Generative Transformer-based Models of Symbolic Polyphonic Music

Vincent Herrmann

Master Thesis

Institute for Music Informatics and Musicology
University of Music Karlsruhe
June 2020

This work was done at the *Bosch Center for Artificial Intelligence* under the supervision of Dr. Michael Herman and Konrad Groh.

Academic Supervisor:
Prof. Dr. Christoph Seibert

I hereby declare that I have written the present paper independently and have not used any sources or aids other than those indicated.

Contents

Chapter 1

Introduction

The topic of this work is developing generative models that can create pieces of music autonomously. There are two basic ways in which music can be generated, although the line between them is not always clear: as a concrete perceptual signal (an audio clip) or in a symbolic representational format, which means producing a musical score instead of an audio file. In recent years there have been successful efforts to generate music in the audio domain (A. v. d. Oord et al., 2016; Dieleman, A. v. d. Oord, and Simonyan, 2018; Vasquez and Lewis, 2019; Dhariwal et al., 2020). Nevertheless, there are important advantages in modelling music symbolically. Such a score can be performed by human musicians as well as by a midi sequencer. Dealing with symbolic music is certainly closer to the way composers go about their work. The general ability to recognize and construct relations between symbols is just as important for many tasks beyond music and is one of the stepping stones on the path to general artificial reasoning (see for example Garcez et al., 2015).

In chapter 2, I present the ideas of generative modelling using some historical examples. When designing a generative model, many factors have to be weighed against each other. One of the most fundamental aspects is how rigid or flexible the exact form of representation is. Since the success of large neural networks and deep learning, it is possible to use extremely general and powerful representations without overwhelming the probabilistic model. This means that by training with large amounts of data, complex and subtle characteristics of music can be captured automatically. A command-based representational format inspired by the MIDI protocol has enormous expressive power and can still be rendered as a one-dimensional sequence, which is important for our models.

The main part of this work is divided into chapters three and four. Chapter 3 is devoted to the Transformer architecture (Vaswani et al., 2017), a kind of neural network that has revolutionized the field of natural language processing in recent years. I examine the individual components of the architecture, especially the attention mechanism, for their function and behaviour. Transformers have already been successfully used to create symbolic music in an autoregressive way (Huang, Vaswani, et al., 2018; Payne, 2019). That is, sequences are generated one item at a time. Besides new visualization methods that allow to understand the way the model builds up an understanding of the input sequences, the main technical innovation that brings a significant improvement in performance is an extension of the input representation with dynamic features. These can be calculated deterministically from the command-based representation and provide each item in the sequence with additional context information that stabilizes the model. The four-part chorales by Johann Sebastian Bach serve as a field of experimentation. But it is also shown that the same principles work for the much more complex compositions of the MAESTRO dataset (Hawthorne et al., 2018).

In chapter 4, I investigate a new approach that allows to learn more abstract and lower-resolution representations, and to convert them back into an understandable form. The sequences of learned representations can then be created by a new latent generative model. This has several potential advantages: The strictly linear (and maybe somewhat unmusical) modelling style of the previous models is broken up. Complex temporal relationships can thus be more easily captured.

In general, it is possible to generate longer coherent sequences because the learned representations are more compact. It also increases the control we have over the generation process.

In order to train generative models for the learned representations without too many problems, these must be sequences of discrete items. Learning discrete representations is, however, in some ways at odds with the basic principle of differentiability, which is a prerequisite for the efficient training of neural networks. Based on the work of A. v. d. Oord, Vinyals, et al., 2017, we investigate several quantization methods that nevertheless allow an undisturbed gradient flow. Learning discrete representations of symbolic music, which has not been done before in this form, presents some challenges that will be explained in more detail. I propose a completely transformer-based quantized auto-encoder architecture. It allows the discrete encoding and decoding of Bach chorales as well as their unconditional generation. This approach, however, still has a lot of potential for further development.

The experiments were programmed using the frameworks PyTorch (for creating and training the neural networks, Paszke et al., 2019) and mido (for processing the MIDI files, `https://github.com/mido`). Audio clips of the presented examples are available online (`vincentherrmann.github.io/mami_thesis`).

Chapter 2

Background

The question “How does music work?” has occupied the minds of people since antiquity. It is not not only asked in a practical, empirical or artistic sense—“How can we produce beautiful sounds?” or “Which combinations of notes seems to convey suspense, which resolution?”. There is often also a more fundamental, almost mathematical aspect—“What are the basic principles that give beauty to music”, “Why do we perceive some combinations or configurations of sounds as music and others not?”.

Trying to scientifically describe, explain and generate music might have been pursued since the advent of science itself. In ancient Greece, Pythagoras saw the motion of the heavenly bodies reflected in the pitches a string can produce (Calter, 1998). One can also argue that such efforts were present from the very beginnings of the field of *computer* science. Ada Lovelace is sometimes described as the first computer programmer in history. She was the associate of Charles Babbage, inventor of the first universal computer, the so-called *Analytical Engine*, which he described in 1837 but was never actually realized in his lifetime. Lovelace explicitly mentions the possibility of using computers to create “scientific pieces of music of any degree of complexity or extent” (Lovelace, 1843). This has indeed been tried in myriads of ways and interest has by no means faded away.

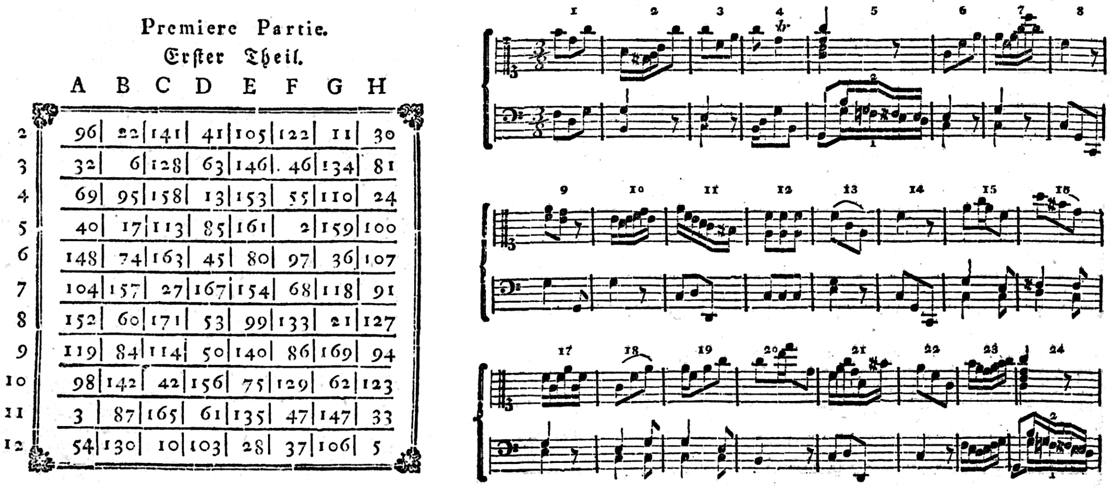
2.1 Generative models of music

Maybe one can frame these aspirations as the quest of finding *generative models* of music. In the broadest sense, a statistical generative model captures the properties and regularities of some musical data by either explicitly or implicitly representing the probability distribution

$$p_{Music}(\mathbf{x}).$$

(cf. Bishop, 2006). Here, $\mathbf{x}$ is any potentially musical artefact. The function p_{Music} states, as it were, how probable it is that a specific $\mathbf{x}$ is a piece of music. In principle, one can sample from this distribution to generate a new piece of music. What exactly p_{Music} expresses depends of course on the *what* exactly can be represented by $\mathbf{x}$, and *how* these supposedly musical artefacts are represented. If, for example, $\mathbf{x}$ can be any conceivable audio clip, the task of differentiating in a robust and general way between music and noise (in our case any non-musical configuration of sounds) is extremely challenging because the space of all possible audio clips is vast. On the other hand, $\mathbf{x}$ could have a representational format so that it can only ever be music. It could, for instance, simply state the opus number and the composer of a piece. Then the generative model has nothing to do at all. All the work has been done on the representational side, by the composers writing the music in the first place as well as by musicologists systematizing it and making it accessible.

Further historic and theoretical reflections would certainly be fascinating. They are, however, not the focus of this work. I will instead present a few historical milestones and highlight their



Premiere Partie.
Erſter Theil.

	A	B	C	D	E	F	G	H
2	96	22	141	41	105	122	11	30
3	32	6	128	63	146	46	134	81
4	69	95	158	13	153	55	110	24
5	40	17	113	85	161	2	159	100
6	148	74	163	45	80	97	36	107
7	104	157	27	167	154	68	118	91
8	152	60	171	53	99	133	21	127
9	119	84	114	50	140	86	169	94
10	98	142	42	156	75	129	62	123
11	3	87	165	61	135	47	147	33
12	54	130	10	103	28	37	106	5

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

Figure 2.1: Lookup table and some of the musical segments from *Anleitung so viel Walzer man will mit Würfeln zu componiren ohne musikalisch zu seyn oder Composition zu wissen* (Mozart, 1790).

basic principles insofar as they will be relevant in the later chapters.

2.1.1 Musical Games of Dice

Some forms of what could be called autonomous composing systems date back to at least the eighteenth century. In so-called *Musikalische Würfelspiele*, or musical games of dice, short segments of music were combined in an aleatoric fashion to give rise to new compositions. These segments were usually crafted by a composer for the specific purpose of fitting together in all possible configurations. Some *Würfelspiele* have been ascribed to Joseph Haydn and Wolfgang Amadeus Mozart (Ratner, 1970).

For this kind of system, only the most simple probabilistic model is required—a die. In other words, the distribution we are modelling is a basic uniform distribution:

$$p(\mathbf{x}_i) = \frac{1}{n}$$

The notation $p(\mathbf{x}_i)$ is short for $p(\mathbf{X}_i = \mathbf{x})$. Here, $\mathbf{X}_i$ is our random variable for the ith item in a sequence. It can have n different values, where n is the number of possible values our random variable $\mathbf{X}$ can have (i.e. the number of faces on the die). $\mathbf{x}$ denotes one specific item out of the n choices. We see that the selection of one individual item does not depend on the other ones, or formally that the items are conditionally independent, meaning that the joint distribution can be factorized (L is the number of items in the sequence we want to generate):

$$p(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_L) = p(\mathbf{x}_1)p(\mathbf{x}_2)...p(\mathbf{x}_L) = \frac{1}{n^L}$$

This, of course, puts a major constraint on the kinds of musical segments that can be used. We can see this from the fact that almost all usually occurring musical segments will not be compatible if one just strings them together in a random order.

In Figure 2.1, we can see an excerpt from the *Würfelspiel* by Mozart that creates short waltzes for piano. The columns show the allowed values of our random variable $\mathbf{X}_i$ at each step. The row is picked by throwing two dice for every step.

It is possible, in principle, to construct elaborate such games that create large numbers of diverse musical pieces, while still relying only on the most basic probabilistic generative model.

This is because it relies purely on the representational structure of the data. With that, I mean that most music, if it does not meet the extremely narrow requirements of having to fit in a certain musical context, can and must not be represented at all. All the musical insight and knowledge is contained in the data representation and none in the generative model.

2.1.2 Markov Chains and Emmy

The table in Figure 2.1 states the values that are allowed depending on the current index i in the sequence. But it is probably more sensible in a musical setting to make the allowed values dependent on the previous segment instead of the absolute position in the sequence. This requires modelling the conditional distribution $p(\mathbf{x}_i|\mathbf{x}_{i-1})$. The columns in the table then do not longer represent the position in the sequence but the value of the previous item.

Such a generative model is also called a *Markov chain* (Markov, 1971). To create new samples from this model, we take the current value (or begin with a generic starting value) and use it to pick the appropriate column. From this column, we sample the value of the next item which then, in turn, determines the column for the item after that, and so on. This is an instance of *autoregressive generation*, which means creating a sequence item for item where each item is dependent on what has already been generated. It will remain a crucial concept for this work.

Instead of building a generative model from scratch by hand, it is possible and perhaps desirable to learn it from existing data—that indeed is the main topic of this work. To learn a Markov chain, we construct a square matrix of zeros, with the number of rows and the number of columns both being the number of distinct values the items in the dataset can have. The value of the current item determines the row and the value of the previous item determines the column. We go sequentially through the dataset and record every item in the matrix by adding the value 1 to the corresponding cell. In the end, the columns of the matrix are normalized so that the sum of each column is 1, giving us the conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$.

The conditioning is still only on the previous item. Because of this, only very short local structures can be modelled. Of course, what can be modelled also depends on what exactly our items are. An item could represent something very generic—for example a single note. Every note will occur many times in the dataset, which means that detailed conditional distributions can be learned. But dependence on just the previous note is not enough to capture anything musically meaningful. On the other hand, if the items each represent, say, a whole bar of music then most items will occur only once in the whole dataset. This leads to deterministic distributions, where only one specific item can follow another. The model then can only replicate existing pieces (it is *overfitting* on the training data). One has to think carefully about the level of detail, the granularity and generality of the representations used for a Markov chain.

The composer David Cope extended the Markov chain approach for his *Experiments in Musical Intelligence* (short *Emmy*) with remarkable results (Cope, 2001). Cope captures the conditional dependencies of several hand-crafted properties of the items (such as tension or resolution at multiple hierarchical levels or occurrences of motives). Additionally, he uses sophisticated rules and pattern matching methods to counteract some of the limitations of Markov chains. As before, one could say that here the important work is done mostly on the representational side.

2.1.3 A Generative Theory of Tonal Music

In linguistics, there have been efforts to find a generative grammar, or the general principles that enable the comprehension and versatile use of language. This approach was prominently represented by Noam Chomsky (Chomsky, 1965). Analogous to the generative grammar of languages, generative theories of music have been developed, most famously by the composer Fred Lerdahl and the linguist Ray Jackendorff ("A generative theory of tonal music", GTTM, Lerdahl and Jackendoff, 1996).

It tries to formally describe how listeners understand, in a mostly unconscious way, the music they are hearing. As the title suggests, the reach of this theory is limited to western classical tonal music. Four important structures are specified, all of them hierarchical to the effect that they consist of nested regions across a wide range of scale: The *grouping structure* expresses segmentation into entities such as motive or phrase. The *metrical structure* discriminates strong and weak beats. *Time-span reduction* uses the grouping and metrical structures to construct a tree where at every level fewer representative elements (so-called "heads") of the corresponding segment are retained. The last structure, *prolongation reduction*, seeks to capture the varying tension and stability of segments. The formation of these structures is governed by a multitude of explicit rules.

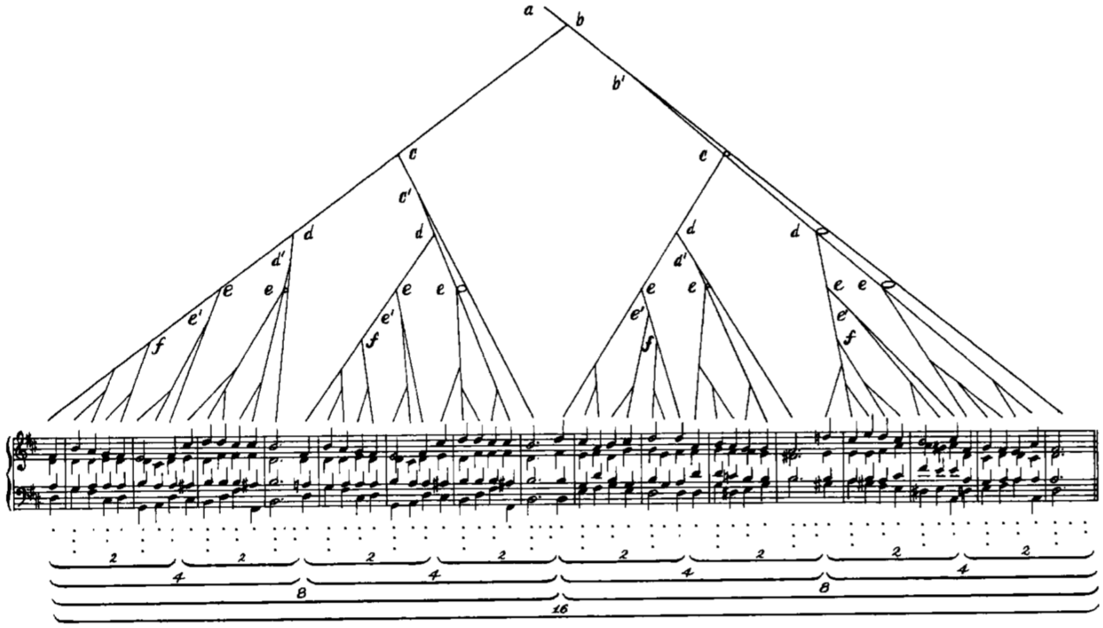


Figure 2.2: Time-span tree of *Oh Haupt voll Blut und Wunden* according to GTTM, with metrical and grouping structure depicted below the staff (taken from Lerdahl and Jackendoff, 1996).

These principles and rules give no direct explicit way of *generating* music. Rather, they allow the examination and evaluation of a potentially musical artefact. If it fits well into the stipulated framework it consequently is likely to be a piece of tonal music. Thus in a sense, this generative theory is akin to a probabilistic generative model which states how well the given data fit into the modelled distribution.

2.1.4 Deep Learning

If we go back to the Markov chain approach, it is natural to ask why we condition only on the direct predecessor and not on more items from the preceding sequence. But in practice, we will quickly encounter limits: We could condition on the two preceding items, $p(\mathbf{x}_i|\mathbf{x}_{i-1}, \mathbf{x}_{i-2})$. This is called a second-order Markov chain and needs a three-dimensional table, which has n^3 cells (n being the number of values an item could have). Or if we model, in general, the whole sequence as $p(\mathbf{x}_L|\mathbf{x}_1, \mathbf{x}_1, ..., \mathbf{x}_{L-1})$, the table will have n^L cells. Such a table is virtually impossible to ever be sufficiently filled, no matter the size of our dataset. This is an instance of the so-called *curse of dimensionality* (Bellman, 1961), which refers to the fact that the number of states a distribution has to cover grows exponentially with the size of dimensions (in our case the number of items in a sequence). Extracting the statistical properties of our dataset by simply counting occurrences, as it is usually done when learning Markov chains, will tell us that the exact sequences that are in the dataset are the only ones allowed.

The way out is not trying to create a giant lookup table but to use a learnable function approximator for p. The input to this function is the sequence up to the current item and the output is the probability distribution over the possible values for the next item. Such a function can be learned using the maximum-likelihood method. This simply means that the higher probability that the function assigns to the items in the dataset, given the context, the better.

This is often formulated as a loss that should be minimized, known as negative log-likelihood (NLL). It is the negative average of the log-probability of all items in the dataset:

$$\mathcal{L}_{NLL} = -\frac{1}{M} \sum_{m=1}^{m=M} \sum_{l=1}^{l=L_M} \frac{1}{L_M} \log p(\mathbf{x}_l^m | \mathbf{x}_1^m, \mathbf{x}_2^m, ..., \mathbf{x}_{l-1}^m)$$

Here M is the number of different pieces in the dataset and L_M is the number of items in each piece.

The principle of autoregressive modelling is a strategy for dealing with the curse of dimensionality as well since it allows us to factorize the unwieldy joint distribution of the sequence (which in its usual form cannot even be handled by neural networks) as many conditional distributions, each of which is more tractable:

$$p(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_L) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)...p(\mathbf{x}_L|\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{L-1}) \tag{2.1}$$

This is an application of the so-called *chain rule* of probability. In chapter 3, we will see how modelling autoregressive conditional distributions looks like in practice (e.g. in Figure 3.10).

In the last years, one class of function approximators has proven itself as particularly adept in overcoming the curse of dimensionality for many kinds of data: deep neural networks. They generally consist of multiple consecutive high-dimensional learnable linear transformations and differentiable nonlinearities. This allows them to be trained end-to-end via stochastic gradient descent on the available data (Schmidhuber, 2015; LeCun, Bengio, and Hinton, 2015).

There are many different types of deep neural networks, some of them I will discuss briefly in the next chapter. One particular type, the $transformer$ architecture (Vaswani et al., 2017) will be described in detail, as it will be heavily used in this work. Notable works after the deep learning revolution (as the sudden practical success of deep neural networks from the year 2011 on is sometimes called) include the recurrent models of Oore et al., 2018, convolutional (Huang, Cooijmans, et al., 2019) and transformer-based models (Huang, Vaswani, et al., 2018; Payne, 2019). For a more detailed overview of the different uses of deep learning for music generation, I refer to Briot, Hadjeres, and F.-D. Pachet, 2017.

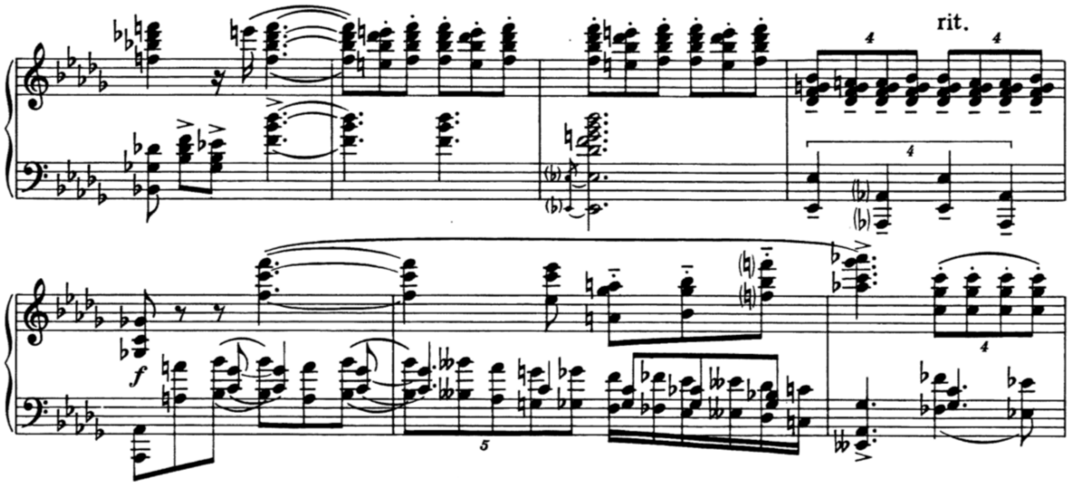


Figure 2.3: A few bars from Scriabin's Piano Sonata No.5 featuring polyphony, various articulations, rhythmic conflict, tempo variation and phrasing (Scriabin, 1907).

2.2 Symbolic Representations of Music

Up until now, we just assumed that we can represent music as a one-dimensional sequence of items. We know this is theoretically the case insofar as all possible sounds can be expressed as a sequence of displacements of a speaker membrane. For this work, however, we are concerned with the symbolic representations of music—what a composer typically would write as a musical score. This reduces the demands on the generative model since no realistic sound has to be produced. But it increases the impact of the representational format.

A musical score is a complex and high-dimensional object. It is not obvious how all the intricacies of a composition (Figure 2.3) can be squeezed into a single sequence of items. Although time is clearly the overarching general criterion for determining the position of an item in the sequence, many things can happen simultaneously.

A few strategies of representing polyphonic musical scores will be presented. The beginning of the four-part chorale *Oh Haupt voll Blut und Wunden* by J.S. Bach (Figure 2.4) serves as a common example.

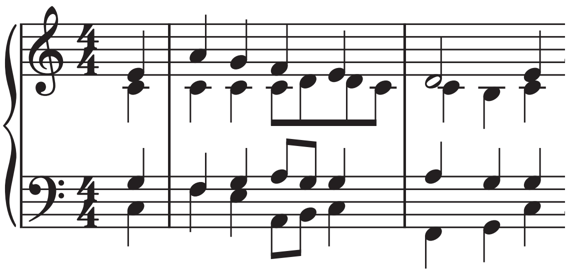


Figure 2.4: The beginning of the chorale *Oh Haupt voll Blut und Wunden*

2.2.1 Voice Grid Representation

A representational format that works well for structurally simple kinds of music is the voice grid (Eck and Schmidhuber, 2002; Hadjeres, F. Pachet, and Nielsen, 2017). The pitches are written as MIDI numbers (0-127) in a grid where each instrument or voice is assigned to a row and each time-step to a column. The length of a time-step should that of the shortest note in the dataset. For many Bach chorales, eighth note steps are sufficient. Here is the voice grid representation of *Oh Haupt voll Blut und Wunden* using eighth note steps:

```
64 64 69 69 67 67 65 65 64 64 62 62 62 62 64 64
60 60 60 60 60 60 60 62 62 60 60 60 59 59 60 60
55 55 53 53 55 55 57 55 55 55 57 57 55 55 55 55
48 48 53 53 52 52 45 47 48 48 41 41 43 43 48 48
```

To convert this two-dimensional representation into a one-dimensional sequence, one has to determine a fixed order. Usually, up-and-right ordering is chosen, i.e. we go from the lowest voice to highest, then the same for the next time step and so forth:

```
48 55 60 64 48 55 60 64 53 53 60 69 53 53 60 69 52 55 60 67 ...
```

One immediately apparent problem is that repeated notes cannot be distinguished from held notes. There are several solutions, maybe the most elegant being the introduction of one additional symbol (“__”). It simply means that the pitch form the previous time step is held.

```
64 __ 69 __ 67 __ 65 __ 64 __ 62 __ __ __ 64 __
60 __ 60 __ 60 __ 60 62 62 60 60 __ 59 __ 60 __
55 __ 53 __ 55 __ 57 55 55 __ 57 __ 55 __ 55 __
48 __ 53 __ 52 __ 45 47 48 __ 41 __ 43 __ 48 __
```

This, however, has the downside that the model may have to look further back to determine which notes are played at a given time step. The voice grid representation quickly comes to its limits if any real polyphony, rhythmical complexity or expressive nuance is required.

2.2.2 Piano Roll Representation

A related way of representing symbolic music is in a so-called piano roll format. The canvas is also a two-dimensional grid, but the rows do not represent instrument or voice but pitch. In this grid, the active pitches at each time step are highlighted. This is exactly the way performances were recorded and reproduced for old player-pianos. Instruments each have their own grids that are stacked together for the full representations, similar to the colour channels of an image. If one stays with the image analogy, the brightness of a pixel (i.e. the value at a grid position) determines the loudness of a certain pitch at a certain time-step. If the value is zero, the given pitch is not played at the current time-step (Figure 2.5).

Despite being quite intuitive, there are several drawbacks: There is no straightforward capacity for any information beyond instrument, timing, pitch and loudness (such as articulation). Again, it cannot be distinguished whether a certain note is held or repeated at a certain time-step. A way around this would be releasing the note for at least one time-step before hitting it again. This alludes to another problem: For sufficiently detailed timing information, the time resolution has to be quite high (say, about ten to a hundred steps per second). This is multiplied with the pitch resolution and the number of instruments, which leads to the grid being very sparse (meaning the overwhelming majority of cells will be empty since only a few notes are active at each point in time).

This is problematic for convolutional neural networks (that would otherwise be the natural choice for this kind of regular grid representation) because very large kernels would be required to capture patterns in the input data. However, in L.-C. Yang, Chou, and Y.-H. Yang, 2017, it is shown that using a Generative Adversarial Network (GAN) approach, simple melodies and chord progressions can be created this way.

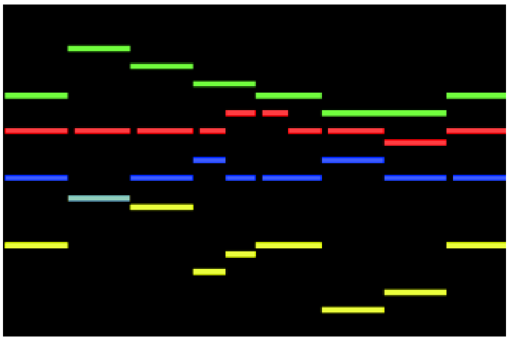


Figure 2.5: Piano roll representation of *Oh Haupt voll Blut und Wunden*

Even worse is autoregressive generation: It is of course possible to give the cells in the grid an ordering and generate them in a sequential fashion (e.g. put out the value for every instrument, do this for every pitch from low to high, and do this again for each time-step). But this leads to extremely long sequences, of which most elements will also be zero.

2.2.3 Note-based Representation

One natural and musically sensible approach could be representing music as a sequence of note objects. Each note could have the following features:

- instrument
- pitch
- loudness
- start time
- duration
- possibly articulation, special playing techniques, ...

These features are not independent. If we want to model a musical piece in this way, the start time of a note, for example, determines the pitches and note durations that should be considered. Similarly, which instrument plays a certain note has some influence on its loudness. This means we would have to regard all possible combinations of features for every note as distinct possibilities. In other words, the joint distribution over all note features has to be modelled:

$$p(\text{note}) = p(\text{instrument}, \text{pitch}, \text{loudness}, \text{start}, \text{duration})$$

For simple kinds of music, such as a Bach chorale, this combinatorial space has a size of about 50000 (for example 4 instruments or voices × 48 pitches × 16 relative start times × 16 durations). For more complex music, e.g. romantic or modern pieces with many instruments and intricate rhythms, the number of possibles notes can easily approach a billion. This cannot be efficiently modelled.

2.2.4 Command-based Representation

The same idea that led us to autoregressive modelling, namely writing the joint distribution as a product of conditional distributions, see 2.1, can be used here again:

$$p(\text{note}) = p(\text{instrument})\; p(\text{pitch}|\text{instrument})\; p(\text{loudness}|\text{instrument}, \text{pitch})\; ...$$

This means we can decompose a note into multiple sequential commands. The size of the distribution that we have to model for each item of the sequence is now only the sum, and not the product, of each feature's number of possible values.

One can think of such a command as changing an implicit state that determines the current instrument, pitch, loudness and time-step. After a command that sets a certain instrument, for example, every following command is applied to this instrument until another instrument change command comes along. And after setting a certain loudness, every note will have this loudness until it is changed with another command.

Additionally, it is helpful to draw some inspiration from the MIDI protocol. There, notes do not have a specified duration, but they are switched on and then switched off by a different command. The timing is controlled by a *wait* command that determines how much time passes until the next command will be executed.

As similar kind of music representation was introduced in Oore et al., 2018, and also used in Huang, Vaswani, et al., 2018, though in both cases only for solo piano, which means that no instrument or voice change commands were required.

index	command	current voice	current time	current pitches
356	voice:Soprano	Soprano	0	
164	note on:64	Soprano	0	64
357	voice:Alto	Alto	0	64
160	note on:60	Alto	0	60, 64
358	voice:Tenor	Tenor	0	60, 64
155	note on:55	Tenor	0	55, 60, 64
359	voice:Bass	Bass	0	55, 60, 64
148	note on:48	Bass	0	48, 55, 60, 64
99	wait:100	Bass	100	48, 55, 60, 64
356	voice:Soprano	Soprano	100	48, 55, 60, 64
292	note off:64	Soprano	100	48, 55, 60
169	note on:69	Soprano	100	48, 55, 60, 69
357	voice:Alto	Alto	100	48, 55, 60, 69
288	note off:60	Alto	100	48, 55, 69
160	note on:60	Alto	100	48, 55, 60, 69
358	voice:Tenor	Tenor	100	48, 60, 69
283	note off:55	Tenor	100	48, 53, 60, 69
153	note on:53	Tenor	100	48, 53, 60, 69
259	voice:Bass	Bass	100	48, 53, 60, 69
276	note off:48	Bass	100	53, 60, 69
153	note on:53	Bass	100	53, 53, 60, 69
99	wait:100	Bass	200	53, 53, 60, 69
356	voice:Soprano	Soprano	200	53, 53, 60, 69
297	note off:69	Soprano	200	53, 53, 60
167	note on:67	Soprano	200	53, 53, 60, 67
357	voice:Alto	Alto	200	53, 53, 60, 67
288	note off:60	Alto	200	53, 53, 67
160	note on:60	Alto	200	53, 53, 60, 67
358	voice:Tenor	Tenor	200	53, 53, 60, 67
281	note off:53	Tenor	200	53, 60, 67
155	note on:55	Tenor	200	53, 55, 60, 67
...				

Table 2.1: Beginning of *Haupt voll Blut und Wunden* in the command-based representational format. The *command* column is the human-readable equivalent of the *index* column. The three rightmost columns give the current state of context information.

The command-based representation is highly flexible. For example, additional types of commands could be added without many difficulties. It is also very concise, as there is no unnecessary or redundant information.

However, the "grammar", with which I mean the rules and principles that determine how a sequence of commands give rise to musical objects (such as note), is quite complex. Also, the splitting up of note objects into elementary commands leads to relatively long sequences. This puts high requirements on our model. Fortunately, there exist neural network architectures that are up to the task, as we will see in the next chapter.

To illustrate this method of representation, let us take a look at the beginning of *Oh Haupt voll Blut und Wunden* again (Table 2.1). We enumerate all commands that are used: 100 *wait* commands, signifying wait times of 10 ms to 1000 ms (command numbers 0-99), 128 *note on* commands (one for each pitch, command numbers 100-227), 128 *note off* commands (also for each pitch, 228-355), four *instrument/voice change* commands (356-359), 32 *loudness* commands (360-391). The loudness commands will in this case not be used since Bach did not usually write any dynamic markings in his chorales.

In principle, the first column contains all the necessary information. The three rightmost columns give the current state that is completely determined by the commands.

However, including this state information can be helpful to the model: The current voice, time position or harmony (which is given by the current pitches) is extremely important for the model at many steps of the sequence. Of course, the model can collect this information from previous commands. But giving it as explicit input features frees up capacities for higher-level modelling and stabilizes the generation process. This can be experimentally validated.

Only this command-based representation will be used for the rest of this work.

Maybe we can see development in the history of autonomous composing systems: A kind of atomization of the representational format, from whole bespoke phrases and very specific constructs (*Musikalisches Würfelspiel*, *Emmy*) to the smallest musical components imaginable (as we just have come to with the command-based representations). As a consequence, less and less external knowledge is encoded into the representations. Instead, the model has to learn all regularities on its own from the data.

Principally, there are two ways to do this: Either we have very large amounts of data, or the model makes a priori some assumptions about the data, explicitly or implicitly, that allow more efficient learning. In practice usually, both are required. The assumptions a model makes are also called *inductive biases* (cf. Benjamin, 2012). They should still be very general so that the same model can be used for broad classes of data. It is not obvious what exactly should be built into the model and what can be inferred from the data. Finding the right inductive biases for music can require some introspection on how we perceive and understand music. On the other hand, we might also be able to infer some things about our own cognition from the principles that work for the models.

With that, the historical development can also be seen as a shift from representational biases of the data (which limits the availability of data) to inductive biases of the model.

Chapter 3

Transformers-based autoregressive models

The command-based symbolic representational system of music is very powerful. In its presented form, or with straightforward modifications and extensions, it can arguably represent most conceivable forms of symbolic musical information. Of course, here we are ignoring indeterministic elements of composition such as improvisational components or free forms.

As already discussed, to use autoregressive modelling, we had to create a one-dimensional sequential representation. This one dimension is not innate to the data domain (as for example space dimensions for images or time for waveforms of sound would be), but an artificially constructed dimension. We might call it the *command dimension.* The underlying music is still on a conceptual level multidimensional (the dimensions being, for example, time, pitch, loudness and so on).

If we think about how human musicians read and understand a music score, there does not seem to be a strictly fixed order in which they parse the individual glyphs and incrementally build up the higher-level concepts in a linear way. Rather, whole notes (possibly with accidentals, articulation and dynamic markings), chords or even bars are consciously perceived as a single unit. These units are for their part read in roughly sequential, probably time-wise, order. Our neural network architecture should reflect these considerations.

3.1 The transformer architecture

For many straightforward tasks, the architectures to choose from are typically *fully connected*, *convolutional*, or *recurrent* neural networks. But all three of do not have all the desired features:

Fully connected networks only work with a fixed input size. Every input element has its own unique place, and there is no weight sharing at all. This means, if the input sequence is shifted by even a single item (which is done all the time for autoregressive modelling), it looks completely foreign to the network. Thus, they are very inefficient, parameter-intensive and prone to overfitting. They will tend to memorize the dataset and will not generalize to unknown data. One could say that this architecture has too little *inductive bias*, it does not take advantage of the specific structure of our data. Even very general assumptions about the data (such as, for example, that it is of sequential nature) can go a long way to creating much more suitable architectures.

Convolutional networks do make such assumptions. They share the weights of the kernels across the command dimension, so the network can handle shifts quite naturally. For the same reason, the input length is not necessarily fixed (although the pooling operations in the higher layer might have to be adapted). Small perturbations or inserted items (such as an additional loudness command), however, are a problem, since a unique kernel would be required for each different possible version or permutation of a specific higher-level unit. Because of that, convolutional networks are not suitable for our highly irregular and dynamic representation.

Recurrent networks, in particular *Long Short-Term Memory* (LSTM) networks (Hochreiter and Schmidhuber, 1997), have the ability to abstract away all these heterogeneities since they can dynamically control the flow of information from one step in the sequence to the next. They are, in fact, Turing-complete universal computers (Siegelmann and Sontag, 1992), so they can, in principle, handle any grammatical system. LSTMs have indeed been successful in generating music (Eck and Schmidhuber, 2002; Oore et al., 2018). In practice, however, the sequential nature of the input is baked very deeply into RNNs. This is problematic insofar as the command dimension, as discussed, is not innate to the data and therefore should be, if possible, abstracted away. In recurrent networks, the information of all previous steps is constantly compressed into a single state vector. This means that the memory from many items long in the past cannot be very strong.

A relatively new neural network architecture, commonly referred to as *transformer* and first presented in Vaswani et al., 2017, solves this memory problem and is able to capture complex, long-ranging and hierarchical relations in sequences. It is now used for many state-of-the-arts results, particularly in the field of natural language processing (Devlin et al., 2018; Raffel et al., 2019; Brown et al., 2020). In the next sections, I will introduce the building blocks of the transformer architecture and how they fit together.

3.1.1 Attention

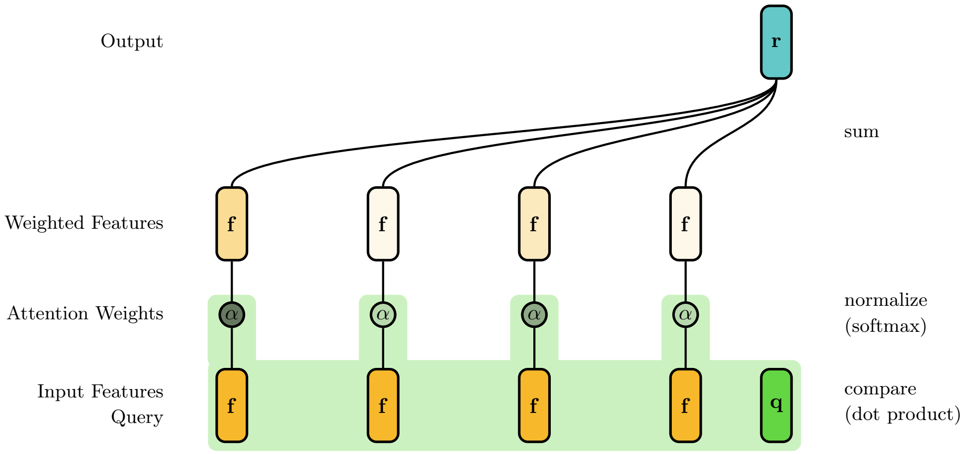


Figure 3.1: Attention mechanism with four input feature vectors and one query.

The concept of attention has been studied in psychology and neuroscience for several decades. At its most general, attention could be described as the dynamic and flexible allocation of limited computational resources. In recent years, related ideas have found their way into architectures of artificial neural networks, creating an important ongoing development in the field. A broad and up-to-date overview of attention in both biological and artificial systems can be found in Lindsay, 2020.

The basic functional principle that emerged as a useful attention mechanism for artificial neural networks is the following: We have a set (meaning in this context a collection without any specific order) of n feature vectors $\mathbf{f}_j \in \mathbb{R}^d$. Each of these vectors gets assigned to it a different scaling factor, or attention weight, α_j. The vectors are multiplied with their attention weights and then added together, resulting in a single output feature vector $\mathbf{r} \in \mathbb{R}^d$.

$$\mathbf{r} = \sum_j \alpha_j \mathbf{f}_j$$

The attention weights are usually normalized to have a total sum of one. This way they can be seen as a distribution over the feature vectors. It can be useful to visualize this distribution and see which feature vectors have been attended to.

A critical part is of course how the attention weights are dynamically calculated. This is done by having one query vector $\mathbf{q} \in \mathbb{R}^d$ that represents the properties that should be given attention to. The query vector is compared to each feature vector $\mathbf{f}_j$ by taking the dot product, which gives us a single scalar value for every comparison. These scalars are normalized using the softmax function, resulting in the attention weights. For better learning behaviour, the scalars are divided by $\sqrt{d}$ where d is the number of elements in each feature vector (a diagram is shown in Figure 3.1).

$$\tilde{\alpha}_j = \frac{1}{\sqrt{d}} \sum_i f_{ij} q_i$$

$$\alpha_j = \text{softmax}(\tilde{\boldsymbol{\alpha}}) = \frac{\exp(\tilde{\alpha}_j)}{\sum_j \exp(\tilde{\alpha}_j)}$$

Where the query vector comes from depends on the context in which the attention mechanism is used. But crucially, it computed based on the current input to the system, which makes the whole operation dynamic. Several additions can be made to this basic mechanism to make it more powerful.

3.1.2 Key-Value Attention

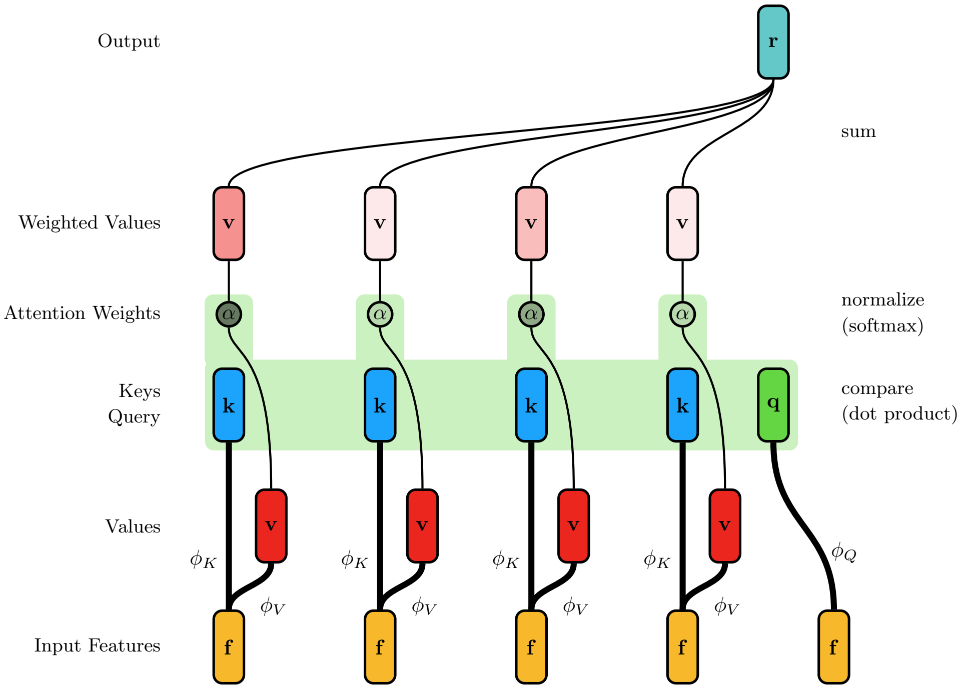


Figure 3.2: Key-value attention mechanism, where the input items are shaped into keys and values by learnt transformations ϕ_K and ϕ_V.

Each feature vector $\mathbf{f}$ contains both the information that is necessary to determine whether some attention weight should be assigned to it and the information that will be passed on to the next processing step if it has indeed been attended to. This means the query has to be constructed in such a way that it ignores some parts of the feature, namely the ones not determining the attention allocation. These parts of the feature vectors that are ignored by the query contain the most relevant new information because most other information is already contained in the query. Therefore, at least some amount of computation is always wasted during the calculation of the attention weights.

To avoid this, we can split each feature vector into two components: a key vector and a value vector. The key vectors are only used to calculate the attention weights by comparing them to the query. The value vectors are scaled by the attention weights and added together, resulting in the output feature vector. The split of the feature vector into key and value vectors is done using learnt transformations ϕ_K and ϕ_V (usually a single linear transformation each). These transformations are shared across all sequence steps. The pre-softmax attention weights are thus calculated as

$$\tilde{\alpha}_j = \frac{1}{\sqrt{d}} \sum_i \phi_K(\mathbf{f}_j)_i \phi_Q(\mathbf{f}_q)_i$$

($\mathbf{f}_q$ is the feature vector from which the query is calculated) and the output vector as

$$\mathbf{r} = \sum_j \alpha_j \phi_V(\mathbf{f}_j).$$

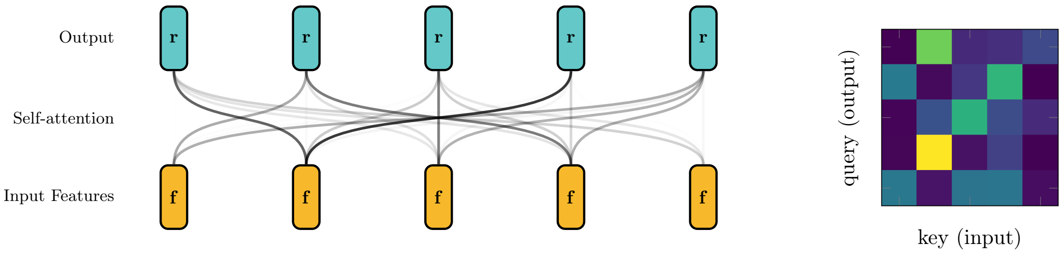
See Figure 3.2 for a depiction. This procedure, as the terms key, value and query suggest, can be interpreted as a smoothed out and differentiable dictionary lookup.

3.1.3 Multi-Head Attention

In its current form, the attention mechanism has a single query that gets compared to all values, resulting in a single set of attention weights. The query can in principle be quite sophisticated, depending on the size d of the query and value vectors.

In practice, however, it turns out to be a more efficient use of the computational resources to split all key, value and query vectors into multiple smaller parts and compute a different set of attention weights as well as a different output vector for each section in parallel. The output vectors are then again composed into one single larger vector. This is called multi-head attention and allows a more fine-grained, modular and specific aggregation of information from different sequence steps. With the presence of multiple heads per layer, the complexity of the information flow between items can be greatly increased. This usually is an advantage but, of course, makes it harder to comprehend and interpret how exactly a result was brought about.

3.1.4 Self-Attention



(a) Self-attention connections

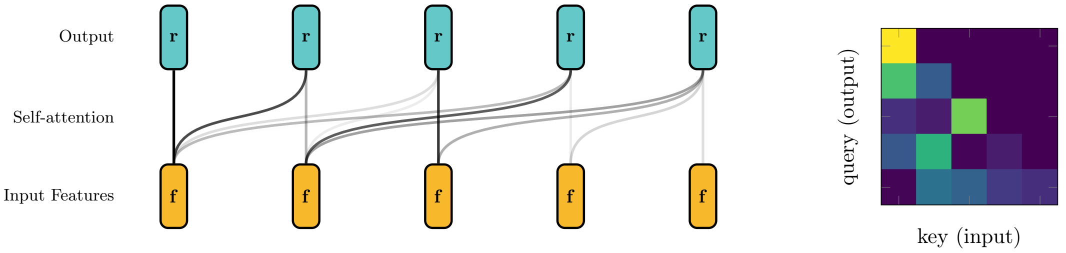
(b) Attention matrix of the self-attention mechanism shown above

Figure 3.3: Simplified depiction of a self-attention mechanism, showing only the weighted connection and the corresponding attention matrix.

As mentioned above, these kinds of attention mechanism do not specify where the input features for either the key-value pairs or the query come from. If we have a set of input feature vectors **f** for the key-value pairs, any one of the same vectors could also be chosen as the basis of a query. If the attention mechanism is executed in parallel for every item **f** as a basis for the key-value pairs as well as for distinct queries, this is called self-attention. The output of a self-attention mechanism is a set of output vectors similar to the inputs. During the calculation, any item of the input set has the potential to interact with any other item (and itself). So self-attention is a way to dynamically share, reshuffle and aggregate information between the items of the input set.

In the self-attention setting, every item is linked as query to all items, itself included, as values by a weighted connection (Figure 3.3a). A clear way to visualize these attention weights is plotting them as an *attention matrix* (Figure 3.3b), where the one dimension specifies the query and the other dimension the key.

3.1.5 Attention Weight Masking



(a) Self-attention connections

(b) Attention matrix of the self-attention mechanism shown above

Figure 3.4: Self-attention mechanism and attention matrix with only causal connections.

It can be necessary to forbid some attention connections, i.e. limit the access of certain queries to some key-value pairs. The most common example is autoregressive modelling: If the input to the self-attention is a sequence and the goal is to predict the next item at each step, we cannot grant the query access to keys and values on its right. These are exactly the values that should be inferred from the past items, so being able to simply copy them from the input would defeat the whole purpose. In this case, a triangular mask has to be laid over the attention matrix, setting all attention weights corresponding to items on the right of the current query to zero. This kind of masking is also called *causal* because it preserves the causal principle that items from the future cannot influence the past.

Depending on the use case, it can also be useful, and sometimes necessary, to limit the access to items that are too long in the past, or only allow access to items in a certain region. In practice, this masking is done by setting the logits $\tilde{\alpha}$ to $-\infty$, which preserves the normalization property of the subsequent softmax operation.

3.1.6 Transformer Architecture

This self-attention module is the at the heart of the transformer architecture (the original publication has the title “Attention is all you need”, Vaswani et al., 2017). A transformer usually consists of alternating self-attention modules and fully connected neural network modules (or multi-layer perceptrons, MLPs). The latter are applied separately to the feature vectors but their weights are shared across the items.

Additional features of the transformer architecture are residual connections and layer normalizations for both the self-attention and the MLP-blocks. *Residual connections* refer to the fact that the output does replace the input of a layer when it is passed on to the next layer, but that

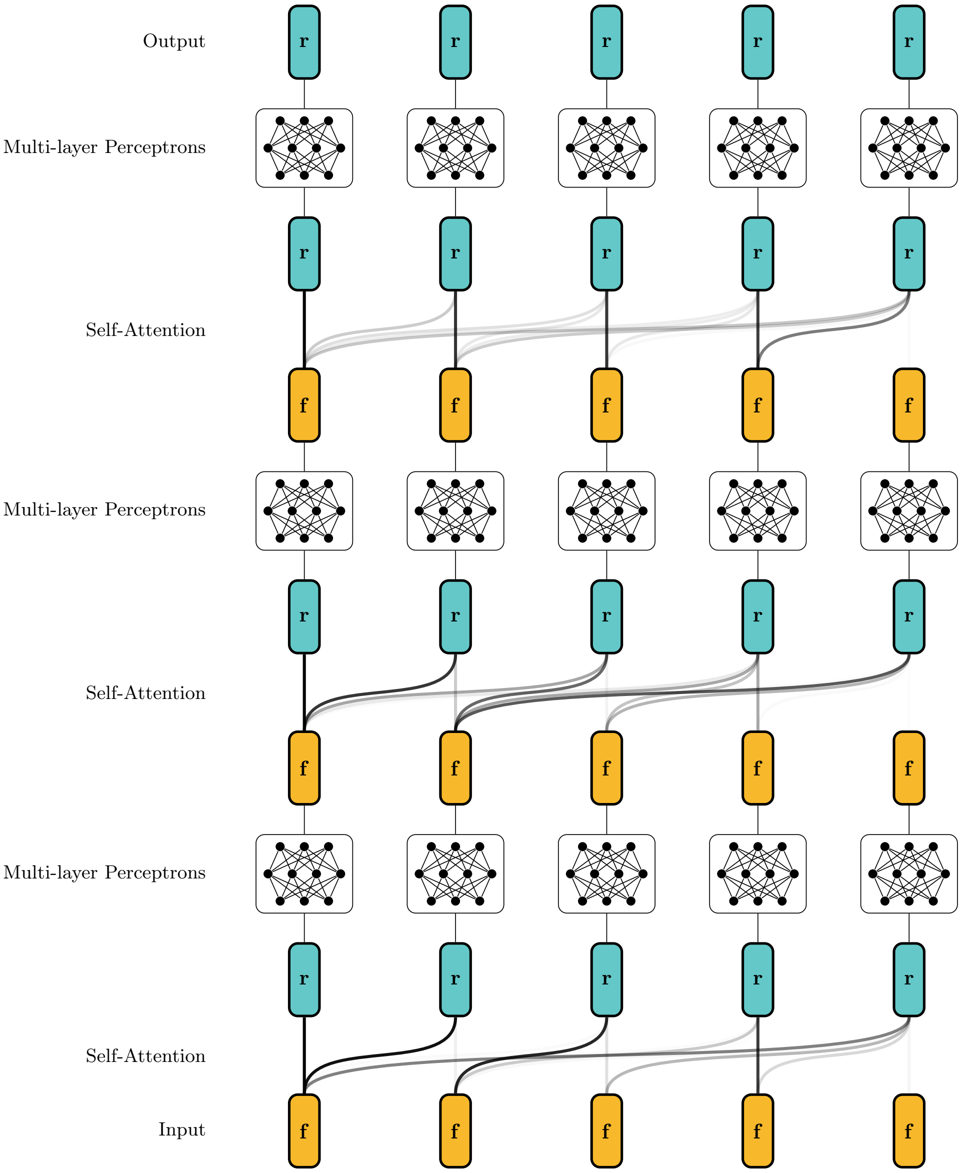


Figure 3.5: Transformer architecture, showing three transformer layers, each consisting a causal self-attention module and two-layer MLPs.

input and output are added together and then passed on. This allows unobstructed information flow from the input to higher layers in the forward pass as well as more stable gradients for the lower layers in the backward pass.

As mentioned above, the self-attention mechanism allows the redistribution and dynamic aggregation of information across columns. The MLPs then process this newly collected information. MLPs are highly flexible (they are, in fact, universal function approximators, see Cybenko, 1989). The advantage of having them shared over columns is that they have to work in many different contexts at the same time and are in this manner forced to learn very general principles (in a way similar to the kernels of convolutional neural networks).

The consecutive application of several self-attention mechanisms results in the information flow following tree-like patterns. If we pick one item of the top and trace back the highly weighted connection down to the input, we will get a tree that branches out at every layer. In the combined attention weights of all layers, every possible tree is contained. This can be seen in Figure 3.6. Each tree, even every branch of each tree, has a weight assigned to it. A transformer learns in a way to construct a suitable soft tree structure for any given context.

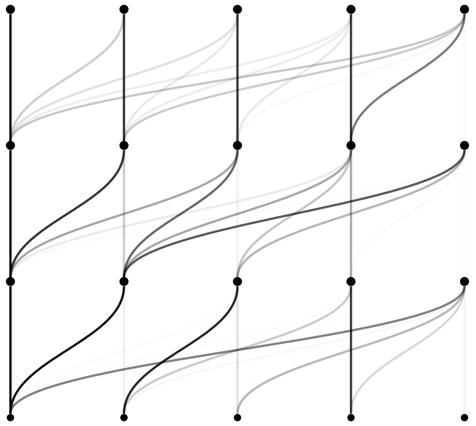


Figure 3.6: Causal self-attention connection of a transformer. It can be interpreted as superimposed weighted tree structures.

3.1.7 Positional Encoding

The attention mechanism takes as input an unordered set of items, so it has no intrinsic capacity for retaining any ordering of the input items. The simplest way to not lose this information is explicitly adding the position of each item as a feature to the inputs. It then has the same status as all other dimensional information of the data, such as in our case pitch, timing, loudness and so on.

The best way of adding the ordering information is most likely not a single number enumerating all items—neural networks are better suited for high-dimensional representations. A common technique is to have an encoding vector $\mathbf{p}_j \in \mathbb{R}^d$ represent the position j of the item. The feature vector $\mathbf{f}_j$ continues to represent the "content" of the item, meaning all non-positional information such as pitch, voice and loudness or in the upper layers complex aggregates thereof. Then $\mathbf{p}_j$ and $\mathbf{f}_j$ are added element-wise, resulting in the effective input vectors for the attention mechanism. In a self-attention setting, this is done before the split into key and query vectors.

If we assume that the transformations ϕ_K and ϕ_Q are linear (which is the usual way of implementing key-value attention), we can decompose the calculation of the attention weights as follows (j being the key-position and k the query position):

$$\begin{aligned}
\tilde{\alpha}_{jk} &= \frac{1}{\sqrt{d}} \sum_i \phi_K(\mathbf{f}_j + \mathbf{p}_j)_i \; \phi_Q(\mathbf{f}_k + \mathbf{p}_k)_i \\
&= \frac{1}{\sqrt{d}} \sum_i \Big(\phi_K(\mathbf{f}_j)_i + \phi_K(\mathbf{p}_j)_i\Big) \Big(\phi_Q(\mathbf{f}_k)_i + \phi_Q(\mathbf{p}_k)_i\Big) \\
&= \frac{1}{\sqrt{d}} \sum_i \quad \Big(\phi_K(\mathbf{f}_j)\phi_Q(\mathbf{f}_k)\Big)_i + \Big(\phi_K(\mathbf{f}_j)\phi_Q(\mathbf{p}_k)\Big)_i + \\
&\qquad\qquad \Big(\phi_K(\mathbf{p}_j)\phi_Q(\mathbf{f}_k)\Big)_i + \Big(\phi_K(\mathbf{p}_j)\phi_Q(\mathbf{p}_k)\Big)_i
\end{aligned} \tag{3.1}$$

The emerging four terms all have distinct properties and functions in calculating the attention weights:

- The first term describes the calculation based only on the feature vectors, without any regard to their position (content-based content selection).

- The second term distributes attention not based on the content of a query, but it searches for certain contents at specific positions in the sequence (position-based content selection). An example could be “at step 19, select all *note on* commands with pitch 61”.
- The third term selects certain sequence positions based on the content of the query (content-based position selection), such as “if the next command might be a voice change, select the item at step 34”.
- The fourth term selects only based on the position of the items (position-based position selection or static positional bias), for example “from step 75 on, select items from the beginning of the sequence”.

We can see immediately that the second and third terms are only of limited use in our case. Addressing an absolute position can only make sense, if at all, for the first few steps of the sequence. After that, the absolute position of an item simply has no meaning in our symbolic music representations (the same is true in natural language for sequences spanning several sentences).

One way of representing the item position is one-hot encoding. Here a different element of a vector is set to one for each input item. This, however, does not generalize to input sequences of different lengths.

Another slightly more elaborate way, also introduced in Vaswani et al., 2017, uses a fixed positional encoding based on the complex exponential function, or sine and cosine functions:

$$PE_{jm} = \exp\left(i\frac{j\pi}{2}\exp(\frac{-sm}{2d})\right) \tag{3.2}$$

Here, i is the imaginary unit, s is a scaling factor (e.g. 10.0) and d is the number of features of the resulting positional encoding, assuming that the real and imaginary parts of the complex numbers are counted as separate values. The index j gives the position in the sequence and m the feature. This encoding has a distinct representation of each position in the sequence that can be added to the corresponding feature vectors (Figure 3.7).

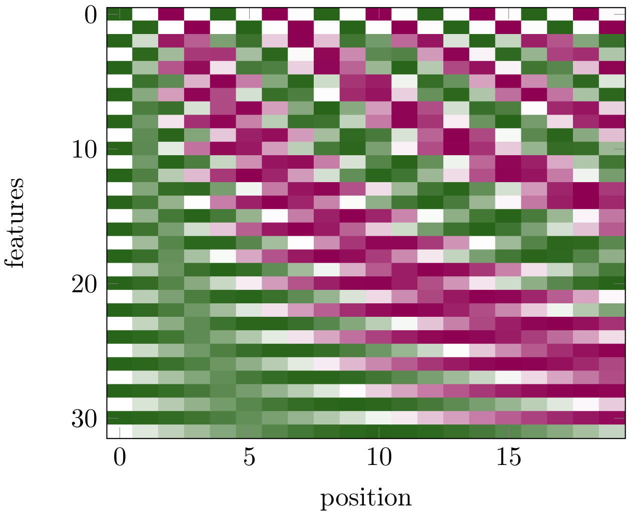


Figure 3.7: Sinusoidal positional encoding according to 3.2

This particular encoding has a major benefit: Often the relative position of one item to another is more important than the absolute position in the sequence. Here, the same transformation can be used to get from any positional encoding to one at a certain distance to it: Simply a multiplication by a constant complex number (in the case of complex features) or a rotational matrix (in the case of real-valued feature pairs) shifts the encoding by a certain number of steps, no matter the absolute position. These are linear transformations and hence can easily be performed by the learnt functions ϕ_K and ϕ_Q.

3.1.8 Relative Attention

It is possible to modify the attention mechanism in a way that is even better suited for domains where the relative position of items is paramount (which is certainly the case for our command-based musical representations).

The idea is that we use a positional encoding $\mathbf{p}$ not to represent a certain absolute position j, but to represent the relative position $k - j$, where j is the position of the value vector and k is the position of the query vector. This approach was introduced in Shaw, Uszkoreit, and Vaswani, 2018 and refined in Huang, Vaswani, et al., 2018; Dai et al., 2019. In this case, equation 3.1 becomes:

$$\begin{aligned}\tilde{\alpha}_{jk} = \frac{1}{\sqrt{d}} \sum_i \quad & \Big(\phi_K(\mathbf{f}_j)\phi_Q(\mathbf{f}_k)\Big)_i + \Big(\phi_K(\mathbf{f}_j)\phi_Q(\mathbf{p}_{j-k})\Big)_i + \\ & \Big(\phi_K(\mathbf{p}_{k-j})\phi_Q(\mathbf{f}_k)\Big)_i + \Big(\phi_K(\mathbf{p}_{k-j})\phi_Q(\mathbf{p}_{j-k})\Big)_i .\end{aligned}$$

The first term is exactly the same. The other three terms are changed:

- The second term (position-dependent content selection) now searches for certain keys at specific positions relative to the query. For example "look for an instrument change at the previous sequence step" or "select Bass notes in the vicinity of the current step".
- The third term (content-based position selection) lets the query select a certain position relative to itself. Examples could be "if the next command might be a voice change, select the previous item" or "if this particular situation calls for, select item the five steps ago".
- The fourth term (position-based position selection) now selects based on only relative position, so it could be called static relative positional bias. Examples are "Only ever select the previous item" or "decrease the attention weight with increasing distance to the current step".

For some not specified reason, in Dai et al., 2019, the mechanism is further simplified, losing the position dependency of the second term and being left only with a static content bias. The models used in this work will have relative attention as stated in 3.1.8. How the construction of the attention matrices from the four terms looks like in practice can be seen in Figure 3.14.

The positional encoding p can be the same as described in (3.2). It does not, however, bring the same benefits as for the absolute positional encoding, since we do no longer have the need of the pseudo-relative attention it enables.

The idea of relative attention can be extended beyond the relative *position* of the items. Natural extensions for the domain of music could be, for example, calculating attention weights based on the relative pitch of two items, or comparing their *time* positions instead of their *sequence* positions. One could even incorporate the command-based principles into the attention mechanisms by automatically increasing the attention weights on those previous items that directly affect the current position (for example the last voice and loudness change commands, as well as the last wait command or the last *note on* and *note off* commands in the current active voice). I tried out several such ideas in preliminary experiments but found that they greatly complicate the model while decreasing computational efficiency and generality, as well as having little to no influence on the performance. In practice, it turns out that extending the input features with relevant state information, as described in section 2.2.4, is both easier and more effective.

3.2 Generative Methods

3.2.1 Autoregressive Generation

What we aim for with training our transformer architecture on the dataset could be called a music language model (MLM), following the terminology of natural language models (LM) such as presented in Radford et al., 2019 and Brown et al., 2020. As described, it models the distribution over the current item of a sequence conditioned on the previous ones $p(\mathbf{x}_i|\mathbf{x}_0, ..., \mathbf{x}_{i-1})$.

Such a model can easily be used for the autoregressive generation of sequences by sampling from the output distributions one item at a time (Figure 3.8).

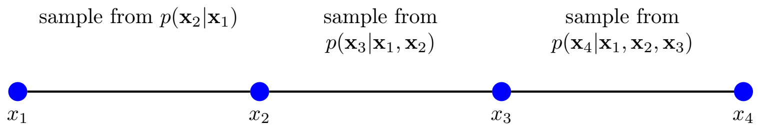

sample from $p(\mathbf{x}_2|\mathbf{x}_1)$
sample from $p(\mathbf{x}_3|\mathbf{x}_1, \mathbf{x}_2)$
sample from $p(\mathbf{x}_4|\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$
x_1
x_2
x_3
x_4

Figure 3.8: Autoregressive generation of a sequence.

The outputs of our model for each step are logits a_i that are then fed into a softmax operation to form a normalized probability distribution.

$$\text{softmax}(\mathbf{a}) = \frac{\exp(a_i/\tau)}{\exp(\sum_j a_j/\tau)}$$

Here, we introduced a new parameter τ, often referred to as *temperature*, that affects the entropy, or the amount of randomness, of the resulting distribution. Intuitively, it does that because the exponential function increases the relative difference between large values and decreases it between small values. A temperature approaching 0 will result in a near-deterministic distribution with the only possible choice being the one with the largest logit. For a high-temperature value, the distribution gets close to uniform. During training, the temperature is set to 1. This usually also works best for generating samples with an MLM that shows good performance (a low negative log-likelihood) on the validation set.

Samples generated using low temperatures will have a high likelihood (according to the model's own assessment), which is per se desirable. The problem is that those samples have a tendency to fall into repetitive, "safe" and uninteresting patterns. Also, the variability between different samples is reduced. In the extreme, the same sequence will be generated every time.

On the other hand, a high temperature increases the probability of obvious mistakes. It can also happen that the sequence drifts into regions that are so different from the training set, and as a result so foreign to the model, that no sensible prediction of the next item can be made. Then there is little chance of recovery.

Nevertheless, it can be interesting to push the temperature upwards to get more unexpected and surprising results. Low temperatures might be needed if there is not enough training data available, resulting in poor test performance. Decreasing the temperature can lead to more coherent samples. It is roughly equivalent to overfitting on the training set, which means that the generated sequences will tend to copy familiar sections.

3.2.2 Beam Search

Generating a sequence can also be seen as a search problem. Namely, of all possible sequences, we want to find the one with the highest total likelihood

$$p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)...p(\mathbf{x}_L|\mathbf{x}_1, ..., \mathbf{x}_{L-1})L.$$

We multiply by the number of sequence steps L, else short sequences would have a clear undesirable advantage.

Autoregressive generation suffers from its greediness. It does not account for the fact that it is possible that selecting a seemingly unsuitable item for the current step can potentially lead to a situation in the future that more than compensates for the current low likelihood. Also, if by chance a bad item is sampled at any point, there is no way of going back and fixing the mistake.

An exhaustive tree search is computationally infeasible since every possible sequence has to be checked. One practicable compromise is stochastic beam search (cf. Kool, Van Hoof, and Welling, 2019). We start by choosing the beam width B and sample B times (without replacement) from $p(\mathbf{x}_1)$, starting B distinct sequences. For the next item in each sequence, we again sample B times, leaving us now with a total of B^2 sequences. From these, we select the B sequences with the highest total likelihood and discard the rest. This procedure is repeated for every generation step (see Figure 3.9).

It is important to increase the temperature when sampling for the stochastic beam search. Else, the results are again quasi-deterministic, since only the best choices are retained.

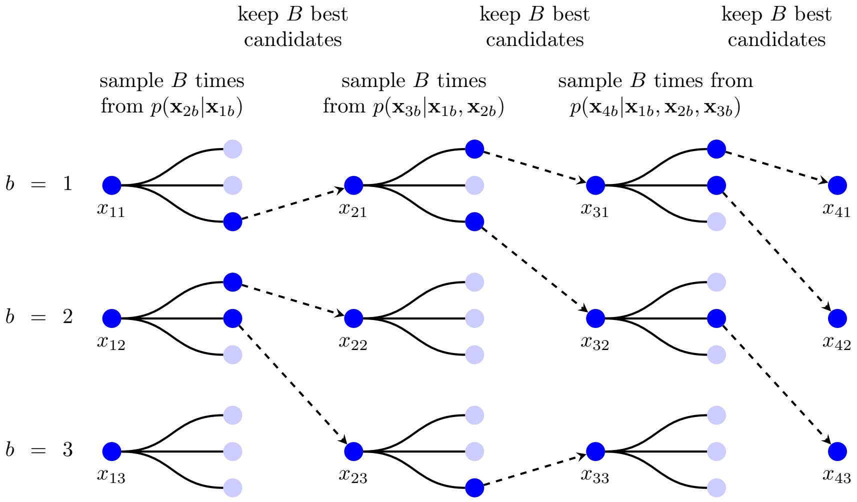

keep B best candidates
keep B best candidates
keep B best candidates
sample B times from $p(\mathbf{x}_{2b}|\mathbf{x}_{1b})$
sample B times from $p(\mathbf{x}_{3b}|\mathbf{x}_{1b}, \mathbf{x}_{2b})$
sample B times from $p(\mathbf{x}_{4b}|\mathbf{x}_{1b}, \mathbf{x}_{2b}, \mathbf{x}_{3b})$
$b = 1$
$b = 2$
$b = 3$
x_{11}
x_{21}
x_{31}
x_{41}
x_{12}
x_{22}
x_{32}
x_{42}
x_{13}
x_{23}
x_{33}
x_{43}

Figure 3.9: Generation of a sequence using stochastic beam search with $B = 3$.

3.3 Bach Chorales Experiment

3.3.1 Dataset

The most important dataset used for this project is the collection of all 376 chorales in four-part harmony written by Johann Sebastian Bach. They are split into a training set (338 chorales) and an evaluation set (38 chorales). They are represented in the command-based representational format, exactly as shown in 2.1. The *command* and the *current voice* features are represented as one-hot vectors, the *current pitches* in a multi-hot vector and the *current time* as a positional encoding similar to the one presented in 3.1. No loudness commands are used since Bach did not write any dynamic markings in the chorales. All chorales are set to have the same tempo. The original key is retained, though during training the piece can be randomly transposed up or down by a minor third or less.

3.3.2 Model Architecture and Evaluation

An autoregressive architecture, consisting of 8 causal transformer layers, is used. The size d of the feature vectors is 128. Learnt linear transformations are used to match the number of input features of the data with d, and to create the right number of output features (the number of available commands). The MLPs in the transformer have a single hidden layer with a size of

512. The attention mechanism has only one head since this makes the model easier to analyse and additional heads did not improve the performance for this dataset. An ADAM optimizer (Kingma and Ba, 2014) is used with a batch size of 32. A learning rate schedule with a warm-up phase (1000 steps) and cosine annealing (100000 steps) is applied. The maximal learning rate is 0.0003. The sequence length, the number of items the attention mechanism can process, is 256. As regularization, in addition to the 10% dropout typically used for transformers, input dropout is used. This means that during training, each input item (all features representing one step of the sequence) has a 20% chance of being set to zero. This reduces the risk of overfitting and forces the model to make robust inferences that rely on more than just very specific previous items.

To examine the effect of the relative attention mechanism, several models were trained. The first model uses the complete relative attention as explained in 3.1.8. Models 2 to 4 are each trained without one of the terms of the relative attention: model 2 without position-based content selection, model 3 without content-based position selection and model 4 without position-based position selection. Model 5 uses no relative attention at all and relies only on the positional encoding added to the input items. Model 6 again uses full relative attention but does not get the additional contextual input features (*current voice*, *current pitches* and *current time*). These additional features are the most significant change to the architecture of the music transformer used in Huang, Vaswani, et al., 2018. Their work is thus comparable to model six. The evaluation results can be found in Table 3.1.

The validation loss is the average negative log-likelihood that the model ascribes to the unfamiliar validation set. A small value means that the model, which has modelled the probability distributions over all items one step in the future, assigns high probability to the actually occurring items in the validation set. Validation accuracy is the average percentage with which the model predicts the next item correctly. This can be broken down further since different kinds of items tend to have different uncertainty. For example, in the context of Bach chorales, a trained model can predict *note off* commands with 100% accuracy. This is because if there is a voice change command and there is a note active in the new voice, the only possible command is to end this note. Of course, if one voice could play multiple notes at once, or if the structure were more polyphonic, this would no longer be true. It can be seen that for Bach chorales, in general, the *note on* commands are hardest to predict. This is not surprising, since they determine the pitch, and with that melody and harmony of the piece.

In Figure 3.10, the modelled distributions for a sequence are shown. We see that the entropy (the degree of uncertainty) of the distributions greatly depends on both the grammatical and the musical context. Some distributions are completely deterministic, such as the ones for the first and last shown step. For the second step, which is the first *note on* command, the distribution has high entropy, since nothing is yet known about the piece. The more notes of the first harmony are determined, the fewer options are there to choose from for the remaining notes.

The performance of the first four models, which all employ some form of relative attention, is very similar. Perhaps surprisingly, the ablated versions (particularly model 2) perform slightly better than model 1, which uses the full relative attention with all terms. This could be due to the fact that any one of the relative attention terms can be functionally replaced by the other terms, which then each have an increased capacity since the same parameter count is used to model three instead of four terms. The difference in performance, however, is so small that it is most likely negligible. Models five and six show significantly worse results, which suggests that both relative attention and the additional contextual input features are indeed helpful.

We can also look at the negative log-likelihood that the models ascribe to the samples they have generated themselves (Figure 3.11a). The performance is for the most part correlated with the validation loss. But interestingly, the absence of relative attention in model five leads to a much more significant deterioration of the generated sequences than the validation loss would suggest. The increased local focus of the relative attention mechanism on the current position seems to stabilize the model when it is freely generating.

Model	NLL	accuracy	note on	wait	voice
1 *full relative attention*	0.172	94.0 %	75.0 %	82.1 %	95.3 %
2 *no position-based content selection*	**0.168**	**94.1 %**	**75.6 %**	**83.8** %	**95.4 %**
3 *no content-based position selection*	0.171	94.0 %	74.8 %	82.9 %	**95.4 %**
4 *no position-based position selection*	0.170	**94.1 %**	75.4 %	82.6 %	**95.4 %**
5 *no relative attention*	0.189	93.4 %	71.6 %	80.5 %	95.3 %
6 *no additional input features*	0.200	92.9 %	73.9 %	81.1 %	95.1 %

Table 3.1: Validation performance of the models trained on the Bach chorales dataset.

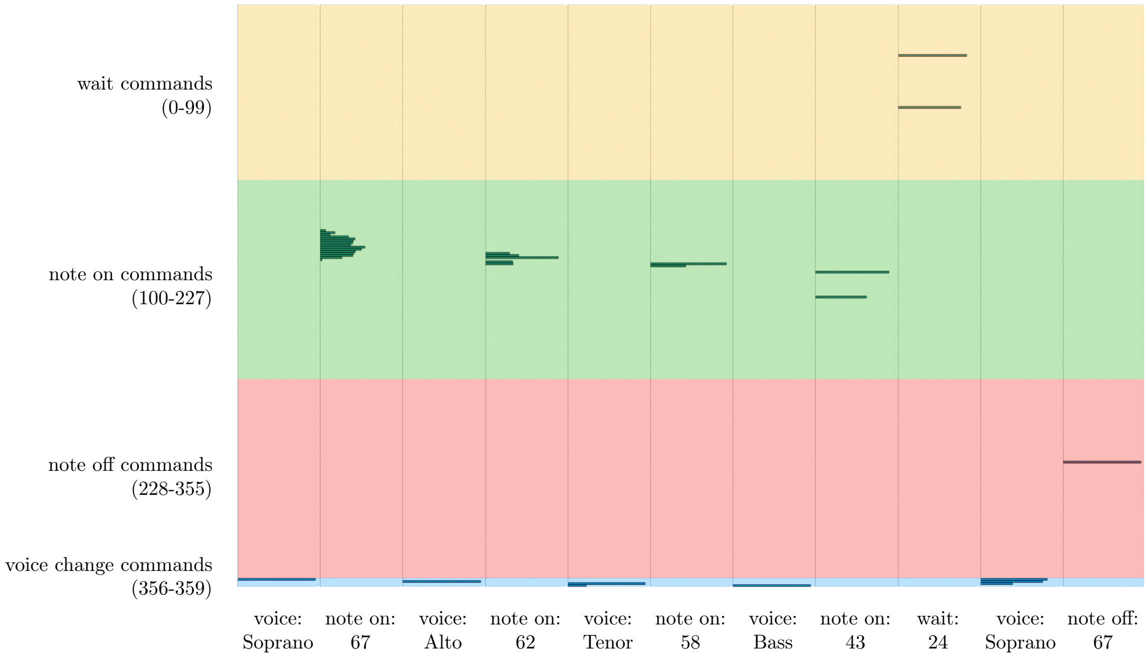


Figure 3.10: The probability distributions given by the trained model for multiple steps of a sequence. In the vertical direction, all possible commands are lined up. The individual columns show the distributions for each step, with the correct command stated at the bottom.

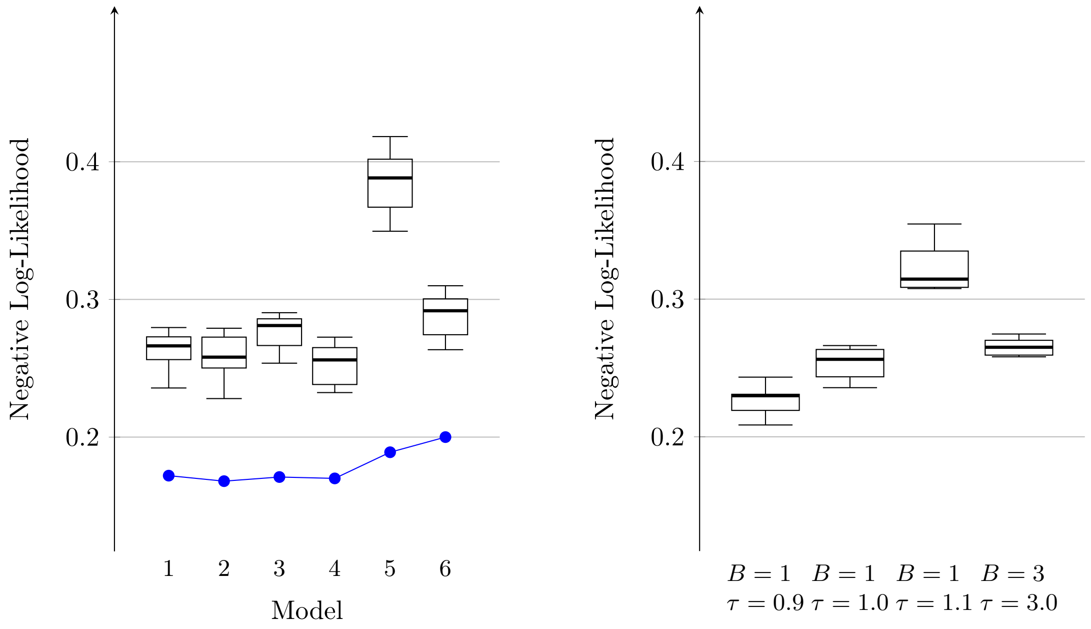
3.3.3 Discussion

From Figure 3.11b, we can see that sequences that are generated using beam search with $B = 3$ and $\tau = 3.0$ have a similar NLL to the ones generated with the natural autoregressive setting ($B = 1$, $\tau = 1.0$). Empirically, however, the beam search samples sound more natural and have fewer obvious mistakes. Chorales generated from scratch by model 1 using beam search are shown in Figures 3.12 and 3.13.

The model mostly adheres to the generally accepted rules of four-part harmony, such as the prohibition of parallel fifths or octaves, the avoidance of crossings with the outer voices and doubling of the third, or singable lines. There are some mistakes, such as the parallel fifths in the second half of bar 12 and the poor voice leading in bar 15 of the first piece, or bar 11 in the second piece. There seems to be a tendency to multiple mistakes at the same spot. A possible explanation for this could be that a single bad item derails the model and in doing so diminishes the quality of the next few items.

The model holds the key it started with rather well, although the second piece suddenly modulates to A flat major at the very end. A general problem is holding onto the metric structure. Because of the flexible dynamic representational format, the model can add or omit a single beat without perturbing the structure too much. For the listener, however, it can be very noticeable. This could be corrected by adding items that, for example, represent bar lines or fermata symbols. But to keep the representation as simple and general as possible, not such items were used.

In the two examples, there is no obvious metric shift. But the phrasing could be more distinct and the section divided more clearly.



(a) Boxplot of the NLL of the generated samples (no beam search, $\tau = 1$) from the different models (black) in comparison with the validation loss (blue). For each model, only the better half of all generated samples is evaluated.

(b) Samples generated with model 1 with different beam widths B and temperatures τ. We see that $B = 3; \tau = 3.0$ is effectively similar default autoregressive generation ($B = 1; \tau = 1.0$).

Figure 3.11: NLL of generated samples.

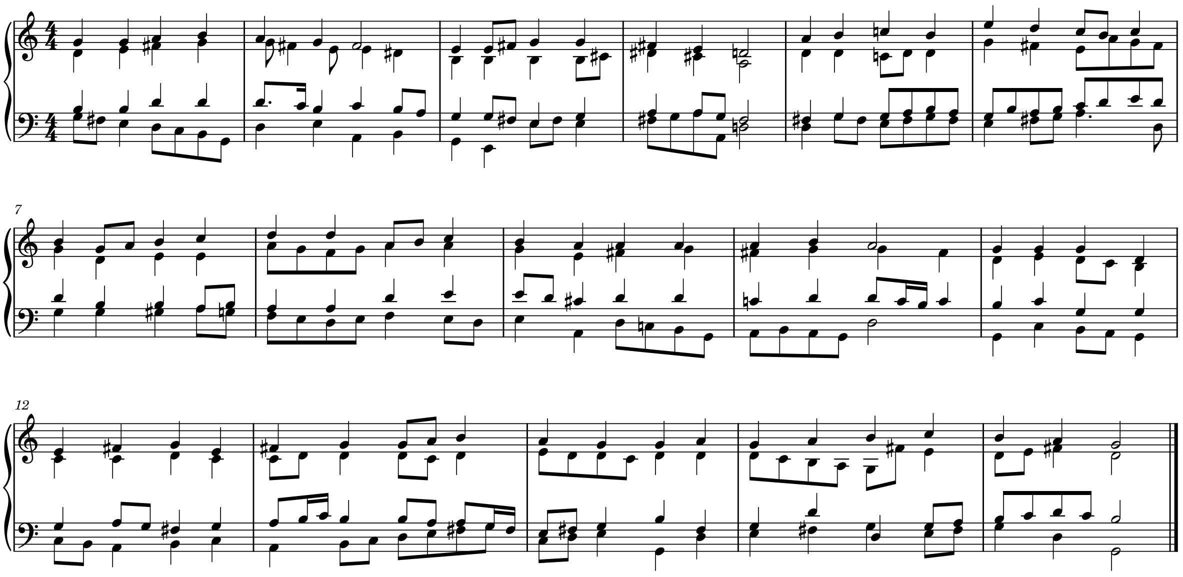


Figure 3.12: First chorale generated by model 1 using beam search with $B = 3$ and $\tau = 3.0$.

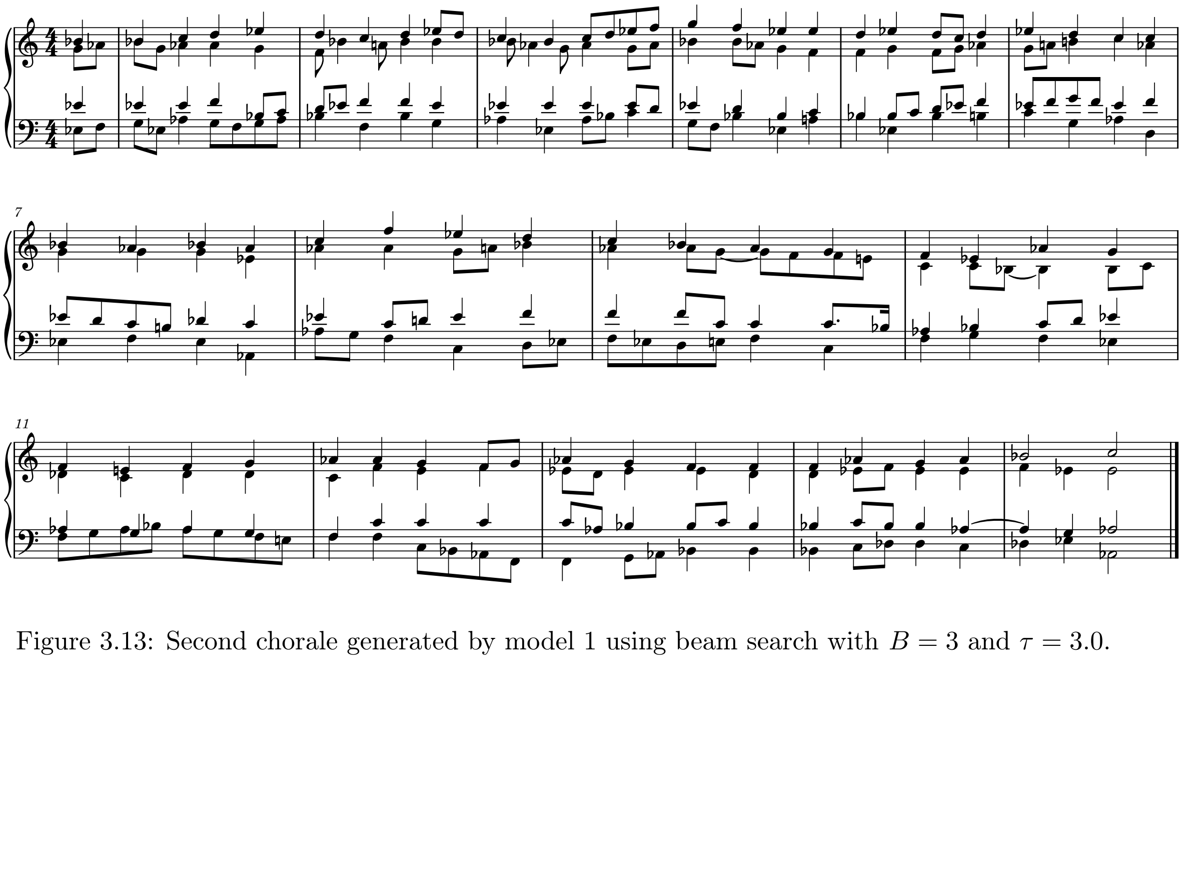


Figure 3.13: Second chorale generated by model 1 using beam search with $B = 3$ and $\tau = 3.0$.

3.3.4 Attention Evaluation

The attention weights that are computed as the model processes data can give, as mentioned before, interesting insights into how information is picked out and aggregated from the different sequence steps.

In this section, we analyse model 6, which is the model with full relative attention but no additional input features (such as current voice, current time or current pitches). This means that any information from a particular item can only be accessed through the attention mechanism, making this particular model easier to interpret.

In Figure 3.14, we see the attention weights of transformer layers 1, 3, 5 and 8 as they are processing the first two bars of *Oh Haupt voll Blut und Wunden*. We can see different patterns emerging: Layer 1 spreads attention over triangular shapes, which correspond to all items that fall on the same metric division. Layer 3 has a much more focussed attention, mostly on either the current item or the last *wait* command. For layers 5 and 8, the attention weight is distributed in a smooth way near the main diagonal, which means that the information of several local items is combined.

Figure 3.15 shows the attention weight of all eight layers in the leftmost column. The other columns show the four terms of the relative attention calculation described in 3.1.8. Column (b) shows the purely content-based selection. Since there is no position information present in the items, this term serves as a bias of what items could be of interest, either in general (e.g. the vertical lines in layer 3) or in a certain situation (e.g. the variation of horizontal lines in layer 7).

Column (c) shows the position-based content selection. In layer 3, there is a highlighted region to the left of the main diagonal, with a distance from the current step of about 15-25. In this region, this attention term looks for specific items and assigns weight to them. Another effect can be seen in layer 4, where there are clear gaps on the main diagonal. The items at those positions seem to be lacking some information that has to be collected from previous steps.

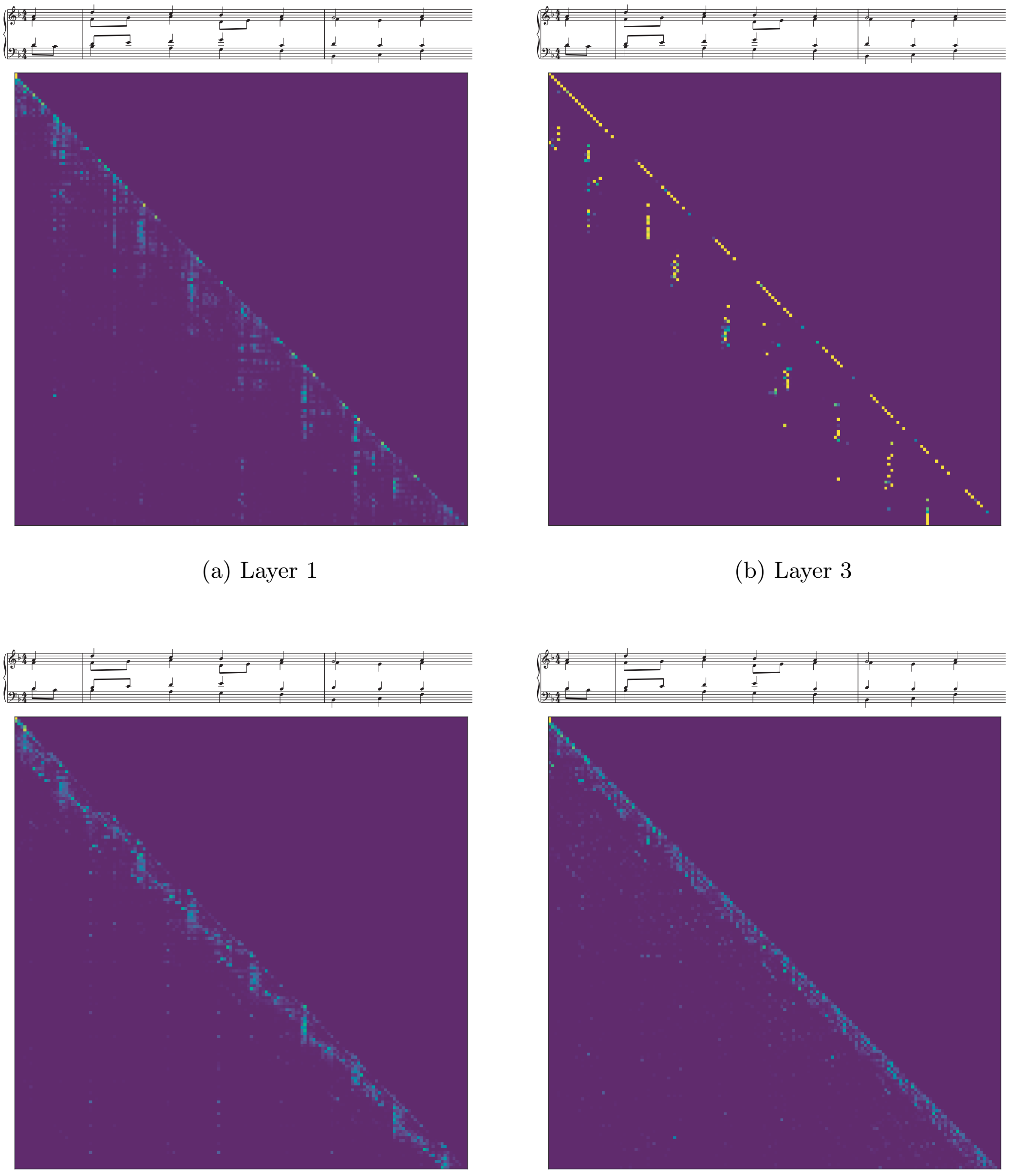
Column (d) shows the content-based position selection. Most striking for this term are the horizontal lines in, for example, layers 4 and 7.

In column (e), the purely position based selection, we can see the positional bias of this attention layer in form of different weighing of the diagonals. This does not depend on the content and will thus be constant for all inputs.

One problem with interpreting the attention weights this way is that, apart from the very first layer, the items over which the attention is distributed do not directly correspond to the input items. The information present at a certain position in the higher layers crucially depends on how exactly the previous layers distributed and aggregated information (cf. Brunner et al., 2019).

We can try to reconstruct this aggregation by plotting attention trees (as alluded to in 3.1.6 and Figure 3.6). This is done by starting with the current position at the top-most layer. Here, the positions with the highest attention weight are selected and a connection is drawn. For each of these positions, from the preceding layer again the positions with the highest weights are chosen and connected. This is repeated recursively until the input layer is reached, resulting in a tree that shows how the information flows through the attention mechanisms from the input sequence to the current output position. Several such trees are shown in Figure 3.16. One thing to note is that every layer has, because of the residual connections, automatically access to the information from the items of the previous layer at the same position. This could be shown as every node of the tree having a straight vertical line downwards (as well as branching out).

The structure of the resulting trees is at least in parts explicable. If we look, for example, at the topmost tree (which belongs to the last Soprano note), we see that at the end of a phrase, there is increased attention on the beginning. For the most part, however, it is difficult to make complete sense of the emerging trees and to determine how closely they mirror, for example, music-theoretical generative principles such as the ones mentioned in 2.1.3.



(a) Layer 1

(b) Layer 3

(c) Layer 5

(d) Layer 8

Figure 3.14: Attention weights of four selected layers for the first phrase of *Oh Haupt voll Blut und Wunden*, roughly aligned with the score.

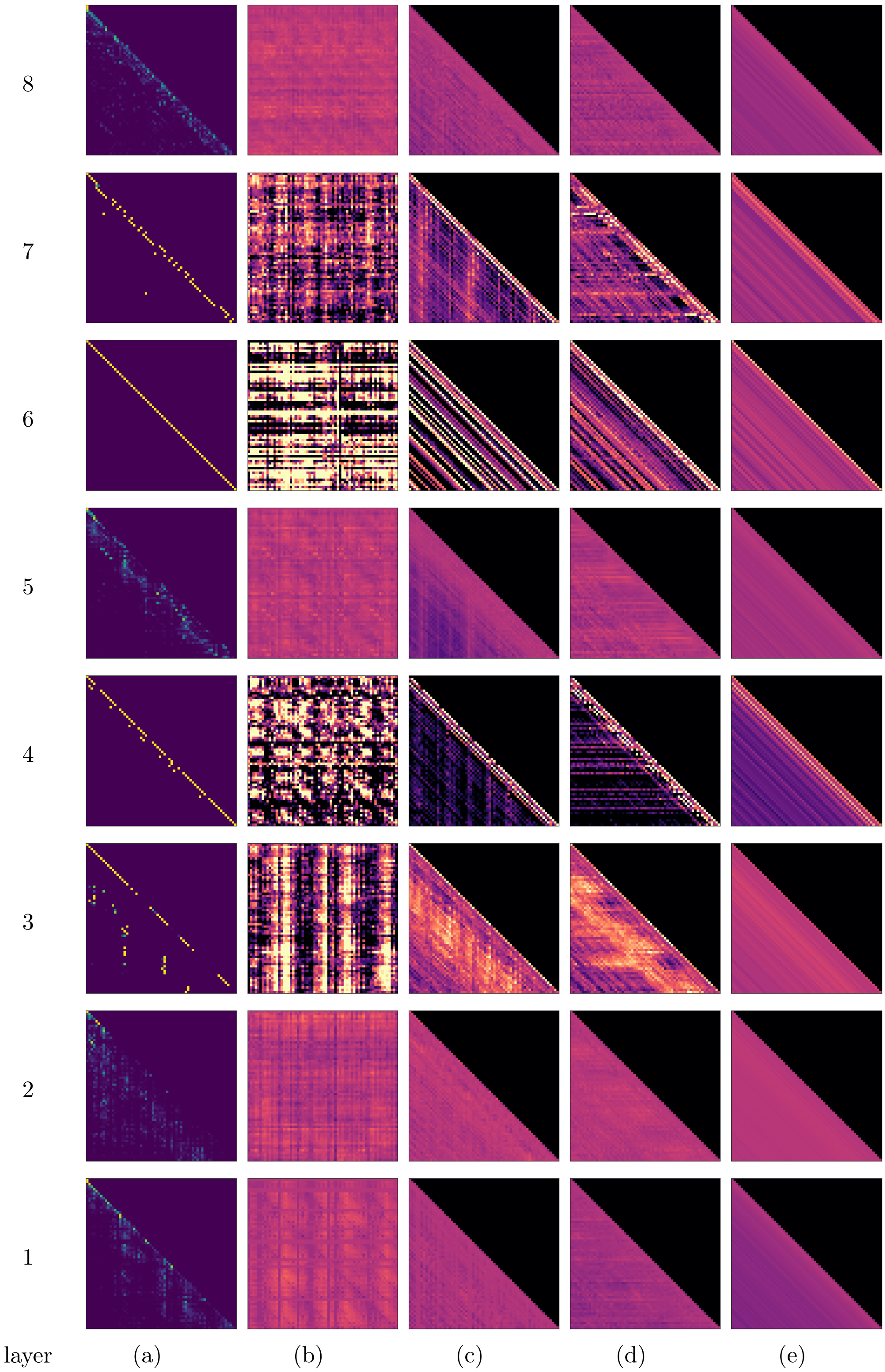


Figure 3.15: Attention weights for the first 64 items of *Oh Haupt voll Blut und Wunden*. Column (a) shows the full attention matrix. The other four columns correspond to the four individual terms: content-based content selection (b), position-based content selection, content-based position selection (c) and position-based position selection (d).

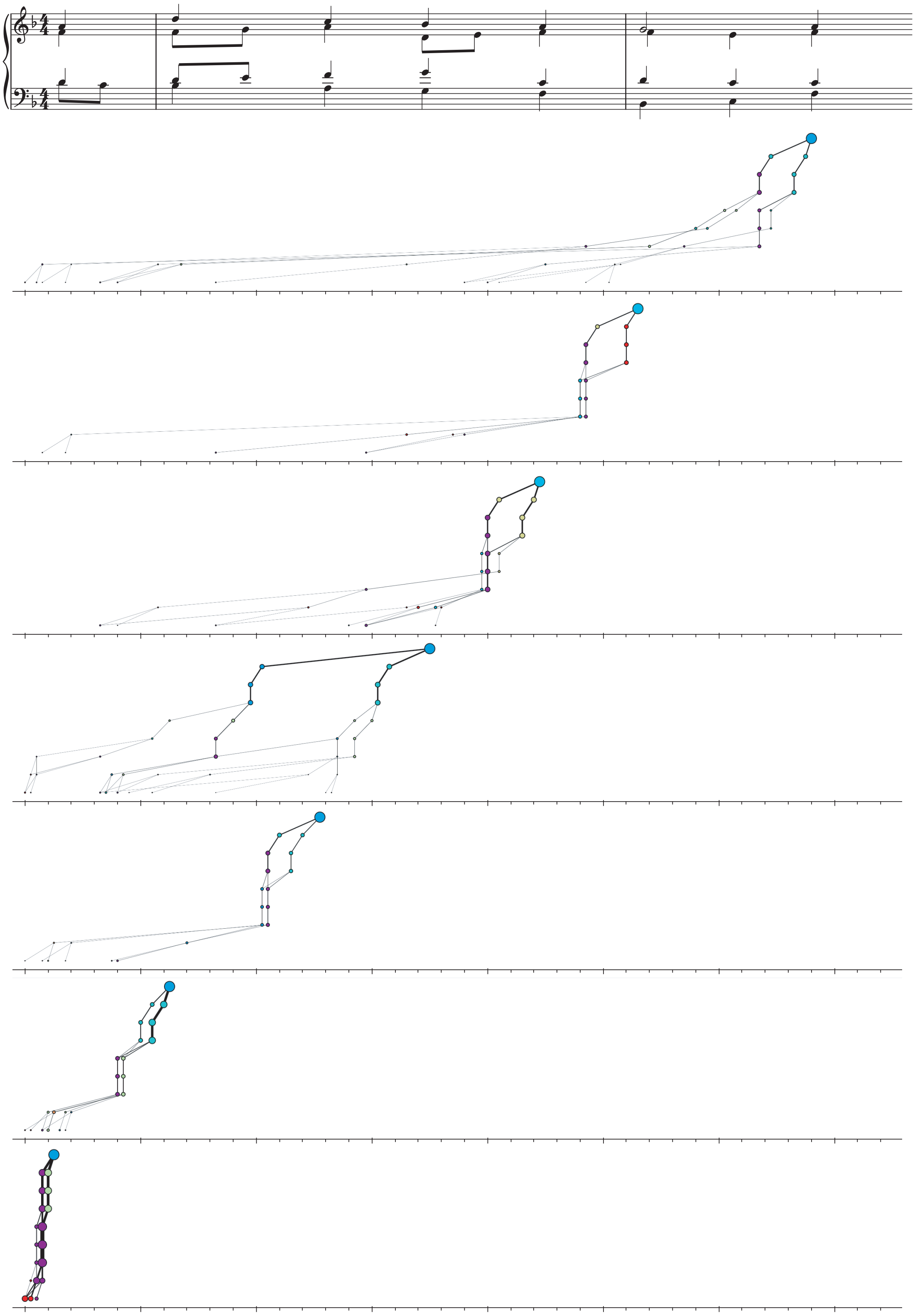


Figure 3.16: Attention trees for the *note on* commands of the Soprano voice.

3.4 MAESTRO Experiment

3.4.1 Dataset

To explore the flexibility and power of both the representational format and the model architecture, I trained a very similar model on the MAESTRO dataset (Hawthorne et al., 2018). This dataset consists of more than 200 hours of piano music that was recorded as MIDI files on a computerized grand piano during several piano competitions. Thus, the data are real performances, including all interpretative details and sometimes wrong notes. Still, they are clearly symbolic and can be brought into the command-based format without any difficulty. The performed pieces come from the classical virtuosic piano repertoire of the 17th to the early 20th century.

Since only works for piano solo are featured in this dataset, there are no voice or instrument change commands. Instead, loudness commands are frequent. There is *current loudness* input feature that states which of the 32 loudness levels was selected most recently. Everything else is the same as described in 3.3.1. The training set includes 967 performances, the validation set 137.

3.4.2 Model Architecture and Evaluation

For the MAESTRO dataset, a substantially bigger model is needed. The building blocks are exactly the same as for the Bach chorales experiments, some of the hyperparameters are, however, changed. There are six transformer layers with $d = 512$. The hidden layers of the MLPs have size 1536. 5000 warm-up and 500000 annealing steps are used. The maximum sequence length is 512. No input item dropout is used.

This model was trained only in the full relative attention version with the additional input features *current time*, *current loudness* and *current pitches.* Of course, no *instrument change* commands are necessary for this piano solo dataset. The negative log-likelihood of the validation set is for this model of course much higher since the data are much more complex. The exact performance is shown in Table 3.2. The *note end* command is no longer deterministic since multiple notes can be active at the same time. For the *wait* and *loudness* commands it is hard to achieve high accuracy since they have relatively high resolution and are noisy for real performance data.

The generated samples of this model have an average negative log-likelihood of 1.40 ± 0.0642 (also taken from the better half of all generated samples).

Model	NLL	accuracy	note on	note off	wait	loudness
full relative attention	1.43	56.9 %	70.5 %	80.90 %	25.4 %	26.1 %

Table 3.2: Validation performance of the model trained on the MAESTRO dataset.

3.4.3 Discussion

In Figure 3.17, a transcription of the performance generated by the model is shown. The output of the model can be converted into a MIDI file. It only describes the pitch, velocity and absolute timing of each note. There is no information about tempo, metric, structured rhythm or dynamic markings. I tried to write it down in a conventional format while capturing details of the performance as accurately and faithfully as possible.

It is not clear how appropriate and insightful a detailed music-theoretical analysis of this piece would be. But I will point out some of the characteristics that might reflect on the capabilities and shortcomings of the generative model.

The first thing to note is that everything can be played by a real-life pianist, there are no impossible passages. In fact, everything is set in a rather pianistic fashion. In the beginning, we have a bass line, a syncopated melody and harmonic accompaniment in the middle. The beginning has an undulating rhythmical pattern with eleven notes on the half-beat and five notes on the quarter beat. It stays consistent across the first five bars, which is very positive. Similar asymmetric rhythms can be found in the works of, for example, Schumann, Liszt, Scriabin or Ravel.

There are some interesting, maybe unconventional harmonic modulations, for instance from an Eb7 chord to g minor with D as a bass note in bar 3 or the harmonic changes in bars 7 and 8. But those are integrated into the melodic flow and coincide with quite natural agogic changes. This means, even if by chance an unexpected or unusual item is sampled at one point, the model subsequently reacts to it and tries to integrate it in a musically sensible way.

The longer second part, beginning in bar 11, is of a quite improvisatory nature, there is no clear metric structure. This does not necessarily mean that it is unrealistic, as there are pieces of the romantic and early 20th-century repertoire where the meter is intentionally obscured. Some of the free transition set in third or sixth intervals are reminiscent of improvisatory passages in works of Chopin or Liszt.

Although the musical surface is rather distinct and consistent, it seems to lack direction, coherence and intent. There are touches of motivic variation, for example from bar 20 to 24. In general though, the melody is meandering and not building any dramatic or structural tension. This is not surprising as the receptive field of the model—how much of the past it can recall—is only a few bars. So if a motive or a phrase is not repeated for a while it will be forgotten. For the same reason, there is no satisfying ending.

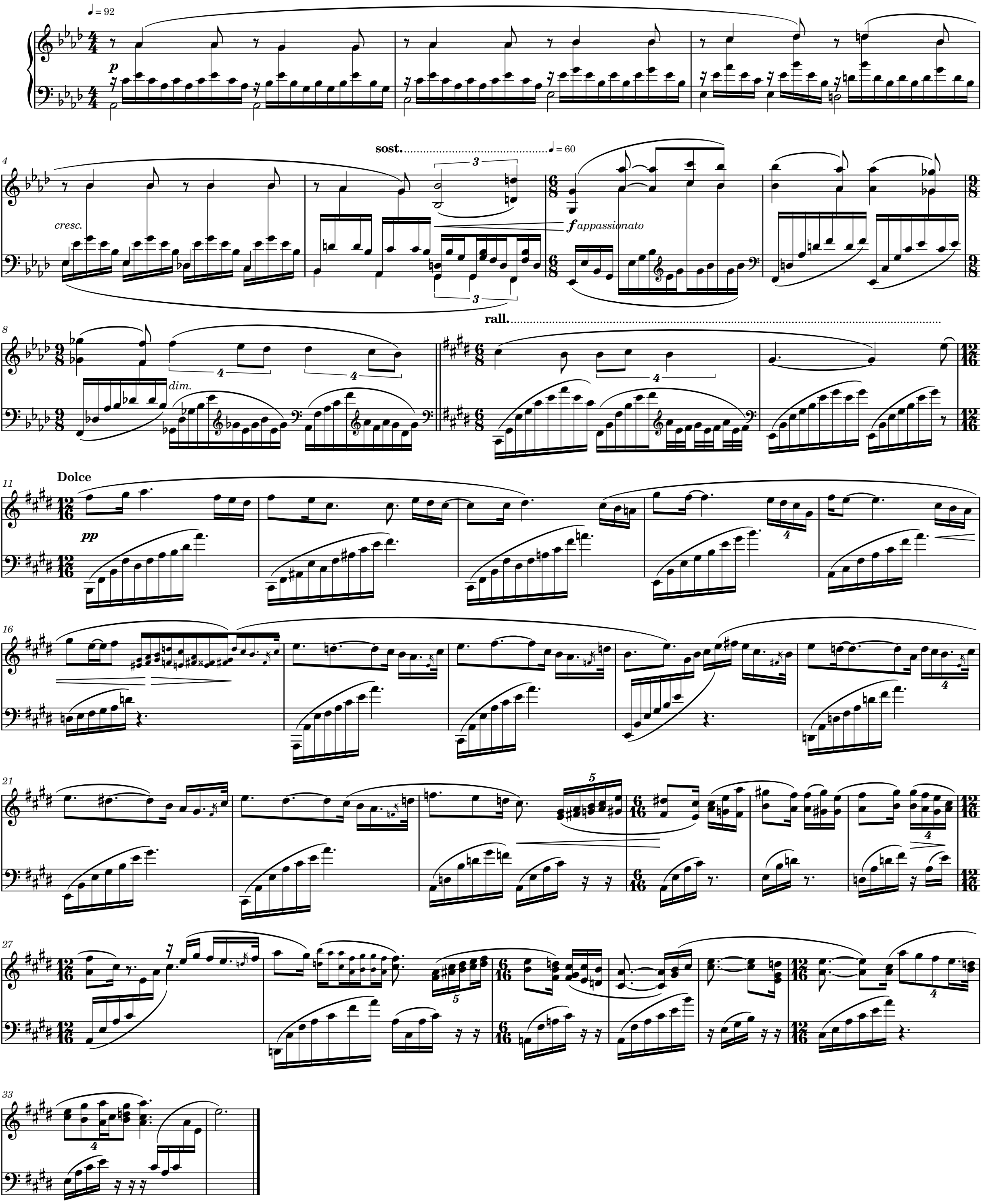


Figure 3.17: Sample generated by the MAESTRO model using beam search with $B = 3$ and $\tau = 3.0$ (manually transcribed).

Chapter 4

Quantized Autoencoders and Discrete Representations

We have seen that transformer-based purely autoregressive generative models give encouraging results. They show no difficulty in handling the intricacies of the command-based representational system. Quite the contrary, they are very well suited to gather information scattered across long sequences and entangled in intricate connections. This also allows them to capture complex musical structures and coherences, at least in the small to medium scale. Additionally, they have proven themselves to be quite general in regard to musical style and genre.

But there are severe limitations. The attention mechanism is hard to scale since memory and computational complexity grow with the square of the sequence length (since we have to construct square attention weight matrices). Long and complex pieces of music can easily require sequences with a length of several hundred thousand items. This is, at the current time, very far from computationally feasible.

Even if it were, this would not result in any real understanding of musical form since such models lack the ability of looking ahead and planning. Even a freely improvising musician usually has at least a rough idea of how their solo might develop. Let alone a classical composer who often starts with the formal structure and fills in the details bit by bit. So some kind of non-sequential processing and high-level organization likely is crucial for the composition of satisfying pieces of music.

Another problem with autoregressive modelling is that the generative process is hard to be controlled. Once started, it will simply produce one item after another until it is stopped or reaches an end token. If to this end no explicit mechanisms are added in some way (such as by conditioning on the composer during training as in Payne, 2019), there is little opportunity to significantly influence the outcome.

One approach addressing those shortcomings is learning a higher-level representational format of the musical data. There, the command-based syntax, for example, as well as other regularities of the data, can be abstracted away. Thus generating music in a suitable learnt representational space is in a sense more pure and direct since there are fewer mundane rules and obstructions. Ideally, we might compare it to the *Würfelspiele* where every configuration results in a viable piece of music.

4.1 Variational Autoencoders

The principle of a probabilistic autoencoder can be stated in the following way: We have an encoder model $q(\mathbf{z}|\mathbf{x})$ that transforms input data $\mathbf{x}$ into representations $\mathbf{z}$ and a decoder model $p(\mathbf{x}|\mathbf{z})$ that reconstructs the original data from the representation. Additionally, we have to define a fixed prior distribution $p(\mathbf{z})$ that regulates which representations $\mathbf{z}$ are allowed.

Our objective is still to maximize the likelihood of the data $\mathbb{E}_{\mathbf{x}} \log p(\mathbf{x}) = \sum_{\mathbf{x}} p(\mathbf{x}) \log p(\mathbf{x})$. It is, however, not clear how to use it for jointly optimizing both the encoder and the decoder model. This can be done with the theory of Variational Autoencoders (or short VAEs, Kingma and Welling, 2013; Kingma and Welling, 2019).

$$\begin{aligned}
\log p(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log p(\mathbf{x})\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x})p(\mathbf{x},\mathbf{z})}{p(\mathbf{x},\mathbf{z})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x},\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})} \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] + \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] + D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x}))
\end{aligned}$$

This not quite where we want to be, since we have no direct access either $p(\mathbf{x},\mathbf{z})$ or $p(\mathbf{z}|\mathbf{x})$. But we can observe that the second term, the Kullback-Leibler divergence of $q(\mathbf{z}|\mathbf{x})$ and $p(\mathbf{z}|\mathbf{x})$, cannot be negative. Therefore, the first term is a lower bound of $\log p(\mathbf{x})$. This term is called the evidence lower-bound (ELBO).

$$\begin{aligned}
\mathcal{L}_{ELBO} &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] \\
&= \log p(\mathbf{x}) - D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x}))
\end{aligned}$$

If we find a way to maximize the ELBO, we are consequently maximizing $\log p(\mathbf{x})$, which is exactly what we want, while also minimizing the D_{KL} term. Our decoder model gives us $p(\mathbf{x}|\mathbf{z})$ and we have a fixed prior $p(\mathbf{z})$. Thus it is convenient to rewrite the ELBO as follows:

$$\begin{aligned}
\mathcal{L}_{ELBO}(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log p(\mathbf{x}|\mathbf{z})\right] + \mathbb{E}_{z\sim q(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\log p(\mathbf{x}|\mathbf{z})\right] - D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))
\end{aligned}$$

Now the first term can be interpreted as the negative reconstruction loss of the autoencoder. It is similar to the loss that we used to train purely autoregressive models, with the difference that the model (the decoder) is now conditioned on the representation generated by the encoder. The second term, the KL divergence of the distribution of the latent variables generated by the encoder and the (fixed) prior $p(\mathbf{z})$, can be seen a form of regularization. It limits the capacity of the latent variables and forces the encoder to compress the information necessary for reconstruction instead of simply storing everything in the $\mathbf{z}$ variables unaltered or merely rearranged.

To generate new samples from a VAE, we first sample latent representations $\mathbf{z}$ from the prior $p(\mathbf{z})$. Those are given as input to the decoder $q(\mathbf{z}|\mathbf{x})$, which transforms them into samples from in data domain.

There are many successful applications of VAEs for learning useful representations and generative models (Ha and Schmidhuber, 2018; Roberts et al., 2018; Gómez-Bombarelli et al., 2018). For rich representations of long sequences, however, the approach comes to its limits. A complex sequence with variable length cannot be efficiently represented by a single latent variable, even if it is very high-dimensional. Instead, a sequence of representations $\mathbf{z}$ is needed, in a way possibly a shorter and compressed version of the input sequence $\mathbf{x}$. Then, for generation, we also have to generate a latent sequence. Generating sequences of multi-dimensional continuous representations, which the $\mathbf{z}$s are, is very challenging.

But we can try to learn discrete representations. Then generating sequences of $\mathbf{z}$ would be similar to the learning of sequences of discrete commands which we did in the last chapter. In fact, exactly the same models can be used.

Also, it is in many ways natural to represent music in a quantized discrete way. Many aspects at least of western classical music are inherently discrete, or categorical, such as pitch, harmony or rhythm. If we want to learn high-level representations, it makes little sense to have a continuous spectrum between, say, the concepts of an avoided and a perfect cadence.

4.2 Quantized Autoencoders

If we only allow a certain number of discrete values for the latent variables $\mathbf{z}$, this automatically limits the information capacity and diminishes the need for further regularization. This also arises naturally from the VAE framework: We set our prior $p(\mathbf{z})$ as a uniform distribution over the available values of $\mathbf{z}$. Then $D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))$ is constant and can be ignored for the optimization.

To construct a neural network architecture capable of encoding and decoding sequences that represent symbolic music bears some challenges. Firstly, we want to have our sequence of latent representations $\mathbf{z}$ to have a lower resolution, since higher-level features should be captured there and some details of the command-based syntax will be abstracted away. Then, the generative model for the latent representations is able to capture longer musical structures with the same sequence length. This means that the encoder has to downsample the input sequence and the decoder has to upsample the latent sequence. In their original design, transformers have no such capability. But a simple learnable down- or upsampling modules can be added between transformer layers to achieve this. A downsampling module consists of a convolutional layer with a small kernel and a max-pool layer. An upsampling module is a single inverse convolutional layer.

Secondly, the decoder still has to be probabilistic. We cannot deterministically transform the latent sequence into a viable reconstruction. This is because we have no reliable way of measuring how similar two sequences are. If we had, we could simply use the difference between original and reconstruction as the reconstruction loss. We can easily see why this is difficult for our kinds of sequences: Let us assume that the reconstruction repeats a note that is held in the original. The effective results are probably almost indistinguishable. But the reconstruction sequence has a few additional commands, which means that after these, every item will be completely wrong in a direct comparison with the original.

To be a tractable probabilistic model, the decoder has to generate the reconstructed sequence in an autoregressive way but still be guided by latent sequence. In principle, this is easy to achieve by having alternating self-attention and attention over the guiding sequence (similar to the transformer decoder architecture in the original publication Vaswani et al., 2017). The problem hereby is, however, that an autoregressive model is on itself quite powerful, as we saw in the last chapter, and does not necessarily need any guidance. It can be easiest for the model to simply ignore the latent sequence. This phenomenon is known as *posterior collapse* (cf. Lucas et al., 2019).

A reasonable solution is to stunt the decoder model, or at least its autoregressive part. The approach we will take is splitting the decoder into two blocks, the *feedforward decoder* and the *autoregressive decoder*. The feedforward decoder transforms the sequence of latent discrete representation $\mathbf{z}$ into an upsampled sequence of continuous guiding elements $\mathbf{g}$ of the same length as the original input sequence. The autoregressive decoder takes as input the guiding sequence $\mathbf{g}$ as well as the reconstruction, up to where it has already been generated and models the distribution over the next item that will be reconstructed. It is, compared to the purely autoregressive models

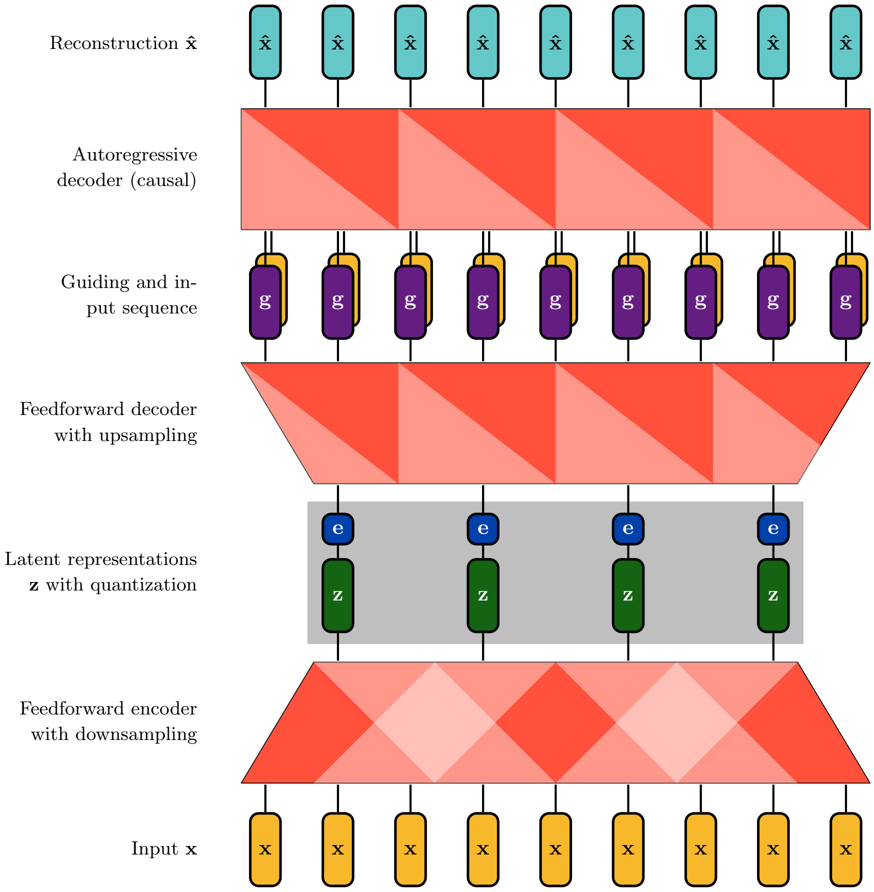

Reconstruction $\hat{\mathbf{x}}$
$\hat{\mathbf{x}}$
Autoregressive decoder (causal)
Guiding and input sequence
g
Feedforward decoder with upsampling
e
Latent representations $\mathbf{z}$ with quantization
z
Feedforward encoder with downsampling
Input $\mathbf{x}$
x

Figure 4.1: Architecture of a quantized autoencoder with a split decoder.

used earlier, less powerful because it has fewer layers and the attention span is shortened. This means that only local structures, such as the syntax of the command-based representation, are managed by the autoregressive decoder. Dependencies over long ranges have to come from the guidance sequence. For a depiction of this architecture, please see Figure 4.1.

Learning discrete representation with a neural network is inherently problematic. The training of a neural network is done by gradient descent. For the calculation of the gradients, every operation in the architecture has to be differentiable. This conflicts with the principle of quantization operations, which have no well-defined gradients since a small change in the input will either have no effect at all or cause a sudden jump. Fortunately, there exist strategies to approximate the gradients necessary for learning discrete representations. Some of them will be presented in the next sections. An analysis of further techniques can be found in Kaiser et al., 2018.

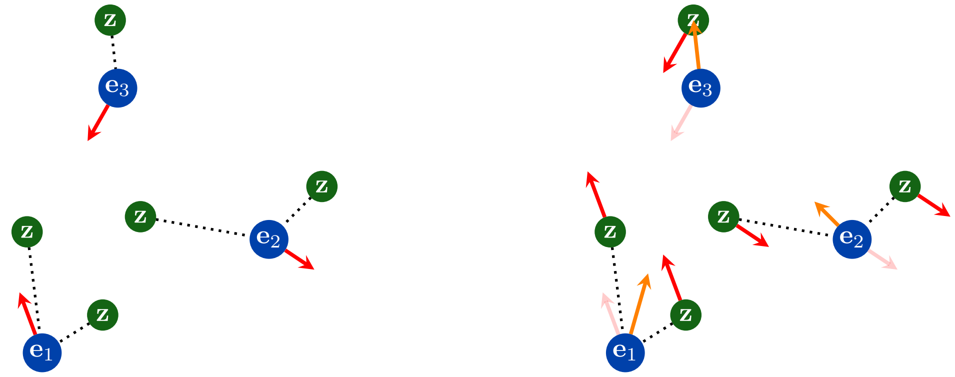
4.2.1 Vector Quantization

One approach uses vector quantization (A. v. d. Oord, Vinyals, et al., 2017). A codebook consisting of K code vectors $\mathbf{e}_k \in \mathbb{R}^D$ is randomly initialized with D being the dimensionality of the codes. The encoder generates, as before, continuous latent representations $\mathbf{z} \in \mathbb{R}^D$. Each of them is then assigned to the closest code vector $\mathbf{e}$, and this code vector, instead of $\mathbf{z}$, is given to the decoder. There are two challenges: Firstly, as mentioned before, the quantization blocks the gradients from the encoder. This is resolved by calculating the gradients for the code vectors that were used and then simply copying them unchanged to the corresponding $\mathbf{z}$ vectors. Sometimes, this is referred to as *straight-through estimation* of the gradients. These gradients can then further be backpropagated through the encoder.

Secondly, we need a way to learn sensible code vectors. They should learn to represent useful information and be spread out such that, for one thing, the distance from each **z** vector to its nearest **e** is not too long and for another thing, not too many **z** vectors are captured by the same **e**. One way to achieve this is an additional loss that is proportional to the average of those distances. This loss affects both the encoder and the position of the code vectors. One problem with this loss is that it can easily be minimized by the encoder giving exactly the same results for every input and one code vector attaining exactly this value. Then, of course, the representations contain no information. The decoder must have learned to rely on the latent representation for there to be any incentive for encoding information. A more stable approach is updating the code vectors not by minimizing the distance loss, but by iteratively moving them towards the mean positions of all **z** vectors that they currently have captured (Razavi, A. v. d. Oord, and Vinyals, 2019).

Still, VQ-VAEs are difficult to train. They tend to not fully, or not at all, use the available codes. There are tricks that can be used during training. For example, once too many **z** vectors are assigned to the same **e**, another unused **e** is set to have the same value as a randomly selected **z**.

Another method is to disable the quantization during the beginning of the training to have the decoder rely on the unperturbed information in the latent representations. When the quantization is switched on, the system is more likely to find a use for the now pruned latent representations and learn to improve them.



(a) The latent features **z** are mapped to the closest embedding vector **e** in the dictionary. In the backward pass, the gradient of the **e** vectors can be calculated (red arrows).

(b) At this point in the backward pass the gradients from the **e** vectors are copied to the corresponding **z** vectors. Then the backpropagation for the encoder can continue. The **e** vectors are iteratively moved towards the mean of the **z** vectors currently assigned to it (orange arrows).

Figure 4.2: Vector quantization with dynamic codes **e** and straight-through estimation of the gradients of the representations **z**.

4.2.2 Argmax Quantization

A simplification of the vector quantization is the argmax quantization, introduced in Dieleman, A. v. d. Oord, and Simonyan, 2018. The encodings **e** are not learned for this quantization operation, but they are fixed K-dimensional one-hot vectors (a different value is set to 1 for every encoding, while all other values are zero). With that, they are the vertices of a $K-1$-dimensional simplex. The **z** vectors have to lie on this simplex, which means they have to represent a probability distribution. In other words, all values must be between 0 and 1, and they must sum to 1. This can be achieved, for example, by the following nonlinearity:

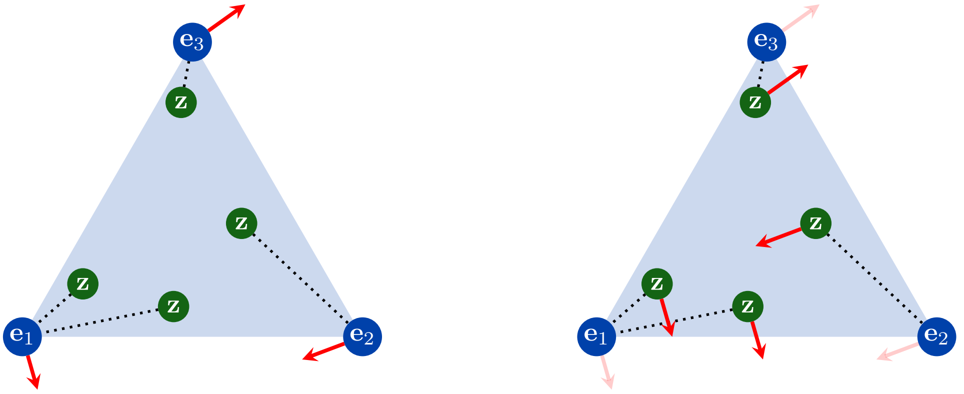
$$f(z_k) = \frac{\max(z_k, 0)}{\sum_j \max(z_j, 0)}$$

The $\mathbf{z}$ vectors are again assigned to the closest $\mathbf{e}$, which is in this case simply the one-hot vector with the 1 at the index of the highest value of $\mathbf{z}$. The gradients are also copied from the embedding vectors to the $\mathbf{z}$ vectors.

An additional loss term can be beneficial to encourage diversity among the different representations that are used. For example, the $l2$-norm of the distance between the average of the $\mathbf{z}$ vectors and a uniform distribution could be used:

$$L = \left\| \frac{1}{N} \sum_n (\mathbf{z}_n) - \frac{1}{K} \right\|_2$$

N is here the number of different vectors $\mathbf{z}_n$ in a given example.



(a) The latent features z are mapped to the closest vertex of the simplex. In the backward pass, the gradient of the e vectors can be calculated (red arrows).

(b) The gradients from the $\mathbf{e}$ vectors are copied to the corresponding $\mathbf{z}$ vectors.

Figure 4.3: Argmax quantization with fixed codes $\mathbf{e}$ and straight-through estimation of the gradients of the representations $\mathbf{z}$.

4.2.3 Gumbel-Softmax Quantization

Another approach relying on the notion of quantization as an $\arg\max$ operation is Gumbel-softmax quantization (Jang, Gu, and Poole, 2016). Here, we are modelling a distribution over all available embedding code vectors $\mathbf{e}$ (they are in this case again not fixed) but only using the one with the highest value. Again, the $\arg\max$ operation is not differentiable, so the gradients cannot flow back to the encoder to let it learn a sensible latent representation.

Let us assume for a moment that the encoder somehow always gave the best possible distribution over the different randomly initialized embeddings. If only ever the embedding with the highest probability value is chosen, this massively limits the ability to learn sensible embeddings. One embedding, for instance, could be the second choice in many important situations, but never the first. Then it will never be updated, despite being potentially very useful if slightly changed.

This can be corrected by adding noise to the distribution over the encodings before choosing the arg max, so that there is a chance for every embedding to be picked. Adding Gumbel-distributed noise to the log-probability distribution has the mathematical feature that the probability of a certain value now being the arg max is the same as its own unperturbed probability. Mathematically, this can be written as

$$p(\arg\max_i[\log z_i + \gamma_i] = k) = z_k.$$

Here, γ_i is a sample drawn from the Gumbel distribution, which has the probability density function $g(x) = \exp\big(-\exp(-x)\big)\exp(-x)$. This means that $p(\log z_i + \gamma_i) = g(\gamma_i - \log z_i)$.

If we know the value of γ_k, we can express the probability that k is largest as the product of the probabilities that any other index j is smaller. This, in turn, is determined by the cumulative distribution function, which is the integral of the probability density function.

$$\begin{aligned}
p(\arg\max_i[\log z_i + \gamma_i] = k|\gamma_k) &= \prod_{j\neq k}\int_{-\infty}^{\log z_k+\gamma_k} g(\gamma_j - \log z_j)\,\mathrm{d}\gamma_j \\
&= \prod_{j\neq k}\int_{-\infty}^{\log z_k+\gamma_k} \exp\big(-\exp(-\gamma_j + \log z_j)\big) \\
&\qquad\qquad \exp(-\gamma_j + \log z_j)\,\mathrm{d}\gamma_j \\
&= \prod_{j\neq k}\int_{0}^{-\exp(-\gamma_k-\log z_k+\log z_j)} \exp(s_j)\,\mathrm{d}s_j \\
&= \prod_{j\neq k}\exp\big(-\exp(-\gamma_k - \log z_k + \log z_j)\big) \\
&= \exp\big(-\sum_{j\neq k}\frac{z_j}{z_k}\exp(-\gamma_k)\big) \\
&= \exp\big(-\frac{1-z_k}{z_k}\exp(-\gamma_k)\big)
\end{aligned}$$

From line two to line three we used integration by substitution with $s_j = -\exp(-\gamma_j + \log z_j)$ and $\mathrm{d}s_j = s_j'\,\mathrm{d}\gamma_j \; = \exp(-\gamma_j + \log z_j)$. The last step takes advantage of the fact that $\sum_j z_j = 1$ since they define a probability distribution. Now we can move on to the unconditional case, where we do not know the value of γ_k. For that, we have to marginalize over all possible values of γ_j.

$$\begin{aligned}
p(\arg\max_i[\log z_i + \gamma_i] = k) &= \int_{-\infty}^{\infty} g(\gamma_k)\exp\big(-\frac{1-z_k}{z_k}\exp(-\gamma_k)\big)\,\mathrm{d}\gamma_k \\
&= \int_{-\infty}^{\infty}\exp(-\gamma_k)\exp\big(-\exp(-\gamma_k)\big) \\
&\qquad \exp\big(-\frac{1-z_k}{z_k}\exp(-\gamma_k)\big)\,\mathrm{d}\gamma_k \\
&= \int_{-\infty}^{\infty}\exp(-\gamma_k)\exp\big(-(1+\frac{1-z_k}{z_k})\exp(-\gamma_k)\big)\,\mathrm{d}\gamma_k \\
&= \int_{-\infty}^{\infty}\exp\big(-\frac{1}{z_k}\exp(-\gamma_k)\big)\exp(-\gamma_k)\,\mathrm{d}\gamma_k \\
&= \int_{-\infty}^{0}\exp\big(\frac{1}{z_k}s_k\big)\,\mathrm{d}s_k \\
&= z_k
\end{aligned}$$

Again, integration by substitution is used, this time with $s_k = -\exp(-\gamma_k)$ and $\mathrm{d}s_k = s_k'\,\mathrm{d}\gamma_k = \exp(-\gamma_k)$. This says that if we take the log-probabilities of a distribution, add Gumbel noise and pick the largest value, every class is picked with exactly the probability of the modelled distribution. This is known as the Gumbel-max trick (Gumbel, 1948). For now, this is simply a different clever way to sample from a distribution. We still have the problem that the arg max operation, just as sampling, is not differentiable. With that, there is no way for the encoder to learn the distributions.

The arg max operation, however, can be replaced by a smooth and differentiable approximation: the softmax function, which we already have encountered. We also saw that by adjusting the temperature parameter τ we can determine how close to arg max the behaviour of the softmax function gets (with being equivalent at the limit for $\tau \to 0$). In practice during training, τ is set according to a schedule, beginning with a relatively high temperature and slowly approaching a small value greater than 0. For testing, the arg max operation is always used.

4.3 Autoencoder Model

As already suggested, learning discrete representations of symbolic music, while attractive, can be challenging. It is in many ways a balancing act: If we have a very rich representational space, with a large number K of codes and the same number of items as the original sequence the model will have little trouble finding an accurate representation from which the original can be faithfully reconstructed. But this defeats our purpose of learning compact and concise representations that can be generated by an independent latent generative model. On the other hand, if our representation space is too restrictive the model has a hard time during training to encode anything useful, especially if the autoregressive decoder is powerful. If it is not, however, the quality of our reconstruction (our decodings) will suffer substantially.

4.3.1 Model Architecture

The goal was to achieve a downsampling factor of four going from the original sequence to the latent sequence. This seems like an appropriate amount since the command-based syntax of the input sequence is quite “wordy”, it uses a lot of items to represent relatively little information. Nevertheless, it proved to be challenging. Only the Gumbel-softmax quantization gives reliable results for a downsampling factor of more than two.

The autoencoder architectures were all trained on the Bach chorales dataset. We already established that the concrete command-based syntax of our data should be abstracted away in the latent representation. There is a way in which we can enforce that: We can express most pieces of music in a multitude of different but equivalent command sequences. For the experiments of the last chapter, we have fixed an instrument order. At a certain time step, first, the Bass commands were given, then the Tenor commands and so on. This ordering, however, is arbitrary and could be replaced with any other, or just be random. The only command items that always maintain their position are the *wait* commands, since they are, in a sense, anchored in the real dimension of time. In between *wait* commands, the items can often be permuted in some way without changing the meaning. What we can do now is having the input sequence in a random but still valid permutation. The reconstruction, however, has to be in another permutation. This should force the model to learn a latent representation sequence that is invariant to the permutations of the input sequence.

We also have to think carefully about the attention mechanisms in the different parts of the model. It is clear that the self-attention in the autoregressive decoder has to be causal (meaning that access to future items is prohibited), just as for the purely autoregressive models. The attention of the encoder should have no such restriction because it is among the encoder’s jobs to rearrange the input commands into a more general and abstract format, which requires the combination of commands both from the past and from the future. If the access to the latent representations from the encoder as well as from the decoder is only via unconstrained attention, there is nothing that gives them the properties of sequentiality, since their ordering does not matter (again, the natural domain of transformers are unordered sets). At least when we want to generate the latent representations this will become highly problematic since we cannot use autoregressive generation for unordered sets. Hence, the feedforward decoder should have only causal access to the latent sequence and the autoregressive decoder’s external attention mechanism over the guidance sequence should be causal as well. This is shown in Figure 4.1.

Model	NLL	accuracy	note on	wait	instrument
Gumbel-quantized Autoencoder	0.0573	98.1 %	93.3 %	98.2 %	99.5 %

Table 4.1: Validation reconstruction performance of the model trained on the Bach chorales dataset.

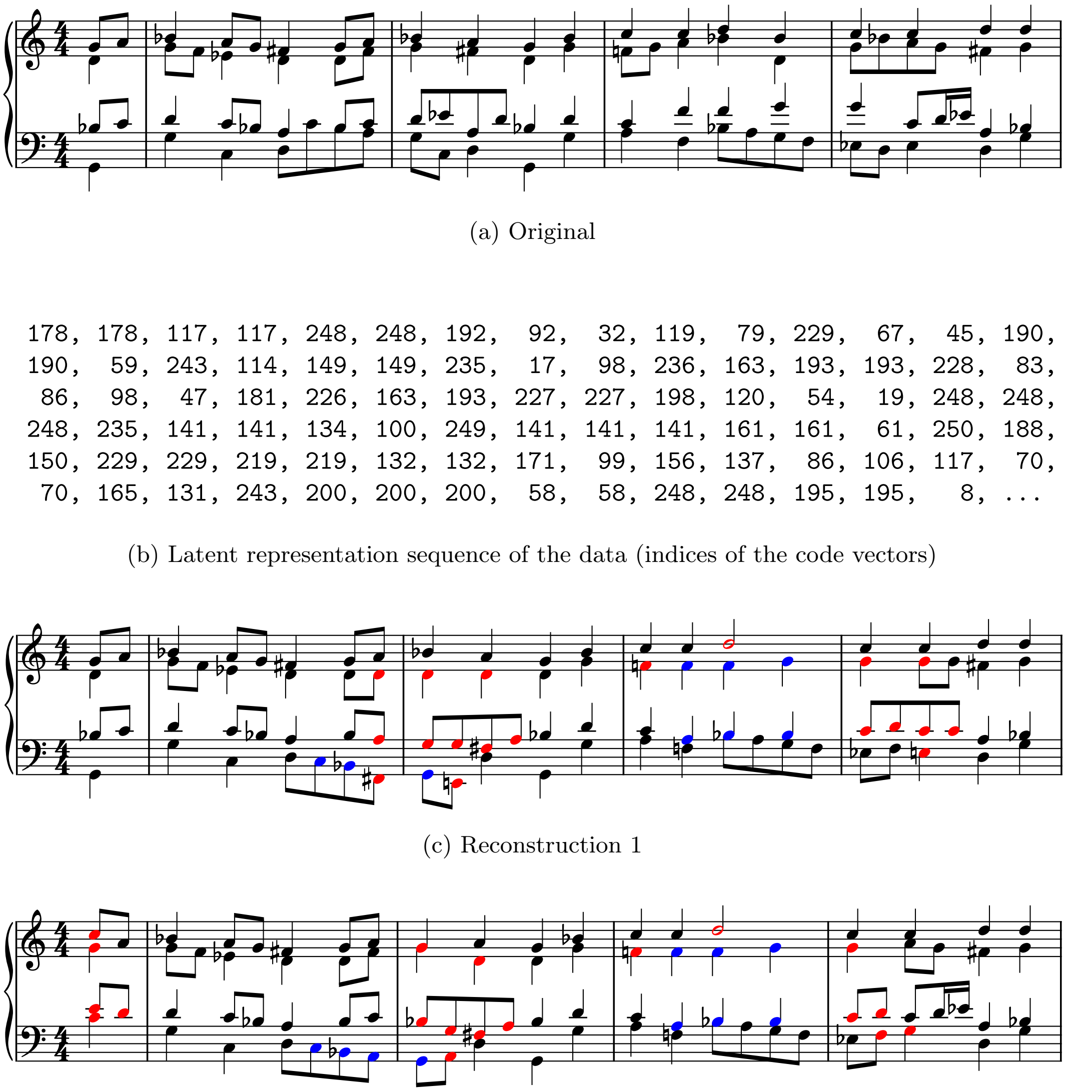
With these considerations, I found the best-working autoencoder architecture to be the following (properties that are not listed are the same as for the purely autoregressive models): The three blocks (encoder, feedforward decoder and autoregressive decoder) each consist of five transformer layers. The autoregressive decoder has both self-attention and external attention over the guidance sequence. The attention mechanisms have all eight heads. Encoder and feedforward decoder have down- or upsampling layers respectively after the second and third transformer layer. The maximum sequence length is 128. For the autoregressive decoder, the external attention span (from how long in the past the model can access items from the guiding sequence) is limited to 32 to further encourage locality in the representations. The softmax temperature τ of the Gumbel quantization start at 4.0 and exponentially decreases over 100000 training steps to 0.5. The quantization is switched on after 5000 steps. For the input sequences to the encoder, the voice order is randomly shuffled. The autoregressive decoder, however, gets the same sequence with fixed voice order.

The reconstruction performance on the validation set is listed in table 4.2. The negative log-likelihood is very low compared to the purely autoregressive models. Empirically I found that the reconstruction performance has to be quite good to yield satisfying results.

4.3.2 Discussion

In Figure 4.4, we see the beginning of a Bach chorale from the validation set, its latent code sequence and two reconstructions from this sequence. The reconstructions were generated using beam search with a beamwidth of 5 and a temperature of 2. Since the autoregressive decoder is probabilistic, reconstructions from the same latent sequence will differ from each other. The encoder learns a lossy compression, so we cannot expect every singly note to be reconstructed perfectly. What we are hoping for is that the reconstructions are musically sensible, have no obvious mistakes and resemble the original closely.

I think we can say that this is mostly the case for these reconstructions. The voice leading is in large parts sensible (the biggest exception being the bass line in Figure 4.4c bar 2). The model sometimes seems to have trouble distinguishing between major and minor chords (in our examples there is some confusion about c minor and c major chords). Other than that, the harmonizations are valid.



(a) Original

```
178, 178, 117, 117, 248, 248, 192,  92,  32, 119,  79, 229,  67,  45, 190,
190,  59, 243, 114, 149, 149, 235,  17,  98, 236, 163, 193, 193, 228,  83,
 86,  98,  47, 181, 226, 163, 193, 227, 227, 198, 120,  54,  19, 248, 248,
248, 235, 141, 141, 134, 100, 249, 141, 141, 141, 161, 161,  61, 250, 188,
150, 229, 229, 219, 219, 132, 132, 171,  99, 156, 137,  86, 106, 117,  70,
 70, 165, 131, 243, 200, 200, 200,  58,  58, 248, 248, 195, 195,   8, ...
```

(b) Latent representation sequence of the data (indices of the code vectors)

(c) Reconstruction 1

(d) Reconstruction 2

Figure 4.4: The beginning of *Beweis dein Macht, Herr Jesu Christ* from the validation set with two reconstruction from the same latent sequence. Differences from the original that are barely noticeable to the listener are marked in blue, more significant deviations in red.

4.4 Latent Generative Model

During training, we assumed that the prior distribution $p(\mathbf{z})$ over latent variables is uniform and that the items of the latent sequence are independent of one another. This was, of course, a simplification. But now that we have a trained autoencoder we can empirically learn the prior by creating a generative autoregressive model for the latent sequences. We compute and save the latent sequences for all pieces of our training and validation set and use them to train the latent generative model (cf. A. v. d. Oord, Vinyals, et al., 2017). These sequences are simply lists of numbers representing the corresponding code vectors (see Figure 4.4b). They have no clear inherent structure (other than being sequential), so no additional information can be added to help the model.

We use an architecture very similar to the ones we already know. The model that works best has seven causal transformer layers (each with 8 attention heads), a feature dimension of 256 and a hidden layers size of 1024 in the MLPs. Full relative attention is used.

4.4.1 Discussion

The negative log-likelihood is quite high compared to the other models (see Table 4.2). The fact that the relevant information has to be encoded into a shorter sequence means that it has to have higher entropy (more information) per item and thus is harder to learn. For the same piece to have the same likelihood in the command-based representation and in the latent representation, we would need the average negative log-likelihood per item of the latent model to be about four times as large, since he original sequence is four times as long. The fact that the average NLL of the latent generative model is considerably higher even than that suggests that the architecture does not yet fully exhaust the potential and there is further room for improvement.

In Figure 4.5, a newly generated piece is shown. The latent sequence was created using beam search with the latent model. A beamwidth of 3 was used while leaving the temperature at 1. This is comparable to purely autoregressive generation with a significantly lower temperature and is required to obtain acceptable results. The generated latent sequence was then given to the decoder that generated the command-based representation, using beamwidth 3 and temperature 3.0.

The quality of the produced sample has to be ranked lower than the ones from the purely autoregressive models of chapter 3. While there are notions of a melody, it is more wooden and unnatural, presumably due to the artificially decreased entropy of the latent generative model. There are undesirable metric shifts, for example in bar 7. The voice leading is also problematic: In the Bass voice, there are several unfavourable jumps, most prominently the major seventh in bar 6. We also have parallel octaves between Soprano and Bass on the first beats of bars 2 and 4. However, it also clearly shows that high-level discrete representations of symbolic music can in principle be learned and generated with this approach.

Model	NLL	accuracy
Latent code model	1.416	65.4 %

Table 4.2: Validation performance of the latent generative model trained on the Bach chorales dataset.

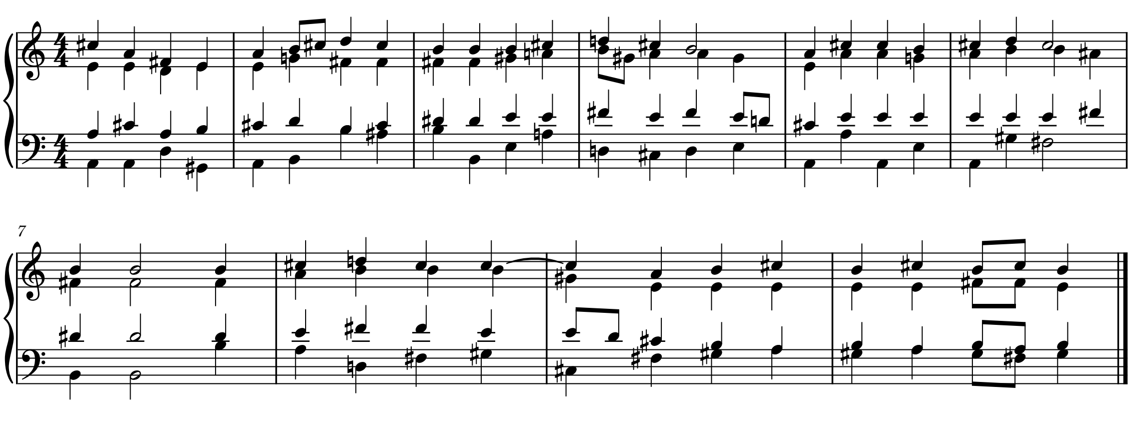


Figure 4.5: A chorale generated using the latent generative model and the decoder.

Chapter 5

Conclusion and Outlook

Generative models of music have a long tradition. Historically, these models relied heavily on what we called *representational bias*, which requires customized creation, selection or processing of the data and can often be limiting or time-consuming. The advent of deep learning techniques, combined with autoregressive modelling, allows a shift to more general representational formats, such as sequences of commands. Capturing the complexities of a musical score in this format, however, leads long sequences with non-trivial syntactic rules.

As has been shown in Huang, Vaswani, et al., 2018 and Payne, 2019, the transformer architecture (which was originally developed for natural language processing tasks) works very well with these kinds of data. In this work, I showed that we can train our model, without any substantial changes in architecture or representational format, on data as different as Bach chorales and virtuoso piano performance and achieve very good results for both of them. Adding contextual state features to the input command items gives the model useful additional information and structure which helps the performance while not adding any complexity to the architecture. We also looked at how the different terms of the relative self-attention mechanism interact and how the model builds tree-like structures, potentially reproducing grammatical, musical or even formal construction of a piece. While the introduced visualization methods can give some intuition, larger-scale quantitative studies will be needed to further clarify how and in what way these models build an understanding of music. For the generation process, I found that beam search with a high-temperature setting leads to more stable and at the same time interesting samples.

With enormous amounts of training data and adequately large models, standard transformers can be scaled to achieve remarkable results (Payne, 2019; Brown et al., 2020). But this approach is still limited. For one, there no infinite supply of high-quality symbolic musical data. Music of a certain composer or a certain style is inherently limited. Moreover, the plain linear progression of autoregressive models does not allow for the entangled, hierarchical relations between musical elements and concepts that are the mark of many musical forms.

Using a Variational Autoencoders, we can learn more abstract representations of the data. From these representations, the original can be reconstructed. This means we can train a generative model that creates music on a higher level of abstraction, which might be closer to how a composer thinks about music. The representation, however, will still consist of a sequence of items. For a standard VAE, the representations will also be high-dimensional continuous vectors. Those representational sequences would be very hard to model.

For this work, I created models to learn discrete representations which can much more easily be created by a latent generative model and which at the same time reflect the many discrete aspects of (at least western tonal) music. For non-symbolic data, this approach was introduced by A. v. d. Oord, Vinyals, et al., 2017. Transferring these techniques to symbolic sequences, despite the fact that the data already is discrete, turned out to be quite challenging. To employ an autoregressive decoder and at the same time prevent posterior collapse, the development of a split decoder architecture was needed. The feedforward decoder transforms the downsampled latent sequence into the guiding sequence, a dense sequence of continuous vectors, which the autoregressive decoder uses to reconstruct the original as faithfully as possible. Of the multiple quantization methods I experimented with, only Gumbel-softmax quantization worked reliably in a high-compression setting.

The status of this approach as presented here can be seen as a proof of concept which still requires more work to fulfil its potential. The bottleneck at the moment seems to be the latent generative model which is not able to find much regularity in the latent sequences. There are many paths that can be explored to either enforce some structure onto the latent sequences or train more powerful models to generate them.

Several autoencoders can be stacked in a hierarchical way to learn even coarser and more abstract features where large coherent forms can be produced. The decoders could be conditioned on certain features of the data. These would then not have to be encoded in the latent representation. Notable results along these lines have already been achieved for less structured data, such as images (Razavi, A. v. d. Oord, and Vinyals, 2019) and audio (Dhariwal et al., 2020). For our application of symbolic music this, could allow, for example, to decode the same representation in the style of different composers or musical genres and give rise to many creative applications.

Bibliography

Bellman, Richard E (1961). *Adaptive control processes.* Princeton university press.

Benjamin, D Paul (2012). *Change of representation and inductive bias.* Vol. 87. Springer Science & Business Media.

Bishop, Christopher M (2006). *Pattern recognition and machine learning.* springer.

Briot, Jean-Pierre, Gaëtan Hadjeres, and François-David Pachet (2017). "Deep learning techniques for music generation–a survey". In: *arXiv preprint arXiv:1709.01620.*

Brown, Tom B, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. (2020). "Language models are few-shot learners". In: *arXiv preprint arXiv:2005.14165.*

Brunner, Gino, Yang Liu, Damián Pascual, Oliver Richter, and Roger Wattenhofer (2019). "On the validity of self-attention as explanation in transformer models". In: *arXiv preprint arXiv:1908.04211.*

Calter, Paul (1998). "Pythagoras & Music of the Spheres". In: *Geometry in Art and Architecture, Retrieved.*

Chomsky, Noam (1965). "Aspects of the theory of syntax". In: *Multilingual Matters: MIT Press.*

Cope, David (2001). *Virtual music: computer synthesis of musical style.* MIT press.

Cybenko, George (1989). "Approximations by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2, pp. 183–192.

Dai, Zihang, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov (2019). "Transformer-xl: Attentive language models beyond a fixed-length context". In: *arXiv preprint arXiv:1901.02860.*

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805.*

Dhariwal, Prafulla, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever (2020). "Jukebox, 2020". In: *URL https://openai.com/blog/jukebox/.*

Dieleman, Sander, Aaron van den Oord, and Karen Simonyan (2018). "The challenge of realistic music generation: modelling raw audio at scale". In: *Advances in Neural Information Processing Systems*, pp. 7989–7999.

Eck, Douglas and Jürgen Schmidhuber (2002). "A first look at music composition using lstm recurrent neural networks". In: *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale* 103, p. 48.

Garcez, Artur d'Avila, Tarek R Besold, Luc De Raedt, Peter Földiak, Pascal Hitzler, Thomas Icard, Kai-Uwe Kühnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver (2015). "Neural-symbolic learning and reasoning: contributions and challenges". In: *2015 AAAI Spring Symposium Series.*

Gómez-Bombarelli, Rafael, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamın Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik (2018). "Automatic chemical design using a data-driven continuous representation of molecules". In: *ACS central science* 4.2, pp. 268–276.

Gumbel, Emil Julius (1948). *Statistical theory of extreme values and some practical applications: a series of lectures.* Vol. 33. US Government Printing Office.

Ha, David and Jürgen Schmidhuber (2018). "World models". In: *arXiv preprint arXiv:1803.10122.*

Hadjeres, Gaëtan, François Pachet, and Frank Nielsen (2017). “Deepbach: a steerable model for bach chorales generation”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 1362–1371.

Hawthorne, Curtis, Andriy Stasyuk, Adam Roberts, Ian Simon, Cheng-Zhi Anna Huang, Sander Dieleman, Erich Elsen, Jesse Engel, and Douglas Eck (2018). “Enabling factorized piano music modeling and generation with the MAESTRO dataset”. In: *arXiv preprint arXiv:1810.12247*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Huang, Cheng-Zhi Anna, Tim Cooijmans, Adam Roberts, Aaron Courville, and Douglas Eck (2019). “Counterpoint by convolution”. In: *arXiv preprint arXiv:1903.07227*.

Huang, Cheng-Zhi Anna, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck (2018). “Music transformer”. In: *arXiv preprint arXiv:1809.04281*.

Jang, Eric, Shixiang Gu, and Ben Poole (2016). “Categorical reparameterization with gumbel-softmax”. In: *arXiv preprint arXiv:1611.01144*.

Kaiser, Łukasz, Aurko Roy, Ashish Vaswani, Niki Parmar, Samy Bengio, Jakob Uszkoreit, and Noam Shazeer (2018). “Fast decoding in sequence models using discrete latent variables”. In: *arXiv preprint arXiv:1803.03382*.

Kingma, Diederik P and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*.

Kingma, Diederik P and Max Welling (2013). “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114*.

— (2019). “An introduction to variational autoencoders”. In: *arXiv preprint arXiv:1906.02691*.

Kool, Wouter, Herke Van Hoof, and Max Welling (2019). “Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement”. In: *arXiv preprint arXiv:1903.06059*.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). “Deep learning”. In: *nature* 521.7553, pp. 436–444.

Lerdahl, Fred and Ray S Jackendoff (1996). *A generative theory of tonal music*. MIT press.

Lindsay, Grace W (2020). “Attention in Psychology, Neuroscience, and Machine Learning”. In: *Frontiers in Computational Neuroscience* 14, p. 29.

Lovelace, Ada (1843). “‘Notes on L. Menabrea’s ‘Sketch of the Analytical Engine Invented by Charles Babbage, Esq.’”” In: *Taylor’s Scientific Memoirs* 3.1843, p. 1843.

Lucas, James, George Tucker, Roger Grosse, and Mohammad Norouzi (2019). “Understanding posterior collapse in generative latent variable models”. In:

Markov, Andrey Andreyevich (1971). “Extension of the limit theorems of probability theory to a sum of variables connected in a chain”. In: *Dynamic probabilistic systems* 1, pp. 552–577.

Mozart, Wolfgang Amadeus (1790). *Musikalisches Würfelspiel*. Berlin: Rellstab.

Oord, Aaron van den, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu (2016). “Wavenet: A generative model for raw audio”. In: *arXiv preprint arXiv:1609.03499*.

Oord, Aaron van den, Oriol Vinyals, et al. (2017). “Neural discrete representation learning”. In: *Advances in Neural Information Processing Systems*, pp. 6306–6315.

Oore, Sageev, Ian Simon, Sander Dieleman, Douglas Eck, and Karen Simonyan (2018). “This time with feeling: Learning expressive musical performance”. In: *Neural Computing and Applications*, pp. 1–13.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. (2019). “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems*, pp. 8026–8037.

Payne, Christine (2019). “MuseNet, 2019”. In: *URL https://openai. com/blog/musenet*.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever (2019). “Language models are unsupervised multitask learners”. In: *OpenAI Blog* 1.8, p. 9.

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu (2019). "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *arXiv preprint arXiv:1910.10683*.

Ratner, Leonard G (1970). *Ars combinatoria. Chance and choice in eighteenth-century music*. na.

Razavi, Ali, Aaron van den Oord, and Oriol Vinyals (2019). "Generating diverse high-fidelity images with vq-vae-2". In: *Advances in Neural Information Processing Systems*, pp. 14837–14847.

Roberts, Adam, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck (2018). "A hierarchical latent vector model for learning long-term structure in music". In: *arXiv preprint arXiv:1803.05428*.

Schmidhuber, Jürgen (2015). "Deep learning in neural networks: An overview". In: *Neural networks* 61, pp. 85–117.

Scriabin, Aleksandr (1907). *Piano Sonata No.5, Op.53*. Leipzig: Edition Peters.

Shaw, Peter, Jakob Uszkoreit, and Ashish Vaswani (2018). "Self-attention with relative position representations". In: *arXiv preprint arXiv:1803.02155*.

Siegelmann, Hava T and Eduardo D Sontag (1992). "On the computational power of neural nets". In: *Proceedings of the fifth annual workshop on Computational learning theory*, pp. 440–449.

Vasquez, Sean and Mike Lewis (2019). "Melnet: A generative model for audio in the frequency domain". In: *arXiv preprint arXiv:1906.01083*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need". In: *Advances in neural information processing systems*, pp. 5998–6008.

Yang, Li-Chia, Szu-Yu Chou, and Yi-Hsuan Yang (2017). "MidiNet: A convolutional generative adversarial network for symbolic-domain music generation". In: *arXiv preprint arXiv:1703.10847*.