TU WIEN Informatics

PAUL-2: A Transformer-Based Algorithmic Composer of Two-Track Piano Pieces

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Felix Schön, BSc.
Registration Number 11777722

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag.rer.nat. Dr.techn. Hans Tompits

Vienna, 5th December, 2022

Felix Schön	Hans Tompits

Erklärung zur Verfassung der Arbeit

Felix Schön, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. Dezember 2022

Felix Schön

Acknowledgements

First and foremost I want to thank my parents who greatly helped me over the course of this thesis and my studies in general by taking a lot of work off my shoulders. I would not have been able to complete my work without their support, and I am very grateful for everything they have done for me.

I want to express my sincere gratitude to my advisor Ao.Univ.Prof. Hans Tompits, with whom working was not only an absolute pleasure but incredibly uncomplicated as well. Having the freedom of being able to explore and work on what I personally am interested in provided great motivation for our past works and this thesis, and I am already looking forward to our next venture!

I want to thank my friends and family who have listened to countless boring (to them) explanations when I had no rubber ducks at hand. I hope that they will continue to provide their support even after being bothered with countless requests to complete my surveys!

I want to thank Mag.art. Wolfgang Schmidtmayr of the University of Music and Performing Arts Vienna and Univ.Prof. Geraldine Fitzpatrick from the Human-Computer Interaction Group at our university for their valuable input.

Lastly I want to thank my piano teacher Nikolaus Karall who introduced me to both the world of music and IT at the same time, and who showed me the ropes when it came to the start of my university education.

Abstract

In this thesis, we introduce the algorithmic composer PAUL-2, which is the direct successor of the system PAUL developed in earlier work. Although both systems are designed for similar goals, PAUL-2 makes use of a significantly enhanced internal architecture and exhibits an improved performance. PAUL-2 is based on a transformer neural-network architecture and implements the relative attention improvements introduced by the Music Transformer model. A transformer is a state-of-the-art sequence-to-sequence architecture capable of generating output sequences of varying length based on a number of input sequences, not necessarily of the same domain. The purpose of PAUL-2 is to generate two-track piano pieces where a particular feature of the system is that the difficulty of the generated output pieces can be set by a parameter. This adjustment possibility facilitates how difficult a piece would be to play for a human pianist and is a central feature for the designated future usage of PAUL-2, which is in an educational context where it is to be used to teach piano students how to sight-read, where students would be provided with computer-generated prompts conforming to their level of proficiency. Here, they have to rely on their sight-reading capabilities alone, as they would not be familiar with the pieces. In a medium-scale study, we evaluated the musical quality of the output of PAUL-2. We found that although participants were able to differentiate between the computer-generated output of PAUL-2 and genuine pieces about three quarters of the time, in general the participants regard the quality of PAUL-2 quite favourably. Furthermore, as part of the development of PAUL-2, we also devised S-Coda, a Python library used to manipulate MIDI files. S-Coda supports many different operations, such as the quantisation of MIDI files, the automatic splitting of sequences into bars, or the visual representation of musical pieces as piano rolls. The usage of S-Coda can greatly reduce the amount of time needed to preprocess music data for machine-learning applications.

Kurzfassung

In dieser Arbeit stellen wir das automatische Kompositionssystem PAUL-2 vor, welches eine direkte Weiterentwicklung des bereits existierenden Systems PAUL ist. Obwohl beide Systeme für dieselben Zwecke konzipiert sind, zeichnet sich PAUL-2 durch eine erheblich verbesserte interne Architektur und eine gesteigerte Performanz aus. PAUL-2 basiert auf einer neuronalen Transformer Netzwerkarchitektur und verwendet Techniken des Music Transformer Systems. Ein Transformer ist eine Sequence-to-Sequence Netzwerkarchitektur auf dem neuesten Stand der Technik, welche erlaubt, Ausgabesequenzen variierender Länge basierend auf mehreren Eingabesequenzen aus möglicherweise unterschiedlichen Domänen zu erzeugen. Die Aufgabe von PAUL-2 ist die Erzeugung von zweispurigen Klavierstücken, wobei ein wesentliches Merkmal des Systems ist, dass der Schwierigkeitsgrad der erzeugten Stücke parametrisierbar ist. Diese Einstellmöglichkeit bezieht sich dabei darauf, wie schwierig es für einen menschlichen Spieler ist, ein erzeugtes Stück zu spielen. Hierbei handelt es sich um ein zentrales Merkmal von PAUL-2 für einen zukünftigen Einsatz als Teil eines Lernsystems um Klavierschüler das Blattlesen beizubringen. Die Schüler würden in einer solchen Umgebung computergenerierte Aufgaben erhalten, welche ihren Fähigkeiten entsprechen und müssten sich dabei allein auf ihre Blattlesefähigkeiten verlassen, da sie die Stücke nicht kennen. In einer experimentellen Studie wurde die musikalische Qualität der von PAUL-2 erzeugten Stücke evaluiert, wobei festgestellt werden konnte, dass die Studienteilnehmer zwar in circa drei Viertel der Fälle zwischen computergenerierten und echten Stücken unterscheiden konnten, aber im Allgemeinen die Qualität der erzeugten Stücke als gut angesehen angesehen haben. Im Zuge der Entwicklung von PAUL-2 wurde auch die Python Bibliothek S-Coda realisiert, welche für das Bearbeiten von MIDI Dateien verwendet werden kann. S-Coda unterstützt viele verschiedene Operationen, wie das Quantisieren von MIDI Dateien, das automatische Aufspalten von Sequenzen in Takte oder die visuelle Repräsentation von musikalischen Stücken als Klavierrolle. Durch die Verwendung von S-Coda ist es möglich, die Zeit für das Vorverarbeiten von Daten für Anwendungen im Bereich des maschinellen Lernens deutlich zu verringern.

Contents

Abstract **vii**

Kurzfassung **ix**

Contents **xi**

1 Introduction **1**

2 Prerequisites **5**
2.1 Music Theory Foundations 5
2.1.1 Properties of Sound 5
2.1.2 Music Theory and Music Notation 7
2.2 Elements of the MIDI Protocol 13
2.2.1 A Brief History of MIDI 13
2.2.2 The MIDI Protocol 13
2.2.3 MIDI Files 16
2.3 Background on Neural Networks 19
2.3.1 Basic Neural Network Architectures 19
2.3.2 The Transformer Architecture 23

3 Computer-aided Algorithmic Composition: A Brief Overview **31**
3.1 Markov Chains 31
3.2 Rule- and Constraint-Based Systems 33
3.3 Evolutionary and Genetic Algorithms 35
3.4 Systems Based on Neural Networks 36
3.5 Other Approaches 38

4 The PAUL-2 System **39**
4.1 Structure and Functionality 39
4.1.1 Overview 39
4.1.2 Architecture 41
4.1.3 Functionality and Motivation 45
4.2 Retrieval and Processing of the Dataset 47
4.2.1 Choice of Dataset Sources 47

4.2.2 Cleansing of the Dataset . . . 49
4.2.3 Preprocessing Procedure . . . 50
4.3 Analysis of the Dataset . . . 52
4.4 Training PAUL-2 . . . 54
4.5 Generated Results . . . 56
4.5.1 Analysis of the Final Networks . . . 57
4.5.2 Generated Compositions . . . 60
4.5.3 Future Improvements . . . 60
4.6 Differences Between PAUL and PAUL-2 . . . 68
4.7 User Study . . . 69

5 The S-Coda Library **77**
5.1 Structure . . . 77
5.1.1 The Message Object . . . 78
5.1.2 The Sequence Objects . . . 78
5.1.3 The Bar, Track, and Composition Objects . . . 79
5.1.4 The MIDI Interface . . . 80
5.2 Sequence Processing . . . 81
5.2.1 Relative Representation Interfaces . . . 81
5.2.2 Absolute Representation . . . 85
5.2.3 Difficulty Assessment . . . 91
5.2.4 Further Functionality . . . 98
5.3 Usage . . . 100

6 Conclusion **105**

Bibliography **107**

CHAPTER 1

Introduction

In the summer of 1956, John McCarthy of Dartmouth College organised a two-month workshop, gathering scientists interested in automata theory, neural networks, and the study of intelligence in general. It was there that the term *artificial intelligence* (AI) was first coined [83]. Ever since its inception, AI has experienced rapid growth in both interest and potential. Notable events include the win of IBM's DeepBlue [19] over chess world champion Garry Kasparov in 1997, or the more recent accomplishment of DeepMind's AlphaGo [90], which beat professional Go player Lee Sedol with a final score of four to one.

With all its potential, it should come as no surprise that artificial intelligence has also been applied to the field of arts, often quite successfully. The field of combining AI and art is generally referred to as *computational creativity* or *artificial creativity* [31]. A well-known application is *neural style transfer*, where the style of one image (usually a famous artwork, e.g., Van Gogh's *The Starry Night*) is transferred and applied to another image. Other applications of AI in art include, e.g., text and image generation. Finally, one subfield of artificial creativity, *algorithmic composition* (AC), refers to the process of composing music by means of formalisable methods. Interpreted literally, AC describes the use of algorithms to compose music, although this definition is quite fuzzy [31].

In fact, basic algorithms have been used to create music, or musical phrases, for centuries now. For instance, Guido d'Arezzo invented a system in the early 11^{th} century, assigning different pitches to each vowel in a body of religious texts. Other famous algorithmic composition techniques include, e.g., *musical dice games*, often attributed to the famous Austrian composers Joseph Haydn or Wolfgang Amadeus Mozart. In these games, the player could roll a die and, according to the result, pick a musical phrase, usually a bar, from a predefined corpus. Arnold Schönberg's *twelve-tone technique* [88] or John Cage's 1951 piano piece *Music of Changes* are other examples for algorithmic composition that stem from a pre-computer era.

Modern approaches for algorithmic composition make use of computer algorithms and artificial intelligence more often than not. The application of artificial intelligence in the field of algorithmic composition is well-studied [31, 60, 20], and especially in recent years impressive results using *artificial neural networks* were achieved. In particular, the *Illiac Suite* [48], composed in 1957, is considered one of the earliest approaches using modern computers for algorithmic composition. OpenAI's MuseNet [74] and the Music Transformer architecture [45] are two very recent examples and represent the enormous developments the field of AC has undergone, being able to generate music with impressive long-term structure.

In previous work [86, 87], we introduced the system PAUL, an algorithmic composer of two-track piano pieces using a *long short-term memory* (LSTM) neural network [43] for the lead track and an LSTM and *encoder-decoder*-based *sequence-to-sequence* neural network [21, 93] for the accompanying track.[1]

A particular feature of PAUL is that it allows for the specification of the desired *complexity* of the output pieces in terms of an input parameter. This parameter represents the *difficulty* of the generated pieces for a human pianist. Three such parameters exist, allowing for the generation of *easy*, *medium*, and *hard* pieces. Each subsequent difficulty class represents a piece that is more challenging to play than a piece assessed with a preceding rating. In a small-scale study, comparing the specified with the perceived complexity, a clear correlation between the two metrics could be observed. Although the results obtained for the lead track were of good, sometimes of great quality, the output produced by the sequence-to-sequence network was less satisfactory. Even though a clear relation between starting and stopping a note could be observed, the output lacked the necessary temporal structure to be considered a valid musical piece.

In order to address the shortcomings of PAUL, in this thesis, we introduce PAUL-2, an algorithmic composer realising the direct successor of PAUL. It makes use of an advanced transformer neural network architecture [97] and the enhancements introduced by Huang et al. [45] by the Music Transformer approach. A transformer architecture constitutes a current state-of-the-art model for deep learning in disciplines such as machine translation. Such architectures are capable of generating an output sequence of varying length based on a set of input sequences of possibly different domains. Instead of using recurrent neural networks that make use of, e.g., LSTM units, it solely employs an attention mechanism to draw dependencies between the input and output sequences.

Although PAUL-2 is the natural evolution of PAUL, the systems themselves are entirely different and do not use any common code base. Indeed, PAUL-2 makes use of state-of-the art approaches when it comes to algorithmic composition and is able to greatly outperform PAUL. In a similar fashion to PAUL, the task of PAUL-2 is to create two tracks of piano music, one representing the lead (played by the right hand), and one representing the accompaniment (played by the left hand). These two tracks should be

[1] PAUL is named after the well-known Austrian pianist Paul Badura-Skoda (6^{th} October 1927 - 25^{th} September 2019).

separate yet exhibit a dependence. Albeit the lead track is generated independently of the accompanying one, for the latter track, the lead one is fed to the neural network as an input, and thus the task of PAUL-2 is to generate an accompanying track that respects the structure and nature of the lead one.

The designated usage of PAUL-2 is to be part in an educational environment, teaching piano students how to sight-read. In such a context, students would be provided with musical phrases generated by PAUL-2. They would then be tasked to play the given prompts and would afterwards be evaluated regarding their performance. The nature of PAUL-2 would ensure that they cannot have seen (and thus perhaps memorised) the pieces beforehand, ascertaining that the students have to rely on their sight-reading capabilities alone. Furthermore, using the difficulty parameter and the results from the evaluation of the student's performance, a system implementing the described approach could ensure that the generated prompts matched the skill level of a student. Repeated inaccuracies in the playing of a student would result in a lowered difficulty score, whilst the opposite would raise it. This way, the system could guarantee that the users would be challenged yet not overwhelmed by the prompts.

PAUL-2 is separated into two distinct parts, viz. P2L and P2A. Here, P2L refers to the part of PAUL-2 responsible for generating *lead sequences*, whilst P2A stands for the component of PAUL-2 which generates *accompanying sequences*, based on a lead sequence input. While P2L is a sequence-to-sequence network, P2A is a multi-sequence-to-sequence network utilising two encoders to generate accompanying tracks. More specifically, both of these components use an encoder to process information about the desired difficulty of the output piece. This information is fed to the decoder that uses relative self-attention to attend to its own inputs. In the next step, the decoder attends to the output produced by the encoder, representing the difficulty parameter. For the P2A component, a second encoder is used to process the lead sequence that PAUL-2 should generate an output for. Lastly, we apply a mask to the output *logit functions*, removing any predictions that would correspond to invalid musical messages.

In contrast to PAUL, PAUL-2 is fully capable of generating valid accompaniments for an input lead sequence. Due to the more advanced technique of using an encoder to inject information about the desired difficulty, we were able to both remove the reliance on separate sets of weights for each of the difficulty classes and to greatly increase the amount of supported difficulty classes. In general, the output of PAUL-2 is of superior quality compared to that of PAUL. In a medium-scale user study evaluating the pieces generated by PAUL-2, a positive assessment regarding the quality of the output of PAUL-2 could be observed.

Since preprocessing the training data for PAUL-2 required a large number of operations on MIDI files [7], we implemented an accompanying tool, named S-Coda, to automatise different functionalities to process such files.[2] Recall that the MIDI protocol (standing

[2]The name "S-Coda" makes reference to the word "Coda", a musical passage at the end of a piece, as well as to the letter "S" in the surname of Paul Badura-Skoda.

for "Musical Instrument Digital Interface") is a common industry standard enabling the interoperation between different hard- and software components for music production [46]. S-Coda is realised as a Python library and serves as a framework for processing music compositions stored in the MIDI file format. Although, as mentioned, it was developed in conjunction with PAUL-2, it is actually a standalone tool and can be used independently of PAUL-2. There is a common notion in the data-science and machine-learning community, expressing the belief that 80 percent of a data scientist's time is spent preprocessing and organising data rather than training machine-learning algorithms. A joke about this fact goes like this, taken from TimeXtender[3]:

> Data scientists spend 80 percent of their time dealing with data preparation problems, and the other 20 percent of their time complaining about how long it takes to deal with data preparation problems.

The purpose of S-Coda is to reduce the time needed for subsequent machine-learning tasks that tackle similar problems. It provides the necessary tools to help data scientists deal with music data, especially data that is of similar nature to our data set. These tools include but are not limited to the *quantisation*, *concatenation*, *merging*, *splitting*, and *visualisation* of musical sequences.

This thesis is organised as follows. Chapter 2 provides the necessary background information on music theory, the MIDI protocol, and neural networks, particularly on the transformer architecture. In Chapter 3, we review some basic approaches to computer-aided algorithmic composition. In Chapter 4, we present PAUL-2, specifying its architecture and functionality. Moreover, we discuss the procedure of the data collection and training of the underlying network of PAUL-2, and provide an analysis and discussion of the results obtained with PAUL-2. Chapter 5, then, discusses S-Coda, detailing the algorithms used for processing the MIDI files and assigning difficulty values. Chapter 6 concludes the thesis with a brief summary and an outlook on future work.

[3] `www.timextender.com`.

CHAPTER 2

Prerequisites

In this chapter, we lay down the necessary background for our subsequent elaboration on PAUL-2. First, in Section 2.1, we discuss some basics of music theory, required for our discussion on PAUL-2. Afterwards, in Section 2.2, given that PAUL-2 relies on input in the MIDI file format, we briefly review the essential elements of the MIDI protocol. Finally, in Section 2.3, we summarise the key concepts of neural networks—in particular, we outline the central features of the transformer model, which underlies the architecture of PAUL-2.

For a more thorough discussion on the topics discussed in this chapter, we refer to the works of Benward [10], Aldwell [1], and Laitz [52] concerning music theory, which Section 2.1 is also based on, to Huber [46] regarding MIDI, and to the well-known textbook on artificial intelligence by Russell and Norvig [83] about neural networks.

2.1 Music Theory Foundations

2.1.1 Properties of Sound

Let us start with a quote from the book by Benward [10]:

> The basic materials of music are sound and time. [...] Sounds are used to structure time in music. Time occurs in the duration of the sounds and the silences between sounds.

The complex interaction of these two aspect is what defines musical works. Although *acoustics* (the science of sound) plays a central part when it comes to sound (e.g., vibration, compression, and rarefaction of the air), we will cover this aspect only superficially, instead we refer to the mentioned works above for more information on the topic.

We attribute four character-defining properties to sound, these being *pitch*, *intensity*, *duration*, and *timbre*.

Arguably the most important of these four properties is the pitch, defining the highness (or lowness) of a sound. Pitch is defined by the *frequency* of a sound, which in turn refers to the number of vibrations (usually measured on a per-second basis) of the sound carrier. The fewer vibrations per second, the lower the resulting sound will be, while a greater amount of vibrations per second will result in a higher sound. The human ear has a hearing range of about 20 to 20,000 hertz (Hz), meaning that it can pick up sounds whose amount of vibrations per second lie within the given interval. In comparison, dogs can hear up to 45,000 Hz, while the hearing range of some whales reaches up to 123,000 Hz [30]. *Tone* describes a sound of definite pitch, and is often used interchangeably with the notion of a *note*, no unambiguous definition of either of the terms is given. We will discuss the concept of notes in more detail in Section 2.1.2.

The intensity refers to the amplitude of the sound wave, and is heard as the loudness or softness of a pitch. This energy, affecting the medium of the sound, can be measured and is given in the unit of decibels (dB). Often, composers use Italian words as instructions on which gradation of intensity to use when performing the piece. For instance, *pianissimo* tells a player to perform very softly, resulting in a sound of about 40 dB, while *fortissimo* is the exact opposite, instructing the player to perform very loudly, at about 100 decibels.

The duration of a sound is an intuitive property, as it refers to the length of time a sound, pitch, note, or tone lasts. The term *pulse* refers to undifferentiated and equally spaced clicks, accentuated pulses (e.g., played more strongly or weakly) are referred to as *beats*. When dealing with patterns of durations, we use the terms *meter* or *rhythm*. Meter generally refers to a grouping of beats into (recurring) patterns, thus defining a sequence of accentuations. An example of a meter would be a group of one strong pulse, followed by two weak pulses. A rhythm on the other hand refers to "the ever-changing combinations of longer and shorter durations and silence that populate the surface of a piece of music" [52]. It is a pattern (of variable length) of uneven durations.

Lastly, timbre refers to the tone quality or colour of a sound. A pianist and a guitarist can both perform the same piece, playing the exact same notes, yet the difference between the used instruments will be apparent due to the timbre of their sound, allowing listeners to distinguish between the instruments. Timbre of a produced sound depends on many factors, e.g., the material of the instrument, or the manner in which the sound-producing element was excited. Timbre is the result of the *harmonic series*, a physical phenomenon resulting when a body of matter vibrates not only as a single unit, but also in smaller sections. A string for example vibrates in halves, thirds, and so on. These multiple vibrations simultaneously produce sounds of different pitches, which are called *partials* or *harmonics*. The combination of these harmonics constitute a single musical tone. Since the first and lowest partial (called the *fundamental*) is also the loudest, it is identified as the tone's pitch by the ear. Although the rest of the produced harmonics are not independently distinguished by the ear, the combination of all the vibrations constitute the timbre of the sound.

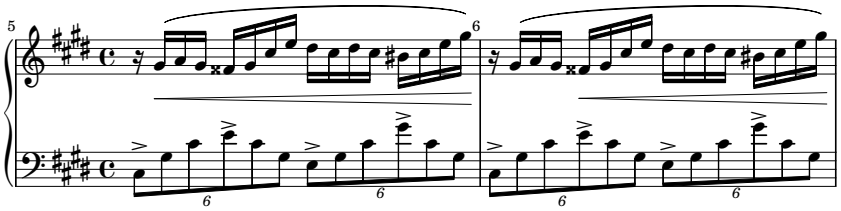


Figure 2.1: Bars 5 through 6 of Chopin's *Fantaisie-Impromptu*.

2.1.2 Music Theory and Music Notation

In a similar way written language is able to record spoken language, music notation is able to do the same for music. In some sense, it is much more complicated than written language, since multiple musical properties are expressed at the same time, using only a few symbols. It can thus be much more precise when it comes to music. For instance, we are able to convey information about pitch and duration of sound at the same time, using a notational element commonly referred to as a *note*.

Figure 2.1 shows an excerpt of Chopin's Opus post. 66, titled "Fantaisie-Impromptu". We will use this excerpt, called a musical *score*, to explain some core concepts of music notation. For now, we note that each of the dots in or between the horizontal lines corresponds to a single note. We used the freely available music notation software LilyPond [84] to create score notations for this thesis.

Staves

A *stave* (or *staff*) consists of five (numbered from bottom to top) equally-spaced horizontal lines upon which musical symbols are placed. Figure 2.1 contains two staves, corresponding to two independent musical tracks. These staves can, e.g., correspond to different instruments, or designate the lead and accompanying track for a piano piece. The vertical line between two or more staves indicates that the music is to be played simultaneously. Additionally, groupings of different tracks (e.g., string instruments in an orchestra) can be achieved using brackets to the left of the staves, although these only serve as visual aids and have no impact on the notated music itself. In a similar fashion, a brace can be used to indicate that two or more staves belong to a single instrument. Figure 2.1, for instance, consists of two staves that are joined using such a brace. The score denotes piano music, each of the tracks is to be played with one hand. Such a grouping of a treble and a bass stave make up a new structure, the *grand stave*, which is most often associated with piano music.

Pitch Classes

In our music system, we use seven letters, A through G, to refer to seven pitch classes. We label notes that are spaced by multiples of 8 using the same letter, since we consider

Figure 2.2: Piano keys and the corresponding note names, taken from Benward [10].

the pitches of these notes to be strongly related. Figure 2.2 shows a piano keyboard where each key is labeled with the corresponding note name. Observe that the labeling restarts after 8 keys.

Often notes spaced exactly one or more *octaves* apart are considered to be of similar pitch, only differing in highness or lowness. The frequencies of notes spaced exactly one or more octaves apart are multiples of each other, the frequency of a note is exactly double that of the note an octave down. For instance, if the frequency of the note corresponding to the first key labeled with "A" in Figure 2.2 is 440 Hz, the frequencies of the second and third key labeled with the same letter are 880 Hz and 1760 Hz, respectively.

In order to differentiate between notes labeled using the same letter, we often append the specific octave of the note to its designation. For instance, the lowest (leftmost) key on a standard-sized piano keyboard (consisting of 88 keys) corresponds to the note A0, the highest (rightmost) key to C8. Commonly C4 is referred to as the "middle C", since it is roughly at the center of a standard-sized piano keyboard.

Clefs

Since the staves alone do not carry information about which notes their lines refer to, we use *clefs* to denote the absolute height of the notes placed on them. Clefs are placed at the beginning of a stave and define the labels and octaves of the lines and spaces of the stave. Several different clefs exist, each defining a different assignment of lines and spaces to note names and octave heights. Two of the most common clefs can be seen in Figure 2.1. The treble track contains a *treble clef* (or *G clef*), which is an ornate version of the letter G. It starts below the first line and terminates around the second line, marking the latter as the position of G4. In a similar fashion, the *bass clef* (or *F clef*) represents an ornate version of the letter F, and can be seen on the bass stave of Figure 2.1. It starts just below the second line and terminates on the fourth line, marking the latter as the position of F3.

Notes whose pitches fall outside the range of the staves use *ledger lines*. These additional lines are added above or blow the stave, and accommodate only a single note. Figure 2.3 shows the usage of these ledger lines by comparing the treble and bass clef. Here, 15 different notes are visualised, once using a G clef, and once using an F clef. We labeled the notes to emphasise the fact that each pair of horizontally aligned notes represents the exact same pitch. Staves are often used to improve the readability of the musical score.

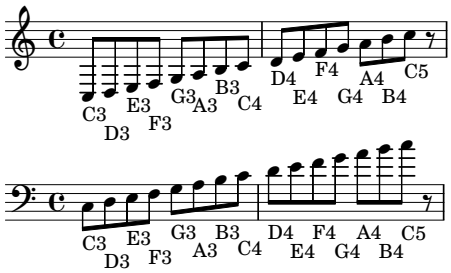


Figure 2.3: A visualisation of different representations of the same notes, using clefs and ledger lines.

Piano music for instance lends itself to the usage of the treble and bass clef, since the range covered by two staves using these staves corresponds roughly to the range covered by a pianist's hands when placing them around C3 and C4 respectively.

Unlike the treble and the bass clef, the *C clef* does not have a fixed position on the stave. Rather, different positionings of the C clef are referred to using designated names, and shift the assignment of note names and octave heights proportionally to the clef's position. For instance, an *alto clef* is a C clef that designates the third line of the staff as the middle C, while the *tenor clef* does the same with the fourth line.

Different clefs are often used for different instruments, e.g., the treble and bass clefs are often used in conjunction with piano music, the alto clef in music for violas, while the tenor clef is often used in music written for the cello, the bassoon, or the trombone. Further clefs, such as the *soprano*, *mezzo soprano*, and *baritone clefs* are used less often. Each of these less frequently used clefs acts in a similar way as the alto and tenor one, designating a particular line of the stave as middle C.

Note Pitches and Intervals

Although in Figure 2.3 notes are placed on each of the five lines of the stave and in each of the four spaces, there are some notes not represented in this figure. This is also the case for Figure 2.2, where only white keys are labeled using note names.

In music notation, we use *accidentals* to raise or lower a pitch by a *half tone*, also called a *half step* or *semitone*. Half tones are the smallest step size representable using the common notation system, although there exist even smaller differentiations of pitch, e.g., *cents*. In the standard tuning system, all semitones are of 100 cents in size.

Using these accidentals, we can notate half tones such as C♯ (C-sharp), D♯ (D-sharp), or B♭ (B-flat) and A♭ (A-flat). A *sharp* accidental, represented using a number sign, raises the value of a note by a half tone, while a *flat* accidental, represented using a small "b", lowers its value by the same amount. Using the *natural* accidental, previous accidentals are canceled, and the corresponding note returns to its original pitch. The *double sharp* or

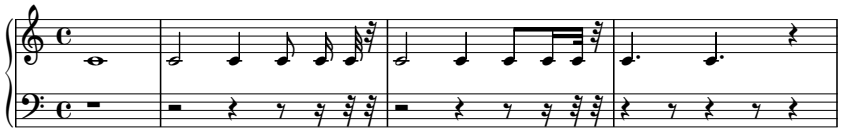


Figure 2.4: An assortment of different notes and rests of different values.

double flat accidentals serve the same purpose as their single-type counterparts, doubling the amount of half tones they modified by. By employing accidentals, we are able to utilise 12 different note classes.

We note that *enharmonic equivalents* are tones of the same pitch, which are labeled using different note names. For instance, if one raises C4 by a half tone to obtain $C^{\sharp}4$, this refers to the same pitch as $D^{\flat}4$, although in one case we speak of C-sharp, while we refer to the other tone as D-flat.

An *interval* is the relationship and distance between two notes. We have already mentioned the concept of an *octave*, which refers to two notes that contain 8 *diatonic notes* between them. Diatonic notes refer to notes with different letter names. Other intervals include the *second* (containing two diatonic notes), the *third*, *fourth*, *fifth*, *sixth* and *seventh*, which conform to the same pattern. We use different prefixes such as *minor*, *major* and *perfect* to differentiate between intervals containing the same amount of diatonic notes.

Note Durations

As stated in Section 2.1.1, one of the core properties of sound is its duration. We notate the duration of a musical element by modifying the appearance of its symbol.

In music notation, *notes* are represented using a combination of noteheads, which are oval in shape and positioned on or between the stave lines, stems, which are thin vertical lines connected to the noteheads, and flags (or beams), which extend from the stems. Both the direction and length of the stems and the usage of beams instead of flags are only of visual nature.

The duration of a note is given by the appearance of its notehead and the amount of flags on the stem. The *value* of a note determines its relative duration, dependent on the *time signature* and the *tempo* of the piece. A *whole note* consists of a hollowed-out notehead and does not have a stem. A *half note* lasts for half the amount a whole note does and consists of the same type of notehead while adding a stem. A *quarter note* continues the pattern, lasting for one fourth the amount a whole note does, while it consists of a solid notehead and a stem. Each subsequent note of smaller note value halves the amount of time it lasts, while adding a flag to its stem to differentiate it. Instead of drawing flags on each of the notes, beams can be used to connect a group of notes together. Here, the number of beams used to connect two or more notes is equivalent to the number of flags a note would otherwise have.

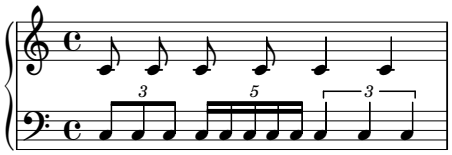


Figure 2.5: An example of irregular divisions of notes, containing two triplets and a quintuplet.

Figure 2.4 shows an assortment of notes and rests of different note values. The lead track contains four sections called *bars*. In this example, each bar can fit exactly one whole note. The first bar contains exactly one such note. The second bar contains (in this order) a half note, a quarter note, an eight note, a sixteenth note, a thirty-second note, and a thirty-second rest to fill out the bar. The third bar contains notes of exactly the same values, using beams instead of flags. Bars one through three of the accompanying track contain *rests* of the same value as their note counterparts. A rest consumes the same amount of time a note of the same value would, without actually playing any note.

The fourth bar contains two *dotted* notes in the lead track. Adding a dot to the right side of a note or a rest increases its duration by half of its original value. In Figure 2.4, two dotted quarter notes are shown. For each of the notes, their total duration is equivalent to the sum of the durations of a quarter note and an eight note, which is demonstrated in the accompanying track. Here, a quarter rest and an eight rest are needed in order to match the amount of time taken up by the dotted note. We note that notes and rests can be dotted more than once. In this case, each dot adds half of the duration the previous dot did.

A *tie* can be used to connect two notes of the same pitch. In this case, a curved line is drawn above or below the notes, indicating that the resulting construct is to be seen as a single note with its duration equal to the combined durations of the connected notes. The dot can be seen as a shorthand of (multiple) notes connected using a tie. Furthermore, ties can be used to explicitly model the concept of multiple *voices*. Voices refer to segments that have their own rhythm, independent of other voices on the same stave. Multiple voices can occur at the same time. No special notation is used to specify a specific voice, notes of different durations are simply stacked to achieve the desired outcome.

Notes are often subdivided into groups of equal notes. In this case, the value of each of the grouped notes is equivalent to the value of the subdivided note divided by the number of notes in the group. For instance, a quarter note may be subdivided into three notes. In this case, each of the three notes of the group has a value of one third of the quarter note. We refer to such divisions as *tuplets* or *irregular divisions* and *irregular subdivisions*, as they allow for rhythms not otherwise possible. Figure 2.5 shows an assortment of subdivisions of quarter notes. The first grouping is referred to as a *triplet*. Using this construct, the three adjusted eights fit into the same amount of time two unmodified

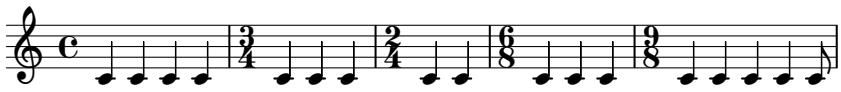


Figure 2.6: Time signatures and resulting capacities of bars.

eights do. The second grouping is a *quintuplet*, five sixteenths fit in the space of four normal sixteenths. The third grouping represents a quarter triplet.

Bars, Time Signatures, and Key Signatures

Notes and rests are the smallest compositional units. We use *bars* (or *measures*) to structure compositions. Bars are marked using vertical lines on the staves. For example, Figure 2.1 contains two separate bars. Each bar has a fixed capacity of note values that it can hold. Furthermore, bars must be filled exactly to capacity, if the melody prematurely ends rests musts be used to fill out the bar.

In order to define the length of all subsequent bars, *time signatures* (or *meter signatures*) are utilised. Time signatures are defined at the beginning of a bar. If no time signature is given for a specific bar, the time signature of the previous bar is assumed. Time signatures consist of an upper and lower numeral. The lower numeral is to be interpreted as a fraction of a whole note, e.g., a "4" corresponds to a quarter note, while an "8" corresponds to an eight note. The upper numeral defines by which amount the note value defined by the lower numeral should be multiplied in order to obtain the capacity of the bar.

For instance, a time signature with an upper numeral of "4" and a lower numeral of "4" corresponds to a bar with a capacity of four quarter notes. This $\frac{4}{4}$ time signature is also known as *common time* and can be abbreviated using the c-shaped symbol seen in the first bar of Figure 2.6. One can observe that four quarter notes fit into the first bar of common time. Since the second bar has a $\frac{3}{4}$ time signature, only three quarters fit into a bar. This time signature is often used for waltzes. This concept works analogously for the subsequent bars of time signatures $\frac{6}{8}$ and $\frac{9}{8}$.

Finally, *key signatures* work in a similar fashion as time signatures do. They define a key signature for all subsequent bars. If no key signature is given (as is the case in, e.g., Figure 2.5), the key of C major is assumed. Key signatures give the necessary sharps or flats that hold for an entire bar, rather than a single note. Using key signatures, visual clutter can be reduced, as we can reduce the amount of explicitly visualised accidentals.

Figure 2.7 shows an example where using a key signature can drastically improve the readability of the score. Here, the C♯-major scale is given twice; once without using any explicitly given key signature, once using the C♯-major signature. Using the key signature, all accidentals used in the following two bars are covered, and no additional

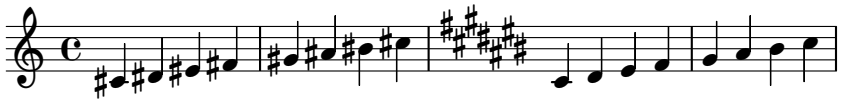


Figure 2.7: C♯-major scale, with and without key signature.

accidentals need to be inserted. For an overview of all possible key signatures we refer, e.g., to the textbook on music theory and practice by Benward [10].

2.2 Elements of the MIDI Protocol

The *Musical Instrument Digital Interface* (or *MIDI* for short) [6] is a digital protocol that enables communication between multiple actors in a music-oriented environment. These actors include electronic hardware and software instruments, performance controllers, digital mixers, and other compatible devices [46]. Today, MIDI is widely used by music artists, composers, stage technicians, and other personnel working in the music industry.

2.2.1 A Brief History of MIDI

Before MIDI became the widely-accepted standard that it is today, synthesisers made by different manufacturers often had no way of communicating with each other. In 1981, Smith and Wood published a paper on the *Universal Synthesizer Interface* [91], allowing for a communication between equipment made by different companies. Over the course of the next two years, a panel of representatives of some of the major electronic instrument manufacturers developed the first version of the MIDI specification [6], based on the work of Smith and Wood.

In 1985, the *MIDI Manufacturers Association* was officially established as a non-profit organisation for promoting the MIDI technology. Today, the association's executive board consists of members of companies such as Yamaha, Microsoft, and Focusrite, among others.

For more than 30 years, no modifications to the MIDI protocol were deemed necessary. At the 2019 National Association of Music Merchants (NAMM) convention, work on an updated version of MIDI was announced. In the fall of the same year, MIDI 2.0 [7] was announced, incorporating, e.g., an increased performance resolution and bidirectionality [46].

2.2.2 The MIDI Protocol

This section will provide a brief overview of the MIDI 1.0 specifications. For more information, we refer to the specification document itself [6], as well as to the book by Huber [46], upon both of which this section is based on.

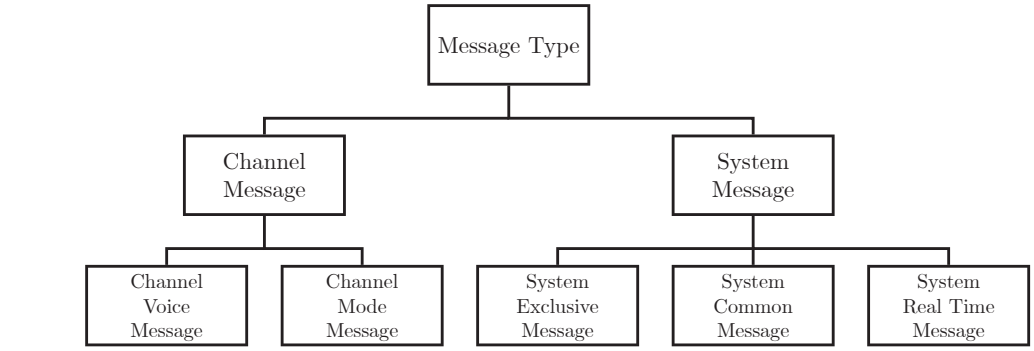


Figure 2.8: The different types of MIDI messages.

Even though the specifications of MIDI 2.0 are publicly available as of the time of writing, they are based on the MIDI 1.0 specifications. MIDI 1.0 is not deprecated but rather serves as groundwork for the improvements made by MIDI 2.0 [7]. Due to this fact, studying the original specifications is a prerequisite to understand how MIDI works in general.

The MIDI Message

Communication between two or more devices supporting the MIDI protocol is based on transferring MIDI messages. Traditionally, these are serially transmitted at a speed of about 31 kilobits (around 4 kilobytes) per second. Furthermore, with cables conforming to the MIDI 1.0 standard, data can only travel unidirectionally; a second cable is needed to communicate bidirectionally.

MIDI messages consist of groups of 8-bit-long information. These groups contain instructions for specific actors in the MIDI chain. Two types of information bytes exist:

(i) The *status byte* identifies the purpose of the data byte by which it is followed. Furthermore, information about which MIDI channel is used is encoded. Each status byte consists of eight bits, in which the most significant bit is always set to 1.

(ii) The *data byte* carries the contents of the message. The receiving actor can parse the data byte and extract the value of the event given by the status byte. Each data byte consists of eight bits, in which the most significant bit is always set to 0.

Due to the fact that for both of the message types the most significant bit is set to a fixed value, the only information carried by this bit is the type of message itself. This leaves 7 bits of information for both of the message types.

Figure 2.8 shows the different types of MIDI messages. MIDI messages can be sorted into one of two main groups: *channel messages* include a channel designation in their

status byte, while *system messages* do not. Thus, channel messages only apply to a specific channel, while system messages are not channel specific and are instead globally transmitted to every actor in the chain. Both channel and system messages can further be categorised. In this section, we provide information only about *channel voice messages*; for more information, we refer to the specification of MIDI 1.0 [6].

For channel messages, the four least significant bits of the status byte correspond to the *channel* of the MIDI messages. Since a total of 2^4 different combinations are possible with four bits, a total of 16 different channels is supported by the MIDI 1.0 specification. These channels can be used to structure and categorise the actors in the chain, e.g., a device set to listen to channel 1 exclusively will ignore all messages that are sent through other channels. This behaviour allows for control of different aspects of an actor using a single source, e.g., a MIDI controller can include a keyboard with which a piano synthesiser is controlled, while its pads control a drum machine. Although in both cases the instruction to play a note is sent, both the piano synthesiser and the drum machine will ignore messages not meant for them.

Channel Voice Messages

Channel voice messages make up the largest amount of messages sent between actors in the MIDI chain. They are used to transmit real-time performance data, e.g., information about when a piano key is pressed and released. They consist of a status byte, followed by two data bytes, requiring around 960 microseconds for transmission given the transfer rate of around 4 kilobytes per second.

Sending a large number of messages simultaneously can sometimes result in a slight yet audible delay. In order to remedy this effect, the *running status* can be employed. In this mode, status bytes stay in effect until overwritten, removing the need of subsequent messages having to include a status byte if they would have used the same one as the previous message.

Seven different types of channel voice messages exist. We will give short descriptions of the two types relevant for this thesis.

Note-On Messages. Note-on messages indicate the start of a MIDI note. This kind of messages are generated, e.g., when pressing a key on the keyboard of a piano MIDI controller. Note-on messages consist of a status byte, a data byte specifying the note number from a range of 0 to 127, and a data byte specifying the *velocity* of the event from a range of 0 to 127.

The interpretation of this velocity byte is left up to the receiving party. Often, the velocity controls the intensity of the sound, e.g., pressing a key on the controller with higher speed results in a higher velocity, which in turn results in generating a louder sound. If the sender is not able to capture velocity information, a standard value of 64 is sent.

Note-Off Messages. Note-off messages indicate the end of a MIDI note. Notes started using the note-on messages will continue to play until the corresponding stop message is received. Note that notes can be stopped using either a note-off message or sending a note-on message with a velocity of 0. An advantage of using the latter approach is the fact that the running status can be employed. The MIDI 1.0 specification states that any device implementing the standard must be able to support both of these options, and treat them equally.

Note-off messages consist of a status byte, a data byte specifying the note number of the note to stop in the same 0 to 127 range, and a data byte specifying the *release velocity* value. This release velocity indicates the speed with which the key was released. If the sender is not able to capture the velocity information, a standard value of 64 is sent.

2.2.3 MIDI Files

In order to preserve MIDI performance data, the MIDI 1.0 specification [6] specifies a file storage system. It states that "the purpose of MIDI files is to provide a way of interchanging time-stamped MIDI data between different programs on the same or different computers" [6]. In comparison to audio recordings, MIDI files preserve the performance itself, e.g., which keys were played, and in which manner they were pressed.

MIDI File Specification

These MIDI files consist of *chunks*. Each chunk consists of a 4-character ASCII type definition, followed by 32 bits indicating the length of the chunk. This length refers to the number of bytes of data in the chunk, excluding the bytes needed for the type and length representation. Two types of chunks exist; *header* chunks and *track* chunks. In the header chunk, general information about the MIDI file is stored, while track chunks contain performance data.

Three different variants of MIDI files exist, these being format 0, format 1, and format 2 files. Format 0 files consist of a head chunk, followed by a single track chunk, thus only allowing for a single track. With format 1 and 2 files, the inclusion of multiple track chunks is possible. With format 1, these track chunks correspond to a collection of tracks from a single musical performance, while with format 2, each of the tracks represents a unique piece, similar to a collection of type 0 tracks.

MIDI files may contain *meta-events*, which provide a way to store more types of information about the performance. These events are marked using an initial byte set to `FF`. Meta-events support variable-length data such as text by using a length argument in a similar fashion as the chunk-length identifier, specifying how many data bytes follow the meta-event. Using meta-events, data such as copyright notices, track names, lyrics, or other text data may be stored. Furthermore, both time signatures and key signatures can be specified using messages of this type.

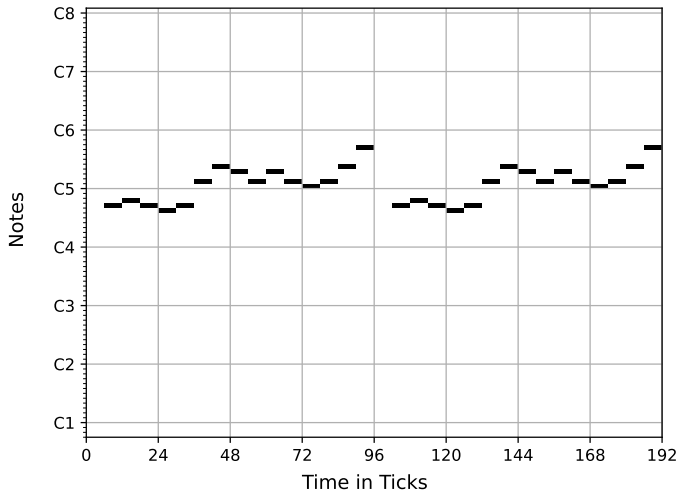


Figure 2.9: Piano roll representation of bars 5 through 6 of Chopin's *Fantaisie-Impromptu*.

Event timings in MIDI files are given using *delta time*, which specifies the amount of time passed since the last event. Each event must include such a delta time; if no time passed since the last event, it must be set to a value of 0. This delta time is given in the unit of *ticks*.

The specification describes two distinct methods of defining the length of a single tick, both dependent on an entry in the header chunk. This entry is two bytes long, with the first bit signaling the method that is to be used.

With the first method, 15 bits in the *division* part of the header chunk correspond to the *ticks per quarter-note* value, defining how many ticks an unmodified quarter note should last. This value is also referred to as *parts per quarter note* or *pulses per quarter note* (PPQN). A PPQN of 24 for example would allow for a resolution of down to sixty-fourth triplets, as the delta time between the on and off messages of the note would be 1.

With the second method, the ticks would correspond to subdivisions of a second. Here, roughly two bytes are used for specifying the amount of *frames per second*, and the resolution of ticks per frame. For instance, a value of 25 frames per second with a resolution of 40 would correspond in one tick lasting a single millisecond.

Piano Roll Representation

Although not a part of the MIDI specification, often *piano roll representations* are used to visually represent MIDI files. In fact, all of the major digital audio workstations (DAWs), such as Ableton Live[1] or Cubase[2], use some form of piano-roll visualisation to represent

[1] `https://www.ableton.com/en/live/`.
[2] `https://www.steinberg.net/cubase/`.

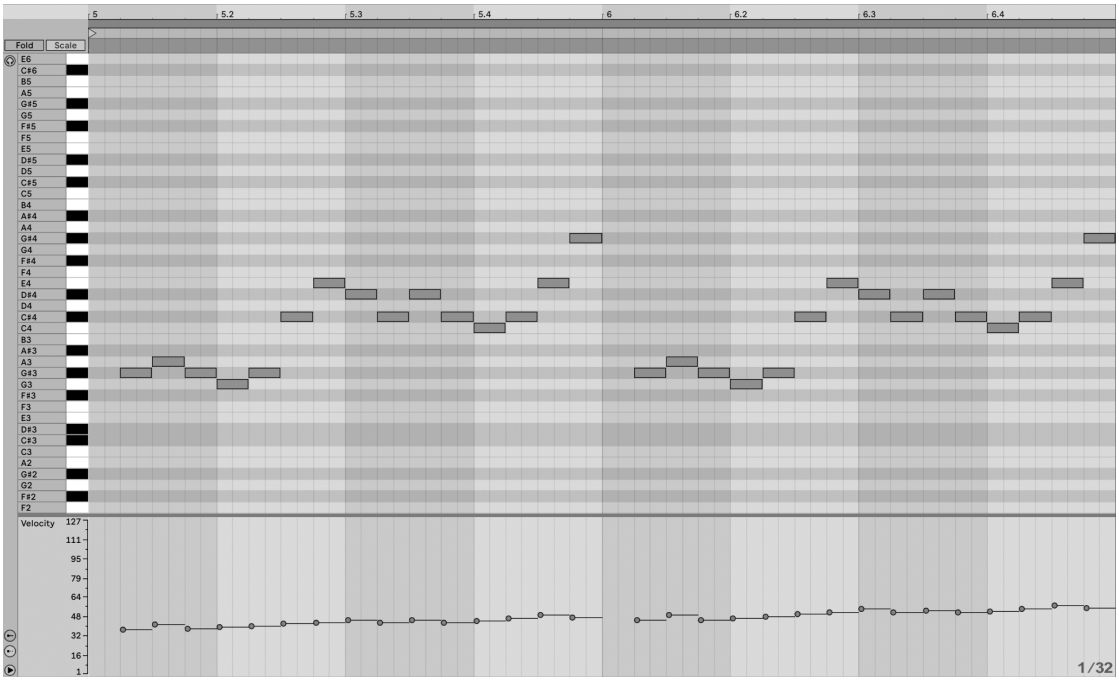


Figure 2.10: Ableton Live's piano roll representation of bars 5 through 6 of Chopin's *Fantaisie-Impromptu*.

any number of MIDI tracks.

Figure 2.9 shows a sample piano roll of Chopin's *Fantaisie-Impromptu*, also depicted in score notation in Figure 2.1. Each of the black rectangles represents a note. The width of the rectangles represents the length of the note, e.g., a note that spans from 24 to 48 ticks is a quarter note, assuming a PPQN of 24. The height of the rectangle always covers a single note class only, since no single note can cover two or more pitches. The position on the horizontal axis gives the moment in time the note is played, while the position on the vertical axis represents the pitch of the note.

With some piano roll representations, the opacity of the rectangles is related to the velocity of the note. Although S-Coda is able to generate such piano rolls, we disabled the feature for all of the piano-roll representations used throughout this thesis. We argue that, since PAUL-2 does not generate any velocity information, representing this information only for some pieces could cause unnecessary confusion.

Figure 2.10 shows a piano roll of the same piece generated by the DAW software Ableton Live. Due to the fact that the piano roll doubles as an interface to edit the MIDI data, a playable keyboard is shown on the left side. Users are able to modify, rearrange, or delete notes directly from the piano roll representation. Furthermore, velocity information is shown at the bottom of the piano roll. Ableton Live does not represent velocity through

the opacity of the notes, but rather in a separate section. In a similar fashion to the notes themselves, users of the DAW are able to edit velocity information directly from the piano roll interface, which would not be easily possible if they used the opacity visualisation.

We will use piano roll representations in conjunction with score representations throughout this thesis. Although they are similar, and it is in fact possible to construct a score representation from a piano roll representation, we argue that they serve different purposes; piano roll representations serve as a quick way of showing the structure of MIDI files, while score serves as an instruction to anyone wanting to perform the piece.

2.3 Background on Neural Networks

The field of artificial intelligence is extensive and *artificial neural networks* (ANNs), or *neural networks* (NNs) for short, show especially good promise when it comes to algorithmic composition, as we noted already in Chapter 1 and as we will discuss in more detail in Chapter 3.

As their name suggests, ANNs draw from concepts in the field of neuroscience, especially from the idea of a *neuron*. These nerve cells found in the brain form a large number (between 10 and 100,000) of connections with other neurons, at intersections called *synapses*. This allows for electrical signals to be propagated through a network of these cells. The electrochemical signals passed from neuron to neuron control the short-term brain activity and can result in long-term changes regarding the structure of the interconnected neurons. The networks made from these cells are thought to form the basis of the learning process [83]. Current estimates on the amount of neurons in the human brain place the number at about 85 to 120 billion [42].

Early attempts at replicating such a biological network date back almost a century. McCulloch and Pitts [59] devised a mathematical model of a neuron in 1943. Since then, substantial advancements have been made in the field of artificial neural networks, leading to the field of *computational neuroscience*. The fact that modern artificial neural networks lend themselves well to parallel computation and are able to learn complex input function makes them a powerful machine-learning tool [83].

In this chapter, we give an overview of how these networks work and discuss the *transformer* architecture [97] in more detail. More specifically, Section 2.3.1 covers the foundations regarding basic ANNs, *recurrent neural networks*, and *encoder-decoder networks*. Then, in Section 2.3.2, we cover transformer networks which constitute the current state of the art as far as machine translation approaches are concerned.

2.3.1 Basic Neural Network Architectures

This section mainly follows the exposition of Russell and Norvig [83] and Goodfellow, Bengio, and Courville [38].

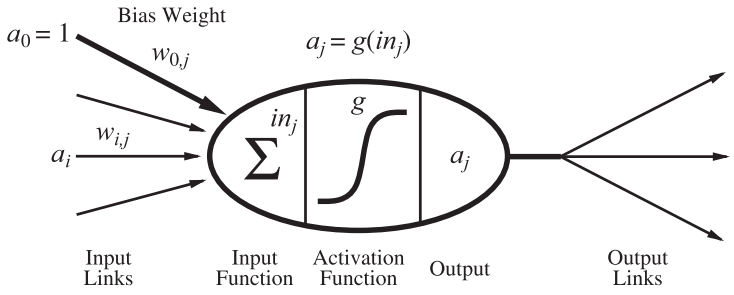


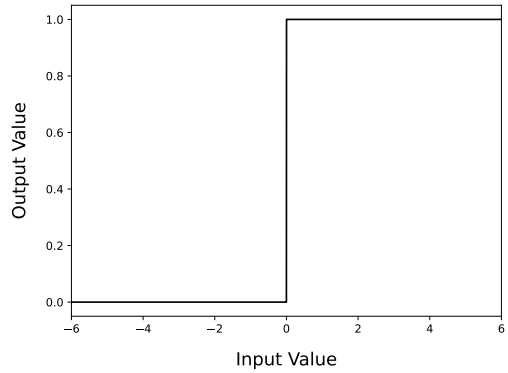
Figure 2.11: A mathematical model of a neuron, taken from Russell and Norvig [83].

Neural networks are composed of special units, the neurons. Figure 2.11 shows such a neuron, which we will now discuss in more detail. Neurons receive input values either from previous units or a global input, which are then scaled and summed up. Following this initial calculation, an *activation function* is applied to the weighted sum. Several different activation functions exist, popular ones include

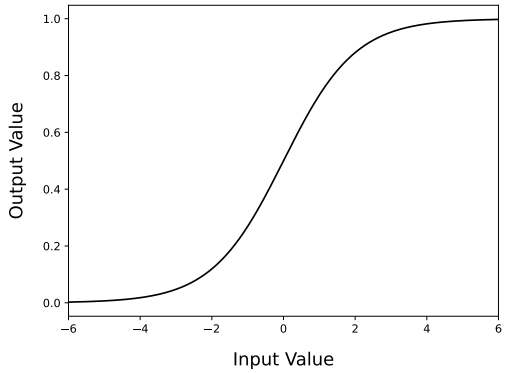
- *threshold functions* (cf. Figure 2.12a) that use a hard cut-off at a certain value,
- *sigmoid functions* (cf. Figure 2.12b) that are fully differentiable, or
- *rectified linear functions* (ReLU), shown in Figure 2.12c, that are computationally cheap compared to sigmoid functions.

Usually, these neurons form a *network*, meaning that each neuron is receiving input values from multiple neurons, while passing its output value to several other neurons as well. In a *feed-forward network*, these neurons can be separated into distinct *layers*. Here, no connections between neurons of the same layer exist, and neurons of a specific layer only receive input from units of the previous layer, while passing their output values only to units of the next layer. Such networks can be represented as a directed, acyclic graph. Feed-forward networks are the most basic kind of neural networks, since their output is only a function of the immediate input, no internal state other than the weights of the network exists, and thus no such state has to be considered for the calculation of the result.

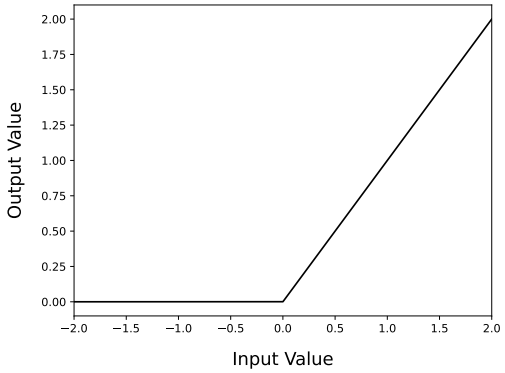
For *multilayer feed-forward neural networks*, i.e., networks that consist of at least one hidden layer, the *universal approximation theorem* [44] holds, which states that multilayer feed-forward networks with at least one hidden layer using arbitrary many squashing functions can arbitrarily approximate any Borel measurable function from one finite dimensional space to another, providing sufficiently many hidden layers are available. Here, by a *squashing function* one understands a mapping $f : \mathbb{R} \to [0, 1]$ such that (i) f is non-decreasing, (ii) $\lim_{x\to\infty} f(x) = 1$, and (iii) $\lim_{x\to-\infty} f(x) = 0$.



(a) A threshold function.



(b) A sigmoid function.



(c) A rectified linear (ReLU) function.

Figure 2.12: Three different activation functions commonly used for neural networks.

In contrast to feed-forward neural networks, *recurrent neural networks* (RNNs) [82] are well-fitted to deal with sequential data such as, e.g., text or audio data. Rather than process a single input, they can handle input sequences of variable length. PAUL-2 does not make use of RNNs and rather solely relies on the *attention* mechanism which we will discuss in Section 2.3.2. For more information about RNNs, cf., e.g., the book by Goodfellow, Bengio, and Courville [38].

Recurrent neural networks are able to both map input sequences to a fixed-size vector and map fixed-size vectors to sequences [38]. In order to map sequences of variable length to other sequences of variable (but not necessarily the same) length, we can use *sequence-to-sequence* models, also referred to as *encoder-decoder*, introduced independently by Cho et al. [21] and Sutskever, Vinyals, and Le [93].

Figure 2.13 shows such an encoder-decoder architecture. The idea behind this architecture is relatively simple: an *encoder* takes a variable-length input sequence $x_0, \ldots, x_n$ and

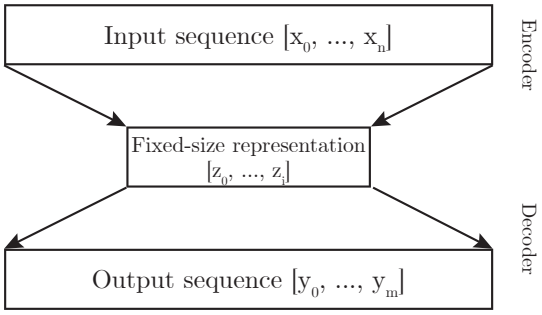


Figure 2.13: Architecture of an encoder-decoder neural network.

converts it to a fixed-size representation $z_0, \ldots, z_i$. In the next step, a *decoder* takes this fixed-size representation as input and generates from it the output sequence $y_0, \ldots, y_m$.

One of the advantages of using such an architecture is the fact that the lengths of the input and output sequences can differ, which is not possible with the architectures introduced before. During the training process, both of the networks can be trained in a joint fashion [38].

Neural networks are a viable tool when it comes to *machine learning*. Provided with a large enough set of labeled training data, neural networks are able to generalise to unseen samples. This learning process is done using the *backpropagation* algorithm, where the ground truth is compared with the prediction made by the network. The differences between these values is then used to update the weights of the network in order to improve its future predictions. In order to enhance the performance of this training procedure, which is extremely computationally heavy, often a *minibatch* (also referred to as a *batch*) of data is processed at once, averaging the differences of all the samples in the batch (cf. the book of Russell and Norvig [83] for more information on the topic). Feeding all the data to the network a single time is referred to as an *epoch*. Usually, several of these epochs are conducted in order to improve the performance of the network.

A core concern when dealing with neural networks is avoiding to *overfit* (or *underfit*) the model. When we train a model to the point that it simply remembers the training data, rather than learn the underlying structure of the dataset, we speak of *overfitting*. When a model has not been shown enough data and it is still able to improve its performance when being trained, we speak of an *underfitting model*. The same can hold for the *capacity* of a model. If the wrong hyperparameters (e.g., amount of layers in a network, amount of units per layer, ...) are chosen, the model may not be able to learn the structure of the dataset.

In order to measure the performance of a model, we commonly split the dataset into two (or more) partitions, forming the *training data* and the *validation data*. During the training process, only the training data is used to train the network. The validation data is solely used to evaluate the performance of the network, e.g., after completing an epoch.

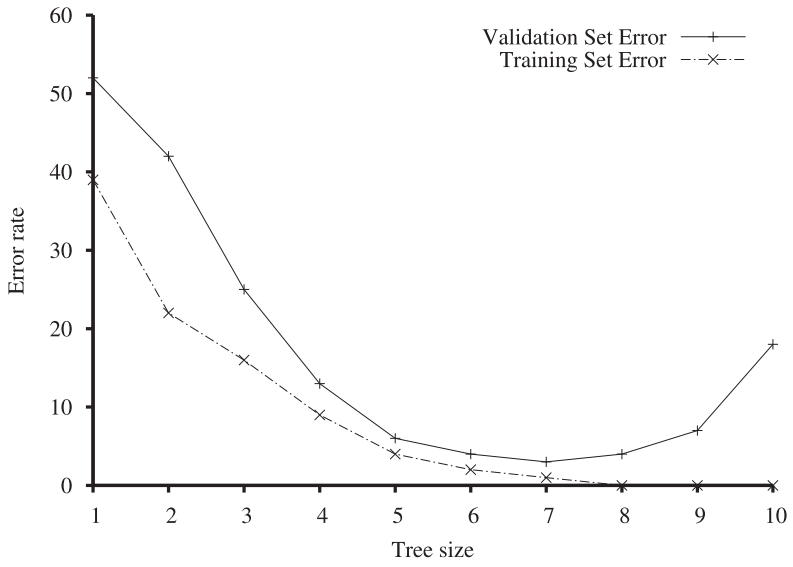


Figure 2.14: Error rates of different decision trees measured on training and validation data, taken from Russell and Norvig [83].

Sometimes a third partition, called the *test data*, is used to evaluate the performance of the network after the entire training process.

Figure 2.14 visualises the concept of under- and overfitting using a *decision tree*, another machine learning approach [83]. Although the training set error continues to decrease with increasing tree size, the validation set error starts to rise for sizes greater than 7. This is a common occurrence for neural networks as well, where after a specific amount of completed epochs, the validation set error starts to increase. At this point, it is essential to stop the training process, since otherwise the performance on unseen data can decrease. Similarly, stopping the training process too early corresponds to tree sizes of 1 through 6 in Figure 2.14. Here, the model could still improve its performance by tweaking the hyperparameters. For neural networks, increasing the capacity of the model or the amount of epochs can remedy an underfitting model.

2.3.2 The Transformer Architecture

The *transformer architecture* was introduced by Vaswani et al. in 2017 [97] and is an architecture based on an encoder-decoder that does not use any recurrent neural networks. A transformer rather relies on the use of the *attention mechanism*, which we will explain below.

Figure 2.15 shows the entire architecture of a transformer. We use this figure to explain all different components of a transformer, in the order of the data flow.

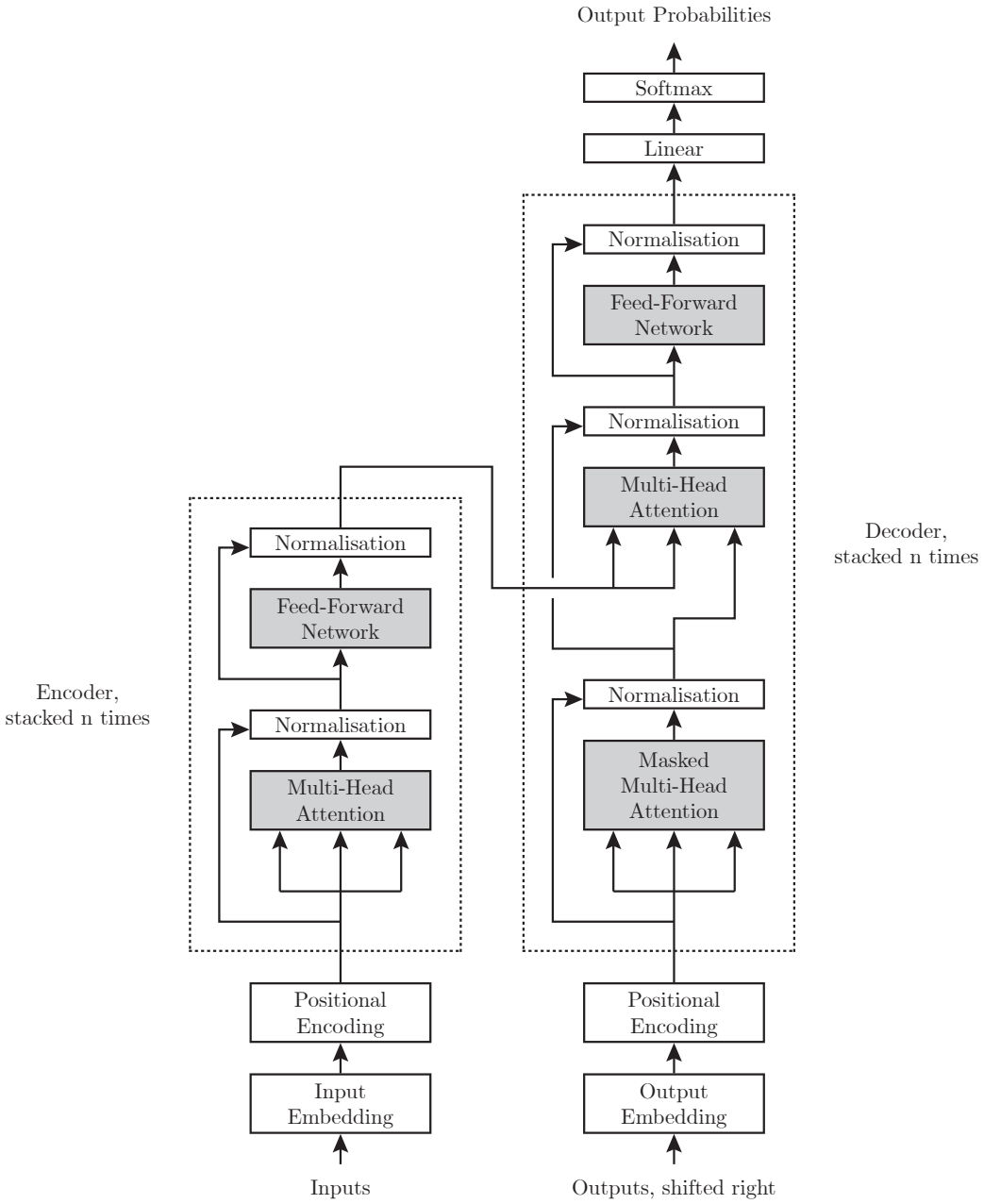


Figure 2.15: The architecture of a transformer.

Input Embedding and Positional Encoding

The input data ("Inputs" in Figure 2.15) comes in the form of $[batch\_size, seq\_len]$, which is a tensor of two dimensions. Here, *seq_len* gives the length of the input sequences,

while *batch_size* is the amount of data in a single batch. That is, a single tensor contains *batch_size* input sequences of length *seq_len*.

In a first step, Vaswani et al. [97] apply a standard embedding layer. The resulting data is of the form

$$[batch\_size, seq\_len, d_{model}], \tag{2.1}$$

where d_{model} is one of the hyperparameters of the model, giving the amount of embedding dimensions. Embeddings of tokens are learned over the course of the training process. After the training, the embedding layer is able to convert the tokens to vectors of length d_{model}, where similar tokens are closer to each other. The embeddings are then scaled by $\sqrt{d_{model}}$.

Due to the fact that a transformer does not use recurrent neural networks, positional information of the tokens need to be injected into the input sequences. This is done using a *positional encoding*. Vaswani et al. [97] use an approach using sine and cosine functions, since they reason that this way the model will be able to extrapolate to longer sequences than seen during the training process. Although they have experimented with learned positional encodings, they state that they have found the two approaches to be nearly identical.

The following two functions are used to inject positional information:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right);$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right).$$

Here, *pos* is the position of the token, while i gives the embedding dimension. The sine function is used for even dimensions, whilst the cosine function is used for odd ones. Vaswani et al. [97] state that they chose these functions since they conjectured it would allow the model to easily learn to attend by relative positions.

Scaled Dot-Product Attention

As stated above, the *attention mechanism* is the core component of the transformer model. Following Vaswani et al. [97], by an *attention function* one understands a mapping assigning a query and a set of key-value pairs to an output, where the query, keys, values, and the output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

However, when defining their concrete attention function, Vaswani et al. [97] combine multiple input vectors into matrices and define the attention function over matrices. More specifically, when given l query and key vectors of dimension d_k, and l value vectors of dimension d_v, let Q, K, and V be the following matrices:

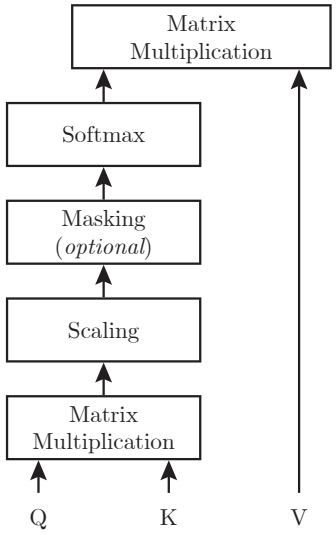


Figure 2.16: The scaled dot-product attention.

- Q is an $l \times d_k$ matrix consisting of l rows of query vectors of dimension d_k;
- K is an $l \times d_k$ matrix consisting of l rows of key vectors of dimension d_k; and
- V is an $l \times d_v$ matrix consisting of l rows of value vectors of dimension d_v.

Then, the *scaled dot-product attention function* is given by

$$\mathit{Attention}(Q, K, V) = \mathit{softmax}\left(\frac{1}{\sqrt{d_k}} QK^T\right)V, \tag{2.2}$$

where K^T is the transponse of K and *softmax* is a function which produces a probability vector based on an input vector $r = (r_1, \ldots, r_m)$ as follows:

$$\mathit{softmax}(r_i) = \frac{e^{r_i}}{\sum_{j=1}^{m} e^{r_j}}, \quad \text{for } 1 \leq i \leq m.$$

Moreover, when the *softmax* function is applied to a matrix, then this means that it is repeatedly applied to the vectors representing the rows or columns of that matrix.

The scaling factor $\frac{1}{\sqrt{d_k}}$ is used in order to combat the problem of vanishing gradients for large values of d_k.

A visualisation of the calculation of the scaled dot-product attention function is depicted in Figure 2.16. Note that the calculation in this figure also contains an optional masking step where $(-1) \cdot 10^{-9}$ is added to those values covered by the mask, representing negative infinity.

Relative Scaled Dot-Product Self-Attention

In 2019, Huang et al. [45] improved upon the work of Shaw, Uszkoreit and Vaswani [89], who introduced a way of taking the relative distances between elements of a sequence into account, by modifying the scaled dot-product attention. The enhanced function is called the *relative scaled dot-product attention* and can be used (almost) interchangeably with the one used in the original transformer paper [97].

The relative attention can be expressed using the following formula:

$$\mathit{RelativeAttention}(Q, K, V) = \mathit{softmax}\left(\frac{1}{\sqrt{d_k}}(QK^\intercal + S^{rel})\right)V,$$

where Q, K, and V are as for equation (2.2). Notice that the main difference to the original attention function is the S^{rel} term, which injects information about the relative distances of elements in the sequence into the attention mechanism. Huang et al. [45] were able to reduce the memory requirement from $O(L^2D)$ provided by Shaw et al. [89] to $O(LD)$, where $L = \mathit{seq\_len}$ and $D = d_{model}$ (cf. also equation (2.1)). In order to do so, they introduced the function *skew* thus:

$$S^{rel} = \mathit{skew}(QE^T).$$

Here, E is a set of learned relative position embeddings, which encode possible pairwise distances between two elements of the sequence. These distances can range from $-max_{len}+1$ to 0, and are ordered in ascending fashion in E, where max_{len} gives the maximum allowed length for any sequence, which has to be known beforehand. If we consider E to be a set of embedding vectors of dimension d_{model}, each element i represents the relative distance $-L + 1 + i$. If we now consider Q to be a vector of the same shape as E, the matrix product QE^T contains all the values we need for the computation of S^{rel}.

Multi-Head Attention

Vaswani et al. [97] found that, instead of performing a single scaled dot-product attention function with queries, keys, and values of dimension d_{model}, it can be beneficial to split up the d_{model}-dimensional embeddings into $d_h = \frac{d_{model}}{h}$-dimensional embeddings, where h is the amount of *heads* for the *multi-head attention*. In a subsequent step, h instances of the dot-product attention function are performed in parallel, each attending to the d_h-dimensional inputs. After the attention function, the intermediate results are concatenated, resulting in a tensor of the original dimensions

$$[\mathit{batch\_size}, \mathit{seq\_len}, d_{model}].$$

Due to the fact that the dimensions of the inputs of the different heads is a fraction of the initial embedding dimension, and that the attention function of the different heads can be computed in parallel, the computational cost of employing multi-head attention is similar to that of performing dot-product attention on the original instance.

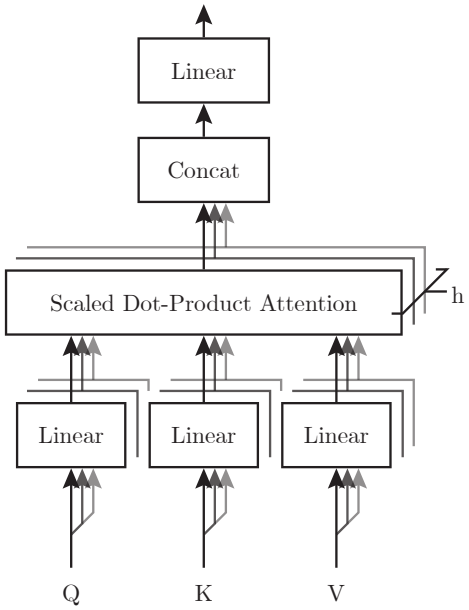


Figure 2.17: The multi-head attention.

Figure 2.17 shows a visual representation of this process. Initially, the values of Q, K, and V are linearly projected using learned weight matrices W_i^Q, W_i^K, and W_i^V (where $1 \leq i \leq h$). In practice, these weight matrices are simple feed-forward networks. For efficiency, W_1^Q through W_h^Q can be implemented using a single network. The same holds for the other weight matrices. After applying the dot-product attention in parallel, the concatenated values are linearly projected again.

Position-wise Feed-Forward Network

For both the encoder and the decoder of a transformer, the subcomponents are followed by a position-wise feed-forward network. This is simply a fully-connected feed-forward network with two layers, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between. While the linear transformations are the same across different positions, they use different parameters from layer to layer.

Encoder and Decoder

The *encoder* and *decoder* layers (shown in Figure 2.15 using the dashed rectangles) are comprised of several subcomponents. The encoder consists of a multi-head attention block, attending to its own inputs. This process is called *self-attention*. In both the encoder and the decoder layers, subcomponents are followed by an application of layer normalisation [8] to the sum of the previous component's output and the previous

intermediate result. Furthermore, both stacks apply a feed-forward network as discussed above.

Vaswani et al. [97] refer to this self-attention mechanism as one of the core components of the transformer architecture, improving the long-term dependencies in sequences and the parallelisability of the model, while at the same time decreasing the computational power required to train such networks. Furthermore, traditionally neural networks have been black-box models, i.e., it is very hard to impossible to interpret trained networks. With attention mechanisms, however, it is possible to extract the attention weights at a given step. This can show which part of, e.g., the input sequence the model is focused on at this step, which can be quite helpful when analysing the model.

The decoder layer uses *masked multi-head attention* for its self-attention mechanism. This mask sets all affected positions to a value representing negative infinity. It is constructed in such a way that the multi-head attention block cannot attend to values succeeding the current element, e.g., if the fifth element of the sequence is evaluated, elements 6 to *seq_len* are masked out.

In the next step of the decoder, the multi-head attention block sets Q and K to the output of the encoder stack and V to its own intermediate result. This way the decoder can attend to the encoded input sequence.

Vaswani et al. [97] use *stacks* of both the encoder and decoder. These stacks consist of subsequent iterations of the same layer, taking as input the output of the previous one. The authors argue that this can increase the performance of the model and used a stack of $N = 6$ layers for their evaluation.

CHAPTER 3

Computer-aided Algorithmic Composition: A Brief Overview

Although algorithmic composition is not an idea of modern origin and several approaches from pre-computer eras exist, as briefly discussed in Chapter 1, we now will give a brief overview about the main proposals for computer-aided algorithmic composition, especially ones making use of artificial-intelligence techniques, which raise to prominence over the last two decades.

Our exposition mainly follows the survey papers by Fernandez and Vico [31] and Carnovalini and Rodà [20]. Other excellent sources discussing AC techniques include, e.g., *The Oxford Handbook of Algorithmic Music* [60], the books by Nierhaus [65] and Briot, Hadjeres, and Pachet [18], as well as the paper by Ji, Luo, and Yang [47].

3.1 Markov Chains

Introduced by the Russian mathematician Andrey Andreyevich Markov in 1906 [57], *Markov chains* are stochastic processes of a certain nature. In their most basic form, a Markov chain specifies a set of *transition probabilities* for a finite (or at most countably infinite) set of states. Here, the transition probability of a transition from one state to another is only dependent on the current state, not the preceding sequence of states. Accordingly, one says that Markov chains (in this form) do not have any memory.

Markov chains can be represented using graphs, where the nodes correspond to states and weighted directed edges correspond to transition relations. Here, the *weight* defines the probability of such a transition. Besides this graph representation, usually Markov chains are represented using *probability matrices*.

Table 3.1: The relative frequency of notes in eleven standardised Stephen Foster songs [66].

Note	B_3	$C_4^\sharp$	D_4	E_4	$F_4^\sharp$	G_4	$G_4^\sharp$	A_4	B_4	$C_4^\sharp$	D_5	E_5
Relative frequency	17	18	58	26	38	23	17	67	42	29	30	17

Table 3.2: The probability of a note following a preceding note, given in sixteenths [66].

Note	B_3	$C_4^\sharp$	D_4	E_4	$F_4^\sharp$	G_4	$G_4^\sharp$	A_4	B_4	$C_4^\sharp$	D_5	E_5
B_3			16									
$C_4^\sharp$			16									
D_4	1	1	2	5	3	1		1		1	1	
E_4		1	6	3	4			1			1	
$F_4^\sharp$			2	4	5	2		2	1			
G_4					4	3		6	3			
$G_4^\sharp$								16				
A_4			1		5	1	1	4	3		1	
B_4			1		1	1		9	2		2	
$C_4^\sharp$									8		8	
D_5								4	7	3	1	1
E_5								6		10		

While basic Markov chains only consider the current state for the next transition, with *higher-order Markov processes*, one can take the last n transitions into account, where n is the order of the process. Such chains are referred to as *n-th order Markov chains* [20, 64].

These n-th order chains are well-fitted for the task of algorithmic composition, since in music, the next note in a sequence of notes is most definitely dependent on the preceding ones. In an AC context, the states of a Markov chain could correspond to a set of notes. Using a transition relation, one could then produce a melody, by simulating a sequence of transitions through the chain.

One of the earliest examples of Markov chains applied to the task of algorithmic composition is the work of Harry Olson [66]. He analysed eleven melodies by the American composer Stephen Foster, and subsequently created Markov chains of different orders. For the sake of standardisation, the melodies were all transposed to the key of D beforehand (cf. Tables 3.1 and 3.2).

Further examples for Markov chains in algorithmic composition include Pinkerton's "Banal Tune Maker" [77], which is based on an analysis of 39 nursery tunes, Pachet's "Continuator" [72], which uses a variable-order Markov chain in order to better deal with sequences of varying length, and others [5, 29, 24, 73].

Although Markov chains are no longer considered state-of-the-art as far as algorithmic

composition is concerned, they can still be useful when it comes to, e.g., generating small parts or higher-level structures of a piece [20].

3.2 Rule- and Constraint-Based Systems

Systems that compose music based on a set of rules or constraints are often a natural choice for algorithmic composition, as music theory itself is based on a set of more or less formalised rules. Although these rules are often violated when human composers construct a new work, they can still serve as a good basis for formal systems composing music. The name *constraint-based system* (CBS), alternatively *rule-based system*, serves usually as an umbrella term for a large class of formal approaches, encompassing a wide variety of techniques where the boundaries between these techniques are often fuzzy.

One famous example for the application of a rule-based system to algorithmic composition are the first two movements of the *Illiac Suite* by Hiller and Isaacson [48], who used classical rules for counterpoint in order to generate musical material [64]. Here, *counterpoint* refers to the musical concept of coordinating several voices [52].

Constraint satisfaction programming (CSP), often referred to as just *constraint programming*, is a natural choice when it comes to AC, as it is well-fitted to conform to and describe the rules of music theory. Anders and Miranda [4] conducted a survey on the application of CSP for AC, and concluded that it is a potent approach.

CHORAL, due to Ebcioğlu [36, 37], is a famous example regarding the application of constraint programming for AC. He designed 350 rules and a custom logic language called BSL (in his own words "a new and efficient logic programming language which is fundamentally different from Prolog" [36]) to realise an expert system capable of writing four-part chorales in the style of Bach. An important aspect of the approach is the intricate balance of the rules: a smaller body of rules allows to be more open for different styles, whilst a larger body can produce results that better conforms to a specific one.

After the success of CHORAL, the application of CSP for AC has experienced an upswing. Tsang and Aitken [95] used Prolog to develop an application for harmonising four-part chorales, although the efficiency of their system was lackluster at best. Ovans and Davison [69, 70] introduced a system which aids a user in composing using the counterpoint technique. In this case, the user is responsible for generating a musical solution, driving the search process, while the system constrains the possible outputs. Further applications of CSP techniques include the approach of Sandred [85], who proposed using CSP for generating rhythm, PiCO [81], an experimental language for music composition integrating constraints, the method of Davismoon [24], in which Markov processes are combined with constraints, and others [2, 3].

A different rule-based paradigm for AC comes in the form of *answer-set programming* (ASP), a declarative problem solving paradigm with roots in logic programming and non-monotonic reasoning [35, 34, 33]. In ASP, problems are encoded using rules and

constraints, and solutions are given in terms of the models (the "answer sets") of the resulting programs.

A well-known example regarding the application of ASP for algorithmic composition is Anton, due to Boenn, Brain, De Vos, and Fitch [16], an automatic composition system that is capable of composing basic melodies with accompaniment, especially first-species counterpoint.[1] First-species counterpoint (also called *1:1 counterpoint*) describes the process of adding a single pitch above or below each pitch of the baseline [52]. Anton is able to compose melodic, harmonic, and rhythmic music using a single framework, and achieves near real-time performance in doing so. Its authors argue that it could thus be used for live performances or concerts. The program supports working with major, minor, Dorian, Lydian, and Phrygian modes [10]. Specific rules handle the progression of melodic and harmonic parts, and select the next note in each part based on the previous one. At any given time, each of these parts is only able to play a single note. Although no fixed number of parts is specified, the system supports solos, duets, trios, and quartets out of the box.

Everardo and Ramírez introduced Armin [75], an ASP-based composer of trance music.[2] Armin is based on Anton, adding some rules which represent knowledge about the trance-music genre. Their application is capable of generating three percussion tracks and a melodic line. A core concern of Armin is the generation of a valid trance piece, which is why it supports generating sequences of sections (e.g., introduction, verse, chorus, breakdown, ...) using Markov chains. Armin's initial core concern is not to compose a piece in its entirety, but rather serve as a template generator. The authors argue that the generated templates can be enhanced and post-processed. Noteworthy is the fact that Armin aims to output audio files in the WAV format, producing sounds from a previously generated score. In order to do so, the score is passed to a parser, which interprets the instrument tracks and synthesises the sound. Armin supports the generation of bass drum, hi-hats, and snare drum percussion tracks.

Opolka, Obermeier, and Schaub introduced chasp [67], standing for "composing harmonies with ASP", which is able to autonomously create pieces of many different genres, in contrast to Anton and Armin, which generate music restricted to specific genres. This flexibility of chasp is possible by focusing on the harmonic basis of a piece. Furthermore, chasp is able to create a chord progression based on a certain user-specifiable length and key. Later in the process, a Python framework is used to apply rhythm to the intermediate output. The authors achieve this by having created LilyPond [84] templates beforehand, based on an understanding of what defines each of the supported genres. Using their Python framework, these files are then populated with the generated chord output, which induces harmony and rhythm.

Figure 3.1 shows an example output from chasp. Here, the limitations of chasp become apparent: since the program only supports a pattern-based creation of musical pieces,

[1] Named after the Austrian composer Anton Friedrich Wilhelm von Webern (3rd December 1883 - 15th September 1945).

[2] Named after Dutch DJ Armin van Buuren (born 25th December 1976).

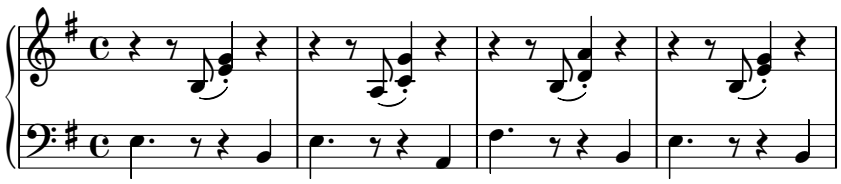


Figure 3.1: An example output generated using chasp [67].

the musical expressability is severely limited. All the bars follow the same rhythmic and melodic pattern, only the base pitch is replaced. We argue that although, e.g., Anton's range of supported genres may not be as wide, its capabilities as algorithmic composer are more impressive, since it is not limited by templates such as these.

3.3 Evolutionary and Genetic Algorithms

Genetic algorithms (GA), or *evolutionary algorithms*, take inspiration from biological sexual reproduction. Here, successor states are created from a set of two parent states, which are combined to create (possibly multiple) children states. These children states usually exhibit similar properties to their parents, combining some of their aspects.

GAs start with a set of (often randomly generated) initial states, called the *population*. Usually, each of these states, or *individuals*, is represented using a finite alphabet, e.g., a string of the numbers 0 and 1, or numbers 1 through 8. Individuals are proportionally selected based on a *fitness* function, assigning each state a value representing their quality. In the *crossover* stage, individuals are combined according to a strategy. Often a crossover point is chosen which divides the parent states in half. One side of the crossover point is then selected for each parent, and then paired with the corresponding other side from the other parent. In a final *mutation* step, each location is subject to a random change with a small probability [83].

One of the main problems of the application of genetic algorithms for algorithmic composition is evaluating how fit any given solution is. Since music (and art in general) is highly subjective, defining a fitness function can be a very hard task, sometimes not even possible. In spite of this limitation, GAs have been used for AC systems.

A famous example is GenJam [12], a system for Jazz improvisations. Here, the mentioned problems regarding defining a fitness function are circumvented by having a human deciding on the quality of the output, an approach also referred to as an *interactive genetic algorithm*. This constitutes a rather severe bottleneck, as such an evaluation is both resource-intensive and slow. Subsequent versions tried to remedy this by introducing a neural network for the fitness assessment [14], although this application was not very successful as the networks failed to generalise the evaluations from the training set. Eventually, the fitness function was removed from GenJam [13]. Thus, the system can no longer be considered as using a genetic algorithm.

Other approaches using GAs include, e.g.,

- the one of Phon-Amnuaisuk, Tuson, and Wiggins [76], who took rules from music theory to design a fitness function, describing preferred and forbidden intervals and patterns,
- the method of Ortega, Alfonso, and Alfonseca [68], in which an entirely different approach is taken by evolving the rules of a grammar used for AC, and
- the approach by Bell [9], who used GAs to evolve parameters of a Markov chain.

3.4 Systems Based on Neural Networks

Many of the recent and very promising works apply neural networks for the task of algorithmic composition. One of the earlier works utilising neural networks is the one of Eck and Schmidhuber [27, 28], who first applied a *long short-term memory* (LSTM) neural network [43] to the task of algorithmic composition. These networks are a type of recurrent neural networks, being able to take its own previous outputs into account when generating the next element in a sequence. Applying LSTM units to such a network improves its long-term structure, since the network is able to control the amount of information taken from the input versus the previously seen data. Note that Schmidhuber is the co-author of the paper where LSTMs were first introduced [43]. Eck and Schmidhuber point out that their system is able to produce music with good timing and proper structure.

An algorithmic composer for multi-track pop music based on a *generative adversarial network* (GAN) is MuseGAN (standing for "multi-track sequential generative adversarial network") [25]. GANs, introduced by Goodfellow et al. [39] in 2014, are networks consisting of two parts, a *generator* and a *discriminator*. The generator's aim is to produce an output of a special nature, whilst the discriminator tries to differentiate between the ground truth and the output of the generator. During the training process, both of the networks take the other one into account, e.g., the generator is trained to produce output that fools the discriminator, while the discriminator is trained to accurately separate real from fake pieces. This way, both of the networks are able to improve based on the performance of the other one. In contrast to PAUL-2, in MuseGAN piano rolls are used for generating music. More specifically, a convolutional neural network is used to generate this visual representation of music, where a black pixel indicates that a specific note is played at that point in time. Figure 3.2 illustrates this concept, where the evolution of MuseGAN's output is clearly visible. While at earlier steps, no clear structure is visible and only random noise is created, at later steps, coherent musical structures can be observed.

OpenAI's MuseNet [74] takes an entirely different approach. That work is based on GPT-2 [79], a large-scale transformer-based unsupervised language model, which is trained to predict the next token in a sequence. GPT-2 supports the creation of 4-minute musical compositions, where a particular feature of this system is that it creates multi-track pieces,

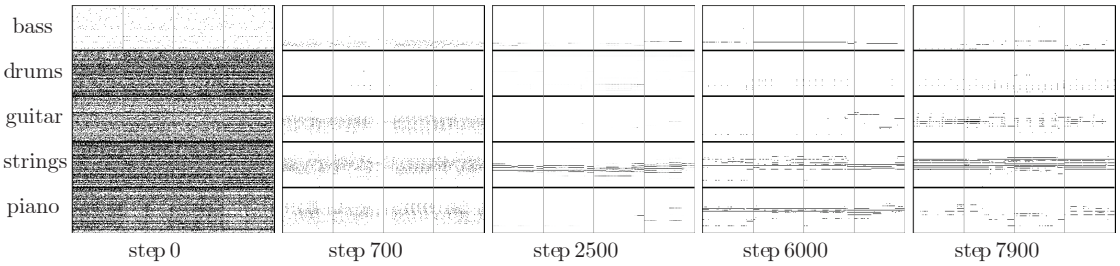


Figure 3.2: Evolution of generated piano rolls by MuseGAN over the course of the training process, taken from Dong et al. [25].

consisting of separate tracks for multiple instruments. Ten different instruments are supported, including piano, strings, drums, guitar, bass, and woodwinds. Furthermore, MuseNet supports the synthetisation of music in styles of various different composers, some classical (e.g., Mozart, Beethoven, or Chopin), but also some modern composers and artists (like Queen, The Beatles, or Adele). In MuseNet, a token-based representation of music data is utilised. Each token can contain a multitude of information, e.g., in case of play messages, the instrument, the velocity of the note, and the note identifier. The following excerpt shows a sample encoding:

```
bach piano_strings start tempo90 piano:v72:G1 piano:v72:G2
piano:v72:B4 piano:v72:D4 violin:v80:G4 piano:v72:G4 piano:
v72:B5 piano:v72:D5 wait:12 piano:v0:B5 wait:5 piano:v72:D5
wait:12 piano:v0:D5 wait:4 piano:v0:G1 piano:v0:G2 piano:v0:
B4 piano:v0:D4 violin:v0:G4 piano:v0:G4 wait:1 piano:v72:G5
wait:12 piano:v0:G5 wait:5 piano:v72:D5 wait:12 piano:v0:D5
wait:5 piano:v72:B5 wait:12
```

Lastly, the enhancements to the transformer architecture, as introduced by Huang et al. [45] in the Music Transformer approach, are widely considered to constitute the state of the art in algorithmic composition. Music Transformer uses an event-based representation using note-on, note-off, and time-shift events, and is able to generate single-track piano music, outperforming what Huang et al. dub the "vanilla transformer" (a transformer model making use of the unmodified attention mechanism), especially when it comes to the long-term structure of musical pieces. The authors note that the output of the "vanilla transformer" quickly deteriorates when generating pieces that are longer than the training data, while Music Transformer is able to compose pieces with consistent style throughout.

3.5 Other Approaches

Note that we have only discussed a small number of approaches used for algorithmic composition, and an even smaller number of concrete systems. Other techniques used for AC include *generative grammars* [54, 92, 71, 40], *L-systems* [78, 58, 63], *transition networks* [22, 23], *self-similarity* approaches [98, 17, 53], *cellular automata* [61, 62], and *agent-based systems* [55, 49, 50].

For exhaustive surveys on the topic, we refer to the works listed at the beginning of this chapter, especially the papers by Fernandez and Vico [31] and Carnovalini and Rodà [20].

CHAPTER 4

The PAUL-2 System

In this chapter, we introduce our main contribution, PAUL-2, an algorithmic composer for two-track piano pieces supporting multiple difficulty levels, based on a transformer model.

The chapter is structured as follows: Section 4.1 covers the structure and functionality of PAUL-2, explaining the details of our implementation. In Section 4.2, we discuss the pipeline used to process the dataset. In Section 4.3, we provide an analysis of the dataset used to train PAUL-2, and Section 4.4 deals with the training procedure and its details. Section 4.5 discusses the generated music files of PAUL-2 and Section 4.6 compares PAUL-2 with its predecessor system PAUL. Finally, Section 4.7 reports on our user study, evaluating the quality of the music files generated by PAUL-2. Our preprocessing library S-Coda will be discussed in Chapter 5.

4.1 Structure and Functionality

In this section, we lay down the internal structure of PAUL-2 and discuss its technical details.

4.1.1 Overview

The following list provides an overview of the dependencies of PAUL-2:

- Python 3.10[1],
- sCoda 1.0[2],

[1] `https://www.python.org/`.
[2] `https://pypi.org/project/sCoda/`.

- TensorFlow 2.9.0[3],
- CUDA 11.7.0[4],
- cuDNN 8.4.1[5], and
- numpy 1.22.3[6].

As mentioned, PAUL-2 is a transformer-based [97] algorithmic composer. It is capable of sequence-to-sequence translations, since it is based on the encoder-decoder paradigm. For our purposes, these sequences consist of either musical information or musical meta-information in the form of difficulty values.

PAUL-2 consists of two distinct models. Both of these models have different internal architectures and support a different set of hyperparameters. Both of these models need to be trained separately and have a different set of weights that can be stored in order to load them at a later date.

The models of PAUL-2 are the following:

(i) P2L, a sequence-to-sequence transformer for composing *lead tracks*, based on an input difficulty sequence, and

(ii) P2A, a multi-sequence-to-sequence transformer for composing *accompanying tracks*, based on an input lead track and a difficulty sequence.

With our preceding system PAUL [86, 87], we needed to train six different models in total, since each model only supported a single difficulty level. This induced a large overhead and lowered the potential of the models, since only sequences that contained subsequent bars of the same difficulty could be used.

With PAUL-2, however, we drop these constraints. For P2L, we use the encoder solely to encode the desired difficulty values of the output lead track, from which we then construct the generated sequence. For P2A, we were inspired by the enhancements introduced by Libovický, Helcl, and Marecek [56], who discussed multi-source sequence-to-sequence translation tasks. We modified the transformer architecture to be able to generate accompanying sequences from both an input sequence consisting of difficulty values and an input sequence consisting of musical information representing the lead track.

We also utilise the enhancements introduced by Huang et al. [45], who introduced the performant relative self-attention mechanism. Using this enhanced attention, the decoder of the transformer is able to better attend to its own input, i.e., the sequence generated

[3] https://www.tensorflow.org/.
[4] https://developer.nvidia.com/cuda-toolkit/.
[5] https://developer.nvidia.com/cudnn/.
[6] https://numpy.org/.

so far. As a result, the structures of the sequences generated by the model contain better long-term dependencies.

We note that at the moment the relative self-attention mechanism does not support look-ahead attention, i.e., relative attention cannot be used in contexts where the entire sequence is able to be attended to. This is the case for self-attention in the sequence encoder of P2A, or the cross-attention between the encoder and decoder of P2A. We argue that, if this would be possible, the quality of the encoding of the input sequence could possibly be improved, leading to higher-quality output.

We used several different resources for our approach which aided us in the practical implementation. Google's TensorFlow [94] provides valuable resources for implementing various architectures using their framework. Especially helpful were the guides on the transformer model[7] and on the attention mechanism in general[8]. Furthermore, the resources provided by Gomatam[9] were of great help when implementing the relative attention mechanism.

We used the TensorFlow framework for the practical implementation of PAUL-2 since it provides a relatively high-level approach for implementing advanced neural network architectures. With this framework, several different layers of different functionality (such as an attention layer, a basic feed-forward layer, a normalisation layer, ...) can be concatenated and automatically trained. Furthermore, TensorFlow allows for the use of graphics-processing units which can speed up the training process immensely.

4.1.2 Architecture

We now cover the structure of P2L and P2A. Figure 4.1 and Figure 4.2 show a visual representation of both models, which we will use to discuss them in more detail.

We start by explaining the four different types of masks used for PAUL-2. Figure 4.3 shows a graphical representation of three types. Some explanation is needed in order to understand the visualisations. Here, on the vertical axis the temporal dimension is shown. Each row corresponds to exactly one step in the process, e.g., the first row represents the mask applied to the sequence when the focus lies on the first element.

Figure 4.3a shows a *single-out mask*, which we are introducing in this thesis. This mask allows only for attending to the current element, no other element is visible to the active one. We use the single-out mask exclusively for masking the difficulty sequences. Due to the fact that we create sequences of difficulty values where each value of index i corresponds to the message of the melodic sequence with the same index, simply applying such a mask ensures that the network matches the musical messages with the correct difficulties.

[7] `https://www.tensorflow.org/text/tutorials/transformer/`.
[8] `https://www.tensorflow.org/text/tutorials/nmt_with_attention/`.
[9] `https://github.com/spectraldoy/MusicTransformerTensorFlow/`.

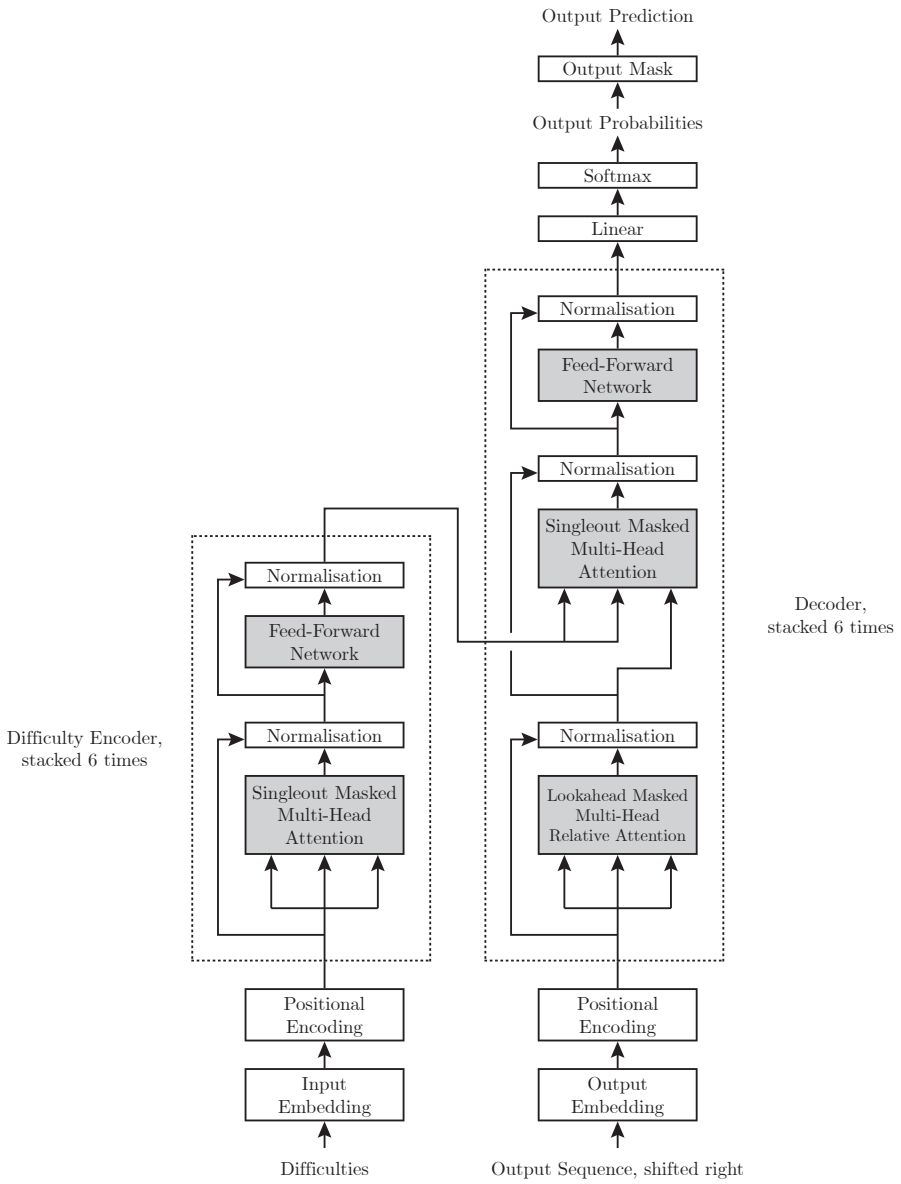


Figure 4.1: The P2L architecture.

Figure 4.3b shows a *look-ahead mask*. This mask is used during the training process, where we feed the network the entire output sequence at once. The mask prevents the network from learning to simply look up the next value it is supposed to predict, rather than try to guess it. It is solely used for the self-attention of the decoder.

The next mask is a *padding mask*, which is not shown in the figures we provide, as it does not have a temporal component. The padding mask simply masks out any position of

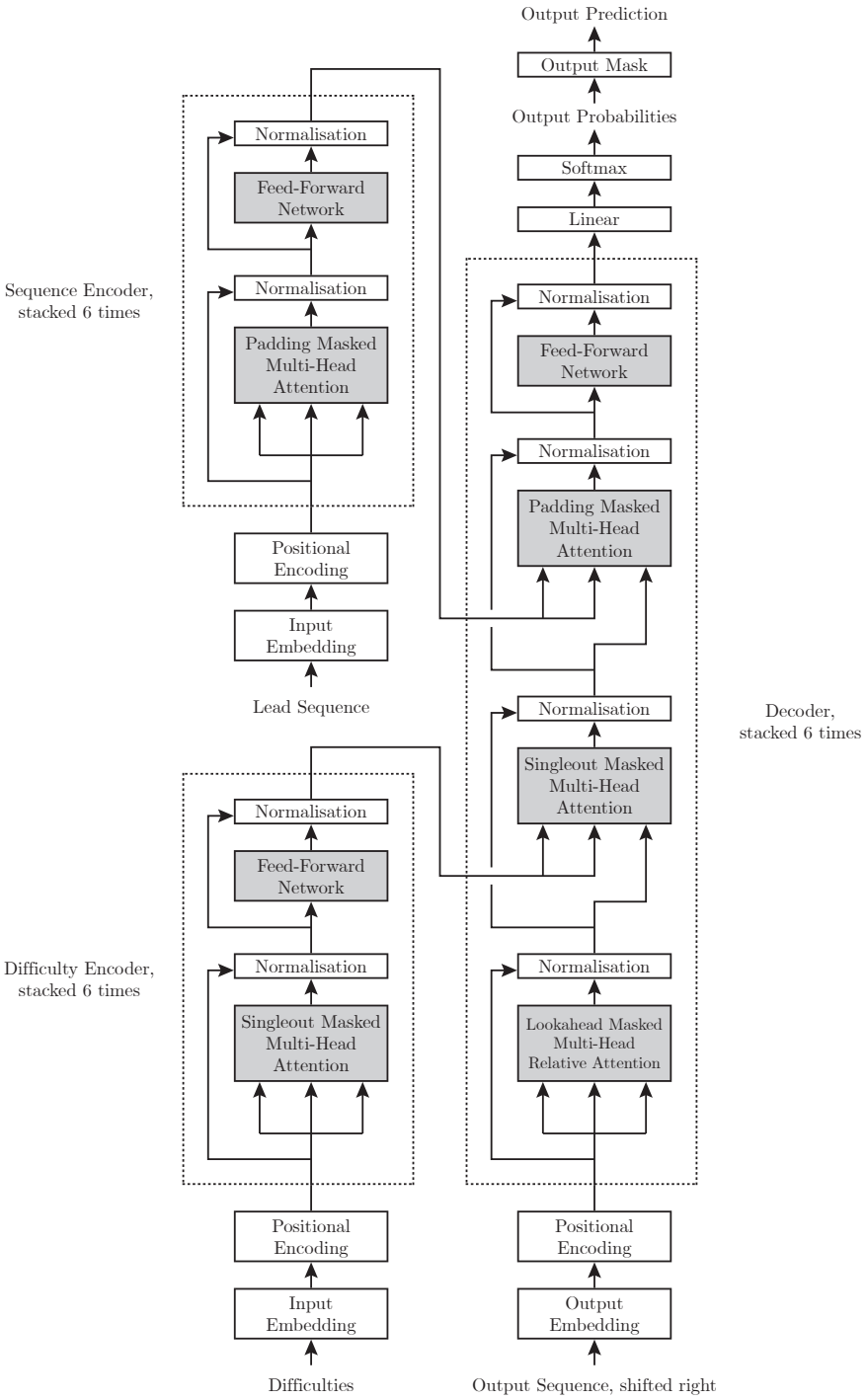
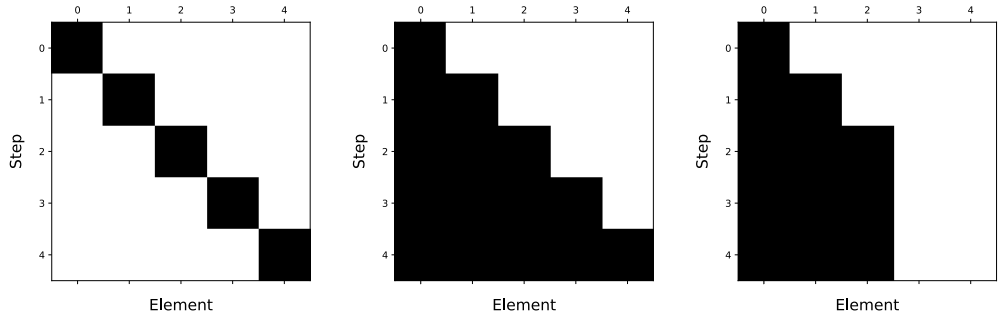


Figure 4.2: The P2A architecture.



(a) A single-out mask. (b) A look-ahead mask. (c) A combined padding and look-ahead mask.

Figure 4.3: Three different masks used for PAUL-2.

the sequence filled with the padding value. This way we can ensure that all the sequences are of equal length, without having to artificially extend them. During the processing of the data, any masked out value is simply ignored.

Figure 4.3c shows a combined mask, made from applying both the padding mask and the look-ahead mask. Note the difference to Figure 4.3b: In our example, the last two values of the sequence were filled with the padding value. This is why they are masked out at the fourth and fifth time step, even though the look-ahead mask would allow for attending to them.

Figure 4.1 shows the architecture of the lead network. It is very similar to the unmodified transformer architecture. The main differences include our usage of the relative attention mechanism and the fact that we utilise the single-out mask instead of a normal padding mask used in the original paper [97]. We note that Huang et al. [45] did not utilise an encoder for their work, since they were only interested in generating single sequences. They did not create accompanying tracks for a musical lead.

Another improvement we introduce in this thesis is the usage of an output mask. Since S-Coda is able to provide a list of possible next messages for an existing sequence, we can utilise this information to restrict the possibilities for the next predicted output message. During the generation process, we create a sequence consisting of all of the messages generated so far. We use this sequence to retrieve the mentioned list of next messages, which we can then use to create a mask of possible next values. Applying this mask to the output predictions of the network ensures that no invalid message is generated, while still taking the different output weights into account. For example, using this process we can prevent that a message to close a note that has not been opened before is generated. In similar fashion, no message to open a note that is still open will be generated this way.

Figure 4.2 shows the architecture of the accompanying network. We use a total of two encoders. These encode both the difficulty of the sequence in a similar fashion as with the

lead network and the lead sequence itself. For the encoding of the lead sequence, we use a standard padding mask for the self-attention, as the encoder is in fact allowed to attend to future positions during the generation process. We then use a serial connection of the attention blocks in the decoder that attend to the output of the encoders. Libovický, Helcl, and Marecek [56] state that the serial approach outperformed other architectures for multi-source machine translation tasks.

Although we only use two distinct encoders, our model is highly scalable and can easily accommodate a larger amount of encoders. In fact, the only element needed to change the amount of encoders is to provide more input sequences and change a few hyperparameters, such as the amount and types of masks to use. Due to this fact, we were able to reuse the same training framework for P2A as we used for P2L.

4.1.3 Functionality and Motivation

As stated previously, the intended usage of PAUL-2 is to be used in a tutoring system, teaching piano students how to perform the act of *sight-reading*. Sight-reading refers to the process of simultaneously reading and performing musical score without having studied or seen it beforehand. It is a rather advanced technique for piano players and can be very useful in several different contexts. Professional pianists for example are expected to be able to sight-read reasonably well and often have to do so when it comes to, e.g., a recording session of a movie score or a soundtrack, as it greatly reduces the time needed to study the piece and sometimes allows for very short-term recordings where the performers were not shown the score beforehand.

Training in order to better perform at sight-reading can be a very tedious and laborious process, as one has to practice on a plethora of different pieces. This process can be very exhausting, as strides regarding the sight-reading technique are often quite small. Furthermore, if a student has neglected the study of sight-reading for years, it can still be the case that their technical proficiency is quite high, allowing them to perform elaborate pieces. These often do not lend themselves well for the study of sight-reading, as the student would be overwhelmed by the additional workload. Studying sight-reading on beginner pieces, on the other hand, could possibly demotivate the student, as they could feel like having to start all over again. As well, returning to their previous habits and performing the more advanced pieces can be quite alluring. Here, a balance between the current sight-reading skill of a student and the difficulty of the pieces has to be found and adjusted over the course of the learning process.

A second complication comes from the fact that musical scores are often quite expensive to acquire. Even though most classical piano compositions are already in the public domain, publishers tend to rework and tweak some small aspects of the score, granting them the copyright of these modifications and allowing them to sell scores for significant sums of money. Acquiring a large amount of sheet music is paramount for training in the act of sight-reading, as with a small corpus of works the students could start memorising

the pieces instead of truly reading the sheet music quite quickly. We note that free online sheet music libraries such as

`https://imslp.org/`

do exist, although the quality of the site's content greatly varies.

This is where we imagine PAUL-2 to come in: We want to be able to provide students with standardised musical prompts of high quality that conform to their skill level. In this case, students would not have to concern themselves with finding challenging prompts that would not overwhelm them, as the system would be able to track their progression and provide them with adequate material to study on. Furthermore, additional motivation in the form of some sort of progress tracker could be supplied, providing the students with immediate feedback on their advancements in the discipline. Finally, since the training material of PAUL-2 is in the public domain, the high cost of acquiring sheet music can be circumvented.

In such a hypothetical system, the students could then use MIDI keyboards to perform the musical prompts, allowing for the system to judge their performances. This data can then be used twofold:

(i) As mentioned before, the system could keep track of the student's skill level. In such a scenario, the system could judge the performance of a student on a given piece. Subsequent error-free performances of prompts of the same difficulty could indicate that a student is not challenged enough by the pieces, thus the difficulty of the prompts can be raised. On the other hand, if the student has difficulties performing the prompts provided, this could indicate that they are overwhelmed by the difficulty of the pieces, which in turn can then be used to lower subsequent difficulties.

(ii) If a student studies sight-reading on their own without being provided immediate feedback by a teacher, it could be the case that they mistakenly learn something wrong. Relearning a previously wrongly learned habit can be quite challenging and should be avoided at all costs, but having a teacher present at all times to provide feedback can be quite expensive. A system as imagined above could immediately show students their shortcomings while also highlighting when they did particularly well.

Although the implementation of a system as described above is not the aim of this thesis, with PAUL-2 we want to show that in theory generating pieces that conform to a specific difficulty level can be achieved. Realising the rest of such a system can then be done relatively straightforward, as we imagine generating the pieces of varying difficulty to be the main challenge.

4.2 Retrieval and Processing of the Dataset

In this section, we discuss the details of the dataset used for the training of PAUL-2. Section 4.2.1 covers our approach regarding the choice of a source of MIDI files to use and in Section 4.2.2 we discuss the procedures applied to the dataset in order to sort out any invalid tracks. Finally, Section 4.2.3 covers the preprocessing procedure applied to the remaining files.

4.2.1 Choice of Dataset Sources

In order to train neural networks, large amounts of input data is necessary. During the training process, these networks learn to imitate the underlying distribution and structure of the training set. After a neural network is sufficiently trained, it can extrapolate this information in order to classify data that has not been seen before. For our purposes this would encompass, e.g., predicting the next note in a musical sequence.

The quality of the data used to train the network is of great importance, as the network can only learn to make correct predictions if the data itself is valid. At the same time, the quantity of the data is of equal importance. If too little data is provided, the network will not be able to learn all the intricacies of the underlying structure. Networks fed with too little data tend to overfit very quickly. This can often be remedied by providing more data to train on.

With PAUL [86, 87], we used a dataset of sequenced MIDI files provided by Bernd Krüger and retrieved from

`http://piano-midi.de/`.

We argued that this singular source provides a well-curated set of MIDI files, containing only sequenced MIDI files. In contrast to MIDI files recorded from human performances, sequenced MIDI files are created manually in a step-by-step process, inserting notes by hand. A great advantage of this technique is the fact that one can insert information, e.g., about which tracks the inserted notes belong to. This is, in general, not possible when recording a MIDI performance. In these cases, most often the output from a singular MIDI keyboard is recorded, which has no means of differentiating between which track a played note should belong to.

It is essential for the training of PAUL-2 that the MIDI files used in our dataset separate the sequences into the lead and the accompanying track. Since we want to be able to generate an accompanying track for a given input lead track, we need to train the network on sequences of similar shape, i.e., provide the network with training data consisting of two separate tracks as well.

We decided on using pieces from Krüger's database again, as they fulfill all the mentioned criteria. Furthermore, they are very well organised when it comes to naming and

separation of the tracks, with only few exceptions. With PAUL, we used a set of approximately 130 classical piano pieces in order to train three different LSTM-based networks. Each of these networks was trained only on pieces conforming to its difficulty level. For the training process of PAUL-2, we decided on using almost the entirety of Krüger's database, consisting of approximately 300 different pieces.

For the earlier training iterations, discussed in more depth in Section 4.4, we settled on using this data source exclusively. We were able to generate pieces of the desired shape and characteristics, i.e., two-track piano pieces conforming to a pre-defined difficulty parameter.

Although with this at hand, we already managed to complete our primary research objective, being the investigation of whether a transformer-based approach can be used to generate such pieces, we were not satisfied with the overall quality of the generated compositions. Even though the temporal structure of the pieces was excellent—an area that PAUL particularly underperformed in—, the harmonics between the lead and the accompanying track were often unpleasant to the ear. We theorised that this could be due to the relatively small amount of data used in our training process.

A dataset of more than 300 pieces might sound sufficient at first, but when compared to state-of-the-art approaches such as, e.g., MuseNet [74]—which used a dataset consisting of more than 40,000 pieces—, it seems rather pale in comparison.

Indeed, for the training of MuseNet, several different sources of data were used, including

`https://www.classicalarchives.com/`,

whose large MIDI database was donated for the development of MuseNet. Although the quality of the data contained on this website seems to be excellent, it is only commercially available. Also, for the development of MuseNet the *MAESTRO* [41] dataset was used, which is freely available. Due to the fact that the *MAESTRO* set is exclusively recorded rather than sequenced, we cannot use it for the training of PAUL-2 since no information about track separation is contained.

We considered using the *GiantMIDI-Piano* [51] dataset since it contains 10,855 MIDI files of piano performances. Unfortunately, the files lack separation into lead and accompanying tracks and do not seem to carry any information about time- or key signatures.

The *Lakh MIDI dataset* [80] is another commonly used dataset, consisting of 176,581 unique MIDI files of various nature. There, 45,129 entries in the dataset have been matched to the *Million Song dataset* [11], a collection of metadata of a million popular music tracks. Although this is a very large amount of tracks, only a small subset of them contains data which is potentially of use for the training procedure of PAUL-2. Ferreira, Lelis, and Whitehead [32] introduced the *ADL Piano MIDI* dataset, which is based on the matched part of the *Lakh MIDI dataset* and contains 11,086 unique piano pieces in MIDI file format. It can be found at its git repository located at

`https://github.com/lucasnfe/adl-piano-midi/`.

We decided on using a dataset composed of a combination of the files from Krüger's database and the *ADL Piano MIDI* dataset. In a next step, as discussed below, we sorted out any file that did not conform to our needs, e.g., having too few tracks or a too large number of empty bars. This left us with a total of 2,147 MIDI files, almost a seventeen-fold increase over the dataset used for PAUL.

4.2.2 Cleansing of the Dataset

A great disadvantage of datasets as large as the *ADL Piano MIDI* dataset is the fact that they often lack uniformity. In our case, many of the files contained consist of only a single track or are not clearly separated into lead and accompanying tracks. We applied an initial sorting procedure to all the files gathered from the sources mentioned above, which we will discuss now.

In the first step, we simply removed any file from the dataset that contains less than two distinct tracks. The validity of this step is indisputable, as these tracks lack the necessary information needed to train PAUL-2.

We then extended this approach to remove any file that contains less than two tracks that contain note information. At first glance, it might seem that this step is identical to the previous one. Often, MIDI files store meta information, e.g., time- and key signatures, in a separate track. Such a file could potentially contain two tracks, yet only one of these tracks would correspond to a musical sequence. We note that this step subsumes the first one. For performance reasons, we decided on applying both of them, as the initial step induced almost no performance overhead.

In a third step, we renamed tracks that only contain meta information, assigning them a unique identifier that can later on be used to convert the MIDI files into `Sequence` objects. We refer to this as a "meta" identifier.

We applied a similar procedure to the tracks containing musical information. Here, we tried to assign unique "piano" identifiers to tracks that unambiguously can be matched to either the lead or the accompanying track based on their names. If this was not possible, we set the identifier of such a track to a value representing the fact that no such unambiguous distinction could be made. We refer to this as an "unknown" identifier.

We then removed any file that has an invalid assignment of identifiers. For example, we consider files that consist of only one piano identifier as invalid. Furthermore, we removed any track with an unknown identifier from files containing two tracks with piano identifiers.

If after these operations a file consists only of tracks with unknown identifiers, no safe assumption about the structure of the piece can be made. In this case, we opted to delete any file consisting of three or more such tracks. Initially, for files that consisted of exactly two tracks where both had been assigned an unknown identifier, we assumed

that they represented valid files. This approach turned out to include too many files that represented invalid compositions, which is why we decided on removing them as well.

In a last step, we removed any file that has tracks that contain too many empty bars. We decided on removing files that contain more than 40% of bars which are empty.

Starting with 11,417 MIDI files, applying all the procedures for sorting out the dataset resulted in a final amount of 2,147 usable MIDI files which form the basis for our dataset.

4.2.3 Preprocessing Procedure

Starting from the 2,147 MIDI files, we applied a data processing pipeline to create sequences that PAUL-2 could be trained on. We applied a $19:1$ split regarding training and validation data, i.e., 5% of the data was not used during the training procedure but rather to evaluate the performance of the network during this step, in order to combat overfitting. In this section, we cover the details of the preprocessing pipeline.

In an initial step, we converted the MIDI files to sequences using the functionality of S-Coda. We then considered using the scaling functionality as a data augmentation technique. Initially, we scaled all files by the scaling factors of 0.5, 1, and 2, resulting in three different sequences per MIDI file. Later on, we opted to drop the scaling entirely, as it introduced too much noise to the generated files. Instead, we only used the normal versions for further processing.

We then extracted chunks of a fixed length of bars from the compositions. We experimented with the amount of bars to extract, but settled on a value of four bars per chunk. We argue that this amount is well-fitted for the output shape of PAUL-2, as it represents a length that lends itself well to the educational application of the composer. Piano students can be shown pieces consisting of four bars, which we argue is an adequate amount for this application. Sequences shorter than four bars could lack longer-term structure, while longer pieces can overwhelm the students.

We experimented with a *stride operator*, defining the distances between starting points of the sequences of bars. For example, by default, the stride parameter equals the total amount of bars per chunk. In this case, every four bars a chunk of the same length is collected. A stride value of smaller than four would result in bars potentially occurring in multiple chunks, where their position would change. A stride of two would result in a chunk containing bars one through four and a chunk containing bars three through six for example. Although PAUL-2 supports the usage of this stride operator, we opted not to use a stride value smaller than the amount of bars per chunk, as the network tended to quickly overfit when using it.

When extracting the bars into chunks, we checked for the amount of empty bars occurring per track. We opted to disregard any chunk where there were more than 40% of empty bars per track. Although some classical compositions, for example, do contain sections containing a few consecutive empty bars, most of the time this step filters out invalid MIDI files that were not yet removed by the cleansing steps.

Table 4.1: Tokens used for PAUL-2 and the corresponding message types.

Token	Message Type
0	padding
1	start
2	stop
3 – 26	wait message
27 – 114	note on
115 – 202	note off
203 – 217	time signature

In the next step, we calculated the difficulty values of each of the bars in a chunk. We created tensors of the same length as the melody ones in which we store the difficulty values. Each entry in such a difficulty sequence represents the difficulty of the current message. Note that S-Coda provides difficulty values per bar, not per message. We chose to store the difficulties in this way due to the fact that it makes the training process easier, as we can simply use a single-out mask to attend to them, without having to mind the bar boundaries. We refer to Section 4.1.2 for more information on the single-out mask.

In the next data augmentation step, we transposed the chunks of bars. This is done in order to avoid having the network learn patterns that are only applicable to certain pitch values, and is a common machine-learning technique when dealing with MIDI data. We opted for transpositions in the range of -5 to 6 half tone steps, spanning an entire octave. The result of this operation are eleven new sequences, for a total of twelve chunks of different pitch values per original chunk.

We then tokenised the bars of the preprocessed chunks in order to store them as tensors. Table 4.1 shows the numerical values used to tokenise the different message types. For example, a wait message with a wait time of 24 ticks would be tokenised as the token “26”.

PAUL-2 expects data that is uniform in shape, i.e., all of the input data needs to be of the same length. Due to this fact, we opted to limit the length of the tokenised bar chunks to 512 per track. We motivate the choice for this value later on in Section 4.3.

After filtering out chunks that do not conform to this limitation on length, we applied a padding operation to the tensors. Here, we added as many padding tokens to the tensors as needed as to have them be of the fixed length.

Lastly, we utilised TensorFlow’s dataset API to efficiently store the tensors to disk. During the implementational phase, we experimented with several different approaches for storing the data. With some of them, the performance for retrieving the data from the disk was subpar, incurring a high penalty on having to perform many different training iterations. We found that using the API we were able to reduce the access times to mere seconds, which greatly aided the training process.

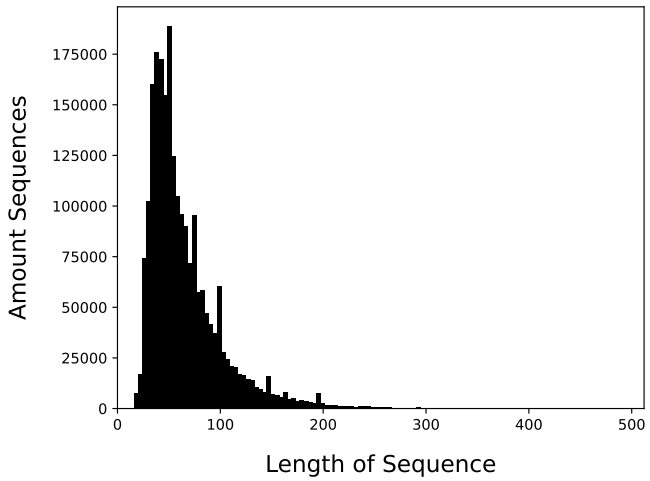


Figure 4.4: Lengths of the sequences used for the training.

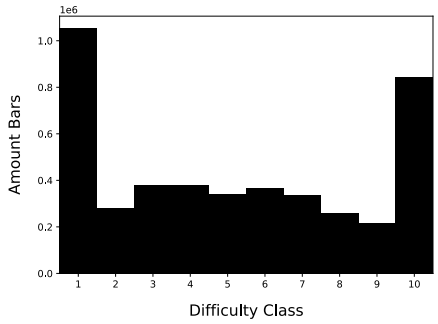
4.3 Analysis of the Dataset

In this section, we provide a brief analysis of the dataset used to train PAUL-2. We believe that it is important to have a general understanding of the dataset in order to be able to interpret the results.

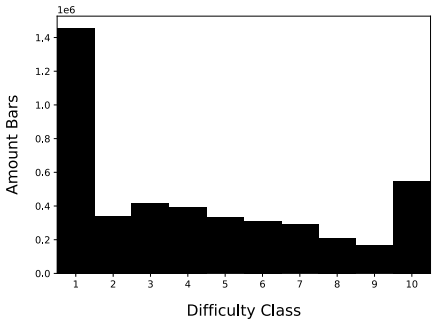
As mentioned in Section 4.2, our dataset consists of about 2,100 MIDI files. After the preprocessing procedure detailed in Section 4.2.3, our data consists of a set of sequences, where each sequence consists of four consecutive bars. As previously stated, we limit the amount of messages per tensor to 512, which is why we disregard sequences longer than that. Figure 4.4 shows the lengths, i.e., the amount of messages of the sequences consisting of four bars each. It is clear to see that the vast majority of the sequences consist of 200 message or fewer, with some outliers consisting of up to 300.

We argue that using a maximum capacity of 512 gives us enough headroom to deal with these outliers, while still keeping the overhead to a minimum. For machine-learning tasks (and computer science in general), it is common to use powers of 2 for any kind of values. We decided to keep this tradition.

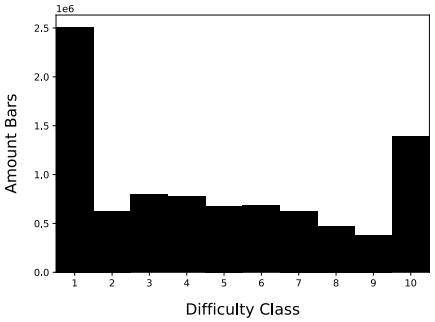
Analysing the difficulties assigned by S-Coda to the set of training data provided additional insights. Figure 4.5 shows the result of this analysis. More specifically, Figure 4.5a shows the amount of bars belonging to a certain difficulty class for the bars belonging to the lead sequences, while Figure 4.5b shows the same for accompanying sequences. Note that the relative proportion of easier sequences is higher for the bars originating from accompanying sequences. We argue that with piano music, the melody is most often played in the lead track, while the accompanying track more often than not contains easier accompanying parts, as the name suggests. Most of the time the melody is more



(a) The amount of bars per difficulty class for lead sequences.



(b) The amount of bars per difficulty class for accompanying sequences.



(c) The amount of bars per difficulty class for both lead and accompanying sequences.

Figure 4.5: The amount of bars per difficulty class for bars of different origins.

difficult to play, which is why the distribution of difficulties of the accompanying track is skewed towards the lower classes.

Figure 4.5c shows the combined amounts of the previous figures. Especially interesting to note is the distribution of the difficulty classes. We initially assumed that the difficulties would follow a standard distribution, with classes 5 and 6 being the most represented. Although one could argue this is roughly the case for classes 2 through 9, classes 1 and 10 are heavily over-represented. We provide two explanations on why this could be the case:

- Due to the fact that our metrics often use certain cut-off points for assigning difficulties, all bars where this measure falls below or above the thresholds will be assigned the exact same difficulty. This way, a large amount of bars could possibly be assigned the same difficulty class even though one could further differentiate them. We argue that although this could be the case, further subdividing this difficulty classification could dilute the training set.

- We note that although S-Coda's metrics are quite in-depth, we do not claim that they perfectly and accurately reflect the "real" difficulty of a piece. For this thesis, our focus rather lies on providing a proof of concept regarding using a transformer architecture to produce pieces of different difficulty classes. We believe that for the majority of bars, the difficulty assignment rather closely corresponds to the real-world difficulty, but we acknowledge that there are some outliers where our metrics miss the mark.

4.4 Training PAUL-2

During the training process, the entire dataset is fed to the networks potentially multiple times. Based on the examples contained in the set, the trainable parameters of the networks are updated in order to reduce the overall *loss* of the network. The loss of a network refers to the quality of its predictions. A large loss value indicates that the network is unable to make correct predictions, while a value of 0 indicates that the network makes perfect predictions.

Updating the weights of the network in order to decrease the loss is an incredibly resource-intensive process. It is not uncommon for a single epoch to take up to several hours of processing time, sometimes even days. Generally, even though CPUs (central processing units) are able to train such networks, they are seldom used. This is due to the fact that GPUs (graphics processing units) are much better equipped to handle such tasks. GPUs usually have a drastically greater amount of processing cores than their CPU counterparts, while also being optimised for matrix calculations. These are of special importance when it comes to neural networks, thus often a great speed-up can be achieved by using GPUs instead of CPUs.

Another advantage of using GPUs over CPUs for the training process is the fact that multiple graphics cards can be used to train the network in parallel. While systems using multiple central processing units do exist, these are more often than not found in the enterprise field instead of the consumer market and as a result quite cost-inefficient. In contrast, systems supporting multiple GPUs are commonplace at the point of writing. Using multiple GPUs allows for an almost linear speed-up of the training process. For the generation or *inference* process, usually only a single GPU is used, since this step is generally not very well parallelisable.

With PAUL, we used an NVIDIA GeForce GTX 1060 6GB[10] that we had available locally in combination with an NVIDIA Tesla K80[11] made available through Microsoft's Azure[12] platform. Note that for the K80, we were limited to only half the available bandwidth and memory capacity, resulting in 12GB of usable memory. In our conclusion [86], we argued that with products like the NVIDIA GeForce RTX 2080 Ti[13] the handling of larger and

[10] https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862/.
[11] https://www.techpowerup.com/gpu-specs/tesla-k80.c2616/.
[12] https://azure.microsoft.com/.
[13] https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305/.

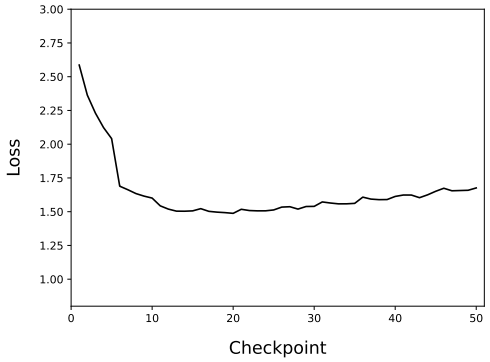
more advanced networks would be possible. Note that upon the end of the lifetime of a product, NVIDIA tends to remove the card's specifications from their website, which is why we provided entries to a third-party database.

With PAUL-2, we were able to utilise two of the most powerful consumer graphics cards available at the time of writing. We used two NVIDIA GeForce RTX 3090[14] cards for the training and inference, which greatly helped with the drastically increased network and dataset dimensions. These cards each possess four times the amount of memory a GTX 1060 6GB does, while more than doubling the amount of memory an RTX 2080 Ti has. Furthermore, the memory speeds, number of cores, and operations per second drastically improved as well. We note that even more powerful cards such as the NVIDIA GeForce RTX 3090 Ti[15] or the NVIDIA A100 PCIe 80 GB[16] exist. The price to performance ratio with these extreme high-end cards is often quite high, with the latter card's MSRP being $13,999. Although commanding a price that is almost ten times that of the RTX 3090's $1,499, the A100 only possesses 3.5 times the amount of memory and less than double the amount of transistors of the latter card.

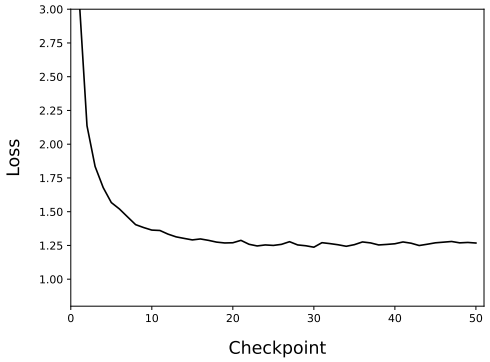
TensorFlow provides a high-level framework supporting different parallelisation approaches. For PAUL-2, we utilised this framework using a specific approach for handling multiple systems, namely the `MultiWorkerMirroredStrategy`. With this strategy, we are able to connect multiple systems (referred to as *workers*), each potentially possessing multiple GPUs, to a computation cluster. In this cluster, each worker is able to contribute computational power to the training process, significantly reducing the time needed to complete an epoch. When experimenting with this set-up, we found that using a set-up of two workers, each providing a single GPU, did not significantly speed up the training process. We argue that the overhead induced by having to distribute the dataset over the network was only just equalised by having double the computational power. We assumed that a larger number of GPUs per worker would remedy this effect. Due to this, we decided on connecting both our GPUs to a single system. This resulted in a speed-up of almost 100%.

The quality of the solutions and the overall performance of neural networks often heavily depend on the selected hyperparameters. With a transformer model, the selection of these parameters can be quite important as well. In the original transformer approach [97], 6 layers, 8 attention heads, a d_{model} value of 512, which corresponds to the dimensionality of the embedding, 2048 units per layer in the feed-forward network, and a dropout rate of 0.1 was used. In machine learning, authors often opt to conduct a simple grid search for hyperparameter selection, select them randomly, or base them on intuition alone. We opted to utilise the `scikit-optimize` Python library, which provides an implementation of a Bayesian optimisation approach, which is widely used for hyperparamter tuning. Here, a probabilistic model of an underlying function is built from a set of input data. For our purposes, this underlying function is the relation of hyperparameters to the

[14] `https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622/`.
[15] `https://www.techpowerup.com/gpu-specs/geforce-rtx-3090-ti.c3829/`.
[16] `https://www.techpowerup.com/gpu-specs/a100-pcie-80-gb.c3821/`.



(a) Performance of the lead network per checkpoint.



(b) Performance of the accompanying network per checkpoint.

Figure 4.6: Final performance of the networks on the validation set per checkpoint.

performance of the network, measured using the validation set. Providing the optimiser with some initial values that were computed manually results in it being able to predict combinations of hyperparameters that are likely to improve either the underlying model or the performance of the network.

We note that evaluating a set of hyperparameters is an incredibly resource-intensive process and quite expensive. In view of this, we were not able to run the optimisation process on the entire dataset. Instead, we opted to run hyperparameter optimisation on a smaller subset, which drastically decreased the time needed to run the computations. For each set of hyperparameters suggested by the optimisation routine, we ran the training process for as long as the model did not show signs of overfitting. We then fed the collected data about the performance of the model to the optimisation routine, which in turn was then able to make more precise predictions.

As a result of this process, we selected the following hyperparameters for our model:

- `num_layers` = 6,
- `num_heads` = 4,
- `dropout_rate` = 0.2,
- $d_{model} = 256$ for P2A and $d_{model} = 512$ for P2L, and
- 512 units per layer in the feed-forward network.

4.5 Generated Results

We are able to report that the output of PAUL-2 is vastly superior to that of PAUL, both in terms of temporal structure and melodic quality. In contrast to our preceding system, PAUL-2 is able to compose accompaniments that represent valid musical sequences.

4.5.1 Analysis of the Final Networks

Figure 4.6 shows the performance of the final networks on the validation set, which contains samples which were never shown to the networks during the training process. For both networks, we opted to run a total of 10 epochs. We used five checkpoints per epoch in order to be able to judge when the networks would start to overfit. Training P2L using these settings took 15 hours of computational time, while training P2A took 10 hours due to the decreased dimensionality. For our final models, we opted to select the checkpoints that induce that smallest validation loss. We use these checkpoints to store the weights of the models. These weights can then be retrieved during the generation process and be used to generate the pieces.

Figure 4.6a shows the performance of the lead network. Here, a clear point from which on the model starts to overfit is discernible. Interestingly, the same does not hold for the performance of the accompanying network, which is shown in Figure 4.6b. The performance of the network seems to stagnate rather than decrease beyond a certain point. We theorise that this could be due to the fact that the capacity of the model is too small, i.e., a larger model would perhaps be able to learn more about the dataset. At this point in time, we are limited by our local hardware, which constrains our choice of hyperparameters. We tried conducting tests with models of slightly increased dimensionality. These also showed no signs of overfitting and would often perform worse on the validation dataset than the model shown in Figure 4.6b which uses the hyperparameters discussed in Section 4.4.

PAUL-2 uses 10 levels of difficulty for the output pieces. In contrast to PAUL, each of the two networks is able to output pieces of all difficulty levels instead of having to rely on a specific set of weights to do so. We argue that this could potentially improve the performance of the networks since each network is shown a greater amount of training data. One drawback of this method is the fact that the networks could be less accurate when it comes to generating pieces of the desired difficulty level. With PAUL, we saw an accuracy of about 66% when being tasked to generate a piece of the second of the three possible difficulty levels. We assume that if PAUL-2 would also only use three difficulty levels, this accuracy could potentially decrease.

We argue that these potential drawbacks are heavily outweighed by the advantages of our method. With PAUL we could only use sequences of bars of the same difficulty level. This approach would not be feasible with PAUL-2, as the pieces are assigned one of ten different difficulty classes. This diversification would result in almost no sequences containing more than a single bar if the old approach was chosen. Furthermore, we argue that the old approach is not scalable when it comes to a greater number of difficulty classes, as the number of training samples per difficulty class linearly decreases with the amount of classes.

Figure 4.7 shows a confusion matrix of the specified difficulty versus the generated difficulty of bars generated using the lead network. For each difficulty class, 80 bars were generated. These bars were generated using only an initial time signature message, no

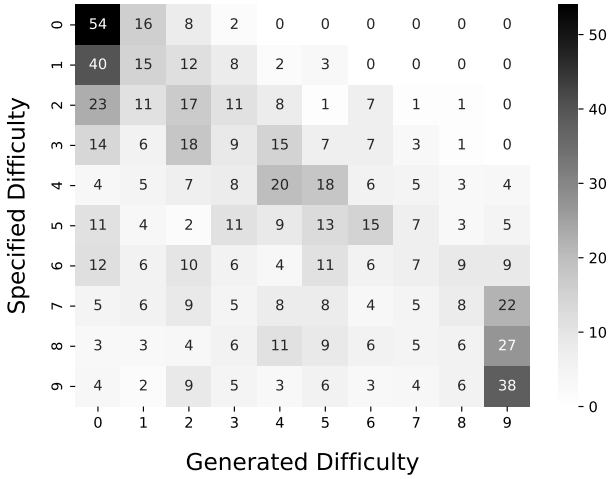


Figure 4.7: Confusion matrix of specified difficulties versus generated difficulties of bars generated by P2L.

previous bars or additional information was provided. For the easier difficulty classes, the network seems to perform quite well. Here, a correlation between specified and generated class is clearly visible. When it comes to the more difficult classes the network tends to either produce bars of the most difficult class or bars that are substantially easier than specified.

At this point, we once again note that PAUL-2 still only serves as a proof of concept, especially when it comes to the difficulty assessment of the bars. It may be that some of our assessments do not accurately reflect the difficulty and as a consequence the networks are not able to correctly learn to generate the desired pieces. Further investigation and research done by musical experts is needed to accurately reflect the real difficulty of piano pieces. For our purposes, the difficulty value assigned by S-Coda suffices. Furthermore, the confusion matrix shown in Figure 4.7 shows promising results and suggests that the network takes the specified difficulty parameter into account when generating the bars.

Figure 4.8 shows the same confusion matrix as Figure 4.7 but for the accompanying network. Note that for this evaluation we could theoretically construct ten such matrices, each showing the results of the generation process for an input sequence of a different difficulty class. For this evaluation, we opted to use the fifth bar of Chopin's *Fantaisie-Impromptu*, as shown in Figure 2.1.

Interesting to note is the fact that the network tends to produce bars that are easier than specified, which could be attributed to the fact that accompanying sequences for piano music are less difficult in general. Other than that, a clear correlation between specified and generated difficulty is visible again. We would argue that for the accompanying network the difficulties are a bit less scattered than for its lead counterpart. A drawback of the network is the fact that it seems to often generate bars of the easiest difficulty.

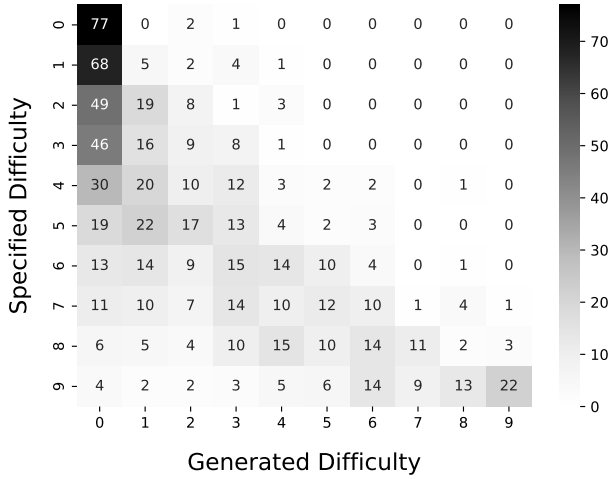


Figure 4.8: Confusion matrix of specified difficulties versus generated difficulties of bars generated by P2A.

This could be due to the fact that the network is prone to generating very sparse or sometimes even empty bars, which are always assigned the easiest difficulty evaluation.

Even though the networks do not produce bars of the desired difficulty all of the time, this can be remedied by applying a specific approach for the generation of pieces. The transformer architecture lends itself very well to parallelisation, as do GPUs with their large number of processing cores in general. We decided to make use of this fact and apply a technique usually not used for machine translation. Instead of creating a single sequence for each of the networks, we opted to generate a multitude at the same time. Experimenting with these settings, we found that we were able to generate 16 such sequences at a time with our local hardware. This allows for the consecutive difficulty assessment of all of these sequences. We are then able to select the generated sequence whose difficulties best conform to the specified parameter, or—if the discrepancy between the difficulty assessment and the specified difficulty is too great—generate new such sequences.

In practice, we do not create sequences of the maximum length (four bars in our case) at a time, but rather split them up into smaller segments. We opted to always start the generational process with a chunk of two bars and reset this number every time a new valid bar or several are found. Consecutive failures of generating bars of the correct difficulty results in a decreased number of bars to generate, which in turn lowers the amount of time needed for each iteration. This way we can ensure that the generated sequences either perfectly or closely match our difficulty specification. We argue that using our approach the architecture is able to scale quite well due to the fact that it is very well parallelisable using additional GPUs and processing power.

4.5.2 Generated Compositions

Figures 4.9 to 4.17 show nine example outputs generated by PAUL-2. We decided on providing three different samples per difficulty class while only covering three difficulty classes instead of providing one sample of each difficulty class. This is due to the fact that we wanted to highlight the program's capabilities of creating vastly different pieces that can be categorised using the same difficulty evaluation. Furthermore, we argue that the differences between two adjacent difficulty classes are often quite ambiguous, especially when only viewed using the piano-roll visualisation. Thirdly, we allow discrepancies of up to one difficulty class in either direction when generating the bars, i.e., a sequence of bars of difficulties 4, 5, and 6 would be a valid output when tasked to generate a sequence of difficulty 5. This greatly helps reduce the time needed to generate new pieces while the pieces in general still conform to the difficulty specification.

As shown in Figure 4.7 and Figure 4.8, PAUL-2 is well capable of generating sequences of bars that conform to a given difficulty specification in general. Here, the program often generates bars of adjacent difficulties to the one specified, but seldom bars whose difficulties are further apart. We exploit this fact by allowing difficulty discrepancies of up to one difficulty class in either direction.

The output of PAUL-2 consists of valid musical pieces that can potentially be used in a tutoring environment. The sequences are characterised by their rhythmic validity. Although (as is the case with most machine-learning applications) PAUL-2 generates failure samples from time to time, more often than not the output of the program is rhythmically consistent between the tracks. We define failure samples as those sequences that lack rhythmic dependence between the tracks, contain too many empty bars, or contain parts that would be unplayable even to a skilled pianist.

Figure 4.18 shows the score representation of three samples of different difficulties generated by PAUL-2. Here, the differences between the difficulties of the samples is more apparent. More complex pieces tend to play more notes at once while also arranging them in more complex rhythms. The piece shown in Figure 4.18c even contains triplets.

Although a clear rhythmic dependence between lead and accompanying track is observable, one drawback of PAUL-2's output is the frequent lack of melodic consistency. PAUL-2 tends to create pieces that can sound unpleasant to the listener's ear due to dissonances between the notes of the two tracks. We note that the individual output of both tracks sound musically valid to our ears, although we believe the output of both MuseNet [74] and the Music Transformer [45] to be superior to ours.

In what follows, we we give an analysis of the reason of this and how this could be circumvented in future versions to further improve the performance of PAUL-2.

4.5.3 Future Improvements

Firstly, we strongly believe that the output of PAUL-2 would be of higher quality if we had access to the same computational power and to datasets of comparable size as the

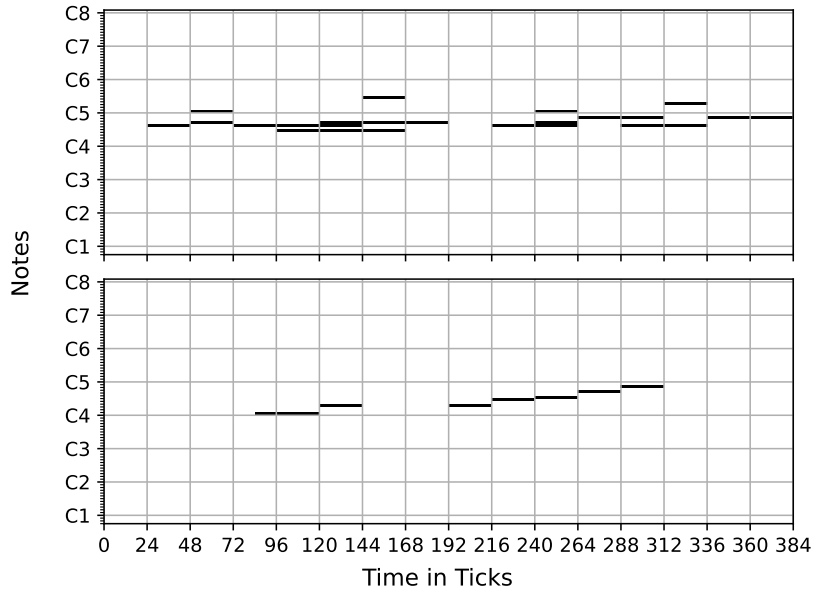


Figure 4.9: First example output of difficulty 2.

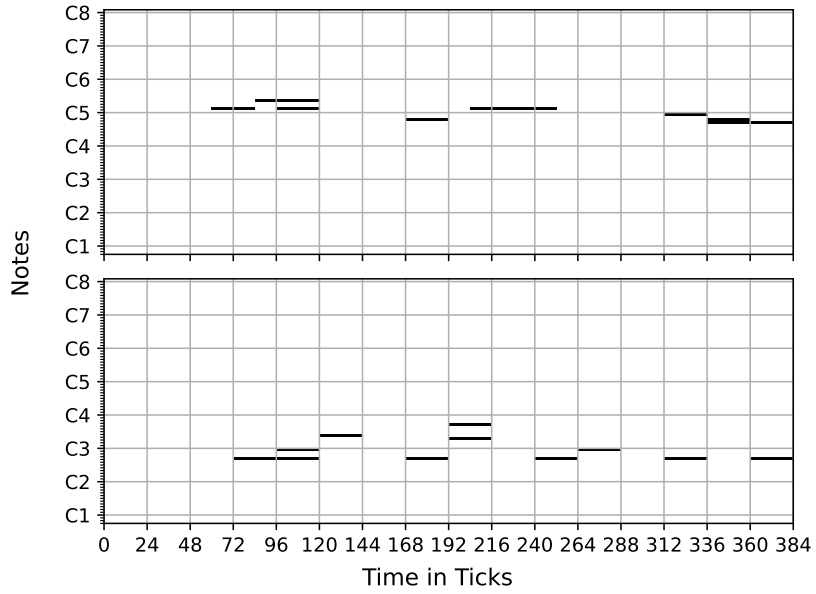


Figure 4.10: Second example output of difficulty 2.

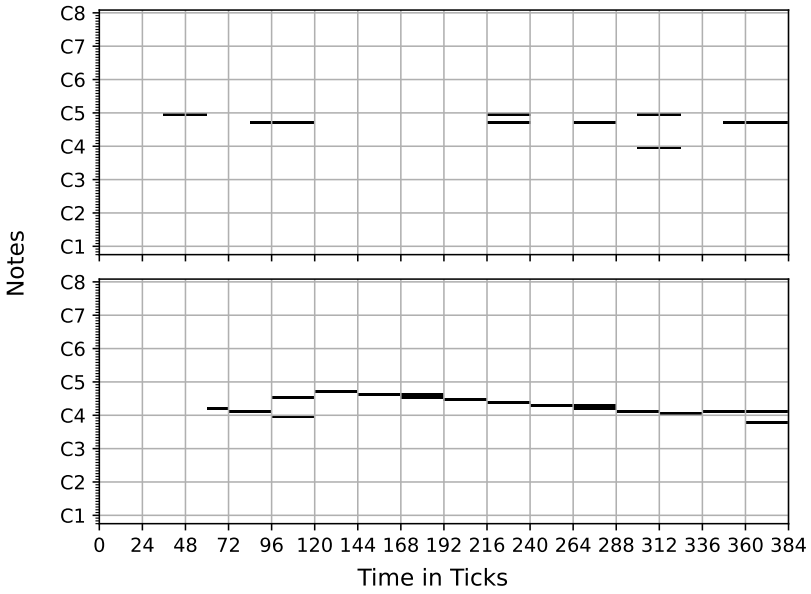


Figure 4.11: Third example output of difficulty 2.

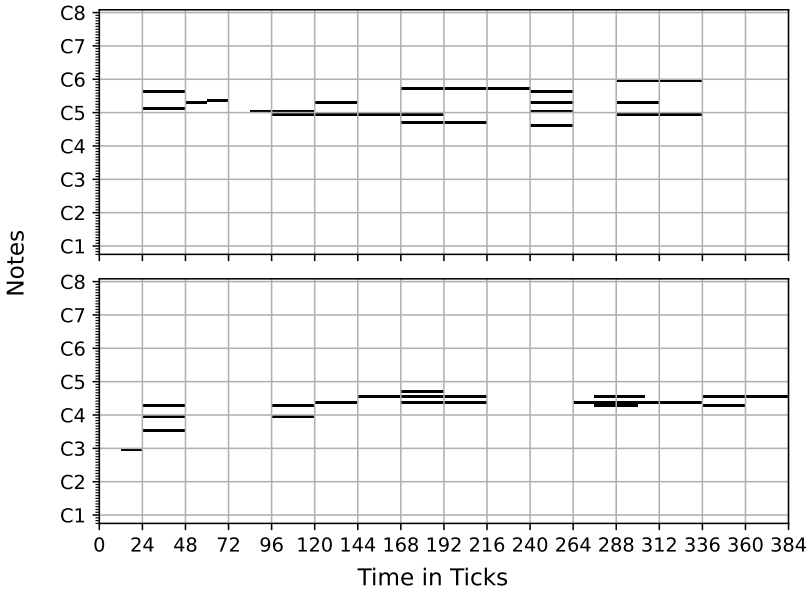


Figure 4.12: First example output of difficulty 5.

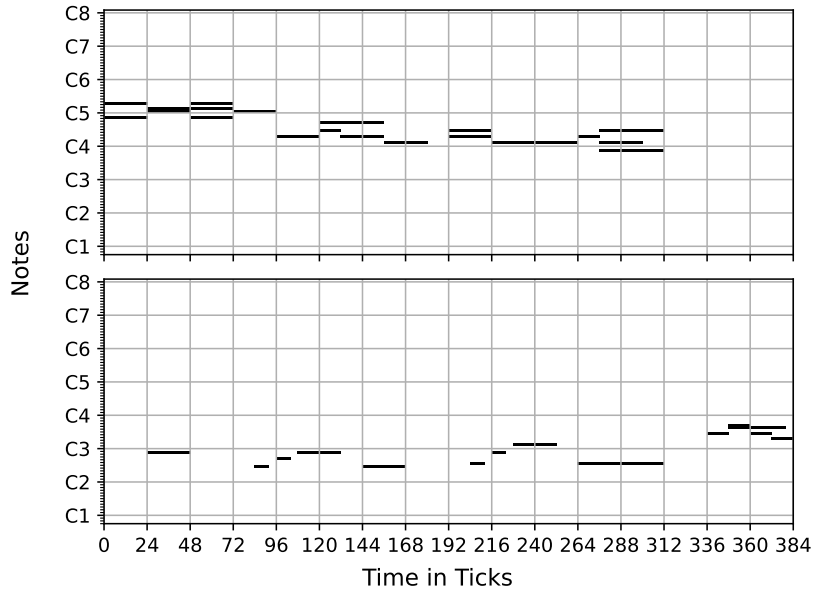


Figure 4.13: Second example output of difficulty 5.

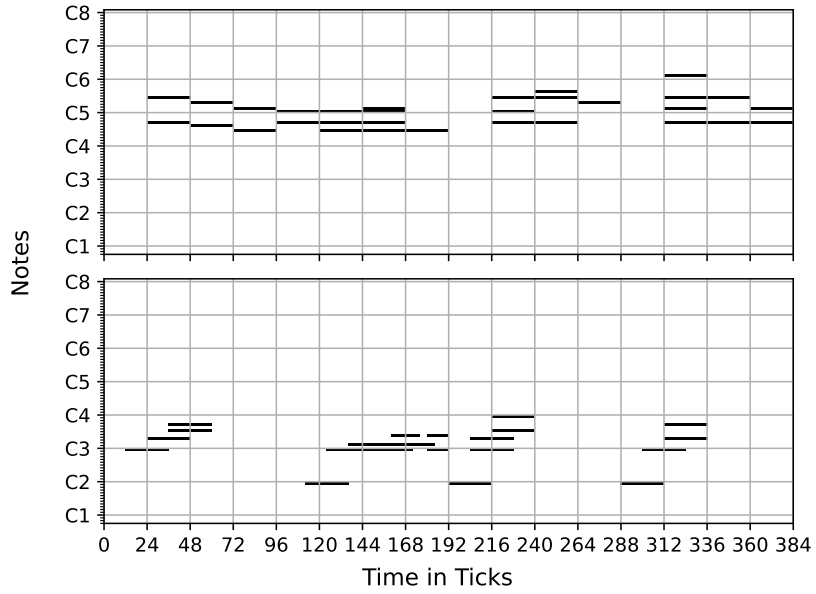


Figure 4.14: Third example output of difficulty 5.

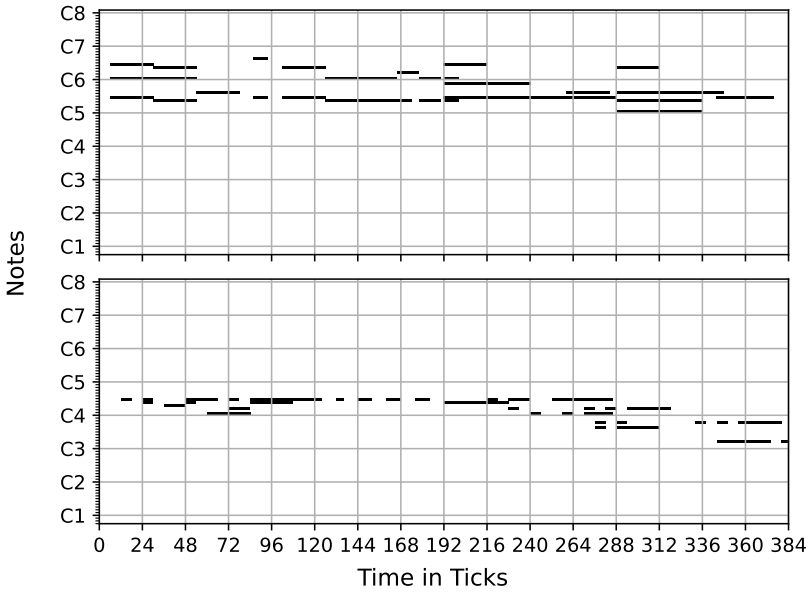


Figure 4.15: First example output of difficulty 8.

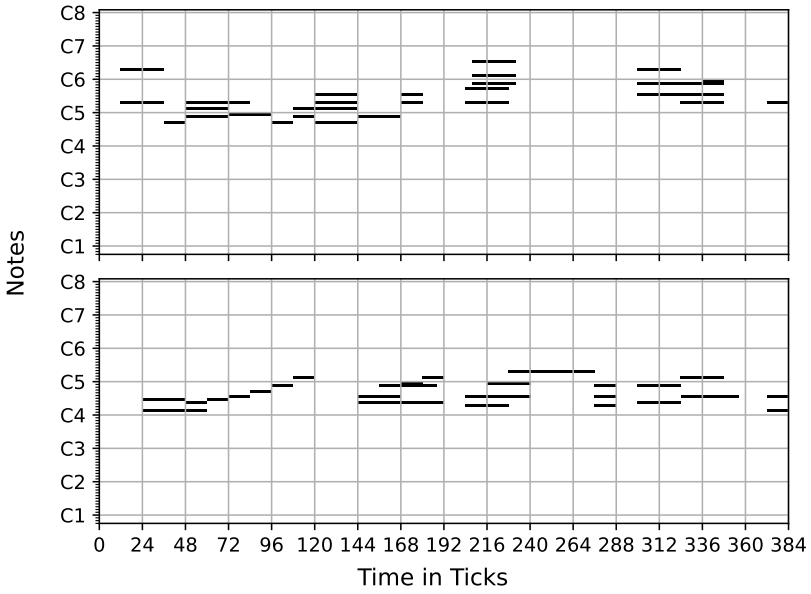


Figure 4.16: Second example output of difficulty 8.

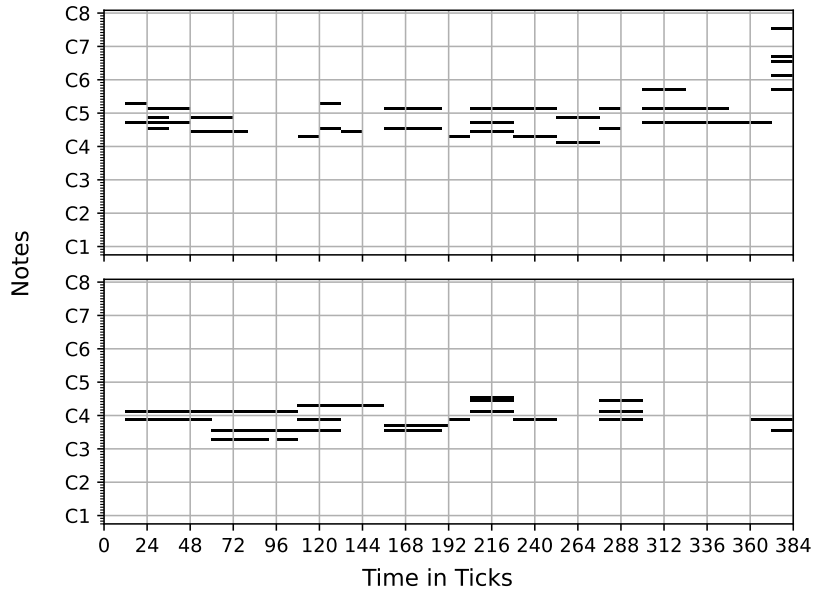
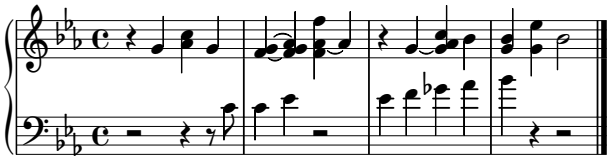


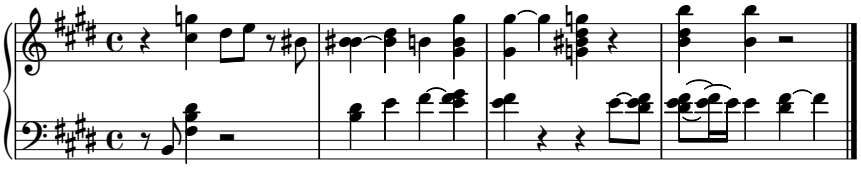
Figure 4.17: Third example output of difficulty 8.

two mentioned works had. Machine learning is an extremely resource-intensive process, competing with state-of-the-art approaches usually takes an amount of computational power that can only be provided by high-profile backers such as Google, who owns Magenta where the Music Transformer originated. We argue that, compared to the vast computational power the mentioned works had at their disposal, the output of PAUL-2 is of excellent quality. For example, with PAUL-2, we used a 6-layer architecture with 4 attention heads, while MuseNet utilised a 72-layer architecture with 24 attention heads, which require vastly more computational power to train.

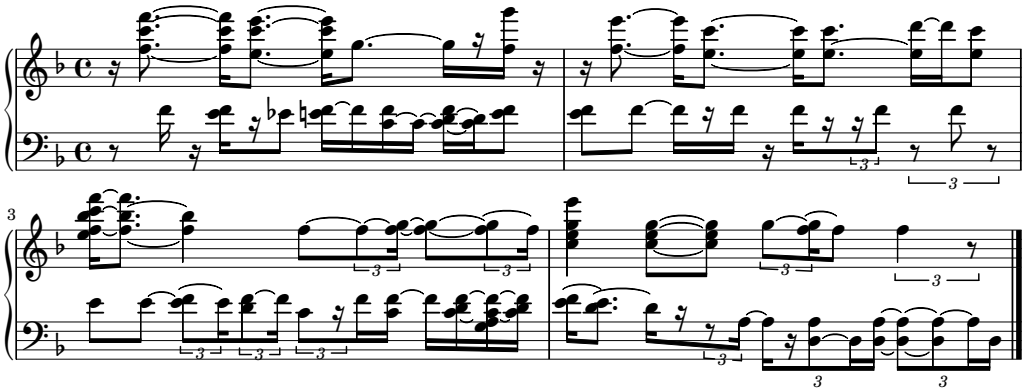
Secondly, we want to highlight the fact that with PAUL-2 we took an entirely different approach as, e.g., MuseNet. Our primary goal was to create an algorithmic composer capable of generating sequences based on a difficulty parameter. To the best of our knowledge, PAUL and PAUL-2 are unique in this regard. We successfully implemented such a composer and showed that it is indeed capable of attending to the provided difficulty specification. Furthermore, in contrast to these state-of-the-art approaches, we utilise not only a single but two encoders to encode both the leading sequence and the difficulty specification. To our understanding, MuseNet and Music Transformer completely drop the need for an encoder, relying only on the decoder to produce the output. We argue that the melodic consistency of PAUL-2's output could be improved if we did the same. For piano sheet-music notation, the existence of two independent tracks is quite important as there often exist rhythmical differences which would clutter the visualisation if not separated.



(a) Score representation of the sample shown in Figure 4.9.



(b) Score representation of the sample shown in Figure 4.12.



(c) Score representation of the sample shown in Figure 4.15.

Figure 4.18: Score representation of three samples created by PAUL-2.

We devised several methods that could potentially improve the melodic consistency of PAUL-2's output. In subsequent future work, we would like to compare different methods of representing the musical sequences. Instead of using absolute values for the pitches, we would like to experiment with representing them in a relative fashion, i.e., storing the changes in pitch rather than the pitches themselves for each note. Furthermore, we would like to research whether a symbolic representation that is closer to the sheet music notation can improve the program's performance. For this endeavor we would like to replace the start, stop, and wait messages used in this thesis by a combination of new types of messages. In fact, we would like to utilise messages that are inspired by the velocity messages as used by Huang et al. [45] and the running-status approach as used in the MIDI protocol. Here, each of these messages would signal the note value of all the following notes until being overwritten. For example, a message could signal that all the

following pitches should be interpreted as quarter notes. This approach, in combination with rest messages, would allow for the same expressiveness as our more conventional approach does while eliminating the need for stop messages. We conjecture that this method could help the network create more rhythmically correct pieces.

MuseScore[17] is a music notation software that allows for automatic conversion of MIDI files to sheet music. The software provides functionality for the automatic separation of MIDI files into lead and accompanying tracks based on some algorithmic procedure. Although this is not an exact procedure, we devised two use cases for a similar approach.

Firstly, as already mentioned, having PAUL-2 generate a single track containing information from both the lead and the accompanying track could remedy its melodic shortcomings. We argued that for its intended use case the separation of music into two distinct tracks is quite important. An algorithmic procedure capable of splitting a single track into a lead and accompanying track could at this point be applied to the output of this theoretical algorithmic composer. The output would be of similar shape as the one PAUL-2 is able to produce at the moment.

Advantages of this method are the theorised improvement in melodic quality and the fact that no overlaps between the two tracks can occur, i.e., it will never be the case that the lead track contains lower pitches at the same time the accompanying track contains higher ones. Such an overlap would be quite challenging, perhaps even impossible to perform by a pianist. The main disadvantage of such an approach is the fact that it will most likely be quite inaccurate in its assessment of which note belongs to which track, as the separation of the two tracks is a complicated task. In theory, a neural network could be trained to handle this task. The main problem with this approach is the same that PAUL-2 has: the available datasets are often of insufficient size or quality.

Secondly, such an algorithmic procedure could be applied to some of the datasets of greater size, producing more data to train on. The same hurdles as with the first application of such a procedure exist, namely the likely inaccuracy of the split. Training the networks on a dataset that is the result of such an operation could result in outputs that exhibit some deficiencies that were introduced by the splitting procedure. For example, this output could contain a greater number of empty bars per track than compared to an output produced by a network trained on unmodified data.

We note that PAUL-2 tends to generate samples that contain a lot of voices which were modeled using ties. This can make the pieces a lot harder to play than they would otherwise be. We conjecture that changing the representation of musical sequences to one closer to the score notation could remedy this fact. We also note that the training corpus contains pieces that further encourage this behaviour. For example, sometimes notes stop shy of the beginning of the next note, inducing a rest that only complicates the piece without providing any musical advantages. A heuristic could be applied to remedy such notes in future iterations of the program.

[17] `https://musescore.org/`.

At the moment PAUL-2 does not explicitly prohibit the generation of pieces containing overlapping tracks. We argue that a simple transposition operation could remedy such tracks. In the future, the relative representation of pitches could also resolve such issues.

We provide the entire source code of PAUL-2 and the discussed generated samples at the following repository:

`https://github.com/FelixSchoen/PAUL-2/.`

4.6 Differences Between PAUL and PAUL-2

We now briefly highlight the main differences between PAUL-2 and its predecessor system, PAUL [86, 87]. Although the general aim of both systems is the same (viz. being the generation of musical prompts of varying, user-specifiable difficulty), the manner in which this is achieved greatly differs.

With PAUL, we employed a basic recurrent network using LSTMs in order to create the lead sequences, and a basic LSTM-based sequence-to-sequence network in order to generate the accompanying sequences. These networks consisted of three (respectively six) LSTM layers with 1024 units per layer. Training these networks took about six hours on a locally available NVIDIA GeForce GTX 1060 6GB[18] GPU for the treble network, while training the accompanying network could only be conducted on a rented NVIDIA Tesla K80[19] GPU. No experimentation regarding hyperparameters was conducted over the course of training PAUL.

With PAUL-2, we eschewed recurrent neural networks in favour of the superior and vastly more advanced transformer architecture [97]. Training was conducted on two locally available NVIDIA GeForce RTX 3090[20] GPUs and took about 10 to 15 hours even though training was performed on a much larger training dataset. The speed-up in training can be attributed to the more advanced architecture which solely uses attention instead of recurrence, and the exceedingly more powerful graphics processing units. With PAUL-2 we experimented with a plethora of different hyperparameters, trying to find the ideal values for our setting.

An enormous advantage of PAUL-2 over PAUL comes in the way it handles the different difficulty classes: with PAUL we made use of a total of three different classes. For each of the two network types, we had to create three different sets of weights corresponding to the three difficulty classes. We could then only ever train these versions of the network on pieces of the same difficulty, which resulted in each of the networks only seeing a fraction of the available data during the training process.

This approach is not scalable for a higher amount of difficulty classes, as this would dilute the training set too much and result in each of the versions of the network only ever

[18] `https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862/.`
[19] `https://www.techpowerup.com/gpu-specs/tesla-k80.c2616/.`
[20] `https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622/.`

receiving a small amount pieces as input data. We furthermore note that even though the amount of difficulty classes used by PAUL was less than a third of the ones supported by PAUL-2, the accuracy with which the network generated pieces was not necessarily better relatively speaking. As shown in Figure 4.7 and Figure 4.8, PAUL-2 is able to very accurately generated pieces of the desired difficulty.

Finally, let us discuss the quality of the generated pieces. In general, the output quality of PAUL-2 is vastly superior to that of PAUL. With our previous system, we were not able to generate valid accompanying sequences, as the MIDI data generated showed a complete lack of structure. This is not the case with PAUL-2, where we are able to generate valid and pleasant musical pieces. We note that the quality of the lead sequences of PAUL-2 is also greatly improved over the one of our previous system. We attribute these advantages to three main aspects:

(i) The more advanced architecture is able to make use of attention to create output pieces of higher quality.

(ii) With a dataset consisting of a much greater amount of pieces, we were able to train the networks to a higher degree.

(iii) The improvements regarding the difficulty classes allowed for a singular version for each of the two networks, resulting in them being shown the entirety of the training dataset and thus having an even greater amount of training data at hand.

4.7 User Study

Although PAUL-2 serves as a proof-of-concept implementation, we were still interested in evaluating its performance. We conducted a medium-scale study with 60 participants in order to assess the musical quality of the program's output. In this study, participants were presented with multiple pairs of short musical phrases. They were then asked to listen to both of them and decide on a sample whether they believed it to be genuine or computer-generated. This approach is somewhat inspired by the Turing Test [96, 83], in which a human interrogator has to assess whether they are communicating with a computer only by passing textual questions and receiving answers of the same kind.

Each of the questions involving the comparison of two musical samples consists of one sample generated by either P2L or P2A and one sample consisting of 4 consecutive bars from the test dataset. This way we can ensure that, when generating the accompaniment, PAUL-2 has not seen the input sequence before. Furthermore, we decided on using samples of different difficulty levels for this study. We settled on using samples of difficulties 2, 5, and 8 in a similar fashion as with the previous evaluations. We argue that this selection allows for the analysis of the impact of the difficulty parameter on the perception of a human listener, while at the same time not diluting the results of the study.

When designing the study, we placed special emphasis on only comparing pieces of the same difficulty level, e.g., no comparisons between a piece of level 2 and 8 are to be

made. In order to do so, we extracted consecutive segments of the desired length and difficulty from the test set. We then randomly drew about 30 samples from this set and decided on the 18 samples that best represented the training set according to our opinion. Note that for these 18 samples, 9 were taken from lead tracks while 9 were taken from accompanying tracks.

We then generated double the amount of lead samples needed per difficulty, e.g., we generated six lead samples of difficulty 2 from which we selected the three samples that best represented the overall performance of PAUL-2 according to our opinion. We decided on this approach as to eliminate possible failure samples.

For the accompanying samples we generated samples based on the same lead sequence the genuine sample is based on. Here, we generated up to three different samples from which we then selected a single sample to be used in the study. We decided on this larger proportion regarding generated to needed samples as we argue that this equalises the fact that for the lead samples we were able to draw 9 samples from as set of 18, while for the accompanying ones we had to draw a single sample from a set of three for a total of nine times.

We then randomly paired up samples of the same difficulty. We argue that randomising this step is quite important, as otherwise deliberate gerrymandering could be exploited in such a fashion as to skew the results of the study. For example, pairing all the most convincing genuine samples with the weakest generated samples could result in preferential treatment of the generated ones for the rest of the matchups.

In the study, participants were presented with 18 different pairs of samples. We interleaved the difficulties of the samples, i.e., the first question consisted of comparing samples of difficulties 2, while the second question compared samples of difficulties 5, and so on. The first nine questions consisted of lead samples, while the last ones consisted of accompanying samples. The users were not told about the differences in difficulty or origins of the prompts. At the end of the study, we asked participants to self-evaluate their level of musical proficiency. Furthermore, we asked them to rate how confident they were in their differentiation between generated and genuine samples. Lastly, participants were asked to rate the quality of the samples they believed to be computer-generated compared to the samples they believed to be genuine.

We note that we deliberately decided on not comparing computer-generated pieces that consist of two tracks with genuine ones. This is due to the fact that although PAUL-2 is able to generate valid two-track piano pieces they are still very easily told apart from genuine piano tracks. We further note that it was never our main goal to generate pieces that would "fool" a target audience, rather to create compositions that would lend themselves well to an educational context.

With PAUL [87], we conducted a study where we tried to evaluate if our difficulty assessments were consistent with the actual difficulty of the pieces. We presented the participants with samples and asked them to the assign them one of the three difficulty classes used for PAUL. We did not conduct such an evaluation with PAUL-2. This is mainly

Table 4.2: The samples used for the study, their difficulties, and the origin of the genuine ones.

Order	Sample 1	Sample 2	Type and Difficulty	Genuine Sample Origin
1	unit	loss	Lead, 2	No Holly for Miss Quinn
2	ear	baseball	Lead, 5	Cleopha
3	agency	idea	Lead, 8	Cleopha
4	assistant	meaning	Lead, 2	Romance
5	buyer	injury	Lead, 5	Don't Speak
6	hotel	year	Lead, 8	Fourth of July
7	context	employee	Lead, 2	The Bare Necessities
8	shirt	college	Lead, 5	One Winged Angel
9	town	patience	Lead, 8	Mine Cart
10	flight	accident	Accompanyment, 2	Chim Chim Chiree
11	instance	message	Accompanyment, 5	Preludio con Fuga
12	version	debt	Accompanyment, 8	Invention 15
13	judgment	delivery	Accompanyment, 2	I Will Always Love You
14	internet	sympathy	Accompanyment, 5	Romance
15	photo	marriage	Accompanyment, 8	Kitten on the Keys
16	method	two	Accompanyment, 2	To Life
17	night	leader	Accompanyment, 5	Wendy's House
18	engine	hearing	Accompanyment, 8	You'll Be in My Heart

due to the fact that we argue the study conducted with PAUL showed that our metrics were rather accurate, and we only improved upon them with PAUL-2. Furthermore, we once again want to emphasise the fact that the difficulty evaluation only serves as a proof of concept that we are able to accommodate such a parameter when generating a musical piece.

Table 4.2 shows the samples used for the study. We used randomly generated nouns to refer to these samples. This way we could guarantee that the nomenclature would not give any hints about the type of sample while still being able to differentiate between them. The underlined samples represent the computer-generated ones, e.g., "unit" is a sample generated by P2L. We also note that we used the services of

`http://random.org/`

to simulate a coin flip whose result we used to decide whether the computer-generated sample should be used for the first or for the second slot. The last column shows the (cleaned) file names of the genuine pieces used for the corresponding question.

Table 4.3 shows the main results of the study. Since we asked the participants to report their self-assessed level of musical proficiency we were able to take this aspect into

Table 4.3: Results of the study per self-reported musical proficiency. Values rounded to two decimal places.

Subset	Selected Generated	Quality Mean	Confidence Mean	Number Samples
Overall Results	26%	3.20 ($\sigma = 1.05$)	2.52 ($\sigma = 1.07$)	60
Proficiency 1	22%	3.44 ($\sigma = 1.01$)	2.56 ($\sigma = 1.01$)	9
Proficiency 2	29%	3.10 ($\sigma = 1.10$)	2.30 ($\sigma = 1.06$)	10
Proficiency 3	32%	3.57 ($\sigma = 0.93$)	2.43 ($\sigma = 0.98$)	21
Proficiency 4	23%	2.75 ($\sigma = 1.13$)	2.56 ($\sigma = 1.15$)	16
Proficiency 5	19%	2.75 ($\sigma = 0.96$)	3.25 ($\sigma = 1.50$)	4

consideration for our analysis. With an overall score of only 26% of the computer-generated samples having been selected as genuine ones, PAUL-2 does not yet pass our "Turing Test". In an ideal case, we would see values closer to 50%, as this would indicate that the participants had to guess rather than recognise which of the samples are genuine. As stated before, our main research goal was not to create an output that would perform better in such a test but rather a composer capable of creating pieces conforming to the difficulty specification, which we have already shown PAUL-2 is capable of. The results obtained by the study still indicate that the quality of PAUL-2's output is quite good, as it often can be confused with genuine pieces.

Interesting to note is the relationship between the self-assessed proficiency and the rate of how often the generated pieces are assessed as genuine ones. We would have assumed that with higher proficiency the amount of computer-generated samples would strictly decline. This is the case with levels three through five, where each consecutive increase in self-reported proficiency reduces the amount of selected computer-generated samples. Interestingly, this is not the case for the first two levels, as there is an increase of wrongly categorised samples from level one to level two. The same holds for the step from level two to level three. We argue that this could be the case due to using self-assessed proficiency only, since this value can widely vary, even between participants of the same (real) level. According to the Dunning-Kruger effect [26], people tend to overestimate their own abilities in subjects they have no expertise in, while people with some experience tend to underestimate themselves. This effect could also have caused these anomalies. We furthermore note that although a sample size of 60 participants is large enough for a thorough analysis, the fact that we use five different categories can dilute the dataset in such a way that the categories themselves become very prone to outliers.

We asked the participants to state how they would rate the quality of the samples they believed to be computer-generated compared to the ones that they believed to be genuine. Table 4.3 shows the results regarding this question. We provide both the mean and the standard deviation of the score out of five. With an overall value of 3.20, we argue that the musical quality of PAUL-2 shows good promise. It is quite important to note that

Table 4.4: Results of the study per difficulty class.

Subset	Selected Generated	Number Samples
Difficulty 2	28%	354
Difficulty 5	26%	349
Difficulty 8	26%	350

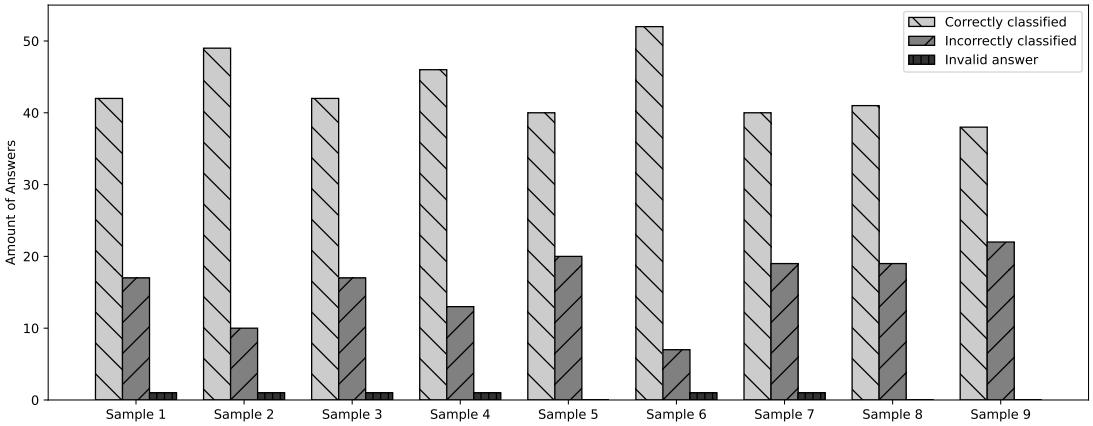
participants were never told which of the samples were genuine and which ones were not. This in turn means that for the answering of this question they had to rely on their previous assessment. We argue that it is likely that for this question the participants will have only considered the worse of the two samples of all the previous questions, independently of whether or not this was actually the computer-generated one or not. This can result in the quality scores being skewed towards a lower quality, which is important to keep in mind when analysing these results.

The same anomaly as discussed with the percentage of the selected generated samples is again observable. Indeed, the average reported quality of participants that rate their musical proficiency at level three is higher than the one of participants of levels two and one. Both the discrepancy and the standard deviation of the second level are especially large. We assume that an even larger sample size could provide more clarity regarding this fact.

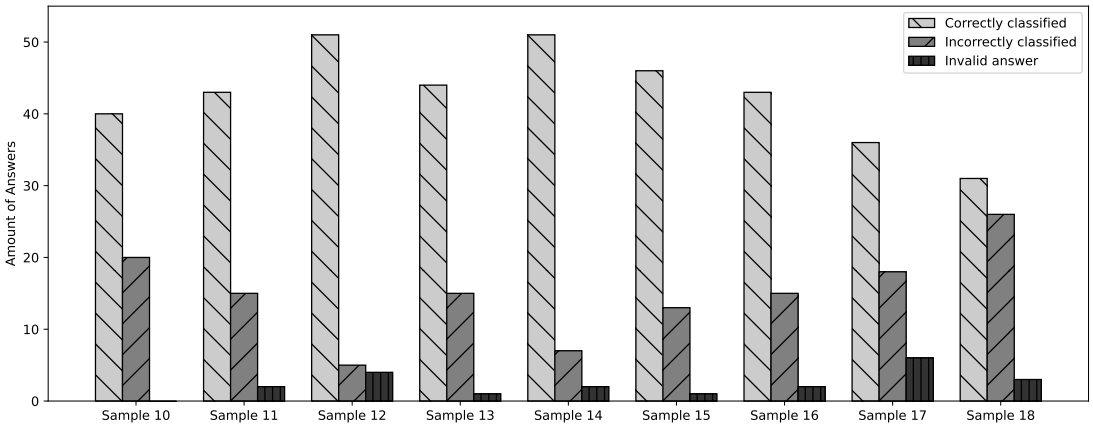
Participants were asked to report their confidence regarding their evaluations on a scale of one to five. We note that for proficiency levels one through four, these scores are quite low. This score is higher for level five but in this case the standard deviation is quite large although the sample size of the category is very small. This leads to believe that although the participants performed quite well regarding the separation of computer-generated and genuine samples, it could be the case that this was based on subconscious feelings alone rather than logical reasoning.

We were able to communicate with some of the participants which often stated that they found it extremely difficult to differentiate between the two samples. One participant went as far as to suggest implementing an answer that allows the subject to state that they deem the samples to be equivalent. Although we considered this approach when designing the study, we deliberately decided against it. We argue that if the participants are forced to choose between the two samples, they can rely on their subconscious feelings as well, which seems to have played an important part for this study. We furthermore argue that if the samples were truly of equal value, this would solidify in a closer percentage of selected computer-generated samples when averaged over a large number of participants and samples.

We would have assumed that the difficulty of the samples would also play a role in the percentage of the correctly classified ones. Table 4.4 shows the results of the study grouped by difficulty class of the samples, which suggests otherwise. There seems to be



(a) Results of the study per lead sample.



(b) Results of the study per accompanying sample.

Figure 4.19: Results of the study per individual sample.

only a small difference in the amount of wrongly categorised samples between the groups of different difficulties. We note that, in our opinion, the output of higher difficulty is quite easily distinguishable from genuine pieces. This is in line with the results observable from the study, although we would have assumed the differences to be more drastic.

At this point, we note that participants were provided with an option stating that they knew either of the pieces, which is why the number of samples per difficulty class of Table 4.4 differs. Furthermore, after some initial problems with the playback of the samples, we asked users that encountered some problems with one or more of the samples of a question to select the same option.

Figure 4.19 shows the results of the study per sample. These samples correspond to

the ones shown in Table 4.2. To improve visual clarity, the bars in Figure 4.19a and Figure 4.19b show the amount of correct answers for each of the questions rather than the participants that selected the first versus the second sample. Notable outliers are samples 12 and 18. Question 12 compared a computer-generated sample of difficulty 8 with an excerpt of Bach's *15 Inventions*. This is very sophisticated musical piece, while the computer-generated sample exhibited some characteristics of a failure sample. Furthermore, a larger percentage of participants compared to the other questions stated that they were familiar with the sample. The genuine sample used for question 18 was quite monotone, which could be the reason for the increased number of incorrect classifications.

All in all, PAUL-2 shows very good promise when it comes to both the musical quality and the eligibility of the samples for usage in an educational context.

CHAPTER 5

The S-Coda Library

In this chapter, we introduce S-Coda, a Python library for preprocessing MIDI files. This preprocessing includes but is not limited to quantisation, consolidation, merging, and difficulty assessment of music data. S-Coda was developed alongside PAUL-2, but it can be used independently of it.

The chapter is organised as follows: In Section 5.1, we lay down the general structure of S-Coda. Section 5.2 covers S-Coda's sequence transformation and assessment capabilities. Finally, in Section 5.3, we provide a short guide on how to use S-Coda.

5.1 Structure

The following list provides an overview of S-Coda's dependencies:

- Mido 1.2.10[1],
- numpy 1.22.3[2],
- matplotlib 3.5.2[3], and
- pandas 1.4.1[4].

S-Coda is freely available from the *Python Package Index*, located at

`www.pypi.org/project/sCoda/`.

[1] `https://mido.readthedocs.io/`.
[2] `https://numpy.org/`.
[3] `https://matplotlib.org/`.
[4] `https://pandas.pydata.org/`.

Table 5.1: Data fields of an S-Coda `Message`.

Field	Description
`message_type`	Determines the type of the message. See Table 5.2 for the possible `message_types`.
`time`	This field can represent both the amount of time to wait for `wait` messages, and the absolute position in time the message occurs at for messages of absolute sequences.
`note`	The value of the note the message represents.
`velocity`	The velocity of the MIDI event.
`control`	Which control value to modulate.
`program`	Which instrument is represented.
`numerator`	The numerator of the time signature.
`denominator`	The denominator of the time signature.
`key`	The key of the key signature.

Furthermore, the entire source code, licensed under the MIT license, is available at the following repository:

`www.github.com/FelixSchoen/sCoda/.`

5.1.1 The Message Object

S-Coda is heavily inspired by conventional MIDI files, which use messages to store information about the contained musical data. With S-Coda, we use `Message` objects for a similar purpose, these are the most atomic elements in the library.

`Messages` contain multiple fields of data, specifying the type and contents of the message that is represented. Table 5.1 lists all of the fields messages can contain. Note that not all of the fields have to necessarily be assigned for any single message, e.g., only messages of the type `time_signature` contain data for the fields `numerator` and `denominator`. The different types of messages are listed in table Table 5.2.

5.1.2 The Sequence Objects

These messages are contained in structures called *sequences*. S-Coda uses two types of sequence objects, these being `RelativeSequence` and `AbsoluteSequence`. The main difference between these two objects is the representation of messages they use. For sequences using the relative representation, messages do not include information about when they occur. We use only `wait` messages to inject temporal information. For sequences using the absolute representation, each message contains the exact amount of time in ticks it occurs at.

Table 5.2: Different types of S-Coda messages.

Message Type	Description
`internal`	Represents a message that is only internally used, e.g., for the conversion between relative and absolute sequences.
`key_signature`	A message that defines a new key signature.
`time_signature`	A message that defines a new time signature.
`control_change`	A message that changes a MIDI property, e.g., the tempo.
`program_change`	A message that gives information about which instrument is represented.
`note_off`	A message that represents the end of a note.
`note_on`	A message that represents the start of a note.
`wait`	A message that represents a time frame between messages.

The usage of these two different representations allows for transformative algorithms of higher quality, since some of them heavily profit from the different representations, be it due to improved efficiency or simplicity.

Due to the fact that the interfaces provided by the two types of sequence representations are mutually exclusive, S-Coda provides an abstraction in the form of the `Sequence` object. This object contains a reference to both a `RelativeSequence` and an `AbsoluteSequence`. The `Sequence` object provides methods of the same name as both the relative and absolute representations. When such a method is called, the call is delegated to the respective sequence object on which the call will then resolve. This way users do not have to concern themselves with the implementational details of S-Coda but can rather use the abstraction layer provided by the `Sequence` object.

S-Coda keeps track of when methods that modify the stored sequence representations are called. When such a call is made, the sequence not affected by the call is marked as *stale*. In order to retrieve any of the different representations, S-Coda uses a special interface that first checks for staleness of the requested sequence. If the sequence is found to be stale, the representation that is up-to-date is used to overwrite the stale one, after a conversion to the correct representation. This way the representations are kept in sync while the performance impact is minimised, due to the fact that representations are only updated when they are requested.

5.1.3 The Bar, Track, and Composition Objects

In contrast to the sequence objects, the `Bar`, `Track`, and `Composition` objects represent much more structured musical data. For instance, a `Bar` object represents a single musical bar with a fixed key- and time signature.

The `Bar` objects contain a reference to a `Sequence` object. Upon creation of a `Bar`, it is ensured that the stored sequence conforms to a set of conditions, e.g., the duration of

the sequence may not exceed the capacity of the bar, or that there is a maximum of one key- and time signature respectively. Sequences that are shorter than the capacity of the bar are padded with `wait` messages to ensure that a bar is always filled to capacity.

Although a `Bar` object provides a reference to its `Sequence` object, and thus the collection of messages could be modified in such a way that the `Bar` no longer represents a valid musical bar, this would violate the software contract. The `Bar` object solely represents a single, unmodifiable musical bar. If the user desires to modify the underlying sequence, a list of bars can be converted to a single sequence, which can then subsequently be modified and converted back to a bar representation again.

The `Composition` object represents a complete musical composition containing multiple tracks. These tracks are represented using the `Track` object, which stores a list of `Bar` objects and a name. S-Coda provides an interface to easily convert a standard MIDI file to a `Composition` object using the `from_file()` method. This method takes as input a list of sublists of indices. Each index refers to a track of the original MIDI file, which is converted to a `Sequence` object. In the next step, sequences created from indices of the same sublist are consolidated. Finally, the created sequences are split up into bars, which are stored using `Track` objects.

5.1.4 The MIDI Interface

Although S-Coda does not directly parse MIDI files and rather uses the Mido [15] library to do so, it provides a layer of abstraction around it, eliminating the need to directly use Mido for MIDI file handling. This is done in order to allow for a potential custom MIDI parser in the near future. At the moment, Mido is a well-supported and documented library that is used for a large number of Python projects dealing with MIDI files.

The MIDI wrapper defines three wrapper classes: the `MidiFile` object provides an interface to open a MIDI file, the `MidiTrack` object represents a track in a real MIDI file, and the `MidiMessage` object represents a message.

When creating a `MidiFile` object from a file, currently Mido is used to read the MIDI file. A `MidiTrack` object is created for each track that is contained in the file. In the next step, each of the raw messages provided by Mido are parsed and stored as `MidiMessage` objects. The `MidiMessage` object contains similar fields to the `Message` one explained in Section 5.1.1. Although this procedure may seem redundant, as a direct conversion from Mido objects to the S-Coda internal ones would be possible, it allows for better scalability, as a future custom MIDI parser could simply replace the Mido parser used by the wrapper elements.

In order to convert the wrapper objects to the S-Coda internal ones, `MidiFile` provides the `to_sequences()` method. Here, empty sequences are created for each of the `MidiTrack` objects contained in the `MidiFile`. Each of the `MidiMessage` objects is then manually converted to a `Message` object using an absolute time representation by setting the corresponding fields and keeping track of the elapsed time. These messages are then inserted into the absolute sequence representation of the created sequences.

5.2 Sequence Processing

S-Coda provides a plethora of interfaces to modify and evaluate sequences. In this section, we discuss a large number of these methods and present some of the algorithms we implemented to do so. Although the order of discussion was chosen rather arbitrarily, we begin by splitting it into two parts: interfaces that utilise the relative representation and interfaces that utilise the absolute representation. We then discuss the difficulty assessment functionality. Lastly, we present functionality that cannot be assigned to either of the previous categories.

5.2.1 Relative Representation Interfaces

Message Adjustment

In this step, the sequence is adjusted in two ways: Firstly, redundant time signature messages, e.g., a message that uses the same numerator and denominator as the current active time signature, are removed. Secondly, we consolidate subsequent wait messages before we split them into parts of maximum length PPQN again. This ensures that wait messages always use the most amount of wait time possible, up to a given maximum amount. We settled on the PPQN value as quarter notes are very common in music notation, and argue that a maximum wait value of 24 constitutes a good compromise between sparsity of wait values and oversaturation.

Concatenation of Sequences

Provided with a list of sequences, in this step we concatenate them according to the order given in the list. In order to do so with a relative sequence representation, all messages of the sequences to concatenate simply have to be inserted into a new sequence, in order of original appearance.

Evaluate Next Valid Messages

Provided with a sequence in a relative representation, S-Coda allows for the evaluation of all next valid messages, i.e., which types of messages can be appended to the sequence without violating basic musical constraints. For example, if in a sequence notes C4 and G4 are opened but not closed again, the messages corresponding to opening these exact notes would not be valid next messages. This also works the other way around: Messages that correspond to closing a note that is currently not open are not valid messages.

In this step, we iterate over the messages contained in the sequence, keeping track of the open notes, the current point in time both over the entire span of the sequence and per bar of the sequence, and the current time signature. Based on this information, we provide the user with a list of viable next messages. Time- and key signature messages are only valid at the border of bars, i.e., they cannot be inserted in the middle of a bar. Furthermore, at least one tick of time has to be waited between opening and closing a note.

The list of valid next messages can be used to generate a mask that can be applied to the output of, e.g., a neural network. This way only valid messages are predicted.

Empty Check

In this simple step, we simply check whether or not the sequence is empty. This is done by iterating over the messages contained. If a `note_on` message is encountered, a negative result is returned. If no such message is found, a positive result is returned.

Padding of Sequences

Provided with a fixed sequence length given in ticks, the current sequence is padded up to the provided time if it currently does not match the length. In order to do so, the sum of all `wait` messages of the sequence is calculated. In a next step, a new `wait` message is inserted. The length of this message is given by the difference of the requested padding time and the sum of the wait times.

Splitting of Sequences

S-Coda provides functionality for splitting sequences into parts of fixed capacity, given in ticks. This can be used for, e.g., splitting a sequence into bars if one knows the locations of the bounds of the bars and their capacities.

Algorithm 5.1 shows how this splitting is handled. We start by initialising a list used for storing the split sequences, a map of open messages, a pointer to a new relative sequence, and a copy of the messages of the current sequence.

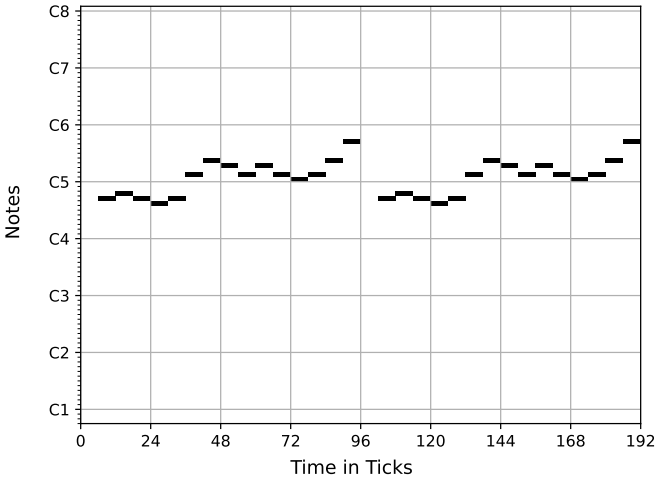
For every capacity defined using the parameter of the method, we try to fit as many messages as possible into an intermediate sequence object. When opening notes, we keep track of which notes have been opened. Likewise, when closing notes we stop keeping track of these notes. Messages that do not affect the temporal structure of the sequence can simply be added to the current intermediate result if we still have some capacity left. Special care goes into handling the `wait` messages.

For each `wait` message, we check if it can still fit into the current intermediate sequence in its entirety. If this is the case, we can simply add it. If this is not the case, we try to fit as much of it as possible, as is defined by the remaining capacity. After this, we stop all current notes and add a command to restart them to the message queue, which will be used to open the notes again for the next part of the split sequence. Furthermore, we also add the rest of the `wait` message to the message queue, and store the current sequence in the output array. In the next step, we update our working memory with the message queue and change the pointer of the current sequence, having it point to the next sequence placeholder.

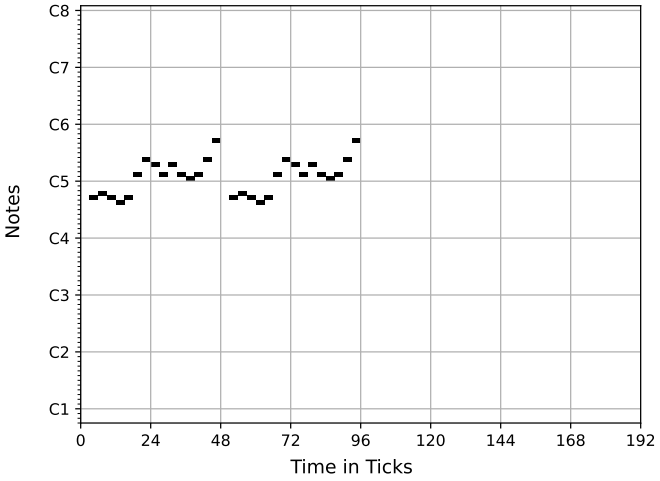
After having iterated over all capacities, we simply add any remaining message still contained in the working memory to the current sequence, which in turn will then be added to the final output array. This way, splitting a sequence using n capacities can

Algorithm 5.1: Splitting of Sequences

```
1  Function split(sequence, capacities):
2      init split_sequences, open_messages
3      current_sequence ← RelativeSequence()
4      working_memory ← copy(sequence.messages)
5      for capacity in capacities do
6          init next_sequence, next_sequence_queue
7          remaining_capacity ← capacity
8          while remaining_capacity ≥ 0 do
9              if end of sequence to split reached then
10                 add current_sequence to split_sequences
11                 current_sequence ← next_sequence
12                 break
13             msg ← next message
14             if msg is note_on then
15                 if remaining_capacity > 0 then
16                     add msg to current_sequence and update open_messages
17                 else
18                     add msg to next_sequence_queue
19             else if msg is note_off then
20                 add msg to current_sequence and update open_messages
21             else if msg is wait then
22                 if msg fits in remaining_capacity then
23                     add msg to current_sequence and update remaining_capacity
24                 else
25                     carry_time ← msg.time - remaining_capacity
26                     add msg with remaining capacity to current_sequence
27                     close open messages, add these messages to
                        next_sequence_queue
28                     add wait msg with carry_time to next_sequence_queue
29                     add current_sequence to split_sequences
30                     insert next_sequence_queue into working_memory
31                     current_sequence ← next_sequence
32                     break
33             else
34                 if remaining_capacity > 0 then
35                     add msg to current_sequence
36                 else
37                     add msg to next_sequence_queue
38     add messages in working_memory to current_sequence
39     add current_sequence to split_sequences
40     return split_sequences
```



(a) The original sequence.



(b) The sequence scaled by a factor of 0.5.

Figure 5.1: Visualisation of S-Coda's scaling capabilities, demonstrated on two bars of Chopin's *Fantaisie-Impromptu*.

result in up to $n + 1$ distinct parts; each of the parts 1 through n will be of the specified length, while the last part contains the rest of the messages of the sequence.

Scaling of Sequences

S-Coda is able to stretch sequences temporally. On a basic level, this is done by scaling the values of the `wait` messages. For any scaling factor $sf \geq 1$ we can simply multiply

the wait values of these messages, resulting in a valid stretched sequence. Scaling the sequence by a factor $sf < 1$ is more complicated, due to the need to preserve time signatures.

Initially, we experimented with simply adjusting the numerator by the scaling factor. One drawback of this approach is the fact that the set of time signatures in a database of MIDI compositions gets diluted—it can create time signatures which were previously not contained, or some that are very uncommon.

We decided to implement an approach that tries to retain as many of the original signatures as possible. Starting at the first bar of the sequence, we retrieve all consecutive bars that would end up in a new bar due to the scaling. We then evaluate if all these $\frac{1}{sf}$ bars are of the same time signature. If this is the case, scaling for these bars can commence analogously to the case $sf \geq 1$. If this is not the case, we apply our initial approach, scaling the numerator of the time signatures according to the scaling factor. If the scaling of the numerator would result in a non-integer value, we instead scale the denominator by the reciprocal of the scaling factor.

Figure 5.1 visualises this process. In Figure 5.1a, the original, unscaled sequence is shown, whilst Figure 5.1b shows the sequence scaled by a factor of 0.5. Here, each note lasts for exactly half the amount of time it originally did, reducing the duration of the sequence by half.

Transposing Sequences

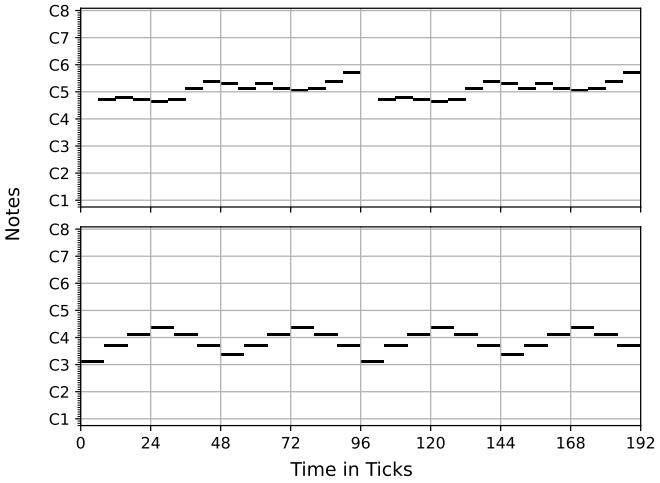
Transposing sequences consists of shifting the contained notes by a fixed value. Using our representation, this is a relatively simple process. In the base case we can simply change the stored note value by incrementing or decrementing it.

Special attention needs to be paid to the case where a note is shifted above or below the bounds of MIDI note values. In this case, we shift the notes by as many octaves as needed to return the note values into the bounds. If such a shifting was necessary, we furthermore quantise the note lengths as discussed in Section 29 and reset any stored difficulty values depending on the pattern of the sequence. Quantising the note lengths is important as otherwise two notes could potentially occupy the same slot, resulting in an invalid MIDI representation.

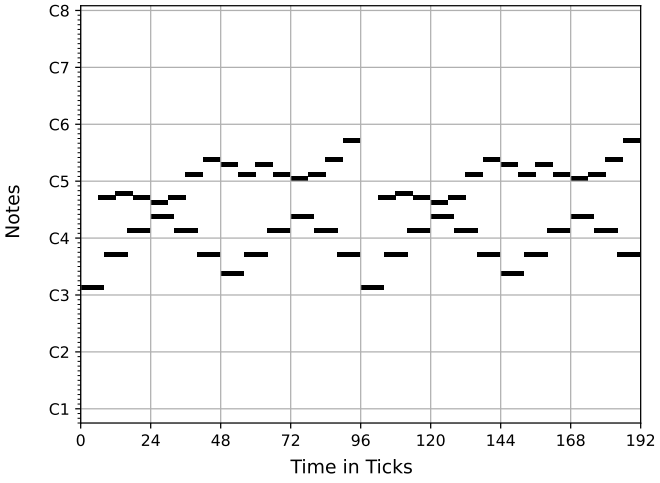
5.2.2 Absolute Representation

Merging of Sequences

Merging two or more sequences results in a new sequence that contains the messages of all the input sequences. In contrast to the concatenation operation, the messages retain their initial places in time. Thus, the overall length of the resulting sequence equals the distance between the earliest message and the latest message of any of the input sequences.



(a) The unmerged sequences.



(b) The result of the merging operation.

Figure 5.2: Visualisation of S-Coda's merging capabilities, demonstrated on two bars of two tracks of Chopin's *Fantaisie-Impromptu*.

Figure 5.2 visualises this process. Here, two bars of the lead and accompanying track of Chopin's *Fantaisie-Impromptu* are shown in Figure 5.2a. Figure 5.2b shows the result of the merge operation. The presented sequence contains notes of both of the initial sequences.

Quantisation of Sequences

Quantisation refers to the act of fitting the beginnings and ends of notes to a fixed-size grid. When human pianists perform a musical piece, they will not be able to time all notes perfectly, resulting in slight delays and shifts of the note timings. Quantising a captured performance can remedy this by nudging the notes to the nearest position on the grid.

S-Coda supports the quantisation of musical pieces. Figure 5.3 visualises this process. In Figure 5.3a an unedited sequence is shown. Here, the beginnings and ends of notes are clearly not fitted to the quarter-note-sized grid. Figure 5.3b shows the sequence after the process of quantisation. The boundaries of the notes are clearly positioned at the grid lines.

Quantisation is an important preprocessing step. If omitted, a machine-learning algorithm based on an unquantised dataset might not be able to accurately learn the correct rhythm, as the slight variations in timing might result in a change of the note values.

Algorithm 5.2 shows how S-Coda quantises sequences. In contrast to PAUL [87], S-Coda is able to accept any combination of step sizes determining the grid the sequence is to be fitted to. For instance, the quantisation process shown in Figure 5.3 was done using quarter note step sizes. If no step sizes are given, S-Coda defaults to quantising to thirty-seconds and thirty-second triplets.

S-Coda calculates the valid position of each message according to the given step sizes. These step sizes determine the grid the sequence is to be fitted to. For `note_on` messages, the valid position with the least distance to its original position is chosen as its new position. Furthermore, if the note has been opened but not been closed before, a `note_off` message is inserted beforehand. In a third step, we check if the new position of the note would induce an overlap with another note. This can happen if, e.g., notes are very close together. In this case we remove the note to be quantised.

For `note_off` messages, we only consider messages that close notes that have been previously opened, but not yet closed. We proceed similarly as with `note_on` messages, in that we calculate the valid message timings. For this it is important to only consider timings that occur after the timing of the corresponding `note_on` message. Finally, we retrieve the valid timing closest to the original one and update the message.

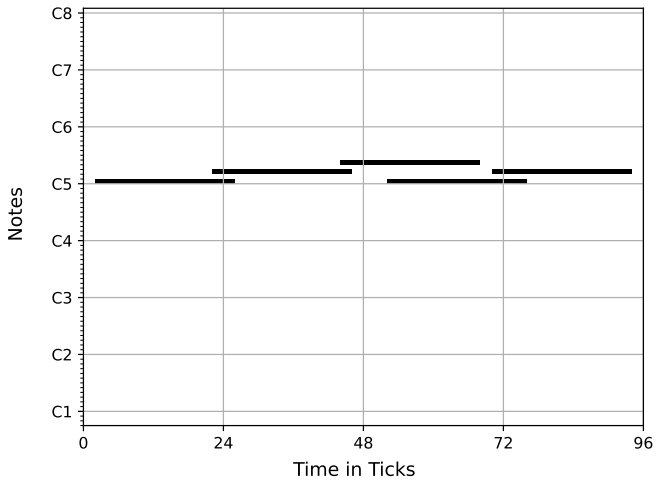
For any other type of message we simply retrieve the position with the minimum distance to the original one, as these messages cannot cause conflicts like `note_on` and `note_off` do. In a final step, we remove notes with invalid, i.e., negative or neutral, lengths.

Quantising Note Lengths

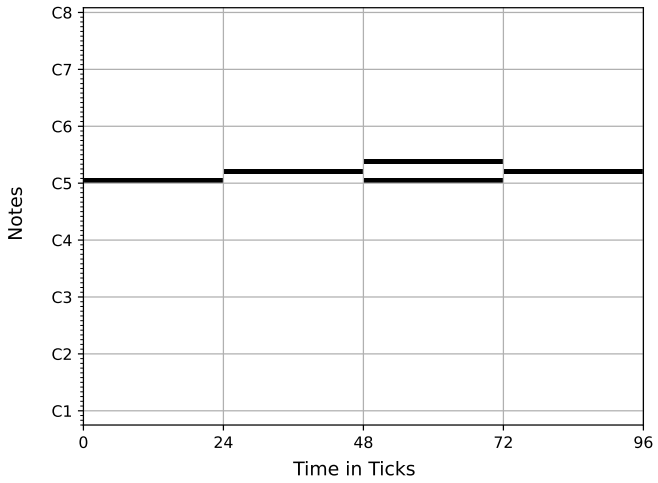
Although the quantisation process ensures that the beginning and end of notes are fitted perfectly to a fixed-size grid, it does not provide any information about the possible resulting note values. Take for instance the sequence shown in Figure 5.4a. Although the

Algorithm 5.2: Quantisation of Sequences

```
1  Function quantise(sequence, step_sizes):
2      init quantised_messages, message_timings
3      for msg in sequence.messages do
4          original_time ← msg.time
5          message_to_append ← copy(msg)
6          positions_left ← [(original_time // step_size) * step_size for step_size
             in step_sizes]
7          positions_right ← [positions_left[i] + step_sizes[i] for i in
             length(step_sizes)]
8          possible_positions ← positions_left + positions_right
9          init valid_positions
10         if msg is note_on then
11             valid_positions += possible_positions
12             message_to_append.time ← valid_positions(min_dist(original_time,
                 valid_positions))
13             close note if still open
14             if no overlap using message_timings detected then
15                 update message_timings
16             else
17                 do not add msg to quantised messages
18         else if msg is note_off then
19             if note is open then
20                 valid_positions ← positions from possible_positions that occur
                     after note open time
21                 message_to_append.time ←
                     valid_positions(min_dist(original_time, valid_positions))
22                 update message_timing
23             else
24                 do not add msg to quantised messages
25         else
26             valid_positions ← possible_positions
27             message_to_append.time ← valid_positions(min_dist(original_time,
                 valid_positions))
28     remove notes that would have invalid lengths
29     sequence.messages ← quantised_messages
```



(a) The original sequence.

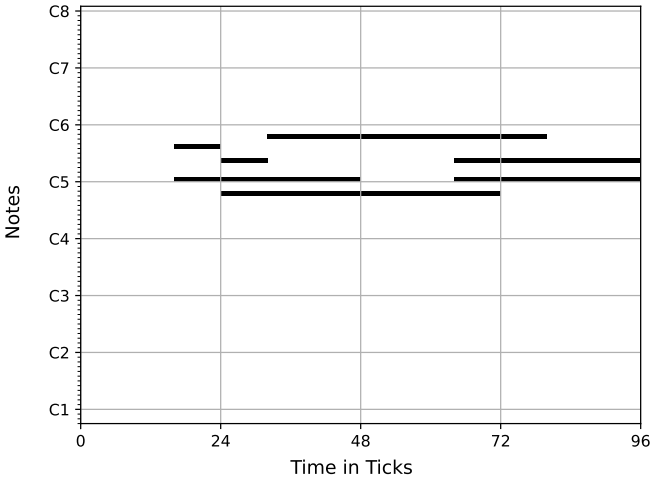


(b) The sequence quantised to quarter notes.

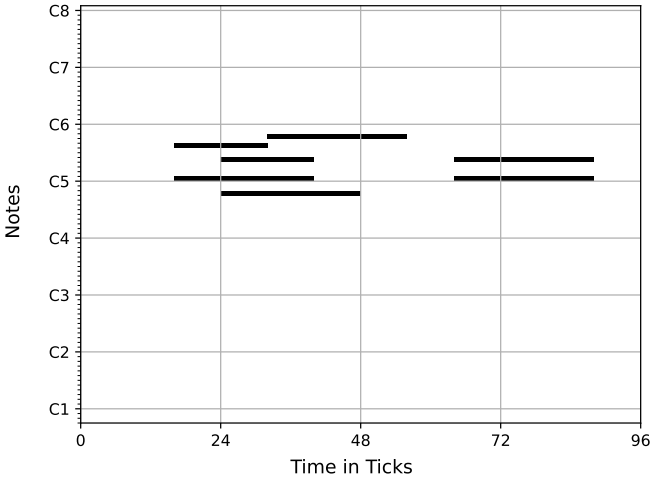
Figure 5.3: Visualisation of S-Coda's quantisation capabilities.

notes are quantised to a quarter note and quarter note triplet grid, the notes are neither quarter notes nor quarter note triplets.

Figure 5.4b shows the same sequence after the note length quantisation procedure. In a similar fashion to sequence quantisation, S-Coda supports variable note length quantisation parameters. In this example, we chose to quantise to quarter notes and quarter note triplets. It is clear to see that all notes displayed in Figure 5.4b are quarter



(a) The original sequence.



(b) Note length quantisation to quarter note and quarter note triplets applied.

Figure 5.4: Visualisation of S-Coda's note-length quantisation capabilities.

notes or quarter note triplets, while being perfectly fitted to the grid.
Algorithm 5.3 shows the procedure for quantising note lengths. First, we construct a list of pairings of `note_on` and `note_off` messages. In each of the elements of the list, the two messages build a matching pair, i.e., no other `note_on` or `note_off` messages of the same note occur between two messages of a pair. For each of these pairs, we then construct a list of valid note durations, based on an input parameter. Furthermore, we

Algorithm 5.3: Quantisation of Note Lengths

```
1  Function quantise_note_lengths(sequence, possible_durations):
2      init quantised_messages
3      notes ← tuples of all matching note_on and note_off messages
4      for pairing in notes do
5          current_duration ← pairing[1].time - pairing[0].time
6          valid_durations ← copy(possible_durations)
7          if current note is not last note then
8              for possible_duration in possible_durations do
9                  if possible_duration would induce clash with next note then
10                     remove possible_duration from valid_durations
11         if length(valid_durations) is 0 then
12             mark current note as to be removed
13         else
14             best_fit ← valid_positions(min_dist(current_duration,
                   valid_durations))
15             correction = best_fit - current_duration
16             pairing[1].time += correction
17     add all unmarked modified notes to quantised_messages
18     add all messages of other types to quantised_messages
19     sequence.messages ← quantised_messages
```

evaluate the current duration of the unquantised note.

Since extending a note could potentially cause a clash with another note that occurs later in the sequence, we first check if the current note is the last of its pitch. If this is not the case, we remove all those durations from the valid durations that would induce a clash with a subsequent note.

If after this procedure no duration is a valid one, we mark the note as to be removed. If there still exist valid durations, we evaluate which of them would induce the smallest change from the current duration. We then correct the duration of the note.

In a final step, we add all notes previously not marked for deletion to an intermediate list. Furthermore, all messages of a different type than `note_on` and `note_off` can also be safely added. We then replace the list of messages of the current sequence with the list of quantised messages.

5.2.3 Difficulty Assessment

S-Coda provides functionality for assessing the difficulty of sequences. This difficulty assessment is used to measure how difficult a given sequence would be to play for a

human pianist. We note that although in theory these assessments work for arbitrary sequences, the standard values were designed with bars of single tracks in mind. S-Coda is designed to primarily handle piano music data.

The output difficulty values range from 0 to 1, a higher value represents a more difficult sequence. S-Coda uses multiple metrics to evaluate difficulty, which will be discussed in this section. These metrics were designed based on multiple factors, such as correspondence with an expert on music theory, personal investigation and experience, and intuition.

Amount of Concurrent Notes and Amount of Overall Notes

When playing from sheet music, the amount of notes to read greatly influences the difficulty of the piece simply due to the fact that more notes have to be read and played in the same time frame. Figure 5.5 visualises this concept. Figure 5.5a shows a sequence of four notes that are trivial to play for any pianist. Although the pattern shown in Figure 5.5b is still very manageable, there is a clear increase in the difficulty of the piece. We note that due to the fact that Figure 5.5b depicts four standard chords, no experienced pianist would have difficulties performing the short sequence. We rather want to emphasise the impact of having to perform a larger amount of notes, even if they all are of the same value.

S-Coda is able to assess the difficulty of a sequence based on both the overall amount of notes occurring in a sequence and the amount of concurrent notes. In order to assess the amount of concurrent notes played, S-Coda determines the transitions between sections of a piece where n notes are played and sections where $n \pm x$ notes are played, where $|n - x| > 0$. For the overall amount of notes, matching pairs of `note_on` and `note_off` messages are counted.

We note that although the overall amount of notes and the amount of concurrent notes are two similar measures, there are some corner cases where they might differ. For example, there is a difference in the difficulty of a sequence consisting of four pairs of two consecutive notes and a sequence consisting of one stack of five notes, followed by three single notes. In both cases, the sequence consists of the same amount of overall notes. These notes can even potentially all be of the same note value. In this case, the second sequence is arguably more difficult to play, since the increase in difficulty induced by having to read five notes at once is not outweighed by only needing to read a single note for the rest of the sequence.

Distances Between Notes

Needing to perform large jumps when performing a piano piece is not a trivial task. It is challenging from both a score-reading and a performance-based point of view. When reading a score, musicians can rely on identifying notes based on the relative distances to other ones. When a score contains large jumps, this is sometimes made more difficult or even no longer possible.

(a) A sequence containing no concurrent notes.

(b) A sequence with concurrent notes that is more difficult to read.

Figure 5.5: The impact of concurrent notes on the difficulty of a sequence.

Furthermore, having to perform large jumps when playing is a challenging task as well. It can be difficult to find the right key to press, as pianists can usually rely on having a frame of reference, i.e., knowing where their fingers are positioned at the moment. When performing large jumps, the pianists often have to shift their gaze from the sheet music to the keyboard in order to find the correct key, disrupting the flow of the performance.

Due to these circumstances, S-Coda tries to incorporate the overall distances between notes into its difficulty assessment. In order to do so, S-Coda collects all notes played at a specific point of time. The program then calculates the differences of note height between the lowest note of the current point of time and the lowest note of the previous point of time. Analogously, the same is done for the highest notes. The maximum of these two measures is then appended to a collection of distances, which is then used to judge the difficulty. S-Coda only considers the top 15% of distances, in order not to have many small jumps dilute the distances.

Key Signatures

As discussed in Section 2.1, key signatures can help declutter the visual appearance of musical scores. The fact remains that musicians have to consider them when performing music, as accidentals are implicitly added for some notes, defined by the signature. Figure 2.7 emphasises this concept. Although the second bar is cleaner due to the absence of the individual accidentals, these still have to be kept in mind when performing the piece. Key signatures that induce a larger number of accidentals, be it sharps or flats, are harder to perform, as the musician has to take the additional modifications to the pitch values into account.

S-Coda tries to reflect this fact in its difficulty assessment by considering the key signatures of bars. Although MIDI files can contain information about the used key signature, not all of them do. In order to remedy this, S-Coda is able to guess the key signature of a specific sequence. If no key signature information is present, S-Coda tries to estimate which signature would be most fitting by counting the number of additional accidentals needed to visualise the score. The signature inducing the least amount of violations is then assumed as the key signature of the piece. If there is a tie, S-Coda prefers signatures with a smaller amount of inherent accidentals.

In order to calculate the difficulty of the piece based on the key signature, the amount

of accidentals induced by it are considered. We do not differentiate between flats and sharps, as we argue that they impact the difficulty of a piece in the same way. The fact that the pitch value of some notes is modified is all that is of essence. S-Coda considers key signatures that induce a larger number of accidentals as more difficult, with C♯ and C♭ being the most difficult as they induce seven accidentals.

Note Classes

If a piece contains, e.g., a large number of notes, many of them played consecutively, it might still be relatively easy to play due to the fact that the same notes appear over and over. It is quite manageable for the performer to see at a glance that the number of overall note classes is quite low, and that they can focus more on, e.g., the rhythm of the piece.

S-Coda considers the amount of overall note classes when assessing the difficulty of a piece. It does not consider notes that are labeled using the same letter but spread an octave apart as the same class. For example, C3 and C4 are considered as different note classes. Even though these notes may sound very similar, they are notated very differently in score notation, having a completely different height. Due to this fact they require additional effort to read.

Note Values

Arguably, one of the most important factors for the difficulty of a piece are the note values of the contained notes. Even difficult patterns can be very manageable if played slowly enough. Since S-Coda makes no assumptions about the tempo of a piece, and thus each quarter note lasts for the same amount of time, it treats pieces with lower average note values as more difficult.

Lower average note values can result in a performer having to both read and play more notes in the same amount of time, a task that can be very challenging. It is quite clear that this metric directly correlates with the difficulty of the piece.

For the assessment, pairings of matching notes are collected in a similar fashion as with, e.g., the note length quantisation algorithm shown in Algorithm 5.3. The lengths of all played notes in ticks is then collected. S-Coda calculates the geometric mean of all the lengths, based on which it calculates the final note value difficulty from.

Rhythm

Special rhythms, especially ones involving subdivisions like triplets, are often seen as quite difficult to play. Sometimes these rhythms result in the performer having to treat each track differently, since the notes between the tracks do not line up. This is often a daunting task, especially for inexperienced pianists.

S-Coda tries to reflect this fact by taking special note values into account. For the calculation, we once again collect pairs of matching note messages as to evaluate their note lengths. Based upon this, we sort the note into one of three bins.

The first bin includes standard note values such as quarter notes, half notes, eighth notes, and so on. These note values are calculated based on an internal value defining the upper and lower multiplicand of the PPQN value, resulting in upper and lower bounds for the note values. For example, by default, S-Coda is equipped to deal with double whole notes down to thirty-second notes, these being of 192 and 3 ticks in length, respectively, assuming a PPQN of 24.

The next bin includes note values of certain tuplets. These values are also calculated automatically, based upon the previous default note values and a list of valid tuplets. By default, S-Coda considers triplets, i.e., three notes in the same amount of time, as two notes. For each of the tuplets, a valid note length is constructed from the list of valid standard note lengths. If we take the above example, S-Coda would construct double whole triplets down to thirty-second triplets, being 128 and 2 ticks in length respectively.

Lastly, S-Coda considers dotted notes as well. The values of these dotted notes are constructed from the list of values of the standard note values. By default, S-Coda considers a single dotted iteration, resulting in note values that are 1.5 times as long. S-Coda only deals with integer tick lengths, thus in its default configuration no thirty-second dotted notes are considered since they would have a note length of 4.5 ticks. Instead, sixteenth-dotted notes are the smallest note value considered for the dotted values.

S-Coda tries to match the note value of each of the occurring notes into one of these three bins. Afterwards, the amount of matched entries is weighted and summed up. The resulting value is then used to estimate the difficulty of the rhythm.

Note Patterns

Musical patterns in compositions can severely reduce the difficulty of a piece. This is often due to the fact that pianists need to learn phrases of smaller length since the same sequence is repeated later on, reducing the overall amount of notes needed to be read.

Figure 5.6 highlights this fact. In Figure 5.6a a bar consisting of twelve notes is shown. These twelve notes can be subdivided into four equal parts of length three. When a pianist wants to perform this piece, they only have to learn this part of three notes, which they will be then able to simply repeat for the rest of the measure. In contrast to this, Figure 5.6b shows a bar consisting of the same amount of notes but without any discernible pattern. In fact, the bar shown contains the exact same notes, only arranged in a different order. Although this measure is still relatively easy to play, a pianist can no longer simply repeat a part of smaller length in order to perform the entire bar.

S-Coda takes this fact into account when calculating the difficulty of a piece by trying to evaluate note patterns of that piece. This is done using recursive applications of

(a) A bar with a note pattern.

(b) A bar without a note pattern.

Figure 5.6: Two bars inspired by Beethoven's *Moonlight Sonata* consisting of the same notes in a different arrangement, resulting in different note patterns.

regular expressions. Regular expressions are sequences of characters used to find patterns in a given string. For our purposes, this string consists of textual representations of the notes in a musical sequence. This textual representation uses the relative distances between notes instead of absolute pitch values in order for the pattern recognition to take small pitch shifts of patterns into account. The following excerpt shows the textual representation of the sequence shown in Figure 5.6a. Note that due to the fact that we use the relative changes in pitches, the first note is only implicitly encoded:

$$+5\ \ +3\ \ -8\ \ +5\ \ +3\ \ -8\ \ +5\ \ +3\ \ -8\ \ +5\ \ +3$$

Algorithm 5.4 shows the procedure we use to determine the difficulty of a sequence regarding its patterns. We use two different pattern matching algorithms, `match_pattern()` and `greedy_match_pattern()`, where `greedy_match_pattern()` does the same as `match_pattern()` with the exception that it only considers a single local match for each pattern length, instead of trying to find all matches of all possible lengths per recursion iteration.

We note that although the non-greedy approach is able to measure the difficulty regarding occurring patterns with higher precision, it incurs a heavy penalty regarding the performance of the evaluation. Longer sequences, especially ones with low average note values and time signatures that allow for a large amount of notes per bar, can take up to 30 seconds per evaluation using this process. This is why we opted to introduce a time limit to this non-greedy approach after which the greedy pattern matching will be applied. By default, S-Coda spends up to five seconds per bar on the non-greedy approach before switching to a greedy one.

After having found a number of pattern matches using either of the mentioned procedures, we calculate the difficulty value of the sequence based on these results. In order to do so, we compare the length of the pattern with the amount of coverage provided by it. For example, a short pattern that covers most of the sequence makes a piece far easier to play than a long pattern that maybe only covers a small part. Based on this relation, we return the difficulty value.

Algorithm 5.5 shows how the pattern matching is conducted in more detail. Since this method is called recursively, we need to define a termination criterion. In our case, this is twofold: we throw a `TimeoutError` if no more time budget is left. Furthermore, we

Algorithm 5.4: Assess Difficulty of Sequence Based on Occurring Patterns

```
1  Function difficulty_pattern(sequence):
2      string_representation ← build string representation from sequence
3      try:
4          results ← match_pattern(string_representation, max_duration)
5      catch timeout when matching pattern:
6          results ← greedy_match_pattern(string_representation)
7      init results_with_coverage
8      for result in results do
9          uncovered ← copy(string_representation)
10         length_pattern ← 0
11         for match in result do
12             uncovered ← remove all covered by match from uncovered
13             length_pattern ← length_pattern + length of match
14         coverage ← (length(string_representation) - length(uncovered)) /
            length(string_representation)
15         add (coverage, length_pattern, result) to results_with_coverage
16     best_fit ← maximum of results_with_coverage measured by coverage /
        length_pattern
17     return difficulty based on best_fit
```

only recurse if patterns can be found on the current working string, which is reduced every iteration.

In a first step, we search for all possible patterns of a specific length. We use the following regular expression to search for patterns in the string representation, where `placeholder` is to be replaced with the length of the desired pattern:

```
(?=(?P<p>(?:[+-]\d+){placeholder})(?:[+-]\d+)*(?P<e>(?P=p)
(?:[+-]\d+)*?)+)
```

This regular expression matches a string that contains a sequence of elements of length `placeholder`, followed by an indeterminate amount of other elements, followed by one or more occurrences of this sequence of elements again. In this case, an element consists of either the plus or minus sign followed by a positive amount of digits.

As long as we find such patterns, we increase the size of the patterns to find and add the found patterns to an intermediary list. Note that this list only contains unique patterns. In a next step, we iterate over all found matches. For each match we restart the procedure on a copy of the current working string where the matched part is removed. We then add the result of the current iteration combined with the result from the recursive procedure to an output list, which is returned at the very end of the procedure.

Algorithm 5.5: Match Pattern in String

```
1  Function match_pattern(string_representation, duration):
2      if duration exceeded then
3          throw TimeoutError
4      init local_matches, results
5      current_pattern_length ← minimum pattern length
6      while matches can be found do
7          matches ← match current_representation with current_pattern_length
8          for match in matches do
9              if match not in local_matches and match does not contain pattern
                itself then
10                 add match to local_matches
11         current_pattern_length ← current_pattern_length + 1
12     init results
13     for local_match in local_matches do
14         if local_match not in results then
15             add local_match to results
16         modified_representation ← remove all covered by local_match from
            string_representation
17         recursive_results ← match_pattern(modified_representation,
            duration)
18         for recursive_match in recursive_results do
19             init result
20             add local_match and recursive_results to result
21             add result to results
22         return results
```

5.2.4 Further Functionality

Splitting into Bars

We provide functionality to split sequences into bars fully automatically, based on `time_signature` and `key_signature` messages. S-Coda is able to process multiple sequences at once. In this case, only one track is expected to carry time and key signature messages. The procedure splits multiple tracks into bars of equal length.

Algorithm 5.6 details how this splitting procedure works. We determine when a time- or key signature change occurs and store the timings in a list. Furthermore, we store the current elapsed time, measured at the bar boundaries. While the tracks are not synchronised, i.e., at least one of the tracks contains more bars, we retrieve the most recent time- and key signature messages in order to determine the currently active signatures. Using this information, we can calculate the capacity of the current bar. We

Algorithm 5.6: Split Sequence into Bars

```
1  Function split_into_bars(sequences):
2      init current_key, current_point_in_time, track_bars
3      current_numerator, current_denominator ← 4
4      time_signature_timings ← get_message_timings(sequences,
         MessageType.time_signature)
5      key_signature_timings ← get_message_timings(sequences,
         MessageType.key_signature)
6      while tracks are not synchronised do
7          time_signature ← retrieve next time signature from
             time_signature_timings
8          key_signature ← retrieve next key signature from key_signature_timings
9          length_bar ← PPQN * (current_numerator / (current_denominator / 4))
10         current_point_in_time ← current_point_in_time + length_bar
11         for sequence in sequences do
12             sequence_split ← sequence.split([length_bar])
13             if sequence_split contains more than 1 sequence then
14                 sequences[i] ← sequence_split[1]
15             else
16                 sequences[i] ← empty placeholder sequence
17             sequence_to_add ← sequence_split[0]
18             quantise sequence_to_add note lengths
19             create bar from sequence_to_add with current time signature and key
                 signature
20             add bar to track_bars[i]
21     return track_bars
```

can then apply the splitting procedure to obtain the bar-sized sequence chunks, which can then be added to the output list.

Piano-Roll Visualisation

The piano-roll representations used throughout this thesis were created using S-Coda. We implemented the functionality to easily visualise a list of sequences. The visualisation makes use of matplotlib[5], where we create subplots for each of the sequences to include. This allows for a fine degree of customisation and expandability.

The piano-roll method accepts multiple input parameters, all but one of them are populated with default values if not specified. Except for the sequences to plot, no

[5] https://matplotlib.org/.

additional parameter must be provided. S-Coda allows for an easy specification of the labels of the axis and the title of the plot.

Furthermore, the function accepts parameters specifying the scale of both the horizontal and vertical dimension. This way, users can specify which range of notes and which time frame to plot. If no range is specified, S-Coda will print the entire MIDI range and show the entire sequence from start to finish.

Lastly, we accept a parameter specifying if the velocity of the notes should be shown. If set, notes with lower velocity will be plotted with more transparency. We did not make use of this feature for this thesis as PAUL-2 does not produce velocity information.

5.3 Usage

Listing 5.1 provides a code snippet illustrating how to use S-Coda. Although showing the entire capabilities of S-Coda would go beyond the scope of this thesis, the provided snippet gives some basic insight regarding its functionality.

We start by loading a list of sequences from the local drive. In order to do so, we need to provide a list of lists of tracks of the MIDI file which correspond to the final output sequences. In this example, track 2 and track 3 of the MIDI file are converted to separate sequences, without any merging operation. Track 1 of the MIDI file only contains meta information, e.g., time signature messages. By default, S-Coda merges all meta messages contained in the tracks marked as meta tracks with the first generated sequence. The second argument of the method provides the list of the tracks marked as meta tracks, in this example only the first track is marked as such.

In the next step, the loaded sequences are quantised. Both quantisation of the messages in general and quantisation of note lengths is done. Note that if no parameters are provided, S-Coda quantises to thirty-seconds. Providing information about how to quantise can result in, e.g., quantisation to quarter notes.

In the next step, we split the loaded sequences into bars. S-Coda is able to do this fully automatically. No specification of bar capacities or time signatures is necessary since this information will be extracted from the sequences themselves. Note that due to the fact that only the first sequence contains information about time signatures, providing only the second sequence for the splitting procedure would not result in a valid split. In this case, both sequences would have to be passed to the method although the result of the split for one of the sequences can be safely discarded.

We then extract three bars and their sequences from the two sequences that we want to apply transformations to. Note that we create copies of the sequences in order not to modify the sequences the `Bar` objects are referencing since this can result in these objects no longer representing valid bars.

For the first sequence, we apply a scaling operation using a factor of 0.5. This results in the sequence having its total duration reduced by half.

Listing 5.1: Example usage of S-Coda

```
# Load sequences from file
sequences = Sequence.from_midi_file("chopin_fantaisie.mid",
    [[1], [2]], [0])

# Quantise sequences
for sequence in sequences:
    sequence.quantise()
    sequence.quantise_note_lengths()

# Separate sequences
sequence_lead = sequences[0]
sequence_acmp = sequences[1]

# Split sequence into bars
bars = Sequence.split_into_bars([sequence_lead,
    sequence_acmp])
lead_bars = bars[0]
acmp_bars = bars[1]

# Obtain bars to process
seq_to_scale = copy(lead_bars[4].sequence)
seq_to_merge = copy(acmp_bars[4].sequence)
seq_to_diff = copy(acmp_bars[2].sequence)

# Scale sequence
seq_to_scale.scale(0.5)

# Transpose sequences
seq_to_scale.transpose(12)
seq_to_merge.transpose(-12)

# Merge bars
merged_sequence = Sequence()
merged_sequence.merge([seq_to_scale, seq_to_merge])

# Create pianoroll and calculate difficulty
Sequence.pianorolls([merged_sequence], x_label="Time in
    Ticks", y_label="Notes", show_velocity=False).show()
print(seq_to_diff.difficulty())
```

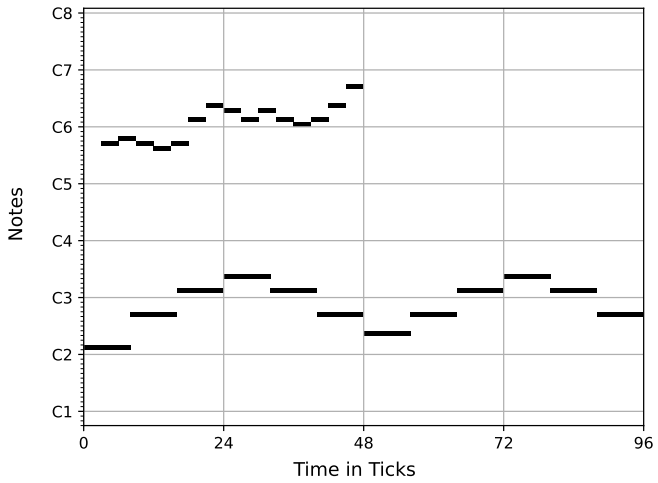


Figure 5.7: Piano roll created by the demonstration of S-Coda, shown in Listing 5.1.

We then transpose both the first and the second sequence by an octave. Note that we raise the pitches of the first sequence while we lower them for the second one. This results in the distances between the notes increasing by two octaves, assuming that all pitch values of the second sequence were lower than the pitches of the first sequence beforehand.

In the third step, we merge the recently transposed sequences into a new sequence. Note that for this, we create a new empty sequence beforehand since this operation is done in-place, i.e., the sequence we apply the operation to will be modified. Merging the sequences results in the new sequence containing the notes of both input sequences without modifying their absolute timings.

We then create a piano-roll visualisation from the modified sequences. For this, we only provide a single sequence to visualise, although we could pass more than one argument here. We set the label of both axes and disable the velocity information. The `pianoroll()` method returns a plot object which can be used to either show the created piano roll or save it to a file.

Saving the created piano roll results in the visualisation shown in Figure 5.7. Here, one can clearly see the increased distances between the two merged partitions and the result of the scaling operation. Initially, the notes ranging from C5 to C7 were spread over the entire range of the bar, after the scaling they only occupy the first half.

In the last step we simply show how to have S-Coda evaluate the difficulty of a sequence. This is done by calling the `difficulty()` method, which in turn calculates the difficulty of the piece based on the metrics discussed in Section 5.2.3.

Finally, we note that the functionality shown in Listing 5.1 by no way covers the entire range of S-Coda's capabilities. When designing and creating the library, we made sure to document all the available functions and methods in order to provide users with information on the interfaces.

CHAPTER 6

Conclusion

In this thesis, we introduced PAUL-2, a transformer-based [97] algorithmic composer of two-track piano pieces utilising relative attention [89, 45], capable of generating both lead and accompanying tracks of piano music. A particular feature of PAUL-2 is the fact that it is able to generate pieces that conform to a given *difficulty specification*, which aims to estimate how difficult the piece would be to play for a human pianist. The future designated usage of PAUL-2 is in an educational context, where it could be used to generate new musical prompts for piano students. These students would then have to rely on their sight-reading capabilities rather than their memory of the piece in order to perform it.

PAUL-2 consists of two distinct networks with their own sets of weights, P2L and P2A. The former is able to generate lead piano tracks from an input difficulty parameter, whilst the latter deals with creating accompaniment for a lead track based on an input musical sequence and a difficulty parameter.

We implemented an approach in the spirit of the approach by Libovický, Helcl, and Marecek [56] in order to create P2A, which is a multi-sequence-to-sequence encoder-decoder network. We furthermore introduced the single-out mask used to attend to a single input only, and an output mask that is applied to the output predictions of the neural networks. This mask ensures that only valid messages are generated, e.g., no notes may be closed that have not been opened before.

PAUL-2 excels at generating pieces of the desired complexity—a strong correlation between specified and generated complexity of the output pieces can be observed. Although the output represents valid musical pieces, the visual representation as musical score can get cluttered due to the fact that often several voices and unusual rhythms are used.

In a medium-scale study with 60 participants, we evaluated the musical quality of the output of PAUL-2. The subjects were asked to differentiate between genuine and computer-generated samples in a "Turing Test"-like fashion. Although the desired percentage of

about 50% was not achieved, the participants often stated that they found the task to be quite challenging. Furthermore, when surveyed on the quality of the generated pieces, the responses of the participants were of favourable nature.

Overall, the quality of the output of PAUL-2 is vastly superior to that of our preceding system, PAUL [86, 87]. We strongly believe that future iterations will be able to improve upon our current work and bring to bear algorithmic composers capable of being used in a practical environment.

We argue that due to the lack of datasets, training a network such as PAUL-2 is quite difficult at the moment. These sets lack both the quality and quantity to achieve results comparable with projects such as, e.g., MuseNet [74].

Furthermore, we introduced S-Coda, a Python framework for handling MIDI files. S-Coda provides a multitude of MIDI transformations and operations which can greatly reduce the time needed to preprocess musical data for machine learning or other applications. Among other things, S-Coda is able to quantise musical sequences, i.e., fit the beginnings and ends of notes perfectly to a fixed-size grid, automatically split sequences into musical bars based on time signature messages, and create visual piano-roll representations from them.

In future work, we intend to improve upon the relative attention mechanism introduced by Shaw, Uszkoreit, and Vaswani [89] and enhanced by Huang et al. [45]. At the moment, this approach is not suited to deal with the multi-track nature of PAUL-2. Furthermore, we intend to experiment with alternative approaches for the symbolic representation of music. We argue that choosing an approach that is more akin to sheet music could potentially enhance the output of a system whose task it is to generate music that is to be visually represented. Lastly we would like to experiment with additional types of attention. We argue that encoding information about the temporal distances of the messages measured in ticks rather than only the amount of messages they are separated by could improve the long-term structure and rhythm of the pieces even more.

Bibliography

[1] Edward Aldwell, Carl Schachter, and Allen Cadwallader. *Harmony & Voice Leading*. Cengage, Boston, MA, USA, fifth edition, 2019.

[2] Torsten Anders and Eduardo R. Miranda. Constraint-based composition in realtime. In *Proceedings of the 34th International Computer Music Conference* (*ICMC 2008*). Michigan Publishing, 2008.

[3] Torsten Anders and Eduardo R. Miranda. Constraint application with higher-order programming for modeling music theories. *Computer Music Journal*, 34(2):25–38, 2010.

[4] Torsten Anders and Eduardo Reck Miranda. Constraint programming systems for modeling music theories and composition. *ACM Computing Surveys*, 43(4):30:1–30:38, 2011.

[5] Christopher Anderson, Arne Eigenfeldt, and Philippe Pasquier. The generative electronic dance music algorithmic system (GEDMAS). In *Proceedings of the 9th Conference on Artificial Intelligence and Interactive Digital Entertainment* (*AIIDE 2013*), volume WS-13-22 of *AAAI Technical Report*. AAAI Press, 2013.

[6] MIDI Manufacturers Association. The Complete MIDI 1.0 Detailed Specification. `https://midi.org/`, 1996.

[7] MIDI Manufacturers Association. MIDI 2.0 Specification Overview. `https://midi.org/`, 2020.

[8] Jimmy L. Ba, Jamie R. Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[9] Chip Bell. Algorithmic music composition using dynamic markov chains and genetic algorithms. *Journal of Computing Sciences in Colleges*, 27(2):99–107, 2011.

[10] Bruce Benward and Marilyn Saker. *Music in Theory and Practice: Volume 1*. McGraw-Hill, New York, NY, USA, eighth edition, 2009.

[11] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Society for Music Information Retrieval Conference* (*ISMIR 2011*), pages 591–596. University of Miami, 2011.

[12] John A. Biles. Genjam: A genetic algorithm for generating jazz solos. In *Proceedings of the 20th International Computer Music Conference* (*ICMC 1994*). Michigan Publishing, 1994.

[13] John A. Biles. Autonomous GenJam: Eliminating the fitness bottleneck by eliminating fitness. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation* (*GECCO 2001*), 2001.

[14] John A. Biles, Peter Anderson, and Laujra Loggi. Neural network fitness functions for a musical IGA. In *Proceedings of the 1996 Symposium on Soft Computing* (*SOCO 1996*). ICSC Academic Press, 1996.

[15] Ole M. Bjørndalen. Mido. `https://mido.readthedocs.io/`, 2022. Accessed: 2022-06-07.

[16] Georg Boenn, Martin Brain, Marina De Vos, and John P. Fitch. Automatic music composition using answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):397–427, 2011.

[17] Tommaso Bolognesi. Automatic composition: Experiments with self-similar music. *Computer Music Journal*, 7(1):25–36, 1983.

[18] Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. *Deep Learning Techniques for Music Generation*. Springer, 2020.

[19] Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[20] Filippo Carnovalini and Antonio Rodà. Computational creativity and music generation systems: An introduction to the state of the art. *Frontiers in Artificial Intelligence*, 3:14, 2020.

[21] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (*EMNLP 2014*), pages 1724–1734. ACL, 2014.

[22] David Cope. Recombinant music: Using the computer to explore musical style. *Computer*, 24(7):22–28, 1991.

[23] David Cope. Computer modeling of musical intelligence in EMI. *Computer Music Journal*, 16(2):69, 1992.

[24] Stephen Davismoon and John Eccles. Combining musical constraints with markov transition probabilities to improve the generation of creative musical structures. In *Proceedings of the 13th Conference on Applications of Evolutionary Computation* (*EvoAPPS 2010*), volume 6025 of *Lecture Notes in Computer Science*, pages 361–370. Springer, 2010.

[25] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. MuseGAN: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the 32nd AAAI Conference on Artificial Intelligence* (*AAAI 2018*), pages 34–41. AAAI Press, 2018.

[26] David Dunning and Justin Kruger. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77(6):1121–1134, 1999.

[27] Douglas Eck and Jürgen Schmidhuber. Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. In *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing* (*NNSP 2002*), pages 747–756. IEEE, 2002.

[28] Douglas Eck and Jürgen Schmidhuber. Learning the long-term structure of the blues. In *Proceedings of the 12th International Conference on Artificial Neural Networks* (*ICANN 2002*), volume 2415 of *Lecture Notes in Computer Science*, pages 284–289. Springer, 2002.

[29] Arne Eigenfeldt and Philippe Pasquier. Realtime generation of harmonic progressions using constrained markov selection. In *Proceedings of the 1st International Conference on Computational Creativity* (*ICCC 2010*), pages 16–25. computationalcreativity.net, 2010.

[30] Richard R. Fay. *Hearing in Vertebrates: A Psychophysics Databook*. Hill-Fay Associates, 1988.

[31] Jose D. Fernández and Francisco J. Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48:513–582, 2013.

[32] Lucas N. Ferreira, Levi H. S. Lelis, and Jim Whitehead. Computer-generated music for tabletop role-playing games. In *Proceedings of the 16th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (*AIIDE 2020*), pages 59–65. AAAI Press, 2020.

[33] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[34] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, Cambridge, England, 2014.

[35] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming* (*ICLP 1988*), pages 1070–1080. MIT Press, 1988.

[36] Kemal Ebcioğlu. An expert system for harmonizing four-part chorales. In *Proceedings of the 12th International Computer Music Conference* (*ICMC 1986*). Michigan Publishing, 1986.

[37] Kemal Ebcioğlu. An expert system for harmonizing chorales in the style of J. S. Bach. *Journal of Logic Programming*, 8(1):145–185, 1990.

[38] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning.* Adaptive computation and machine learning. MIT Press, 2016.

[39] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. In *Proceedings of the 28th Conference on Neural Information Processing Systems* (*NIPS 2014*), pages 2672–2680, 2014.

[40] Masatoshi Hamanaka, Keiji Hirata, and Satoshi Tojo. Fatta: Full automatic time-span tree analyzer. In *Proceedings of the 33rd International Computer Music Conference* (*ICMC 2007*). Michigan Publishing, 2007.

[41] Curtis Hawthorne, Andriy Stasyuk, Adam Roberts, Ian Simon, Cheng-Zhi A. Huang, Sander Dieleman, Erich Elsen, Jesse H. Engel, and Douglas Eck. Enabling factorized piano music modeling and generation with the MAESTRO dataset. In *Proceedings of the 7th International Conference on Learning Representations* (*ICLR 2019*). OpenReview.net, 2019.

[42] Suzana Herculano-Houzel. The human brain in numbers: A linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, 3, 2009.

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[44] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[45] Cheng-Zhi A. Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer: Generating music with long-term structure. In *Proceedings of the 7th International Conference on Learning Representations* (*ICLR 2019*). OpenReview.net, 2019.

[46] David M. Huber. *The MIDI Manual: A Practical Guide to MIDI within Modern Music Production*. Routledge, New York, NY, USA, fourth edition, 2021.

[47] Shulei Ji, Jing Luo, and Xinyu Yang. A comprehensive survey on deep music generation: Multi-level representations, algorithms, evaluations, and future directions. *CoRR*, abs/2011.06801, 2020.

[48] Lejaren A. Hiller Jr. and Leonard M. Isaacson. Musical composition with a high-speed digital computer. *Journal of the Audio Engineering Society*, 6(3):154–160, July 1958.

[49] Alexis Kirke and Eduardo R. Miranda. Emergent construction of melodic pitch and hierarchy through agents communicating emotion without melodic intelligence. In *Proceedings of the 37th International Computer Music Conference* (*ICMC 2011*). Michigan Publishing, 2011.

[50] Alexis Kirke and Eduardo R. Miranda. A multi-agent emotional society whose melodies represent its emergent social hierarchy and are generated by agent communications. *Journal of Artificial Societies and Social Simulation*, 18(2), 2015.

[51] Qiuqiang Kong, Bochen Li, Jitong Chen, and Yuxuan Wang. Giantmidi-piano: A large-scale MIDI dataset for classical piano music. *Transactions of the International Society for Music Information Retrieval*, 5(1):87–98, 2022.

[52] Steven G. Laitz. *The Complete Musician: An Integrated Approach To Tonal Theory, Analysis, and Listening*. Oxford University Press, Oxford, England, third edition, 2012.

[53] Jeremy Leach and John Fitch. Nature, music, and algorithmic composition. *Computer Music Journal*, 19(2):23–33, 1995.

[54] Fred Lerdahl and Ray S. Jackendoff. *A Generative Theory of Tonal Music*. The MIT Press. MIT Press, London, England, June 1996.

[55] George E. Lewis. Too many notes: Computers, complexity and culture in "Voyager". *Leonardo Music Journal*, 10:33–39, 2000.

[56] Jindrich Libovický, Jindrich Helcl, and David Marecek. Input combination strategies for multi-source transformer decoder. In *Proceedings of the 3rd Conference on Machine Translation* (*WMT 2018*), pages 253–260. Association for Computational Linguistics, 2018.

[57] Andrey Andreyevich Markov. Extension of the law of large numbers to quantities depending on each other. *Proceedings of the Physical and Mathematical Society at Kazan University, 2nd series*, 15:135–156, 1906. In Russian.

[58] Stephanie Mason and Michael Saffle. L-systems, melodies and musical structure. *Leonardo Music Journal*, 4:31, 1994.

[59] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[60] Alex McLean and Roger T. Dean, editors. *The Oxford Handbook of Algorithmic Music*. Oxford University Press, Oxford, 2018.

[61] Eduardo R. Miranda. Cellular automata music: An interdisciplinary project. *Interface*, 22(1):3–21, January 1993.

[62] Eduardo R. Miranda. Cellular automata music: From sound synthesis to musical forms. In *Evolutionary Computer Music*, pages 170–193. Springer London, 2007.

[63] Gary L. Nelson. Real time transformation of musical material with fractal algorithms. *Computers & Mathematics with Applications*, 32(1):109–116, July 1996.

[64] Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer, Vienna, 2009.

[65] Gerhard Nierhaus. *Patterns of Intuition: Musical Creativity in the Light of Algorithmic Composition*. Springer, 2015.

[66] Harry Ferdinand Olson. *Music, Physics and Engineering*. Dover Publications, New York, 1967.

[67] Sarah Opolka, Philipp Obermeier, and Torsten Schaub. Automatic genre-dependent composition using answer set programming. In *Proceedings of the 21st International Symposium on Electronic Art* (*ISEA 2015*), pages 627–632, Brighton, UK, 2015. ISEA International.

[68] Alfonso Ortega, Rafael Sánchez Alfonso, and Manuel Alfonseca. Automatic composition of music by means of grammatical evolution. In *Proceedings of the 2002 International Conference on Array Processing Languages: Lore, Problems, and Applications* (*APL 2002*), pages 148–155. ACM, 2002.

[69] Russell Ovans. Musical composition as a constraint satisfaction problem. In *Proceedings of the 16th International Computer Music Conference* (*ICMC 1990*). Michigan Publishing, 1990.

[70] Russell Ovans and Rod Davison. An interactive constraint-based expert assistant for music composition. In *Proceedings of the 9th Canadian Conference on Artificial Intelligence* (*CSCSI 1992*), pages 76–87, 1992.

[71] François Pachet. Computer analysis of jazz chord sequence: Is solar a blues? In *Readings in Music and Artificial Intelligence*, pages 85–113. Harwood Academic publishers, 2000.

[72] François Pachet. Interacting with a musical learning system: The continuator. In *Proceedings of the 2nd International Conference on Music and Artificial Intelligence* (*ICMAI 2002*), volume 2445 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2002.

[73] François Pachet, Pierre Roy, and Gabriele Barbieri. Finite-length markov processes with constraints. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence* (*IJCAI 2011*), pages 635–642. IJCAI/AAAI, 2011.

[74] Christine Payne. MuseNet. `https://openai.com/blog/musenet`, 2019. Accessed: 2022-05-13.

[75] Flavio O. E. Pérez and Fernando A. A. Ramírez. Armin: Automatic trance music composition using answer set programming. *Fundamenta Informaticae*, 113(1):79–96, 2011.

[76] Somnuk Phon-Amnuaisuk, Andrew Tuson, and Geraint A. Wiggins. Evolving musical harmonisation. In *Proceedings of the 4th International Conference on Artificial Neural Nets and Genetic Algorithms* (*ICANNGA 1999*), pages 229–234. Springer, 1999.

[77] Richard C. Pinkerton. Information theory and melody. *Scientific American*, 194(2):77–87, February 1956.

[78] Przemyslaw Prusinkiewicz. Score generation with L-systems. In *Proceedings of the 12th International Computer Music Conference* (*ICMC 1986*). Michigan Publishing, 1986.

[79] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. `https://openai.com/blog/better-language-models/`, 2019.

[80] Colin Raffel. *Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching*. PhD thesis, Columbia University, 2016.

[81] Camilo Rueda, Gloria Alvarez, Luis Quesada, Gabriel Tamura, Frank D. Valencia, Juan Francisco Díaz, and Gérard Assayag. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. *Constraints*, 6(1):21–52, 2001.

[82] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[83] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, fourth edition, 2021.

[84] Jean A. Samra, Colin Campbell, Dan Eble, Jonas Hahnfeld, Phil Holmes, David Kastrup, Werner Lemberg, Han-Wen Nienhuys, and Francisco Vila. LilyPond. `https://lilypond.org/`, 2022. Accessed: 2022-05-23.

[85] Örjan Sandred. Interpretation of everyday gestures: Composing with rules. In *Proceedings of the Music and Music Science Conference, Stockholm*, 2004.

[86] Felix Schön. PAUL: An algorithmic composer of two-track piano pieces using recurrent neural networks. Bachelor's Thesis, Technische Universität Wien, Institute of Information Systems, E193-03, 2020.

[87] Felix Schön and Hans Tompits. PAUL: An algorithmic composer for classical piano music supporting multiple complexity levels. In *Proceedings of the 21st Portuguese Conference on Artificial Intelligence* (*EPIA 2022*), volume 13566 of *Lecture Notes in Computer Science*, pages 415–426. Springer, 2022.

[88] Arnold Schönberg. *Style and Idea*. Philosophical Library, 1950.

[89] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 16th Annual Conference of the North American Chapter of the Association for Computational Linguistics* (*NAACL 2018*), pages 464–468. Association for Computational Linguistics, 2018.

[90] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[91] Dave Smith and Chet Wood. The 'USI', or universal synthesizer interface. *Journal of the Audio Engineering Society*, October 1981.

[92] Mark J. Steedman. A generative grammar for Jazz chord sequences. *Music Perception*, 2(1):52–77, 1984.

[93] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Proceedings of the 27th Annual Conference on Neural Information Processing Systems* (*NIPS 2014*), pages 3104–3112, 2014.

[94] TensorFlow. `https://www.tensorflow.org/`, 2022. Accessed: 2022-05-11.

[95] Chi Ping Tsang and M. Aitken. Harmonizing music as a discipline in constraint logic programming. In *Proceedings of the 17th International Computer Music Conference* (*ICMC 1991*). Michigan Publishing, 1991.

[96] Alan M. Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[97] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems* (*NIPS 2017*), pages 5998–6008, 2017.

[98] Richard F. Voss and John Clarke. "1/f noise" in music and speech. *Nature*, 258(5533):317–318, November 1975.