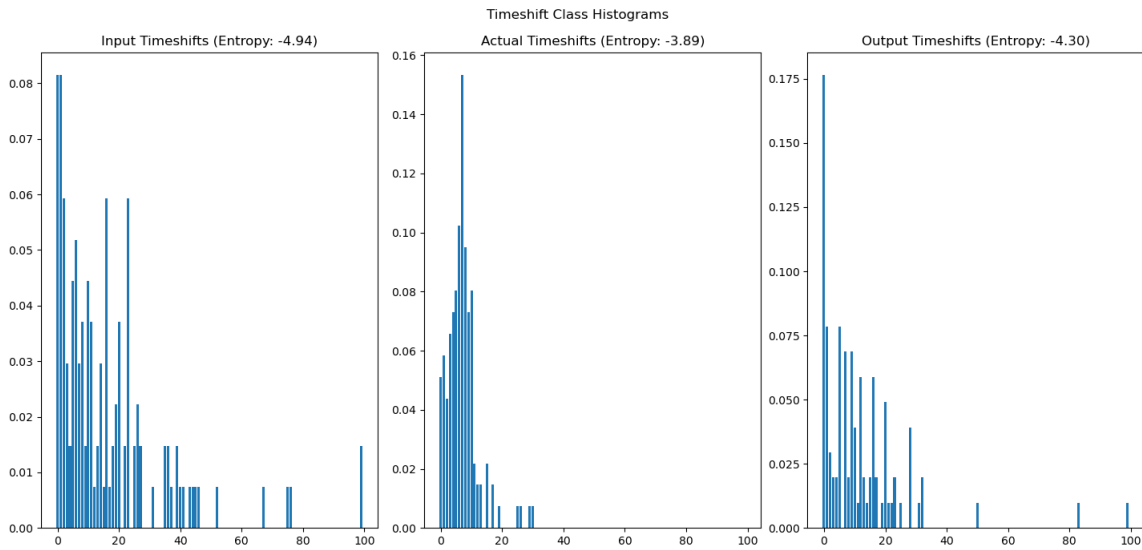


Figure 7.7: Time-Shift Event histogram from model trained with an in-memory dataset

7.5 Conclusions

I successfully implemented a Transformer model that can create expressive performances, operating on a mere fraction of the dataset. However, the Transformer is still limited and is nowhere close in performance to examples generated by Huang et.al and the Google team. I find that the original in-memory dataset implementation works best given our resources, and postulate that given more time, and resources and by including data augmentation, the model is likely to perform much better.

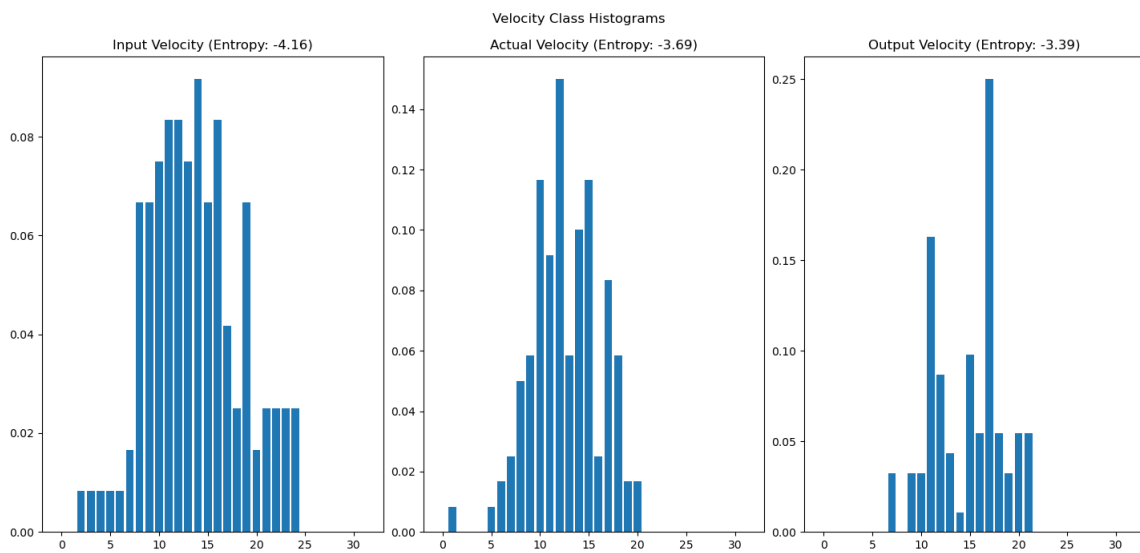


Figure 7.8: Velocity Event histogram from model trained with an in-memory dataset

Chapter 8

Reflections

To round out the work described in this paper, I discuss implementation changes that might make our model more successful, as well as future directions for the project as it currently stands.

8.1 Avenues for different implementations

- **A Different Dataset** — While the MAESTRO dataset is especially fantastic for the general task of expressive music generation, it is less incredible when we consider the overarching goal of this project: to create a model that can improvise *alongside a human performer*. To this end, the MAESTRO dataset is severely limited because it is composed only of classical music, the genre of choice for improvisation until about the late 17th century. Instead, a dataset of expressive, MIDI-aligned jazz performances would be far better suited to this task, especially MIDI with multi-instrument tracks that would allow a model to learn how to accompany another player, rather than just generate an improvised response. Some examples of Jazz datasets include the WJazzD dataset (Wu and Yang 2020) and the PiJAMA dataset (Edwards, Dixon, and Benetos 2023), but both of these include solo performances only, and do not have the same level of granularity for expressive performance generation at the same level as the MAESTRO dataset
- **A Different MIDI representation** — While the encoding proposed by Oore et.al. holds promising results, it also comes with clear downsides, resulting in very long event sequences that can explode a Transformer’s memory requirements. Furthermore, a note’s attributes may be very far apart in a sequence (eg. note-on and note-off) and such long-distance relationships are difficult to model, especially with fixed attention sizes. To deal with this, the Google Brain team later proposed a new representation, `NoteTuple`, that groups a note’s attributes into one event, resulting in a model that requires fewer attributes and has faster generation. This involves swapping out the model’s output softmax

with a Neural Auto-regressive Distribution Estimation Model, or NADE, to predict a note’s attributes (Hawthorne 2018)

- **A Different ML framework** — While the choice to use Tensorflow seemed clear-cut when I started the senior project, I would now choose PyTorch if I were allowed to re-implement the project. This is for many reasons, but mostly around Tensorflow’s ability to communicate with a machine’s CUDA drivers and use GPUs for training. Many days and hours were spent uninstalling and reinstalling CUDA drivers and CUDNN packages, digging into Tensorflow and Nvidia documentation, and changing low-level parameters on the Linux machine I was using, only for the Tensorflow library to ignore the GPUs connected to the computer. By contrast, I managed to get GPU support up and running in PyTorch almost immediately. I also realized after the fact that being able to export the model to a Javascript environment was not unique to Tensorflow; PyTorch models can also be exported in an ONNX format and run in a Javascript environment using `ONNX.js`.
- **A Different Model architecture** — While Transformer models have been the rage in machine learning, the baseline Transformer model that we use is rather outdated. More complex variants of the Transformer-like the **Switch Transformer** (Fedus, Zoph, and Shazeer 2021), **Transformer-XL** (Dai et al. 2019), and the **Perceiver** (Jaegle et al. 2021) have made massive strides in improving the training time and inference time of Transformer based architectures. In particular, the Perceiver architecture has been used to great effect by researchers from the Magenta project to improve on the Long Context Music Transformer created by Huang et.al. Conversely, it may be the case that Transformer-based architectures are simply too large to use in a real-time setting when not powered by production-grade GPUs and servers. LSTM models like those developed by Oore et.al. or even Hidden Markov Models might be better suited to the real-time performance domain.

8.2 Future Work

- **Sharded Dataset** — One dataset implementation that I had yet to implement was the sharded dataset, with generated sequences stored across multiple different files. TensorFlow has built-in support for sharding datasets (especially in large production environments). This would be a very interesting opportunity to pursue especially given the surprisingly robust output from our model trained with the in-memory dataset on far fewer files overall.
- **Pretraining** — Our model can stand to gain from pretraining practices that are already extremely common within the NLP domain. Rather than train new word embeddings, most NLP models today make use of embeddings that have already been trained by very large encoder models such as RoBERTa which are great at learning embedded input representations of sequential data. Similarly, we can use a BERT-based model to pretrain useful MIDI embeddings which can

then be transferred to our Transformer model. MIDI2vec (Alam et al. 2022), is one such study that is working to create reliable embedded representations of symbolic MIDI data.

- **The Performance Interface** — One very obvious part of the project that went unimplemented due to lack of time was connecting the Max/MSP performance interface to the model. This would require further testing and development of the performance interface, especially if we hope to implement Polyphonic Pitch Detection. One very interesting tool that may be of interest is Basic-Pitch, developed by the Spotify Audio Intelligence Lab (Bittner et al. 2022). This is a lightweight model that does exceedingly well at instrument-agnostic multi-pitch estimation, converting raw audio fragments directly to the MIDI domain.
- **Decoder-only architectures and Beam Sampling** — Another avenue of experimentation is exploring Decoder-only architectures, similar to ChatGPT. Making use of pre-trained weights from an Encoder model like RoBERTa, a Decoder-only model could be small enough to enable real-time generation due to constrained model size. Furthermore, this would enable the model to be able to generate output in the absence of input, similar to Voyager. We can also further improve the model output without adding to its size by using Beam search¹ to sample from the model.

¹**Beam search** is a heuristic search algorithm that is used to find the most likely sequence of decisions in such generative models. Unlike the simple greedy search algorithm implemented in our project which only keeps the single best option at each step, beam search keeps track of multiple alternatives, which is specified by the "beam width."

Chapter 9

Appendix

9.1 Detailed Directory Structure

```
(Root)
├── PyModel
│   ├── Analysis
│   ├── Data
│   │   ├── raw
│   │   └── processed
│   ├── Dataset
│   ├── KerasTransformer
│   ├── CustomTransformer
│   ├── midi-neural-preprocessor
│   ├── models
│   │   └── :
│   ├── samples
│   │   └── :
│   ├── utils
│   ├── train.py (Custom training loop)
│   ├── train-v2.py (Training loop with Model.fit())
│   └── :
├── TSImprovisor
│   ├── main.ts
│   ├── improviser.ts
│   ├── utils.ts
│   ├── interface.maxpat
│   └── :
```

9.2 PyModel

9.2.1 Dataset Implementations

BaseDataset.py

```
1 import os
2 import json
3 import random
4 import numpy as np
5 import pickle
6 from CustomTransformer.params import midi_test_params_v2, Params
7 import json
8 from time import time
9
10 class BaseDataset():
11     def __init__(self, p:Params,
12                 path="./data/processed",
13                 data_format="pickle",
14                 min_duration=None,
15                 min_event_length=None,
16                 logger = None):
17
18         self.logger = logger
19         self.data_format = data_format
20         self.maestroJSON = self.get_maestroJSON()
21         self.fileDict =
22         ↪ self.get_encoded_files(path,min_duration,min_event_length)
23         self.params = p
24
25         random.seed(self.params.seed)
26
27     def get_maestroJSON(self, path="./data/raw/maestro-v3.0.0.json"):
28         with open(path) as f:
29             data = json.load(f)
30         return data
31
32     def get_encoded_files(self,path,min_duration,min_event_length):
33         if not os.path.exists(path):
34             os.mkdir(path)
35
36         #strip the year from the midi filename path, add .pickle to the end
37         ↪ and add to base path
38         lambda_func = lambda x: os.path.join(path, x.split('/')[-1] + '.' +
39         ↪ 'pickle')
40
41         #From the json files, get the indexed midi filenames
42         midi_filenames_from_json = { int(key) : lambda_func(value) for key,
43         ↪ value in self.maestroJSON['midi_filename'].items() }
```

```

43     dup_dict = {}
44     for key, value in midi_filenames_from_json.items():
45         dup_dict.setdefault(value, set()).add(key)
46     result = [key for key, values in dup_dict.items() if len(values) >
47               ↪ 1]
48     assert len(result) == 0, f"Duplicate midi filenames found: {result}"
49
50     #Remove midi files that are too short in length by duration
51     if min_duration is not None:
52         i = 0
53         while i < len(midi_filenames_from_json):
54             duration = self.maestroJSON['duration'][f'{i}']
55             if duration < min_duration:
56                 self.log_or_print(f"Removed file
57                 ↪ {midi_filenames_from_json[i]}, Duration of is
58                 ↪ {duration}, which is less than {min_duration}")
59                 del midi_filenames_from_json[i]
60                 i+=1
61
62             self.log_or_print(f"Number of midi files remaining after
63             ↪ removing files less than {min_duration} seconds:
64             ↪ {len(midi_filenames_from_json)}")
65
66     #Remove midi files that are too short in length by number of events
67     if min_event_length is not None:
68         i = 0
69         while i < len(midi_filenames_from_json):
70             with open(midi_filenames_from_json[i], 'rb') as f:
71                 if self.data_format == "pickle":
72                     data = pickle.load(f)
73                 else:
74                     data = np.load(f, allow_pickle=True)
75             if len(data) < min_event_length:
76                 self.log_or_print(f"Removed file
77                 ↪ {midi_filenames_from_json[i]}, Length of is
78                 ↪ {len(data)}, which is less than {min_event_length}")
79                 del midi_filenames_from_json[i]
80                 i+=1
81
82             self.log_or_print(f"Number of midi files remaining after
83             ↪ removing files less than {min_event_length} events:
84             ↪ {len(midi_filenames_from_json)}")
85
86     return midi_filenames_from_json
87
88     def log_or_print(self, log_str, isWarning=False):
89         if self.logger:
90             if isWarning:
91                 self.logger.warning(log_str)
92             else:
93                 self.logger.info(log_str)
94         else:
95             print(log_str)

```

```

86
87     def format_dataset(self,x, y):
88         return (
89             {
90                 "encoder_inputs": x,
91                 "decoder_inputs": y[:, :-1],
92             },
93             y[:, 1:],
94         )
95
96 if __name__ == "__main__":
97
98     #Testing creating base dataset - testing speed of np files vs lists
99     p = Params(midi_test_params_v2)
100
101     start = time()
102     dataset = BaseDataset(p, data_format='numpy',
103         ↪ min_event_length=p.encoder_seq_len*2)
104     end = time()
105     print(f"Time taken to create dataset with numpy format: {end-start}")
106
107     start = time()
108     dataset = BaseDataset(p, data_format='pickle',
109         ↪ min_event_length=p.encoder_seq_len*2)
110     end = time()
111     print(f"Time taken to create dataset with pickle format: {end-start}")

```

TestDataset.py

```

1 import tensorflow as tf
2 import random
3 import pickle
4 import numpy as np
5 from .BaseDataset import BaseDataset
6 from CustomTransformer.params import Params, midi_test_params_v2
7 import resource
8 import sys
9 import time
10 import os
11
12 MAJOR_SCALE = [24, 26, 28, 29, 31, 33, 35]
13 MINOR_SCALE = [24, 26, 27, 29, 31, 32, 34]
14 MAJOR_ARPEGGIO_7 = [24, 28, 31, 35]
15 MINOR_ARPEGGIO_7 = [24, 27, 31, 34]
16 MAX_VAL = 127
17
18
19 def memory_limit(percent):
20     #Limit max memory usage to half
21     soft, hard = resource.getrlimit(resource.RLIMIT_AS)
22     # Convert KiB to bytes, and divide in two to half

```

```

23 resource.setrlimit(resource.RLIMIT_AS, (int(get_memory() * 1024 *
↪ percent), hard))
24
25 def get_memory():
26     with open('/proc/meminfo', 'r') as mem:
27         free_memory = 0
28         for i in mem:
29             sline = i.split()
30             if str(sline[0]) in ('MemFree:', 'Buffers:', 'Cached:'):
31                 free_memory += int(sline[1])
32     return free_memory # KiB
33
34 class TestDataset(BaseDataset):
35     def __init__(self,
36                 p:Params,
37                 data_format="pickle",
38                 min_duration=None,
39                 min_event_length=None,
40                 logger=None,
41                 num_files_by_split=None):
42
43         super().__init__(
44             p=p,
45             data_format=data_format,
46             min_duration=min_duration,
47             min_event_length=min_event_length,
48             logger=logger)
49
50         self.data = {
51             'train': [],
52             'validation': [],
53             'test': []
54         }
55         self.retrieve_files_by_maestro_split()
56
57         if num_files_by_split != None:
58             self.data['train'] =
↪ self.data['train'][0:num_files_by_split['train']]
59             self.data['validation'] =
↪ self.data['validation'][0:num_files_by_split['validation']]
60             self.data['test'] =
↪ self.data['test'][0:num_files_by_split['test']]
61
62     def constructScales(self, scale):
63         num_iterations = (MAX_VAL - scale[-1]) // 12
64         single_scale = []
65
66         for i in range(num_iterations):
67             for note in scale:
68                 single_scale.append(note + i*12)
69

```

```

70     all_scales = [[note + i for note in single_scale] for i in
71     ↪ range(12)]
72     return single_scale, all_scales
73
74 def mockTfDataset_from_scale(self, scale, seq_len, stride=1):
75     single_scale, all_scales = self.constructScales(scale)
76     all_sequences = self.rolling_window(all_scales, seq_len*2, stride)
77     random.shuffle(all_sequences)
78     x, y = [], []
79     for seq in all_sequences:
80         x.append(seq[:seq_len])
81         y.append([1] + seq[seq_len:] + [2])
82
83     split_1 = int(0.8*len(x))
84     split_2 = int(0.9*len(x))
85
86     train_x , val_x, test_x = x[0:split_1], x[split_1:split_2],
87     ↪ x[split_2:]
88     train_y , val_y, test_y = y[0:split_1], y[split_1:split_2],
89     ↪ y[split_2:]
90
91     train = tf.data.Dataset.from_tensor_slices((train_x, train_y))
92     val = tf.data.Dataset.from_tensor_slices((val_x, val_y))
93     test = tf.data.Dataset.from_tensor_slices((test_x, test_y))
94     return train, val, test
95
96 def rolling_window(self, sequence, seq_len, stride=1):
97     ls = []
98     for i in range(0, len(sequence) - seq_len + 1, stride):
99         ls.append(sequence[i: i + seq_len])
100     return ls
101
102 def get_all_sequences_by_split(self, split, seq_len, stride=1):
103     print("Getting all sequences for split:{}".format(split))
104     all_sequences = []
105     counter = 0
106     for path in self.data[split]:
107         with open(path, 'rb') as f:
108             event_sequence = pickle.load(f)
109             sequences = self.rolling_window(event_sequence, seq_len * 2,
110             ↪ stride)
111             all_sequences.extend(sequences)
112             counter+=1
113             print("Finished processing {} files".format(counter))
114
115     return all_sequences
116
117 def rolling_window_np(self, seq, seq_len, stride=1):
118     ## Convert each list to a NumPy array of type uint16
119     # ls = []
120     ##if sequence is a list and not a numpy array , convert to numpy
121     ↪ array

```

```

117     # if type(seq) == list:
118     #     seq = np.array(seq, dtype=np.uint16)
119
120     # for i in range(0, len(seq) - seq_len + 1, stride):
121     #     ls.append(seq[i: i + seq_len])
122     # return np.array(ls)
123
124     # Ensure the sequence is a numpy array
125     if type(seq) == list:
126         seq = np.array(seq, dtype=np.uint16)
127
128     # Compute the shape of the resulting 2D array after applying the
129     ↪ rolling window
130     shape = seq.shape[:-1] + (seq.shape[-1] - seq_len + 1, seq_len)
131
132     # Compute the strides to be used for creating the rolling window
133     ↪ view
134     strides = seq.strides + (seq.strides[-1],)
135
136     # Create the rolling window view using as_strided
137     return np.lib.stride_tricks.as_strided(seq, shape=shape,
138     ↪ strides=strides)
139
140 def get_all_sequences_by_split_np(self, split, seq_len, stride=1):
141     print("Getting all sequences for split:{}".format(split))
142     all_sequences = []
143     counter = 0
144     for path in self.data[split]:
145         with open(path, 'rb') as f:
146             event_sequence = np.load(f, allow_pickle=True)
147             sequences = self.rolling_window_np(event_sequence, seq_len * 2,
148             ↪ stride)
149             all_sequences.append(sequences)
150             counter+=1
151             print("Finished processing {} files".format(counter))
152
153     # Concatenate all sequences and shuffle
154     all_sequences_np = np.concatenate(all_sequences, axis=0)
155     return all_sequences_np
156
157 def mockTfDataset_from_encoded_midi_np(self, stride=1):
158     datasets = {}
159
160     for key in self.data.keys():
161         sequences = self.get_all_sequences_by_split_np(key,
162         ↪ self.params.encoder_seq_len, stride)
163         x = sequences[:, :self.params.encoder_seq_len]
164         y = np.pad(sequences[:, self.params.encoder_seq_len:],
165         ((0, 0), (1, 1)),
166         mode='constant',
167         constant_values=(1, 2))
168

```



```

164         datasets[key] = tf.data.Dataset.from_tensor_slices((x, y))
165
166     return datasets['train'], datasets['validation'], datasets['test']
167
168 def mockTfDataset_from_encoded_midi_pickle(self, stride=1):
169     datasets = {}
170
171     for key in self.data.keys():
172         sequences = self.get_all_sequences_by_split(key,
173             ↪ self.params.encoder_seq_len, stride)
174         x = [seq[:self.params.encoder_seq_len] for seq in sequences]
175         y = [[1] + seq[self.params.encoder_seq_len:] + [2] for seq in
176             ↪ sequences]
177         datasets[key] = tf.data.Dataset.from_tensor_slices((x, y))
178
179     return datasets['train'], datasets['validation'], datasets['test']
180
181 def mockTfDataset_from_encoded_midi(self, stride=1):
182     if self.data_format == 'pickle':
183         return self.mockTfDataset_from_encoded_midi_pickle(stride)
184     elif self.data_format == 'numpy':
185         return self.mockTfDataset_from_encoded_midi_np(stride)
186     else:
187         raise Exception("Invalid data format")
188
189 def mockTfDataset_from_encoded_midi_path(self, path, stride=1):
190     with open(path, 'rb') as f:
191         event_sequence = np.load(f, allow_pickle=True)
192         sequences = self.rolling_window_np(event_sequence,
193             ↪ self.params.encoder_seq_len * 2, stride)
194         x = [seq[:self.params.encoder_seq_len] for seq in sequences]
195         y = [[1] + seq[self.params.encoder_seq_len:] + [2] for seq in
196             ↪ sequences]
197         dataset = tf.data.Dataset.from_tensor_slices((x, y))
198
199     return dataset
200
201 def retrieve_files_by_maestro_split(self):
202     for i in self.fileDict.keys():
203         if self.maestroJSON['split'][f'{i}'] == 'train':
204             self.data['train'].append(self.fileDict[i])
205         elif self.maestroJSON['split'][f'{i}'] == 'validation':
206             self.data['validation'].append(self.fileDict[i])
207         elif self.maestroJSON['split'][f'{i}'] == 'test':
208             self.data['test'].append(self.fileDict[i])
209         else:
210             raise Exception(f"Invalid mode found:
211                 ↪ {self.maestroJSON['split'][f'{i}']}")
212
213 def __repr__(self) -> str:
214     return "<TestDataset has {} files for training, {} files for
215         ↪ validation, {} files for testing>".format(

```

```

210     len(self.data['train']),
211     len(self.data['validation']),
212     len(self.data['test']))
213
214
215 if __name__ == '__main__':
216     p = Params(midi_test_params_v2)
217     os.environ["CUDA_VISIBLE_DEVICES"]="1"
218
219     # =====Test creating scales for easier problem =====
220     # dataset = TestDataset(p, data_format='numpy',
221     ↪ min_event_length=p.encoder_seq_len*2)
222     # train,val,test, = dataset.mockTfDataset_from_scale(MAJOR_SCALE, 12, 2)
223
224     # ===== Test rolling window efficiency (numpy vs list)
225     ↪ =====
226     #Compare time between rolling window and rolling window npwith encoded
227     ↪ midi data
228
229     dataset = TestDataset(p, data_format='numpy',
230     ↪ min_event_length=p.encoder_seq_len*2)
231     with open('...', 'rb') as f:
232         event_sequence = pickle.load(f)
233
234     start = time.time()
235     all_sequences = dataset.rolling_window(event_sequence, 1000,1)
236     end = time.time()
237     print("Time taken for rolling window with encoded midi
238     ↪ data:{}".format(end - start))
239
240     start = time.time()
241     all_sequences_np = dataset.rolling_window_np(event_sequence, 1000,1)
242     end = time.time()
243     print("Time taken for rolling window np with encoded midi
244     ↪ data:{}".format(end - start))
245
246     #check equality across all sequences
247     for i in range(len(all_sequences)):
248         # print("Sequence {} is equal to
249         ↪ {}".format(all_sequences[i],all_sequences_np[i]))
250         assert np.array_equal(all_sequences[i],all_sequences_np[i])
251
252     # =====Test tf dataset=====
253     start = time.time()
254     dataset = TestDataset(p, data_format='pickle',
255     ↪ min_event_length=p.encoder_seq_len*2,
256     ↪ num_files_by_split={'train':2,'validation':1,'test':1})
257
258     #limit num files considered
259     print(dataset)
260     train,val,test = dataset.mockTfDataset_from_encoded_midi()
261     end = time.time()

```

```

253 print("Time taken for creating tf dataset:{}".format(end - start))
254
255 start = time.time()
256 dataset = TestDataset(p, data_format='numpy',
↳ min_event_length=p.encoder_seq_len*2,
↳ num_files_by_split={'train':2,'validation':1,'test':1})
257
258 print(dataset)
259 train_np,val_np,test_np = dataset.mockTfDataset_from_encoded_midi()
260 end = time.time()
261 print("Time taken for creating tf dataset:{}".format(end - start))
262
263 # train = train.shuffle(len(train))
264 train = train.batch(16, drop_remainder=True)
265 train = train.map(dataset.format_dataset)
266
267 # val = val.shuffle(len(val))
268 val = val.batch(16, drop_remainder=True)
269 val = val.map(dataset.format_dataset)
270
271 # train_np = train_np.shuffle(len(train_np))
272 train_np = train_np.batch(16, drop_remainder=True)
273 train_np = train_np.map(dataset.format_dataset)
274
275 # val_np = val_np.shuffle(len(val_np))
276 val_np = val_np.batch(16, drop_remainder=True)
277 val_np = val_np.map(dataset.format_dataset)
278
279 for x , x_np in zip(train.take(len(train)),
↳ train_np.take(len(train_np))):
280     assert
↳ np.array_equal(x[0]['encoder_inputs'],x_np[0]['encoder_inputs'])
281     assert
↳ np.array_equal(x[0]['decoder_inputs'],x_np[0]['decoder_inputs'])
282     assert np.array_equal(x[1],x_np[1])
283
284
285 #####Test if we can construct fully in memory dataset with np using
↳ stride of 2#####
286 # memory_limit(0.8)
287 # strategy = tf.distribute.MirroredStrategy()
288
289 # with strategy.scope():
290 #     try:
291 #         dataset = TestDataset(p, data_format='numpy',
↳ min_event_length=p.encoder_seq_len*2,
↳ num_files_by_split={'train':120,'validation':15,'test':15})
292
293 #         train_np,val_np,test_np =
↳ dataset.mockTfDataset_from_encoded_midi_np(2)
294 #     except MemoryError:
295 #         print("Memory error raised")

```

```

296 #         print("Memory usage:{}".format(get_memory()))
297 #         sys.exit(1)

```

RandomDataset.py

```

1 from .BaseDataset import BaseDataset
2 from CustomTransformer.params import Params, midi_test_params_v2
3 import tensorflow as tf
4 import numpy as np
5 import random
6 import pickle
7
8 class RandomCropDataset(BaseDataset):
9     def __init__(self, p: Params, mode, min_duration=None,
10         ↪ min_event_length=None, num_files_to_use=None, logger=None):
11         super().__init__(p=p, min_duration=min_duration,
12         ↪ min_event_length=min_event_length, logger=logger)
13         self.data = []
14         self.mode = mode
15         self.num_files_to_use = num_files_to_use
16         self.retrieve_files_by_maestro_split()
17         self.stats = {}
18
19         random.shuffle(self.data)
20         if num_files_to_use is not None:
21             self.data = self.data[:num_files_to_use]
22
23     def retrieve_files_by_maestro_split(self):
24         if self.mode not in ["train", "validation", "test"]:
25             raise Exception(f"Invalid mode passed: {self.mode}")
26
27         for i in self.fileDict.keys():
28             if self.maestroJSON['split'][f'{i}'] == self.mode:
29                 self.data.append(self.fileDict[i])
30
31     def make_gen_callable(self, _gen):
32         def gen():
33             for x,y in _gen:
34                 yield x,y
35         return gen
36
37     def get_batch(self,length, collect_stats=False):
38         batch_data = []
39         for _ in range(self.params.batch_size):
40             file_data = random.choice(self.data)
41             sequence = self.extract_random_crop(file_data,length,
42             ↪ collect_stats=collect_stats)
43             batch_data.append(sequence)
44         return np.array(batch_data,int)
45
46     def batch_generator(self,length,collect_stats=False):
47         return self.make_gen_callable(

```

```

45     self._batch_generator(length, collect_stats=collect_stats)
46 )
47
48 def _batch_generator(self, length, num_tokens_to_predict=None,
49 ↪ collect_stats=False):
49     if num_tokens_to_predict is None:
50         num_tokens_to_predict = length
51     while True:
52         data = self.get_batch(length+num_tokens_to_predict,
53 ↪ collect_stats=collect_stats)
54
55         x = data[:, 0:length]
56         y = data[:, length:length+num_tokens_to_predict]
57
58         #attach start and end tokens to each y sequence
59         y = np.array([[self.params.token_sos] + list(seq) +
60 ↪ [self.params.token_eos] for seq in y])
61         yield self.format_dataset(x,y)
62
63 def extract_random_crop(self, file_data, length, collect_stats=False):
64     with open(file_data, 'rb') as f:
65         data = pickle.load(f)
66
67     start_index = random.randint(0, len(data))
68     end_index = start_index + length
69
70     # Handle the padding scenario
71     #If the end index is greater than the length of the file and the
72     ↪ pad length is less than half the window size, pad the sequence
73     ↪ with 0s
74     #Otherwise, recursively find a new random crop
75     if end_index > len(data):
76         pad_length = end_index - len(data)
77         if pad_length <= length / 2:
78             sequence = np.pad(data[start_index:], (0, pad_length),
79 ↪ 'constant', constant_values=0)
80         else:
81             return self.extract_random_crop(file_data, length,
82 ↪ collect_stats) # Recursively find a new random crop
83     else:
84         sequence = data[start_index:end_index]
85
86     if collect_stats:
87         if file_data not in self.stats:
88             self.stats[file_data] = [len(data), (start_index, end_index)]
89         else:
90             self.stats[file_data] =
91 ↪ self.stats[file_data].append((start_index, end_index))
92
93     return sequence

```

```

89     def __repr__(self):
90         return "<RandomCropDataset_{} has {} files>".format(self.mode,
91             ↪ len(self.data))
92
93 if __name__ == "__main__":
94     p = Params(midi_test_params_v2)
95     p.batch_size = 2
96
97     train_dataset = RandomCropDataset(p, 'train',
98     ↪ min_event_length=p.encoder_seq_len*2, logger=None)
99
100 train = train_dataset._batch_generator(10)
101 for _ in range(5):
102     example = next(train)
103     print(example)
104     print(example[0]["encoder_inputs"].shape)
105     print(example[0]["decoder_inputs"].shape)
106     print(example[1].shape)
107
108 train = train_dataset.batch_generator(10)
109
110 #convert to tf.data.Dataset
111 train = tf.data.Dataset.from_generator(
112     train,
113     output_signature=(
114         {
115             'encoder_inputs': tf.TensorSpec(shape=(p.batch_size, 10),
116             ↪ dtype=tf.int32),
117             'decoder_inputs': tf.TensorSpec(shape=(p.batch_size, 11),
118             ↪ dtype=tf.int32)
119         },
120         tf.TensorSpec(shape=(p.batch_size, 11), dtype=tf.int32))
121     )
122 )
123 for ex in train.take(5):
124     print(ex)
125
126 #=====
127 #Test to check stats
128 #=====
129 train = train_dataset._batch_generator(1000, collect_stats=True)
130 for _ in range(1000):
131     example = next(train)
132
133 for k,v in train_dataset.stats.items():
134     if v == None:
135         print(k)

```

SequenceDataset.py

```
1 from .BaseDataset import BaseDataset
2 import tensorflow as tf
3 from CustomTransformer.params import Params, midi_test_params_v2
4 import numpy as np
5 import random
6 import pickle
7 import time
8
9 class FileData():
10     def __init__(self, file_path):
11         self.path = file_path
12         self.current_note_index = 0
13
14 class SequenceDataset(BaseDataset, tf.keras.utils.Sequence):
15     def __init__(self, p:Params, mode, min_duration=None,
16 min_event_length=None, num_files_to_use=None, logger=None):
17         super().__init__(
18             p=p,
19             min_duration=min_duration,
20             min_event_length=min_event_length,
21             logger=logger)
22
23         self.data = []
24         self.complete_files = []
25         self.num_files_to_use = num_files_to_use
26         self.mode = mode
27         self.retrieve_files_by_maestro_split()
28         self.convert_all_to_FileData()
29
30         random.shuffle(self.data)
31
32         if num_files_to_use is not None:
33             self.data = self.data[0:num_files_to_use]
34
35     def __len__(self):
36         return self.calculate_num_batches(self.params.encoder_seq_len, 1)
37
38     def retrieve_files_by_maestro_split(self):
39         if self.mode not in ["train", "validation", "test"]:
40             raise Exception(f"Invalid mode passed: {self.mode}")
41
42         for i in self.fileDict.keys():
43             if self.maestroJSON['split'][f'{i}'] == self.mode:
44                 self.data.append(self.fileDict[i])
45
46     def __getitem__(self, idx):
47         x,y = self.seq2seq_batch(self.params.batch_size,
48             ↪ self.params.encoder_seq_len)
49         return (
50             {
51                 "encoder_inputs": x,
```

```

51         "decoder_inputs": y[:, :-1],
52     },
53     y[:, 1:],
54 )
55
56 def convert_all_to_FileData(self):
57     self.log_or_print("Converting all files to FileData objects")
58     for i in range(len(self.data)):
59         self.data[i] = FileData(self.data[i])
60
61 def get_batch(self, batch_size, length):
62     data = []
63
64     #select k files from the list of files
65     #allows us to grab a batch from the same file multiple times
66     for _ in range(batch_size):
67         if len(self.data)==0:
68             self.reset()
69         file = (random.choice(self.data))
70         data.append(self.extract_sequence_v2(file, length))
71
72     return np.array(data,int)
73
74 #extract_sequence, grabs a sequence of size 'length' from file, starting
75 ↪ at the start index 0. Then, shifts the sequence down by 1
76 def extract_sequence_v2(self, file_data, length):
77     #Grab a random sample of length len from a while
78     with open(file_data.path, 'rb') as f:
79         data = pickle.load(f)
80
81     #pick up from the last recorded start_index
82     start_index = file_data.current_note_index
83
84     #if the start index + length is less than the file length, we can
85     ↪ grab a sequence of length
86     if start_index + length < len(data):
87         #extract a sequence of length len from the file
88         data = data[start_index:start_index + length]
89
90         #update the start index for the next sequence
91         file_data.current_note_index += 1
92     else:
93         #We either 1) perfectly hit the last event in the sequence with
94         ↪ a full sequence 2) we hit the end of the file early and need
95         ↪ to pad with zeros
96         #if there is not enough data left in the file (only possible
97         ↪ with stride > 1) then start from start index and take the
98         ↪ remaining events in the file, padding the remaining sequence
99         ↪ with zeros
100        data = data[start_index:]
101        while len(data) < length:
102            data = np.append(data, self.params.pad_token)

```



```

96         self.move_to_complete_list(file_data)
97     return data
98
99     #Move a fileData object from the model_data[mode] list to the
100    ↪ complete_files list
101    def move_to_complete_list(self,file_data):
102        if file_data not in self.complete_files:
103            self.complete_files.append(file_data)
104        else:
105            self.log_or_print("File{file_data.path} already in
106            ↪ complete_files list", isWarning=True)
107
108        try:
109            self.data.remove(file_data)
110        except:
111            self.log_or_print(f"File{file_data.path} was already removed
112            ↪ from not data list", isWarning=True)
113
114    #if training decoder, num_tokens_to_predict is 0, we teacher force the
115    ↪ entire sequence
116    #if training regular transformer, length and num_tokens_to_predict are
117    ↪ the same
118
119    #first half is passed to encoder as context
120    #second half passed to decoder for teacher forcing
121    def seq2seq_batch(self, batch_size, length, num_tokens_to_predict=None):
122        if num_tokens_to_predict is None:
123            num_tokens_to_predict = length
124        data = self.get_batch(batch_size, length+num_tokens_to_predict)
125        x = data[:, 0:length]
126        y = data[:, length:length + num_tokens_to_predict]
127
128        #attach start and end tokens to each y sequence
129        y = np.array([[self.params.token_sos] + list(seq) +
130        ↪ [self.params.token_eos] for seq in y])
131        return x, y
132
133    def calculate_num_batches(self, seq_len, stride):
134        num_examples = 0
135        for file in self.data:
136            with open(file.path, 'rb') as f:
137                data = pickle.load(f)
138                #seq_len multiplied by two for the encoder and decoder seq
139                ↪ respectively
140                num_examples += (len(data) - seq_len*2) // stride
141        num_batches = int(num_examples / self.params.batch_size)
142        remaining_examples = num_examples % self.params.batch_size
143
144        # print(f"Number of batches: {num_batches}, Number of remaining
145        ↪ examples: {remaining_examples}")
146        assert num_batches*self.params.batch_size + remaining_examples ==
147        ↪ num_examples

```

```

139
140     return num_batches
141
142     #Reset all indices
143     def reset(self):
144         self.log_or_print("Reset Called, resetting fileData objects")
145         self.data += self.complete_files
146         self.complete_files = []
147         for file in self.data:
148             file.current_note_index = 0
149         random.shuffle(self.data)
150
151     def __repr__(self):
152         return "<SequenceDataset_{} has {} files>".format(self.mode,
153             ↪ len(self.data))
154
155 if __name__ == "__main__":
156     p = Params(midi_test_params_v2)
157
158     data = SequenceDataset(p, 'train', min_event_length=p.encoder_seq_len*2,
159         ↪ num_files_to_use=5)
160     print(data)
161
162     #####
163     #Testing out methods required for keras.utils.sequence
164     #__getitem__ and __len__ methods
165     #__calculate_num_batches method calculates the number of batches in the
166     ↪ dataset, which is needed for __len__
167     #__getitem__ returns a batch of data, given an idx
168
169     #Should show multiple sequences and an increasing note_index, before
170     ↪ they are shifted to the complete list
171     #####
172     num_batches = data.calculate_num_batches(p.encoder_seq_len, 1)
173     print(num_batches)
174
175     time.sleep(2)
176     for i in range(p.epochs):
177         print("Epoch:",i)
178         for i in range(num_batches):
179             batch = data.__getitem__(i)
180             print(batch.shape)
181             if i % 100 == 0:
182                 print(f'Batch: {i}')
183                 # print(f'encoder_input shape:
184                 ↪ {batch[0]["encoder_inputs"].shape}')
185                 # print(f'decoder_input shape:
186                 ↪ {batch[0]["decoder_inputs"].shape}')
187                 # print(f'decoder_output shape: {batch[1].shape}')
188             for file in data.data:
189                 print(file.path,file.current_note_index)
190             print(f"Complete files:{len(data.complete_files)}=====")

```

```

85         for file in data.complete_files:
86             print(file.path)
87     print(len(data.complete_files))
88     data.on_epoch_end()

```

9.2.2 Custom Transformer Implementation

PositionalEncodingLayer.py

```

1 from tensorflow.keras.layers import Layer, Embedding
2 import numpy as np
3 import tensorflow as tf
4 from .utils import check_shape
5 from matplotlib import pyplot as plt
6
7 class PositionEmbeddingFixedWeights(Layer):
8     def __init__(self, seq_len, vocab_size, output_dim, **kwargs):
9         super(PositionEmbeddingFixedWeights, self).__init__(**kwargs)
10        self.model_dim = output_dim
11        self.vocab_size = vocab_size
12        self.seq_len = seq_len
13
14        # Initialize the positional encoding matrices
15        self.position_embedding_matrix =
16            ↪ self.get_positional_encoding(seq_len, output_dim)
17
18        # Input embedding layer
19        self.input_embedding_layer = Embedding(
20            input_dim=vocab_size,
21            output_dim=output_dim,
22        )
23
24    def get_config(self):
25        config = super(PositionEmbeddingFixedWeights, self).get_config()
26        config.update({
27            'seq_len': self.seq_len,
28            'vocab_size': self.vocab_size,
29            'output_dim': self.output_dim,
30            'input_embedding_layer': tf.keras.saving.serialize_keras_object(
31                ↪ self.input_embedding_layer),
32            'position_embedding_layer':
33                ↪ tf.keras.saving.serialize_keras_object(
34                ↪ self.position_embedding_layer),
35        })
36        return config
37
38    #Based on "attention is all you need"
39    #Given a input sequence of size L and model dimension, initialize the
40    ↪ positional encoding matrix
41
42    #  $P(k, 2i) = \sin(k/n^{(2i/d)})$  ,  $P(k, 2i+1) = \cos(k/n^{(2i/d)})$ 

```

```

38     # k = position, where k < L/2 (since we alternate between sin and cos)
39     # d = dimension of the output embedding space
40     # n user defined scalar, 10000 in the paper
41     # i = dimension index, where i < d/2
42     @classmethod
43     def get_positional_encoding(self, size, output_dim, n=10000):
44         P = np.zeros((size, output_dim))
45         for pos in range(size):
46             for i in range(output_dim // 2):
47                 denominator = np.power(n, (2 * i) / output_dim)
48                 P[pos, 2 * i] = np.sin(pos / denominator)
49                 P[pos, 2 * i + 1] = np.cos(pos / denominator)
50
51         # Ensure the data type matches TensorFlow expectations
52         P = P.astype(np.float32)
53         return P
54
55     def call(self, inputs):
56         # Generate a sequence of position indices
57         position_indices = tf.range(start=0, limit=tf.shape(inputs)[1],
58             ↪ delta=1)
59         # Embed the input tokens and the positions
60         embedded_input = self.input_embedding_layer(inputs)
61         #Scale the input embedding by sqrt(model_dim)
62         embedded_input *= tf.math.sqrt(tf.cast(self.model_dim, tf.float32))
63         # Sum the token embeddings and position embeddings
64         return embedded_input + self.position_embedding_matrix

```

MultiHeadAttentionLayer.py

```

1  import tensorflow as tf
2  from tensorflow.keras.layers import Layer, Dense
3  from .utils import check_shape
4  from .params import Params
5  import json
6
7  @tf.keras.saving.register_keras_serializable()
8  class MultiHeadAttentionLayer(Layer):
9      def __init__(self, p:Params, isRelative=False, **kwargs):
10         super(MultiHeadAttentionLayer, self).__init__(**kwargs)
11
12         if isRelative:
13             raise NotImplementedError("Relative attention not yet
14             ↪ implemented")
15
16         self.num_heads = p.num_heads
17         self.key_dim = p.key_dim
18         self.value_dim = p.value_dim
19         self.model_dim = p.model_dim
20
21         #Model dimensionality must be divisible by the number of heads

```

```

21     assert self.model_dim % self.num_heads == 0
22
23     #Initialize linear layers for projecting queries, keys, values, and
24     → output
25     self.W_query = Dense(self.key_dim)
26     self.W_key = Dense(self.key_dim)
27     self.W_value = Dense(self.value_dim)
28     self.W_out = Dense(self.model_dim)
29
30     def scaled_dot_product_attention(self, q, k, v, mask=None):
31         #Q, K, V all have shape [batch_size, num_heads, seq_len,
32         → dim_per_head]
33
34         #First multiply queries by keys to get similarity scores and
35         → normalize
36         attention_weights = tf.matmul(q, k, transpose_b=True) /
37         → tf.math.sqrt(tf.cast(self.key_dim, tf.float32))
38
39         #Mask if required (Eg. decoder layer), prevent attention from future
40         → outputs
41         #Essentially multiply by an extremely small negative number to
42         → remove future values from softmax calculation
43         if mask is not None: attention_weights += -1e9 * mask
44
45         #Use softmax to get attention weights in terms of probability
46         → distribution
47         attention_weights = tf.nn.softmax(attention_weights)
48
49         #Multiply by values to get context vector
50         context_vector = tf.matmul(attention_weights, v)
51
52         return context_vector
53
54     def reshape_tensor(self, tensor):
55         '''
56         Eg. for a single input query of size 5(seq len),16(query dim) > 80
57         → elements
58         Therefore if 8 heads, each head will have 8/8 = 10 elements
59         10 elements > 2 sequences of 5 elements, per batch
60         '''
61         tensor = tf.reshape(tensor, (tf.shape(tensor)[0],
62         → tf.shape(tensor)[1], self.num_heads, -1))
63         check_shape("reshaped_tensor", tensor, (p.embedding_dim, p.seq_len,
64         → p.num_heads, int(p.batch_size/p.num_heads)))
65
66         tensor = tf.transpose(tensor, perm=[0,2,1,3])
67         check_shape("transposed_tensor", tensor, (p.embedding_dim,
68         → p.num_heads, p.seq_len, int(p.batch_size/p.num_heads)))
69
70         return tensor
71
72     def concat_heads(self, tensor):

```

```

62     tensor = tf.transpose(tensor, perm=[0,2,1,3])
63     tensor = tf.reshape(tensor, (tf.shape(tensor)[0],
    → tf.shape(tensor)[1], self.key_dim))
64     return tensor
65
66     def get_config(self):
67         config = super(MultiHeadAttentionLayer, self).get_config()
68         config.update({
69             'num_heads': self.num_heads,
70             'key_dim': self.key_dim,
71             'value_dim': self.value_dim,
72             'model_dim': self.model_dim,
73             'W_query': tf.keras.saving.serialize_keras_object(self.W_query),
74             'W_key': tf.keras.saving.serialize_keras_object(self.W_key),
75             'W_value': tf.keras.saving.serialize_keras_object(self.W_value),
76             'W_out': tf.keras.saving.serialize_keras_object(self.W_out),
77         })
78         return config
79
80     def call(self, inputs, mask=None, **kwargs):
81         """
82         input: a list of tensors, representing [queries, keys, values]
83         mask: for masked multi head attention in decoder
84
85         """
86         q,k,v = inputs[0], inputs[1], inputs[2]
87         # for tensor in [q,k,v]:
88         #     check_shape("input_tensor", tensor, (p.batch_size,
    → p.seq_len, p.embedding_dim))
89
90         #First project through linear layers
91         q,k,v = self.W_query(q), self.W_key(k), self.W_value(v)
92
93         #Reshape to [batch_size, num_heads, seq_len, dim_per_head] for dot
    → product attention
94         q,k,v = self.reshape_tensor(q), self.reshape_tensor(k),
    → self.reshape_tensor(v)
95         # for tensor in [q,k,v]:
96         #     check_shape("reshaped_query",
    → tensor, (p.embedding_dim, p.num_heads,
    → p.seq_len, int(p.batch_size/p.num_heads)))
97
98         #compute scaled dot product attention for each head
99         attention = self.scaled_dot_product_attention(q, k, v, mask)
100
101         #concat attention representations across each head
102         concat_attention = self.concat_heads(attention)
103         # check_shape("attention", concat_attention, (p.batch_size,
    → p.seq_len, p.embedding_dim))
104
105         #pass through final linear layer
106         output = self.W_out(concat_attention)

```

```

107     # check_shape("output",output,(p.batch_size, p.seq_len,p.model_dim))
108
109     return output

```

Decoder.py

```

1 import tensorflow as tf
2 from tensorflow.keras.layers import Layer, Dropout
3 from .MultiHeadAttentionLayer import MultiHeadAttentionLayer
4 from .FeedForwardLayer import FeedForward
5 from .AddNormalizationLayer import AddNormalization
6 from .PositionalEncodingLayer import PositionEmbeddingFixedWeights
7 from .utils import check_shape
8 from .params import baseline_test_params, Params
9
10 @tf.keras.saving.register_keras_serializable()
11 class DecoderLayer(Layer):
12     def __init__(self, p:Params, **kwargs):
13         super(DecoderLayer,self).__init__(**kwargs)
14
15         self.seq_len = p.decoder_seq_len
16         self.model_dim = p.model_dim
17         self.dropout_rate = p.dropout_rate
18         self.feed_forward_dim = p.feed_forward_dim
19
20         #First Multiheaded attention layer - Causal self attention (masked)
21         self.masked_mha_layer = MultiHeadAttentionLayer(p,isRelative=False)
22         self.dropout1 = Dropout(self.dropout_rate)
23         self.add_norm1 = AddNormalization()
24
25         #Second Multihead attention layer - encoder-decoder attention or
26         → cross attention
27         self.mha_layer = MultiHeadAttentionLayer(p,isRelative=False)
28         self.dropout2 = Dropout(self.dropout_rate)
29         self.add_norm2 = AddNormalization()
30
31         #Feed forward layer
32         self.feed_forward = FeedForward(self.feed_forward_dim,
33         → self.model_dim)
34
35         self.dropout3 = Dropout(self.dropout_rate)
36         self.add_norm3 = AddNormalization()
37
38     def get_config(self):
39         config = super(DecoderLayer, self).get_config()
40         config.update({
41             'seq_len': self.seq_len,
42             'model_dim': self.model_dim,
43             'dropout_rate': self.dropout_rate,
44             'feed_forward_dim': self.feed_forward_dim,
45             'masked_mha_layer': self.masked_mha_layer.get_config(),
46             'mha_layer':self.mha_layer.get_config(),

```

```

45         'feed_forward': self.feed_forward.get_config(),
46     })
47     return config
48
49     def call(self, x, encoder_output, lookahead_mask, padding_mask,
50 ↪ training):
51         attention_output1 = self.masked_mha_layer([x, x, x], lookahead_mask)
52         attention_output1 = self.dropout1(attention_output1,
53 ↪ training=training)
54         addnorm_output1 = self.add_norm1(x, attention_output1)
55
56         attention_output2 =
57 ↪ self.mha_layer([addnorm_output1, encoder_output, encoder_output],
58 ↪ padding_mask)
59         attention_output2 = self.dropout2(attention_output2,
60 ↪ training=training)
61         addnorm_output2 = self.add_norm2(addnorm_output1, attention_output2)
62
63         ff_output = self.feed_forward(addnorm_output2)
64         ff_output = self.dropout3(ff_output, training=training)
65         final = self.add_norm3(addnorm_output2, ff_output)
66         return final
67
68     class Decoder(Layer):
69         def __init__(self, p: Params, **kwargs):
70             super().__init__(**kwargs)
71             self.decoder_seq_len = p.decoder_seq_len
72             self.model_dim = p.model_dim
73             self.dropout_rate = p.dropout_rate
74             self.decoder_vocab_size = p.decoder_vocab_size
75             self.num_decoder_layers = p.num_decoder_layers
76
77             #Create positional encoding layer
78             self.positional_encoding =
79 ↪ PositionEmbeddingFixedWeights(self.decoder_seq_len,
80 ↪ self.decoder_vocab_size, self.model_dim)
81
82             #N decoder stacks
83             self.decoder_layers = [
84                 DecoderLayer(p) for _ in range(self.num_decoder_layers)]
85
86         def get_config(self):
87             config = super(Decoder, self).get_config()
88             config.update({
89                 'decoder_seq_len': self.decoder_seq_len,
90                 'decoder_vocab_size': self.decoder_vocab_size,
91                 'model_dim': self.model_dim,
92                 'dropout_rate': self.dropout_rate,
93                 'num_decoder_layers': self.num_decoder_layers,
94                 'positional_encoding': self.positional_encoding.get_config(),
95                 'decoder_layers': [layer.get_config() for layer in
96 ↪ self.decoder_layers.layers]
97             })

```



```

89
90 def call(self, x, encoder_output, lookahead_mask, padding_mask,
↪ training):
91     positional_encoding_output = self.positional_encoding(x)
92
93     for layer in self.decoder_layers:
94         output = layer(
95             positional_encoding_output,
96             encoder_output=encoder_output,
97             lookahead_mask=lookahead_mask,
98             padding_mask=padding_mask,
99             training = training
100         )
101     return output

```

Encoder.py

```

1 import tensorflow as tf
2 from tensorflow.keras.layers import Layer, Dropout, Input
3 from tensorflow.keras.models import Model
4 from .MultiHeadAttentionLayer import MultiHeadAttentionLayer
5 from .FeedForwardLayer import FeedForward
6 from .PositionalEncodingLayer import PositionEmbeddingFixedWeights
7 from .AddNormalizationLayer import AddNormalization
8 from .utils import check_shape
9 from .params import baseline_test_params, Params
10
11 @tf.keras.saving.register_keras_serializable()
12 class EncoderLayer(Layer):
13     def __init__(self, p:Params, **kwargs):
14         super(EncoderLayer, self).__init__(**kwargs)
15         self.seq_len = p.encoder_seq_len
16         self.model_dim = p.model_dim
17         self.dropout_rate = p.dropout_rate
18         self.feed_forward_dim = p.feed_forward_dim
19
20         #EncoderMultihead attention layer - Global self attention (fully
↪ autoregressive)
21         self.mha_layer = MultiHeadAttentionLayer(p, isRelative=False)
22         self.add_norm1 = AddNormalization()
23         self.dropout1 = Dropout(p.dropout_rate)
24
25         #Feed forward layer
26         self.feed_forward = FeedForward(p.feed_forward_dim, p.model_dim)
27         self.add_norm2 = AddNormalization()
28         self.dropout2 = Dropout(p.dropout_rate)
29
30
31 def get_config(self):
32     config = super(EncoderLayer, self).get_config()
33     config.update({
34         'seq_len': self.seq_len,

```

```

35         'model_dim': self.model_dim,
36         'dropout_rate': self.dropout_rate,
37         'feed_forward_dim': self.feed_forward_dim,
38         'mha_layer': self.mha_layer.get_config(),
39         'feed_forward': self.feed_forward.get_config(),
40     })
41     return config
42
43     def call(self, x, padding_mask, training):
44         attention_output = self.mha_layer([x, x, x], padding_mask)
45         # check_shape("attention_output", attention_output, (x.shape[0],
46         ↪ self.seq_len, self.model_dim))
47         attention_output = self.dropout1(attention_output,
48         ↪ training=training)
49         #the input itself + the scaled attention values
50         addnorm_output = self.add_norm1(x, attention_output)
51
52         ff_output = self.feed_forward(addnorm_output)
53         ff_output = self.dropout2(ff_output, training=training)
54         #the previous addnorm output + values from the feedforward network
55         final = self.add_norm2(addnorm_output, ff_output)
56         return final
57
58     class Encoder(Layer):
59         def __init__(self, p:Params, **kwargs):
60             super(Encoder, self).__init__(**kwargs)
61             self.encoder_seq_len = p.encoder_seq_len
62             self.model_dim = p.model_dim
63             self.dropout_rate = p.dropout_rate
64             self.encoder_vocab_size = p.encoder_vocab_size
65             self.num_encoder_layers = p.num_encoder_layers
66
67             #Create the positional encoding layer
68             self.positional_encoding =
69             ↪ PositionEmbeddingFixedWeights(self.encoder_seq_len,
70             ↪ self.encoder_vocab_size, self.model_dim)
71             #N encoder layers
72             self.encoder_layers = [EncoderLayer(p) for _ in
73             ↪ range(self.num_encoder_layers)]
74
75         def get_config(self):
76             config = super(Encoder, self).get_config()
77             config.update({
78                 'encoder_seq_len': self.encoder_seq_len,
79                 'encoder_vocab_size': self.encoder_vocab_size,
80                 'model_dim': self.model_dim,
81                 'dropout_rate': self.dropout_rate,
82                 'num_encoder_layers': self.num_encoder_layers,
83                 # 'positional_encoding' is also a layer and needs to handle its
84                 ↪ config
85                 'positional_encoding': self.positional_encoding.get_config(),

```

```

80         'encoder_layers': [layer.get_config() for layer in
81                             ↪ self.encoder_layers.layers]
82     })
83
84     def call(self, x, padding_mask, training):
85         positional_encoding_output = self.positional_encoding(x)
86
87         for layer in self.encoder_layers:
88             output = layer(
89                 positional_encoding_output,
90                 padding_mask=padding_mask,
91                 training = training
92             )
93         return output

```

FeedForwardLayer.py

```

1  from tensorflow.keras.layers import Layer, Dense, ReLU
2  import tensorflow as tf
3  from .utils import check_shape
4  import json
5
6  @tf.keras.saving.register_keras_serializable()
7  class FeedForward(Layer):
8      def __init__(self, d_in, d_out, **kwargs):
9          super(FeedForward, self).__init__(**kwargs)
10         self.d_in = d_in
11         self.d_out = d_out
12
13         # First fully connected layer takes in input dimensions
14         self.dense1 = Dense(self.d_in)
15         # Second fully connected layer, based on model output dimensions
16         self.dense2 = Dense(self.d_out)
17         self.activation = ReLU()
18
19     def get_config(self):
20         config = super(FeedForward, self).get_config()
21         config.update({
22             'd_in': self.d_in,
23             'd_out': self.d_out,
24             'dense1': tf.keras.saving.serialize_keras_object(self.dense1),
25             'dense2': tf.keras.saving.serialize_keras_object(self.dense2),
26         })
27         return config
28
29     def call(self, x):
30         #Send inputs through the first fully connected layer
31         output = self.dense1(x)
32         #Apply ReLU activation
33         output = self.activation(output)
34         #Send inputs through the second fully connected layer
35         output = self.dense2(output)

```

```
36     return output
37
```

AddNormalizationLayer.py

```
1 from tensorflow.keras.layers import Layer, LayerNormalization, Add
2 import tensorflow as tf
3 from .utils import check_shape
4 import json
5
6 @tf.keras.saving.register_keras_serializable()
7 class AddNormalization(Layer):
8     def __init__(self, **kwargs):
9         super(AddNormalization, self).__init__(**kwargs)
10        # Layer normalization layer
11        self.layer_norm = LayerNormalization()
12        # Add layer - the add layers ensure that keras masks are properly
13        ↪ propagated
14        self.add = Add()
15
16    def call(self, x, sublayer_x):
17        #Skip connection
18        output = self.add([x + sublayer_x])
19        return self.layer_norm(output)
20
21    def get_config(self):
22        base_config = super(AddNormalization, self).get_config()
23        config = {
24            "layer_norm":
25            ↪ tf.keras.saving.serialize_keras_object(self.layer_norm),
26        }
27        return {**base_config, **config}
```

Model.py

```
1 import tensorflow as tf
2 from tensorflow.keras.models import Model
3 from tensorflow.keras.layers import Dense
4 from CustomTransformer.Encoder import Encoder
5 from CustomTransformer.Decoder import Decoder
6 from CustomTransformer.utils import padding_mask, lookahead_mask
7 from CustomTransformer.params import baseline_test_params, Params
8
9 class TransformerModel(Model):
10    def __init__(self, p:Params, **kwargs):
11        super(TransformerModel, self).__init__(**kwargs)
12        self.debug = p.debug
13        self.model_dim = p.model_dim
14        self.encoder = Encoder(p)
15        self.decoder = Decoder(p)
16        self.dense = Dense(p.decoder_vocab_size)
```

```

17     self.train_loss = tf.keras.metrics.Mean(name="train_loss")
18     self.train_accuracy = tf.keras.metrics.Mean(name="train_accuracy")
19     self.val_loss = tf.keras.metrics.Mean(name='val_loss')
20
21     def call(self, input_data, training):
22         encoder_input, decoder_input = input_data
23         padding = padding_mask(encoder_input)
24         #tf.maximum returns tensor maximum element wise - lookahead mask is
25         → a upper triangular matrix of ones to prevent decoder from
26         → looking ahead
27         lookahead = tf.maximum(padding_mask(decoder_input),
28                                → lookahead_mask(decoder_input.shape[1]))
29
30         #encoder takes in encoder input and padding mask
31         encoder_output = self.encoder(encoder_input, padding, training)
32
33         #decoder takes in decoder input, encoder output, lookahead mask, and
34         → padding mask
35         decoder_output = self.decoder(decoder_input, encoder_output,
36                                       → lookahead, padding, training)
37         return self.dense(decoder_output)
38
39     #Overriden train_step method to ensure model can be run using
40     → model.fit()
41     def train_step(self, data):
42         train_batchX, train_batchY = data
43         encoder_input = train_batchX
44         decoder_input = train_batchY[:, :-1]
45         decoder_output = train_batchY[:, 1:]
46         with tf.GradientTape() as tape:
47             #generate prediction
48             prediction = self((encoder_input, decoder_input), training =
49                               → True)
50
51             #compute loss
52             loss = self.compute_loss(decoder_output, prediction)
53
54             # Compute the training accuracy
55             accuracy = self.compute_accuracy(decoder_output, prediction)
56
57             gradients = tape.gradient(loss, self.trainable_variables)
58             self.optimizer.apply_gradients(zip(gradients,
59                                               → self.trainable_variables))
60             self.train_loss.update_state(loss)
61             self.train_accuracy.update_state(accuracy)
62             return {"train_loss": self.train_loss.result(), "train_accuracy":
63                    → self.train_accuracy.result()}
64
65     #Overriden test_step method to ensure model can be evaluated using
66     → model.evaluate() and model.predict()
67     def test_step(self, val_data):
68         val_batchX, val_batchY = val_data

```

```

59     encoder_input = val_batchX
60     decoder_input = val_batchY[:, :-1]
61     decoder_output = val_batchY[:, 1:]
62     prediction = self((encoder_input, decoder_input), training = False)
63
64     loss = self.compute_loss(decoder_output, prediction)
65     self.val_loss.update_state(loss)
66     return {"val_loss": self.val_loss.result()}
67
68     #Custom compile method to ensure model can be run using model.fit() and
69     ↪ can receive custom loss and accuracy functions
70     def compile(self, optimizer, loss_fn, accuracy_fn):
71         super().compile(optimizer = optimizer)
72         self.optimizer = optimizer
73         self.compute_loss = loss_fn
74         self.compute_accuracy = accuracy_fn
75
76     @property
77     def metrics(self):
78         return [self.train_loss, self.train_accuracy, self.val_loss]

```

utils.py

```

1 import tensorflow as tf
2 from tensorflow.keras.losses import sparse_categorical_crossentropy
3
4 def check_shape(name, tensor, expectedshape):
5     assert tensor.shape == expectedshape, f" {name} expected shape
6         ↪ {expectedshape}, shape: {tensor.shape}"
7
8 def padding_mask(input):
9     # Mask out padding values by marking them with True
10    mask = tf.math.equal(input, 0)
11
12    # Cast the mask to float32, true values are cast to 1.0 and false values
13    ↪ are cast to 0.0
14    mask = tf.cast(mask, tf.float32)
15
16    # Add extra dimensions to add the padding to the attention logits
17    return mask[:, tf.newaxis, tf.newaxis, :]
18
19 def lookahead_mask(shape):
20     # Mask out future entries by marking them with a 1.0
21     return 1 - tf.linalg.band_part(tf.ones((shape, shape)), -1, 0)
22
23 #define the custom loss function
24 def custom_loss(y_true, y_pred):
25     # Create mask so that the zero padding values are not included in the
26     ↪ computation of loss
27     padding_mask = tf.math.logical_not(tf.equal(y_true, 0))
28     padding_mask = tf.cast(padding_mask, tf.float32)

```

```

27
28 # Compute a sparse categorical cross-entropy loss on the unmasked values
   ↪ - logits = True if we do not have the softmax in our model
29 loss = tf.keras.losses.sparse_categorical_crossentropy(y_true, y_pred,
   ↪ from_logits=False) * padding_mask
30
31 # Compute the mean loss over the unmasked values
32 return tf.reduce_sum(loss) / tf.reduce_sum(padding_mask)
33
34 def custom_accuracy(y_true,y_pred):
35 # Create mask so that the zero padding values are not included in the
   ↪ computation of accuracy
36 padding_mask = tf.math.logical_not(tf.equal(y_true, 0))
37
38 # Find equal prediction and target values, and apply the padding mask
39 accuracy = tf.equal(tf.cast(y_true,tf.int64), tf.argmax(y_pred, axis=2))
40 accuracy = tf.math.logical_and(padding_mask, accuracy)
41
42 # Cast the True/False values to 32-bit-precision floating-point numbers
43 padding_mask = tf.cast(padding_mask, tf.float32)
44 accuracy = tf.cast(accuracy, tf.float32)
45
46 # Compute the mean accuracy over the unmasked values
47 return tf.reduce_sum(accuracy) / tf.reduce_sum(padding_mask)

```

9.2.3 Keras Transformer Implementation

LearningRateScheduler.py

```

1 from tensorflow.keras.optimizers.schedules import LearningRateSchedule
2 from tensorflow import math
3 import tensorflow as tf
4
5 class LRScheduler(LearningRateSchedule):
6     def __init__(self, d_model, warmup_steps=4000, **kwargs):
7         super(LRScheduler, self).__init__(**kwargs)
8         self.d_model = tf.cast(d_model, tf.float32)
9         self.warmup_steps = warmup_steps
10
11     def __call__(self, step_num):
12         # Linearly increasing the learning rate for the first warmup_steps,
   ↪ and decreasing it thereafter - taken directly from Vaswani et
   ↪ al.
13         arg1 = tf.cast(step_num,tf.float32) ** tf.cast(-0.5, tf.float32)
14         arg2 = tf.cast(step_num, tf.float32) * tf.cast((self.warmup_steps **
   ↪ -1.5), tf.float32)
15
16         return tf.cast((self.d_model ** -0.5) * math.minimum(arg1, arg2),
   ↪ tf.float32)

```

baselineEncoder.py

```
1 import keras
2 from CustomTransformer.params import Params
3 from tensorflow.keras import layers
4 import tensorflow as tf
5
6 class TransformerEncoder(tf.keras.layers.Layer):
7     def __init__(self, p:Params, **kwargs):
8         super().__init__(**kwargs)
9         self.model_dim = p.model_dim
10        self.feed_forward_dim = p.feed_forward_dim
11        self.num_heads = p.num_heads
12
13        self.attention = layers.MultiHeadAttention(
14            num_heads=self.num_heads, key_dim=self.model_dim
15        )
16        self.dense_proj = keras.Sequential(
17            [
18                layers.Dense(self.feed_forward_dim, activation="relu"),
19                layers.Dense(self.model_dim),
20            ]
21        )
22        self.layernorm_1 = layers.LayerNormalization()
23        self.layernorm_2 = layers.LayerNormalization()
24        self.dropout1 = layers.Dropout(p.dropout_rate)
25        self.dropout2 = layers.Dropout(p.dropout_rate)
26        self.supports_masking = True
27
28    def call(self, inputs, mask=None):
29        attention_output = self.attention(query=inputs, value=inputs,
30            ↪ key=inputs)
31        attention_output = self.dropout1(attention_output)
32        proj_input = self.layernorm_1(inputs + attention_output)
33
34        proj_output = self.dense_proj(proj_input)
35        proj_output = self.dropout2(proj_output)
36
37        return self.layernorm_2(proj_input + proj_output)
38
39    def get_config(self):
40        config = super().get_config()
41        config.update(
42            {
43                "model_dim": self.model_dim,
44                "feed_forward_dim": self.feed_forward_dim,
45                "num_heads": self.num_heads,
46            }
47        )
48        return config
```


baselineDecoder.py

```
1 import keras
2 from CustomTransformer.params import Params
3 from tensorflow.keras import layers
4 import tensorflow as tf
5
6 class TransformerDecoder(layers.Layer):
7     def __init__(self, p:Params, **kwargs):
8         super().__init__(**kwargs)
9         self.model_dim = p.model_dim
10        self.feed_forward_dim = p.feed_forward_dim
11        self.num_heads = p.num_heads
12
13        self.attention_1 = layers.MultiHeadAttention(
14            num_heads=self.num_heads, key_dim=self.model_dim
15        )
16
17        self.attention_2 = layers.MultiHeadAttention(
18            num_heads=self.num_heads, key_dim=self.model_dim
19        )
20
21        self.dense_proj = keras.Sequential(
22            [
23                layers.Dense(self.feed_forward_dim, activation="relu"),
24                layers.Dense(self.model_dim),
25            ]
26        )
27        self.layer_norm_1 = layers.LayerNormalization()
28        self.layer_norm_2 = layers.LayerNormalization()
29        self.layer_norm_3 = layers.LayerNormalization()
30
31        self.dropout1 = layers.Dropout(p.dropout_rate)
32        self.dropout2 = layers.Dropout(p.dropout_rate)
33        self.dropout3 = layers.Dropout(p.dropout_rate)
34
35        self.add = layers.Add() # instead of `+` to preserve mask
36        self.supports_masking = True
37
38    def call(self, inputs, encoder_outputs, mask=None):
39        attention_output_1 = self.attention_1(
40            query=inputs,
41            value=inputs,
42            key=inputs, use_causal_mask=True
43        )
44        attention_output_1 = self.dropout1(attention_output_1)
45        out_1 = self.layer_norm_1(self.add([inputs, attention_output_1]))
46
47        attention_output_2 = self.attention_2(
48            query=out_1,
49            value=encoder_outputs,
50            key=encoder_outputs,
51        )
```

```

52     attention_output_2 = self.dropout2(attention_output_2)
53     out_2 = self.layernorm_2(self.add([out_1, attention_output_2]))
54
55     proj_output = self.dense_proj(out_2)
56     proj_output = self.dropout3(proj_output)
57
58     return self.layernorm_3(self.add([out_2, proj_output]))
59
60     def get_config(self):
61         config = super().get_config()
62         config.update(
63             {
64                 "model_dim": self.model_dim,
65                 "feed_forward_dim": self.feed_forward_dim,
66                 "num_heads": self.num_heads,
67             }
68         )
69         return config

```

baselineModel.py

```

1  import keras_nlp
2  import keras
3  from CustomTransformer.params import Params, midi_test_params_v2
4  from .baselineEncoder import TransformerEncoder
5  from .baselineDecoder import TransformerDecoder
6  from tensorflow.keras import layers
7  import tensorflow as tf
8
9  class PositionalEmbedding(layers.Layer):
10     def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
11         super().__init__(**kwargs)
12         self.embedding_layer = keras_nlp.layers.TokenAndPositionEmbedding(
13             vocabulary_size=vocab_size,
14             sequence_length=sequence_length,
15             embedding_dim=embed_dim,
16             mask_zero=True,
17         )
18         self.sequence_length = sequence_length
19         self.vocab_size = vocab_size
20         self.embed_dim = embed_dim
21
22     def call(self, inputs):
23         return self.embedding_layer(inputs)
24
25     def compute_mask(self, inputs, mask=None):
26         return tf.math.not_equal(inputs, 0)
27
28     def get_config(self):
29         config = super().get_config()
30         config.update(
31             {

```

```

32         "sequence_length": self.sequence_length,
33         "vocab_size": self.vocab_size,
34         "embed_dim": self.embed_dim,
35     }
36 )
37     return config
38
39 def createBaselineTransformer(p:Params):
40     encoder_inputs = keras.Input(shape=(None,), dtype="uint16",
41     ↪ name="encoder_inputs")
42     x = PositionalEmbedding(p.encoder_seq_len, p.encoder_vocab_size,
43     ↪ p.model_dim)(encoder_inputs)
44     for _ in range(p.num_encoder_layers):
45         x = TransformerEncoder(p)(x)
46     encoder_outputs = x
47     encoder = keras.Model(encoder_inputs, encoder_outputs)
48
49     decoder_inputs = keras.Input(shape=(None,), dtype="uint16",
50     ↪ name="decoder_inputs")
51     encoded_seq_inputs = keras.Input(shape=(None, p.model_dim),
52     ↪ name="decoder_state_inputs")
53
54     x = PositionalEmbedding(p.decoder_seq_len, p.decoder_vocab_size,
55     ↪ p.model_dim)(decoder_inputs)
56     for _ in range(p.num_decoder_layers):
57         x = TransformerDecoder(p)(x, encoded_seq_inputs)
58     decoder_outputs = layers.Dense(p.decoder_vocab_size,
59     ↪ activation="softmax")(x)
60     decoder = keras.Model([decoder_inputs, encoded_seq_inputs],
61     ↪ decoder_outputs)
62
63     decoder_outputs = decoder([decoder_inputs, encoder_outputs])
64     transformer = keras.Model(
65         [encoder_inputs, decoder_inputs], decoder_outputs,
66         ↪ name="transformer"
67     )
68
69     return transformer
70
71 if __name__ == "__main__":
72     p = Params(midi_test_params_v2)
73     model = createBaselineTransformer(p)
74     model.summary()

```

9.3 Analysis

Analysis.py

```
1 class Analysis:
2     def __init__(self):
3         self.seq_data = {
4             'input': [],
5             'output': [],
6             'actual': []
7         }
8         self.pitch_list = [ "C", "C#/Db", "D", "D#/Eb", "E", "F", "F#/Gb",
9             ↪ "G", "G#/Ab", "A", "A#/Bb", "B"]
10        self.pitch_class_dict = {i: self.pitch_list[i] for i in
11            ↪ range(len(self.pitch_list))}
12
13    def load_encoded_MIDI_seq(self, input_seq, output_seq, actual_seq):
14        self.seq_data['input'] = input_seq
15        self.seq_data['output'] = output_seq
16        self.seq_data['actual'] = actual_seq
17        # self.seq_data['input'].append(input_seq)
18        # self.seq_data['output'].append(output_seq)
19        # self.seq_data['actual'].append(actual_seq)
20
21    def decode_single_event(self, event):
22        valid_value = event
23        if event in range_note_on:
24            return ('note_on', valid_value)
25
26        elif event in range_note_off:
27            valid_value -= RANGE_NOTE_ON
28            return ('note_off', valid_value)
29
30        elif event in range_time_shift:
31            valid_value -= (RANGE_NOTE_ON + RANGE_NOTE_OFF)
32            return ('time_shift', valid_value)
33
34        else:
35            valid_value -= (RANGE_NOTE_ON + RANGE_NOTE_OFF +
36                ↪ RANGE_TIME_SHIFT)
37            return ('velocity', valid_value)
38
39    def get_histograms(self):
40        histograms_from_seq = {key:
41            ↪ self.calculate_event_distribution(self.seq_data[key]) for key in
42            ↪ self.seq_data}
43        return histograms_from_seq
44
45    def calculate_event_distribution(self, seq):
46        note_on_counts = [0] * 12
47        note_off_counts = [0] * 12
48        time_shift_counts = [0] * 100
```

```

44     velocity_counts = [0] * 32
45
46     for event in seq:
47         event_type, event_value = self.decode_single_event(event)
48         if event_type == 'note_on':
49             pitch_class = event_value % 12
50             note_on_counts[pitch_class] += 1
51         elif event_type == 'note_off':
52             pitch_class = event_value % 12
53             note_off_counts[pitch_class] += 1
54         elif event_type == 'time_shift':
55             time_shift_counts[event_value] += 1
56         elif event_type == 'velocity':
57             velocity_counts[event_value] += 1
58         else:
59             raise Exception("Invalid event type decoded")
60     #normalize by sum of all notes - to turn into a probability
61     → distribution
62     note_on_counts = np.array(note_on_counts) / np.sum(note_on_counts)
63     note_off_counts = np.array(note_off_counts) /
64     → np.sum(note_off_counts)
65     time_shift_counts = np.array(time_shift_counts) /
66     → np.sum(time_shift_counts)
67     velocity_counts = np.array(velocity_counts) /
68     → np.sum(velocity_counts)
69
70     #create dictionaries to be used for histogram
71     note_on_dict = {self.pitch_class_dict[i]: note_on_counts[i] for i in
72     → range(len(note_on_counts))}
73     note_off_dict = {self.pitch_class_dict[i]: note_off_counts[i] for i
74     → in range(len(note_off_counts))}
75
76     time_shift_dict = {i: time_shift_counts[i] for i in
77     → range(len(time_shift_counts))}
78     velocity_dict = {i: velocity_counts[i] for i in
79     → range(len(velocity_counts))}
80     return {'note_on': note_on_dict, 'note_off': note_off_dict,
81     → 'time_shift': time_shift_dict, 'velocity': velocity_dict}
82
83 def get_entropies(self, histograms):
84     input_histograms, output_histograms, actual_histograms =
85     → histograms['input'], histograms['output'], histograms['actual']
86     all_entropies = {
87         'input': {},
88         'output': {},
89         'actual': {}
90     }
91     for key, histogram in input_histograms.items():
92         all_entropies["input"][key] = sum([histogram[k] *
93         → np.log2(histogram[k]) for k in histogram.keys() if
94         → histogram[k] > 0])
95     for key, histogram in output_histograms.items():

```

```

84     all_entropies["output"][key] = sum([histogram[k] *
    ↪ np.log2(histogram[k]) for k in histogram.keys() if
    ↪ histogram[k] > 0])
85     for key, histogram in actual_histograms.items():
86         all_entropies["actual"][key] = sum([histogram[k] *
    ↪ np.log2(histogram[k]) for k in histogram.keys() if
    ↪ histogram[k] > 0])
87     return all_entropies
88
89     def plot_values(self, histograms, entropies, name):
90         for key, histogram in histograms.items():
91             plt.bar(histogram.keys(), histogram.values())
92             plt.title(f'{name}-{key} (Entropy value: {entropies[key]})')
93             plt.show()

```

9.4 TSImprovisor

Improvisor.ts

```

1 import * as core from "@magenta/music/node/core";
2 import { MusicRNN } from "@magenta/music/node/music_rnn";
3
4 const _ = require('lodash');
5 const Detect = require('tonal-detect');
6 const Tonal = require('tonal');
7 //require("@tensorflow/tfjs-node");
8
9 //import { zeros } from "@tensorflow/tfjs-node";
10 import { TimeSettings, ChordProg, NumBeats, BeatValue, StepsPerQuarter,
    ↪ Note, logErrorToMax } from "./utils"
11 import type { NoteSequence } from "@magenta/music/node/protobuf/index.d.ts";
12 import { tensorflow } from "@magenta/music/node/protobuf/proto";
13
14 export class Improvisor {
15     model: MusicRNN;
16     timeSettings: TimeSettings;
17     currentChordProg: ChordProg;
18     inputNotes: Note[];
19     quantizedInput: NoteSequence;
20
21     constructor(timeSettings: TimeSettings) {
22         this.model = this.loadModel('...')
23         this.timeSettings = timeSettings;
24         this.currentChordProg = [];
25         this.inputNotes = [];
26         this.quantizedInput = core.sequences.createQuantizedNoteSequence(
27             this.timeSettings.stepsPerQuarter,
28             this.timeSettings.qpm
29         );
30     }

```

```

31
32 loadModel(path:string):MusicRNN {
33     let rnn = new MusicRNN(path);
34     rnn.initialize();
35     return rnn;
36 }
37
38 updateTimeSettings(timeSettings: TimeSettings) {
39     this.timeSettings = timeSettings;
40 }
41
42 updateChordProg(chordProg: ChordProg) {
43     this.currentChordProg = chordProg;
44 }
45
46 quantizeInputNotes(){
47     try{
48         if (!this.model){ throw new Error("Model not loaded!"); }
49         if (!this.model.isInitialized){ throw new Error("Model not
↳ Initialized!");}
50         if (!this.timeSettings){ throw new Error("Time Settings not
↳ initialized!");}
51         if (!this.inputNotes){ throw new Error("No input notes!"); }
52
53         if (this.inputNotes){
54             let notes = [];
55             for (let i = 0; i < this.inputNotes.length; i++){
56                 notes.push({
57                     pitch: this.inputNotes[i].pitch,
58                     startTime: this.inputNotes[i].startTime,
59                     endTime: this.inputNotes[i].startTime +
↳ this.inputNotes[i].duration,
60                 });
61             }
62
63             const unquantizedSequence = {
64                 notes:notes,
65                 tempos: [
66                     {
67                         time: 0,
68                         qpm: this.timeSettings.qpm
69                     }
70                 ],
71                 totalTime: 60 / this.timeSettings.qpm *
↳ this.timeSettings.numbeats,
72             }
73
74             let quantizedSequence =
↳ core.sequences.quantizeNoteSequence(unquantizedSequence,
↳ this.timeSettings.stepsPerQuarter);
75             this.quantizedInput = quantizedSequence;
76         } else {

```

```

77         throw new Error("input notes could not be Quantized!");
78     }
79     } catch (error : any) {
80         console.log(error);
81     }
82 }
83
84 async generateNewSequence(): Promise<tensorflow.magenta.INoteSequence >
85 ↪ {
86     let midiNotes = this.quantizedInput.notes.map(n => n.pitch);
87     console.log("midinotes",midiNotes);
88     const notes = midiNotes.map(Tonal.Note.fromMidi);
89     console.log("notes",notes);
90     const possibleChords:ChordProg = Detect.chord(notes);
91     console.log("chords",possibleChords);
92
93     let stepsToGenerate = this.getNumStepsToGenerate();
94     try {
95         //Provide quantized note sequence, steps to generate,
96         ↪ temperature and chord progression
97         return await this.model.continueSequence(
98             this.quantizedInput,
99             stepsToGenerate,
100             1.2,
101             possibleChords
102         );
103     } catch (error: any) {
104         console.log(error);
105         //return empty sequeunce
106         return core.sequences.createQuantizedNoteSequence(
107             this.timeSettings.stepsPerQuarter,
108             this.timeSettings.qpm
109         );
110     }
111 }
112
113 //Necessary because tempo is in quarter notes per minute
114 /*
115 eg.
116 3/4 with 12 steps per quarter = 36
117 6/8 has 6 beats, so each must have only 6 steps per quarter to have 36
118 ↪ steps
119 */
120
121 getNumStepsToGenerate(): number {
122     let numSteps: number = this.timeSettings.numbeats *
123     ↪ this.timeSettings.stepsPerQuarter;
124     if(this.timeSettings.beatvalue == 8) numSteps /= 2;
125     return numSteps;
126 }
127 }

```


Main.ts

```
1 import { Improvisor } from "./improvisor";
2 import { NumBeats, BeatValue, StepsPerQuarter, TimeSettings, Note} from
  ↳ "./utils";
3
4 const maxApi = require("max-api");
5 let improvisor: Improvisor;
6 //retrieve time Settings from Max Patch, initialize Improvisor
7 maxApi.addHandler("setTimeSettings", (numbeats:NumBeats,
  ↳ beatvalue:BeatValue, stepsPerQuarter:StepsPerQuarter, qpm:number) => {
8   //measured beat is currently not used
9   const timeSettings:TimeSettings = {
10     numbeats: numbeats,
11     beatvalue: beatvalue,
12     stepsPerQuarter: stepsPerQuarter,
13     qpm: qpm
14   }
15   if (improvisor) {
16     improvisor.updateTimeSettings(timeSettings);
17   } else {
18     improvisor = new Improvisor(timeSettings);
19   }
20   console.log(improvisor.timeSettings);
21 });
22
23 maxApi.addHandler("getNotes", (...midiNotes: number[]) => { //need to use
  ↳ spread operator to grab all values
24   if (midiNotes.length == 0){
25     return
26   }
27   let notes = [];
28   for (let i = 0; i < midiNotes.length; i+=4) {
29     let note : Note = {
30       pitch: midiNotes[i],
31       velocity: midiNotes[i+1],
32       duration: midiNotes[i+2]/1000,
33       startTime: midiNotes[i+3] //already in seconds
34     }
35     if (improvisor.inputNotes === null) {
36       improvisor.inputNotes = [];
37     }
38     notes.push(note);
39   }
40   improvisor.inputNotes = notes;
41   console.log("INPUT NOTES\n",improvisor.inputNotes)
42   improvisor.quantizeInputNotes();
43   console.log("QUANTIZED INPUT NOTES\n", improvisor.quantizedInput);
44 });
45
46 maxApi.addHandler("generateSequence", async() => {
47   let generated = await improvisor.generateNewSequence();
48   console.log("GENERATED NOTES\n", generated)
```

```
49   maxApi.outlet(generated);
50 }
```

utils.ts

```
1  const maxApi = require("max-api");
2
3  export type NumBeats = 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 12;
4  export type BeatValue = 4 | 8;
5  export type StepsPerQuarter = 1 | 2 | 4 | 6 | 8 | 12; //let 12 be default
   → (whole, half, triplet half, 4th, triple 4th, 8th, triplet 8th, 16th,
   → triplet 16th)
6  //Array of strings representing chords
7  export type ChordProg = Array<string>;
8  export interface Note {
9    pitch: number;
10   velocity: number;
11   duration: number;
12   startTime: number; //start time from start of the bar
13 }
14
15 export interface TimeSettings {
16   numbeats: NumBeats
17   beatvalue: BeatValue
18   stepsPerQuarter: StepsPerQuarter
19   qpm: number //quarter notes per minute
20 }
21
22 export function logErrorToMax(error: any) {
23   maxApi.post(error, maxApi.POST_LEVELS.ERROR);
24 }
```

Bibliography

- [1] Eric J. Humphrey, Juan Pablo Bello, and Yann LeCun. “Moving beyond feature design: Deep architectures and automatic feature learning in music informatics”. English (US). In: *Proceedings of the 13th International Society for Music Information Retrieval Conference, ISMIR 2012*. Proceedings of the 13th International Society for Music Information Retrieval Conference, ISMIR 2012. 13th International Society for Music Information Retrieval Conference, ISMIR 2012 ; Conference date: 08-10-2012 Through 12-10-2012. 2012, pp. 403–408. ISBN: 9789727521449.
- [2] George E. Lewis. “Too Many Notes: Computers, Complexity and Culture in ”Voyager””. In: *Leonardo Music Journal* 10 (2000), pp. 33–39. URL: <http://www.jstor.org/stable/1513376>.
- [3] John Biles. “GenJam: A Genetic Algorithm for Generating Jazz Solos”. In: (July 1994).
- [4] Ashish Vaswani et al. “Attention Is All You Need”. In: (2023). arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs.CL].
- [5] Sageev Oore et al. “This Time with Feeling: Learning Expressive Musical Performance”. In: (2018). arXiv: [1808.03715](https://arxiv.org/abs/1808.03715) [cs.SD].
- [6] Joseph Weizenbaum. “ELIZA—a computer program for the study of natural language communication between man and machine”. In: *Communications of the ACM* 9 (1966), pp. 36–45. URL: <https://api.semanticscholar.org/CorpusID:1896290>.
- [7] Bruce G. Buchanan and Edward H. Shortliffe. “Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley series in artificial intelligence)”. In: 1984. URL: <https://api.semanticscholar.org/CorpusID:59779332>.

- [8] Bruce G. Buchanan. “A (Very) Brief History of Artificial Intelligence”. In: *AI Mag.* 26 (2005), pp. 53–60. URL: <https://api.semanticscholar.org/CorpusID:8005552>.
- [9] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536. URL: <https://api.semanticscholar.org/CorpusID:205001834>.
- [10] Ian Goodfellow et al. “Generative Adversarial Networks”. In: *Commun. ACM* 63.11 (Oct. 2020), pp. 139–144. ISSN: 0001-0782. DOI: [10.1145/3422622](https://doi.org/10.1145/3422622). URL: <https://doi.org/10.1145/3422622>.
- [11] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: [10.1162/neco.1997.9.8.1735](https://doi.org/10.1162/neco.1997.9.8.1735). eprint: <https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf>. URL: <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [12] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: [1312.6114](https://arxiv.org/abs/1312.6114) [stat.ML].
- [13] Stephan R. Richter et al. *Playing for Data: Ground Truth from Computer Games*. 2016. arXiv: [1608.02192](https://arxiv.org/abs/1608.02192) [cs.CV].
- [14] John Biles. “Interactive GenJam: Integrating Real-time Performance with a Genetic Algorithm”. In: (Oct. 1998). URL: <https://genjamorg.files.wordpress.com/2019/07/bilesicmc98.pdf>.
- [15] Cheng-Zhi Anna Huang et al. “Music Transformer”. In: (2018). arXiv: [1809.04281](https://arxiv.org/abs/1809.04281) [cs.LG].
- [16] Curtis Hawthorne et al. “Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset”. In: (2019). URL: <https://openreview.net/forum?id=r11YRjC9F7>.
- [17] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. “Self-Attention with Relative Position Representations”. In: (June 2018), pp. 464–468. DOI: [10.18653/v1/N18-2074](https://doi.org/10.18653/v1/N18-2074). URL: <https://aclanthology.org/N18-2074>.
- [18] Cycling ’74. *Max 8*. <https://cycling74.com/products/max>. [Computer software]. 2023.
- [19] Junyoung Chung et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In: (Dec. 2014).

- [20] Oleksii Kuchaiev and Boris Ginsburg. “Factorization tricks for LSTM networks”. In: *CoRR* abs/1703.10722 (2017). arXiv: [1703.10722](https://arxiv.org/abs/1703.10722). URL: <http://arxiv.org/abs/1703.10722>.
- [21] Noam Shazeer et al. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer”. In: *CoRR* abs/1701.06538 (2017). arXiv: [1701.06538](https://arxiv.org/abs/1701.06538). URL: <http://arxiv.org/abs/1701.06538>.
- [22] Yoon Kim et al. “Structured Attention Networks”. In: *CoRR* abs/1702.00887 (2017). arXiv: [1702.00887](https://arxiv.org/abs/1702.00887). URL: <http://arxiv.org/abs/1702.00887>.
- [23] Andrew N. Robertson and Mark D. Plumbley. “POST-PROCESSING FIDDLE : A REAL-TIME MULTI-PITCH TRACKING TECHNIQUE USING HARMONIC PARTIAL SUBTRACTION FOR USE WITHIN LIVE PERFORMANCE SYSTEMS”. In: *International Computer Music Association* (2009).
- [24] John Thickstun, Zaid Harchaoui, and Sham Kakade. *Learning Features of Music from Scratch*. 2017. arXiv: [1611.09827](https://arxiv.org/abs/1611.09827) [[stat.ML](https://arxiv.org/abs/1611.09827)].
- [25] Valentin Emiya et al. *MAPS - A piano database for multipitch estimation and automatic transcription of music*. Research Report. July 2010, p. 11. URL: <https://inria.hal.science/inria-00544155>.
- [26] Meinard Müller et al. “Saarland Music Data (SMD)”. In: Jan. 2011.
- [27] Kishore Papineni et al. “BLEU: A Method for Automatic Evaluation of Machine Translation”. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL ’02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. DOI: [10.3115/1073083.1073135](https://doi.org/10.3115/1073083.1073135). URL: <https://doi.org/10.3115/1073083.1073135>.
- [28] Chin-Yew Lin. “ROUGE: A Package for Automatic Evaluation of Summaries”. In: *Annual Meeting of the Association for Computational Linguistics*. 2004. URL: <https://api.semanticscholar.org/CorpusID:964287>.
- [29] Lucas Theis, Aäron van den Oord, and Matthias Bethge. *A note on the evaluation of generative models*. 2016. arXiv: [1511.01844](https://arxiv.org/abs/1511.01844) [[stat.ML](https://arxiv.org/abs/1511.01844)].
- [30] Shih-Lun Wu and Yi-Hsuan Yang. *The Jazz Transformer on the Front Line: Exploring the Shortcomings of AI-composed Music through Quantitative Measures*. 2020. arXiv: [2008.01307](https://arxiv.org/abs/2008.01307) [[cs.SD](https://arxiv.org/abs/2008.01307)].
- [31] Drew Edwards, Simon Dixon, and Emmanouil Benetos. “PiJAMA: Piano Jazz with Automatic MIDI Annotations”. In: Zenodo, Sept. 2023. DOI: [10.5281/zenodo.8354955](https://doi.org/10.5281/zenodo.8354955). URL: <https://doi.org/10.5281/zenodo.8354955>.

- [32] Curtis Hawthorne. “Transformer-NADE for Piano Performances”. In: 2018. URL: <https://api.semanticscholar.org/CorpusID:197468126>.
- [33] William Fedus, Barret Zoph, and Noam Shazeer. “Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity”. In: *CoRR* abs/2101.03961 (2021). arXiv: [2101.03961](https://arxiv.org/abs/2101.03961). URL: <https://arxiv.org/abs/2101.03961>.
- [34] Zihang Dai et al. “Transformer-XL: Attentive Language Models beyond a Fixed-Length Context”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 2978–2988. DOI: [10.18653/v1/P19-1285](https://doi.org/10.18653/v1/P19-1285). URL: <https://aclanthology.org/P19-1285>.
- [35] Andrew Jaegle et al. “Perceiver: General Perception with Iterative Attention”. In: *CoRR* abs/2103.03206 (2021). arXiv: [2103.03206](https://arxiv.org/abs/2103.03206). URL: <https://arxiv.org/abs/2103.03206>.
- [36] Mehwish Alam et al. “MIDI2vec: Learning MIDI Embeddings for Reliable Prediction of Symbolic Music Metadata”. In: *Semant. Web* 13.3 (Jan. 2022), pp. 357–377. ISSN: 1570-0844. DOI: [10.3233/SW-210446](https://doi.org/10.3233/SW-210446). URL: <https://doi.org/10.3233/SW-210446>.
- [37] Rachel M. Bittner et al. *A Lightweight Instrument-Agnostic Model for Polyphonic Note Transcription and Multipitch Estimation*. 2022. arXiv: [2203.09893](https://arxiv.org/abs/2203.09893) [cs.SD].