TU WIEN Informatics

PAUL: An Algorithmic Composer of Two-track Piano Pieces using Recurrent Neural Networks

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Felix Schön
Registration Number 11777722

to the Faculty of Informatics

at the TU Wien

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

Vienna, 25th June, 2020

Felix Schön | Hans Tompits

Erklärung zur Verfassung der Arbeit

Felix Schön

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, 25. Juni 2020

Felix Schön

Kurzfassung

Algorithmische Komposition bezieht sich auf den Prozess, Musik mittels mathematischer Regeln zu erschaffen, sei dies von Grund auf oder auf Basis bereits bestehender Kompositionen. Sie kann für verschiedene Anwendungen eingesetzt werden, zum Beispiel um dynamisch Musik basierend auf Kontext zu erstellen, menschliche Komponisten in ihrer Arbeit zu unterstützen, oder aber als Kunstform. Verschiedene Anwendungen welche diese Technik implementieren existieren, wie etwa OpenAI's MuseNet, oder Music Transformer. In dieser Arbeit stellen wir eine Applikation vor, PAUL, welche in der Lage ist, im Stil klassischer Klaviermusik zu komponieren. Im Gegensatz zu anderen verfügbaren Implementationen ist das Ziel von PAUL zwei verschiedene jedoch voneinander abhängige Spuren zu erstellen, ähnlich zu der Art wie Klaviermusik aufgebaut ist. Hierfür werden zwei verschiedene neuronale Netzwerke eingesetzt um diese Aufgabe zu bewältigen: ein gewöhnliches rekurrentes neuronales Netzwerk, welches LSTM Einheiten verwendet um die Melodiespur zu erzeugen, und ein sequence-to-sequence Modell, welches die Begleitspur erstellt. Diese Netzwerke werden auf Basis eines Korpus aus bereits bestehenden Kompositionen im MIDI Format trainiert, welche zuerst vorverarbeitet werden. Diese Vorbearbeitung besteht aus dem Verschmelzen verschiedener Spuren welche der selben Hand zugeordnet sind, dem Quantisieren der daraus entstehenden Sequenzen, und dem Bewerten derer Komplexität. Diese Komplexitätsbewertung wird später dafür verwendet, Stücke auf Basis einer Komplexitätseingabe zu erstellen. Die erzielten Ergebnisse sind von angemessener bis guter Qualität im Bezug auf die relative Simplizität der Modelle.

Abstract

Algorithmic composition refers to the process of creating music using a mathematical rule set, be it from scratch or on the basis of a pre-existing composition. It can be used for different applications, such as dynamically creating music based on context, support human composers in their task, or as a form of art. Different applications implementing this technique exist, such as OpenAI's MuseNet or Music Transformer. In this thesis, we present an application, PAUL, capable of composing in the style of classical piano music. In contrast to other available implementations, the goal of PAUL is to be able to create two separate yet dependent tracks, similar to how piano sheet music is constructed. Two different neural networks are employed in order to manage this task: a general recurrent network using LSTM units creating the lead track, and a sequence-to-sequence model creating the accompanying track. These networks are trained on a corpus of pre-existing compositions stored in the MIDI file format which are preprocessed before training the networks. Preprocessing consists of merging tracks belonging to the same staff, quantizing the resulting sequences, and judging their complexity. The complexity rating is later on used in order to create pieces from a complexity rating input. The results are of reasonable to good quality compared to the relative simplicity of the models.

Contents

CHAPTER 1

Introduction

Algorithmic composition (AC) refers to the process of composing music by means of formalizable methods. Early approaches of AC are Guido d'Arezzo's *Micrologus de disciplina artis musicae*, published around the year of 1025, Mozart's *dice game*—a simple way of composing musical pieces using a pre-defined set of elements, which would be randomly chosen by dice roll and then concatenated [33]—, Schönberg's *twelve-tone technique*, or Cage's *aleatoric music*. All these examples stem from a pre-computer era and make use of analogue techniques in order to generate music. While early work in AC using computer programs was pioneered by Gottfried Michael Koenig [23], modern AC methods encompass techniques including but not limited to Markov models [19], generative grammars [28], transition networks [9], genetic algorithms [8], cellular automata [4], and neural networks [36]. For a comprehensive survey of implementations of AC, we refer to Fernandez and Vico [15]. For a detailed overview about AC, we refer, e.g., to *The Oxford Handbook of Algorithmic Music* [32].

In this thesis, we introduce an automatic composer program, called PAUL, to create new musical pieces in the style of classical piano music.[1] The output of PAUL consists of MIDI files containing two distinct tracks, similar to traditional two-handed piano music. The approach underlying PAUL is to employ two separate artificial neural network models—one to generate the lead track and one to compose the accompanying bass track—, that have been trained on examples of classical piano music by several different and influential human composers.

PAUL provides an input parameter specifying the complexity of the generated music. This complexity specification is intended to consist of three different levels of complexity, each more difficult and demanding than the preceding one. This property should be measured according to the difficulty of playing the generated piece on a regular piano,

[1] The name of the program is with all due reference to the famous Viennese pianist Paul Badura-Skoda (1927–2019).

e.g., a very demanding piece should be given a “hard” rating while an introductory piece should receive an “easy” rating.

In contrast to many of the existing music generators, such as MuseNet [36], Clara [38], and Music Transformer [26], which are capable of intricate music composition but only compose a single track per instrument, PAUL provides a generator that can produce two distinct yet dependent tracks. This is done due to considerations of employing PAUL in an educational sight-reading program in the near future. This program should teach piano students to better perform on sight-reading tasks, where the student has to simultaneously read and play the provided sheet music. Using PAUL, the composition of new and not memorized pieces can be assured.

Considering the application described above, another aim of our work is to provide the possibility of specifying the complexity of the generated pieces. This is done in order to be able to accommodate for the players skill level.

The thesis is organized as follows. In Chapter 2, we provide some background on related work. Afterwards, Chapter 3 discusses the required foundations of music theory and some information about MIDI. Chapter 4 gives the theoretical background of the employed neural network models and Chapter 5 covers the implementational details of our program PAUL. Chapter 6 concludes the thesis with a short assessment and an outlook to future work.

CHAPTER 2

Related Work

We start with a selected overview of some implementations using different approaches to algorithmic composition. For a comprehensive discussion on algorithmic composition, we refer to the survey paper by Fernandez and Vico [15].

In general, algorithmic composition can be divided into two distinct categories: The first, *computer-aided algorithmic composition* (CAAC), refers to the process of supporting a human composer in their task. This can be achieved via the means of automating monotone tasks or providing raw starting material in order to kick-start the compositional process. There are many different software products that incorporate this CAAC concept [15]. The second category encompasses AC whose aim it is to generate music from the ground up, in order to create new pieces without human input or intended to be modified by a human composer. The practical part of this thesis makes use of the latter approach, in order to generate finished pieces of specifiable length.

Due to the fact that algorithmic composition can be categorized to the discipline of *computational creativity*, which refers to the computational synthesis of works of art, the "creative process" in the generation alone is enough to justify its existence. This does not mean that AC cannot be applied for or used in other fields. Use cases for algorithmic composition include but are not limited to the following applications [15]:

- AC can be used as a tool to support human composers in their task. In particular, as stated above, CAAC aims to ease the process of composition by providing either raw material or automating certain tasks.
- Although a discipline on its own, *procedural audio* aims to dynamically create music that fits a certain context. Video games for example can use this technique to generate music based on the current scenario. In an action-packed sequence, the music could swell depending on the ongoing events, or in a horror game, the music could react to certain cues.

Table 2.1: Relative frequency of notes in eleven Stephen Foster songs, recreated from Olson [35].

Note	B3	C#4	D4	E4	F#4	G4	G#4	A4	B4	C#5	D5	E5
Rel. Frequency	17	18	58	26	38	23	17	67	42	29	30	17

- The creation of different musical sounds and not entire pieces is the subject of *sound synthesis*, which can be seen as an extension of algorithmic composition.
- Finally, *music analysis* is often composed of similar approaches as is AC.

2.1 Markov Models

Introduced by Andrey Andreyevich Markov [30], these models are a stochastic procedure used in many different fields, including natural language processing. A Markov model (often also referred to as *Markov chain*) specifies a set of *transition probabilities*, indicating the probabilities of transitioning from one state to another. These basic models only account for the current state when calculating transition probabilities.

Nierhaus [33] refers to the work of Harry F. Olson [35], who utilized a Markov model for music generation in 1952. Olson analysed eleven different melodies created by Stephen Foster. In his analysis he focused on pitches and note values in order to model rhythm. For the analysis of the corpus, all the compositions were transposed to a key of `D major`. Table 2.1 for example shows the result of his work regarding pitches. Based on this analysis he created Markov models of the first and second order. The *order* of a Markov chain refers to the amount of previous states influencing the transition to the next one [15]. These models were able to generate simple yet harmonious melodies.

Eigenfeldt and Pasquier [12] developed a real-time music generator using *variable-order* Markov models, which are—in contrast to non variable-order models— not fixed to a certain amount of previous values that influence the transition to the next one. After analysing already existing data they were able to generate a set of chords. In addition a user is able to input three different metrics influencing the generation, these being harmonic complexity, tension between the transitions and a desired bass-line.

2.2 Generative Adversarial Networks

MuseGAN by Dong et al. [10] is based on *generative adversarial networks*, a technique first introduced by Goodfellow et al. [22]. These networks make use of a generative part—in this case composing music—, and an adversarial part, discriminating against the generated data and trying to differentiate between "real" data, and the one constructed by the internal generator. The hope is that by pitting these two parts against each other both the generative and adversarial parts will improve over time, resulting in

compositions of higher quality. Goodfellow et al. give an example of a game between counterfeiters (representing the generative aspect) trying to produce fake currency, and the police force (representing the adversarial aspect). Due to the competition both teams strive to achieve better results, and thus improve their methods, until the fake product is indistinguishable from the real one.

MuseGAN [10] is able to compose pieces consisting of multiple tracks corresponding to different instruments such as drums, guitar or piano. The representation of music is done via a *multiple-track piano-roll*, similar to the one introduced in Section 3.2. The application is able to generate music from scratch or human input, both using a random noise vector as a basis. MuseGAN makes use of three distinct internal models, specifying how many generators and discriminators are used, and how they communicate. The *jamming* model imitates a group of musicians improvising together. Here, multiple generators work independently to create tracks, and these are judged by independent discriminators. The *composer* model is its opposite; a single generator and discriminator are responsible for the creation and judgement of all of the tracks. The *hybrid* model serves as a combination of these two approaches. Several generators share an *intra-track* random vector, which is supposed to unify the creation. After that, a single discriminator judges the result.

Furthermore, they propose interesting measurements of quality of the generated data. These metrics include: The ratio of empty bars, the number of pitch classes per bar, notes longer than thirty-second notes, the tonal distance and for drum tracks a drum pattern. Although these metrics cannot be applied to all algorithmic composition tasks, they can serve as a source of inspiration, and similar metrics are employed for this project in order to judge complexity.

2.3 Answer-Set Programming

Answer-set programming (ASP) is a declarative problem solving paradigm with roots in logic programming and non-monotonic reasoning [18, 16, 17]. The basic idea of ASP includes describing the problem via rules and constraints. Using an answer-set solver, answer sets are then generated from these specifications. These *answer sets* represent valid solutions to the given problem [13].

Boenn et al. [5] introduced Anton (named after Anton von Webern), an automatic composition system which is capable of creating simple melodies. In contrast to older versions of Anton, the newest iteration includes support for rhythmic patterns and is able to work with major, minor, Dorian, Lydian and Phrygian modes. Using an elaborate rule set the program chooses the next note in each part based on the previous one. Such a decision is represented using the fact `chosenNote(P,T,N)`, where `P` specifies the part, `T` stands for the time and `N` for the played note. At any given time each part is only able to play one note.

Another ASP based solution was introduced by Everardo and Aguilera [14]. The proposed

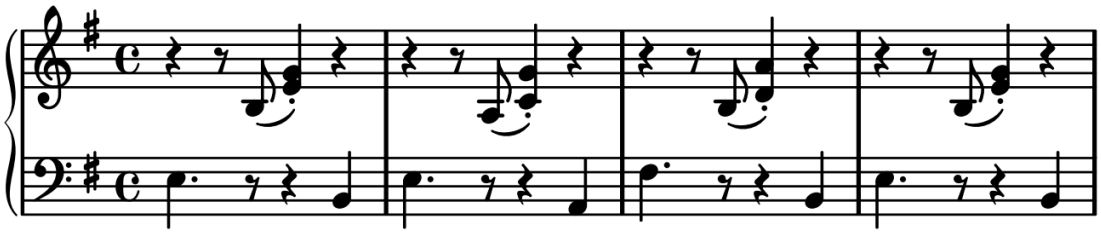


Figure 2.1: An example output generated using *chasp*, reprinted from Opolka et al. [37].

composer, Armin, is capable of generating trance music. It is based on the already introduced Anton model, with additional knowledge regarding trance music. Armin is able to generate three percussive and one melodic track. One described feature consists of better transitions between pieces of a track, such as the transition from a verse to the chorus. At the moment, Armin intends to create a template upon which a trance composer can then build.

Opolka et al. [37] introduced chasp, which is an acronym of "composing harmonies with ASP". It focusses on generating simple accompanying tracks of different genres from chord progressions. These chord progressions are generated using ASP and a set of rules. These rules describe structures and properties of chords. By choosing an eligible chord and appending it to the generated ones, a corpus of chords is created. The system chasp uses LilyPond—the same notation tool used for this thesis—to visually represent the results in human readable score. LilyPond allows for use of templates, which can be populated with notes. Using the notes from the generated chords the LilyPond templates can be filled out, creating valid files containing not only information about which notes to play but also musical rests in order to provide rhythm.

This way harmonious pieces with a very simple structure can be created. Figure 2.1 shows an example output of chasp. For information on how to read score refer to Section 3.1. Based on this example, the pattern-based creation becomes evident; all bars consist of the same pattern applied to a different set of notes making up a chord.

2.4 Neural Networks

Several implementations based neural networks, the technique used in our approach, have been developed. Christine Payne's system Clara [38] makes use of an LSTM network in order to generate piano and chamber music, and is based on the PyTorch[1] library, developed by Facebook. In her paper [38], she proposes two different notations that heavily inspired the representation in our work, these being Chordwise and Notewise.

As their respective name suggests, they are based on a collective chord representation and singular note representation respectively. Due to the fact that Chordwise produces

[1] `https://pytorch.org/`.

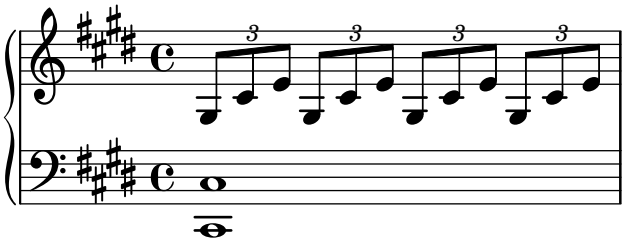


Figure 2.2: First measure of Beethoven's Opus 27 No. 2, first movement, created using LilyPond.[2]

musical rests too frequently and has a very high vocabulary size (approximately 55.000 words), Notewise was used for the majority of generated samples [38].

To illustrate the Notewise notation, Figure 2.2 shows an excerpt of Beethoven's Moonlight sonata (for information on how to read score, refer to Chapter 3). In Notewise notation, the first five notes would be translated to the following representation:

```
p8 p15 p19 wait4 endp19 p22 wait4 endp22 p24 wait4 endp24
```

Clara uses a vocabulary size of approximately 180 words and is able to represent 62 different notes [38] and note values down to sixteenth triplets.

OpenAI's MuseNet [42] is based on a general-purpose unsupervised transformer model trained to predict the next element in a sequence, and works on a very similar principle as Clara. In addition to Clara's play, stop, and wait commands, it supports the specification of different instruments, tempo, and velocity control. The following extract shows an example MuseNet representation [36]:

```
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
```

Due to these additions and the larger scale of the model, MuseNet is able to generate more advanced musical pieces. Furthermore, users are able to interact with the system and create new pieces from pre-written start sequences on their website [42].

Similarly to MuseNet, MusicTransformer [26] makes use of an attention-based neural network, that is able to produce musical pieces with long-term coherence. It uses a representation similar to Simon and Oore's Performance RNN [40]. This composer uses 128 note-on events, the same amount of note-off events, 100 time-shift events in increments

[2] https://lilypond.org/.

of 10ms to 1 second instead of tempo relative representation like MuseNet and Clara, and 32 velocity events, corresponding to the respective MIDI velocities distributed over the same amount of bins. Due to its nature, Music Transformer is able to produce long musical pieces that do not lose their coherence or deteriorate halfway through.

CHAPTER 3

Music Theory and MIDI Foundations

In this chapter, we provide the necessary information in order to understand the inner workings of PAUL. In Section 3.1, we cover music theory, and Section 3.2 gives the basics of MIDI.

3.1 Music Theory Foundations

Unless specified otherwise, the subsequent discussion follows the one given by Boone and Schonbrun [6].

Even though music can be categorized in a large amount of different genres, there is a general consensus about a few different styles, that significantly influence and characterize musical pieces. The *baroque* era is considered to have lasted from around 1600 to 1760, and is complex, soaring and heavily ornamented. It laid the groundwork for the music that followed. Notable composers include Johann Sebastian Bach, Antonio Vivaldi and George Handel. Contrary to popular belief, in the *classical* era (approximately 1760–1820) composers tried to strip down their music to the basic components, favouring strong leitmotifs and melodies. From this rather simple basis they added and decorated using musical elements, to create grand and fancy works of art. Notable composers include Wolfang Amadeus Mozart, Ludwig van Beethoven and Franz Joseph Haydn. In the *romantic* era (1780–1900) music tried to evoke particular feelings and emotions in its listeners. Motives like love, spirituality, nature and the likes were core focuses of this style. Noteable composers include Johannes Brahms, Pyotr Ilyich Tchaikovsky, and Richard Wagner. Composers of the *modern* era often try to differentiate themselves from the giants that came before them, experimenting with new sounds and harmonies. Notable composers include Richard Strauss, Claude Debussy, and Igor Stravinsky.

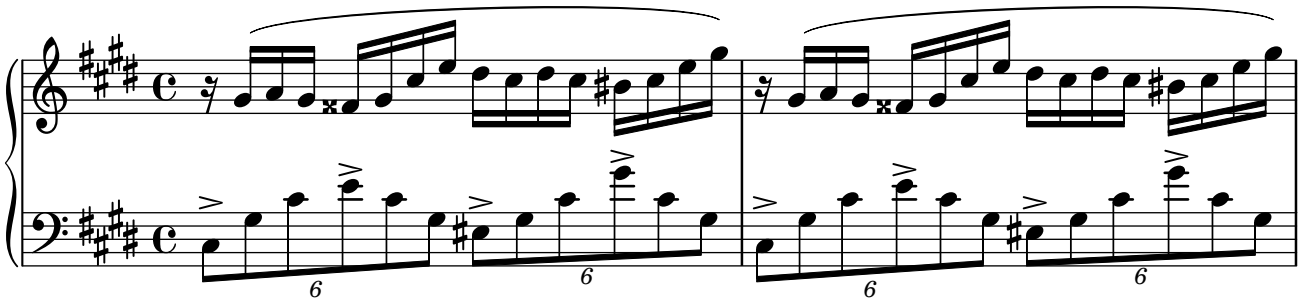


Figure 3.1: Fifth to sixth measure of Chopin's Fantaisie-Impromptu in C♯ minor, op. posth. 66, created using LilyPond.

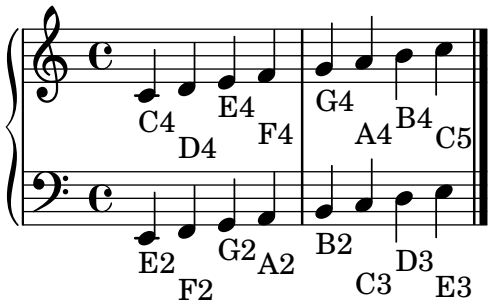


Figure 3.2: Different note names using treble and bass clef, created using LilyPond.

Figure 3.1 shows an exemplary excerpt of Frédéric Chopin's *Fantaisie-Impromptu*, in order to explain the musical foundations based on it. A *note* is simply one dot written on or between a line. More in-depth differentiations will follow later. The excerpt is composed of two musical *staffs*, a *treble* (on Figure 3.1 the upper staff) and a *bass* staff. In piano music these two staffs correspond to the notes played with the right and left hand respectively; a note written on the treble staff should be played with the right hand, analogously for the bass staff.

At the beginning of every staff is a *clef* that tells the player how to read the following elements. The clef defines the lines and spaces following it with note names. The treble clef for instance (shown on the treble staff) circles the note `G4`, which is why it is also referred to as the *g cleff*. The bass clef denotes different note names and pitches, the latter being about an octave lower than the treble clef's. A bass clef denotes that notes written on the line passing through its dot should be read as an `F3`. The *c clef* is another type of clef that is typically used for violas. Due to this thesis' focus on piano music, it is only listed for the sake of completion. Using this information players can infer pitches of other notes on different lines.

Figure 3.2 shows an octave worth of `C major` starting at `C4` and `E2` respectively, to better represent the differences between the clefs. This figure should also help visualize the distance between the two clefs, `C4` marks the lower end of the treble clef's range and

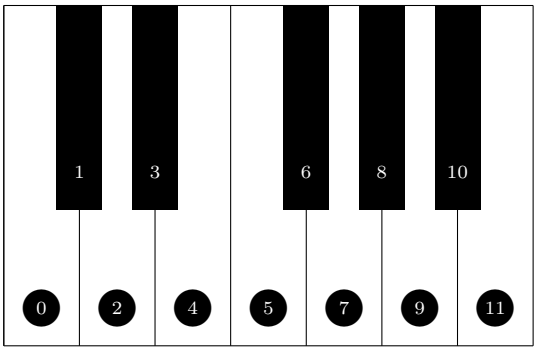


Figure 3.3: One octave of a piano keyboard with numbered keys.

Table 3.1: Notes on the piano keyboard.

Note Name	Number on Keyboard
C	0
C♯/ D♭	1
D	2
D♯/ E♭	3
E	4
F	5
F♯/ G♭	6
G	7
G♯/ A♭	8
A	9
A♯/ B♭	10
B	11

simultaneously the upper end of the bass clef's range. This means the distance between the two staffs is not scaled properly to the distance of the notes themselves.

The brace connecting the two staffs is called the *grand staff*. It is often used in piano writing, or for instruments that can play more than one staff at the same time (e.g., organs).

Accidentals are special symbols usually written in front of single notes in order to change the pitch of a tone. A *sharp* accidental (♯) raises the pitch of the following note by a half step. A *flat* accidental (♭) lowers the pitch of the following note by the same amount. The respective double accidentals (𝄪 and 𝄫) raise or lower the pitch by two half steps each. The *natural* accidental (♮) cancels previously applied accidentals.

Key signatures are a musical property, and can be used to ease the representation of certain pieces. Pieces in the same key use the same accidentals and are based on the same underlying musical *scale*. This fact can be denoted at the beginning of a piece using a number of accidentals at the start of a piece, to the right of the clef. This tells the reader to apply these accidentals to all other notes in the piece if not stated otherwise. Chopin's *Fantaisie-Impromptu* for example is in the key of `C♯-minor`, thus using accidentals for `F♯`, `C♯`, `G♯` and `D♯`.

At this point one has to understand distances in the music system. The western music system uses 12 distinct pitch classes. A jump from one class to the next is constituted of a half step. The piano keyboard is a good visual guide to help understand this concept; each of its keys (black keys included) represents one pitch, and the distance between two adjacent keys is one half step. Figure 3.3 shows an example piano keyboard with numbered keys. Table 3.1 matches these numbers with the respective note names. Note that these note names are not unambiguous but dependent on context; a `C` increased

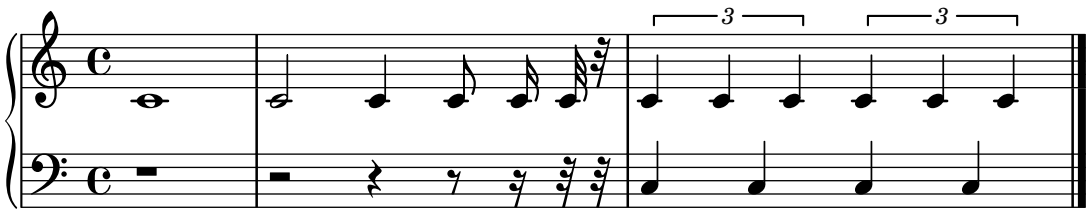


Figure 3.4: An assortment of notes with different note values and rests, created using LilyPond.

with a sharp accidental becomes a C♯, a D decreased with a flat accidental becomes a D♭.

In order to control rhythm and timing we use *note values* and *rests*. The note value of a given note defines its length; how long it is played and the time until the next note can be played. We use different appearances of notes in order to denote note values; Figure 3.4 shows an assortment of the most common note values and rests. The treble staff contains (in this order) in the first bar: a whole note, in the second bar: a half note, a quarter note, an eight note, a sixteenth note, a thirty-second note and a thirty-second rest. The bass staff contains the respective rests. As the name suggests, western music uses a fractional system, where each note value is half the length of its predecessor. This means that two half notes take up the same amount of time as one whole note, analogously for all the other values. Rests follow the same principle, they are a tool to denote rhythm without playing notes. A rest consumes the same amount of time a normal note would, but without playing that note.

Music is divided into *bars*, these are pieces of equal length. Figure 3.4 for example is divided into three distinct bars, all four quarter beats in length. The length of a bar in a piece of music can be noted using *time signature*, which is written next to the clef. The most common time signature is called *common time* and encompasses four quarter beats in a bar. Due to its ordinariness it is usually denoted using the 𝐂 symbol, just like in all the examples given up to this point, although it can be written as $\frac{4}{4}$ as well. Other common time signatures include $\frac{3}{4}$ time and $\frac{2}{2}$ time, the latter also often being represented by 𝐂. In general arbitrary combinations of numerators and denominators are possible for time signatures. In order to fill it, the second bar of Figure 3.4 contains an additional thirty-second rest.

Tuplets are a way of dividing note values into subdivisions otherwise unachievable. A composer denotes a tuplet using a number over a group of notes, thus telling the reader that the note values of the grouped values should be interpreted differently. Tuplets are given as fractions, denoting how many of the tuple's note values should correspond to the normal values. For example, a 3:2 tuplet (referred to as a *triplet*) denotes that three notes of a value should be played in the same amount of time as two of the same value. Other common tuplets include *duplets* (ratio of 2:3), *quadruplets* (ratio of 4:3) and *quintuplets* (ratio of 5:4) [20]. Bar three of Figure 3.4 contains two triplets (encompassing six notes in total) that equal the four quarter notes in length. Often the denominator is

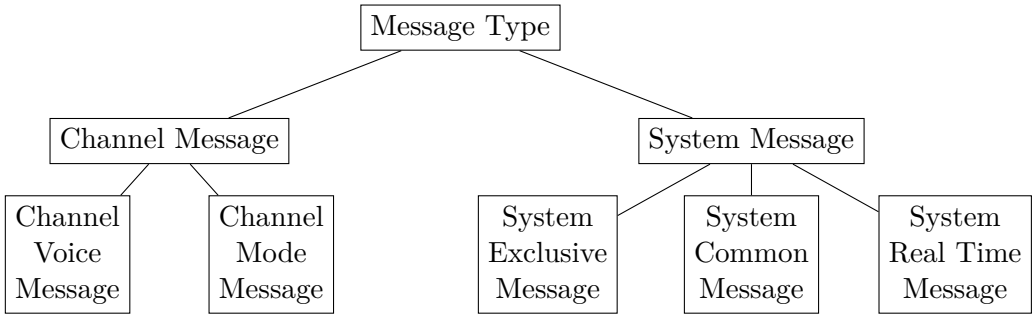


Figure 3.5: Hierarchy of MIDI messages, recreated from the MIDI 1.0 specifications [27].

left out, in that case the next lower power of 2 can be assumed as denominator, although there are exceptions to this rule, like the duplet and quadruplet.

Although there is much more to music and music theory in general (for instance intervals, attenuation, advanced polyphony and much, much more) this section should provide the reader with the necessary amount of information to understand the underlying principles of the program.

3.2 MIDI Foundations

The Musical Instrument Digital Interface[1] (MIDI) is a far spread and widely accepted protocol by computer-aided musicians and composers. MIDI has been conceived and created in the 1982-1983 time frame. It offers a very efficient method of storing and representing musical performance data and is used by both human performers and composers, and computer programs such as digital audio workstations (often referred to as *DAW*s) [27]. Most applications today use version 1.0 of the MIDI protocol, although specifications of version 2.0 have been released and made publicly available [2].

Although MIDI has been created for musical applications, this does not mean that it can't be employed in different fields. It is used in the lighting control and video field to a great extent [31]. Due to the nature of this project this section will solely focus on musical applications of the MIDI protocol.

MIDI data is generally composed of three distinct data groups; the *status message* byte which identifies the type of message sent, and two *data bytes* which convey the message itself. An exception to this rule are real-time and exclusive messages. MIDI messages are sent over one or more of 16 distinct channels, which can be used by the receiving end to control different properties or parameters, or assigned to specific inputs [27].

Figure 3.5 shows the hierarchy of different MIDI messages, based on which the different use cases will be exemplified. *Channel messages* are messages that apply to a specific channel, their status byte includes the channel descriptor. *System messages* on the other

[1] `midi.org`.

hand are not channel specific, and their status byte does not include channel identification. *Channel voice messages* make up the majority of MIDI messages, they contain musical performance data. *Mode messages* contain data specifying how instruments receiving channel voice messages should respond and process them, and are not important for the scope of this project [27].

As stated previously, channel voice messages make up the majority of messages sent. They contain *note on* and *note off* messages, which tell an interpreter to play or stop playing a certain note. For note on and off messages the two data bytes include information about which note to play, and how to play it. Data byte 1 specifies the key pressed and data byte 2 the *velocity*. Velocity can be interpreted as the strength or pressure of a key press [31]. Both key press and velocity accept values in the range 0-127, allowing for 128 different keys played at 128 different velocities. This allows for some headroom regarding a traditional piano keyboard, which is made of 88 keys. MIDI number 21 correlates to the lowest pitch on the piano keyboard, an `A0`, number 108 to a `C8`. More detailed information regarding MIDI number mapping to piano keys can be found at the website of the UNSW Sydney [43].

Other types of messages include *pitch wheel change* messages, which control deliberate changes in pitch, *aftertouch* messages, which send velocity information about already pressed keys, and *control change* messages, which are used to communicate with knobs and sliders on an external interface [31].

In addition to these messages, the so called *meta-messages* can contain information like tempo, time signature, key, copyright, track names, markers or simple text. Meta-events are marked by a status byte set to `FF`. Additionally to the fixed-length data (the identifier, the type of event and the length of the event) meta-messages include a variable-length part, used, e.g., for text representation [27].

MIDI files are of one of three different types: type 0, type 1, and type 2. Due to the fact that after its specification type 2 never really caught on, an explanation will be omitted. In type 0 files all sequenced tracks are stored on a single track, they do not support a multi track data structure. Type 1 MIDI files on the other hand do. They can contain different tracks with separate track information like names, notes, controller information and program changes. Not all DAWs support both type 0 and type 1 files [31]. Ableton Live 10[2] for example, the DAW used for this project, does allow for the import of type 1 tracks, but can only export type 0 files.

Additionally MIDI files specify their *resolution*, also referred to as *PPQN* (*parts per quarter note* or *pulses per quarter note*). This is the amount of discrete time steps per quarter note. The duration of one time step is directly linked to the variable beats per minute setting, specifying the amount of quarter notes played per minute. Common PPQN include 960 or 480 [31]. A low PPQN value can result in an artificial sound, due to the fact that the notes sound too *quantized*. Quantization, in general, is the act of

[2] `https://www.ableton.com/en/live/`.

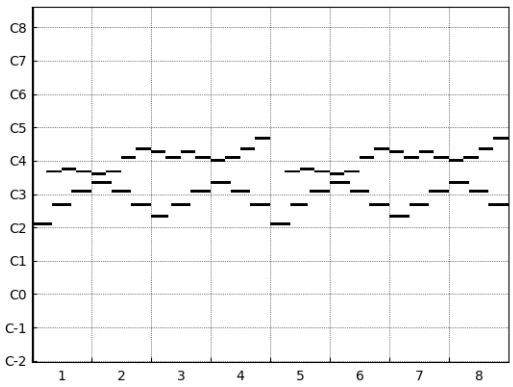


Figure 3.6: Fifth to sixth measure of Chopin's Opus post. 66 represented as piano roll, created using Pypianoroll [11] from MIDI sources retrieved from Krueger's database [29].

perfectly trimming the start and end of a note to a fixed grid, for example a thirty-seconds grid.

A common way of graphically representing MIDI files (at least note on and note off events) is using a *piano roll*. This representation plots note events on a two dimensional x-y graph, where the y-axis represents different pitches or piano keys, and the x-axis the time in beats. Take for example two bars of Chopin's Opus post. 66, represented in Figure 3.1. Figure 3.6 shows these two bars using a piano roll representation, created using the Python library *Pypianoroll* [11] from the classical archives of sequenced MIDI music by Krueger [29]. Though a completely different kind of representation, the overall temporal structure is still visible. This kind of representation will be used throughout this thesis in order to show and compare different MIDI files.

Using the information provided by both Section 3.1 and Section 3.2 readers will be able to understand the musical aspects regarding the inner workings of the program. Chapter 4 will cover the machine learning and neural network aspects of the program.

CHAPTER 4

Artificial Neural Network Foundations

As their name suggests, *artificial neural networks* draw inspiration from theories of neuroscience, the study of the nervous system. Research in this field revealed that the brain consists of *neurons*, nerve cells that conduct electrical signals. A single neuron can make connections with 10 up to 100.000 other neurons. These points of contact are dubbed *synapses*. Using this elaborate network, signals are propagated from one neuron to the next, which both control short-term brain activity and enable long-term changes in the structure of these networks. Although these single units are rather simple, their combination allows for complex processes of thoughts, and consciousness. [39] The estimated number of neurons in a human brain is not unambiguous, many different publications state a number of about 100 billion neurons. More recent and educated guesses have even placed this number to be around 120 billion neurons, although other papers state a number of about 85 billion. [24]

Results from the field of neuroscience have led to mathematical models of *artificial neural networks*, early attempts at implementing artificial intelligence. Since then, these models have gotten more complex and better at handling specific tasks, leading to the field of *computational neuroscience*. The network's abilities such as toleration of noisy inputs or the ability to learn are what makes them a valuable tool [39]. Today they are widely used in a plethora of different fields such as computer vision, text recognition and generation, translation, speech recognition and many more.

We first cover the foundations of basic neural networks in Section 4.1, afterwards, in Section 4.2, we deal with *recurrent neural networks*, and Section 4.3 discusses *sequence-to-sequence* models.

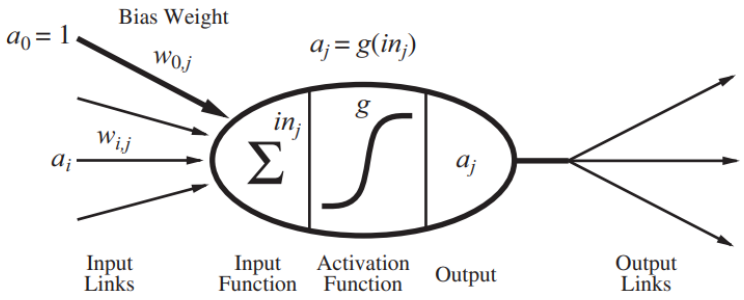


Figure 4.1: The basic mathematical model of a neuron, reprinted from Russell and Norvig [39].

4.1 Basic Neural Networks

From now on, the term "neural network" refers to artificial neural networks rather than biological ones. Unless specified otherwise, the discussion in this section follows Russell and Norvig [39].

As stated above, neural networks are made from an amalgamation of neurons. Figure 4.1 shows the mathematical model of such a unit. Every unit receives a number of inputs from previous units, referred to by a_i. These are scaled by their respective weights, $w_{i,j}$. In addition every unit also has a bias weight $w_{0,j}$ with a fixed input of 1. Each neuron computes the weighted sum of all its inputs using

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i$$

and applies an activation function $g(in_j)$ to compute the output a_j. Commonly used activation functions include threshold functions, which output either 0 or 1, sigmoid functions, which output values between 0 and 1, and rectified linear units (also referred to as ReLU), which are a combination of a threshold function for values smaller than or equal to 0 and a linear function for values greater than 0. Neurons that implement a threshold function are referred to as *perceptrons*, those with sigmoid functions as *sigmoid perceptrons*. Using these nonlinear activation functions the network is able to represent a nonlinear function as well.

Neural networks can further be classified into two distinct categories: *feed-forward networks* and *recurrent networks*. The former is a network made out of neurons that only have connections in one direction, thus the name. This way every node receives input from a previous layer and passes output to the next one. Feed-forward networks form a directed acyclic graph and don't have an internal state.

Recurrent networks are able to feed output back into the network, forming cycles. This way, the output of a recurrent network is dependent on its state, which in turn is dependent

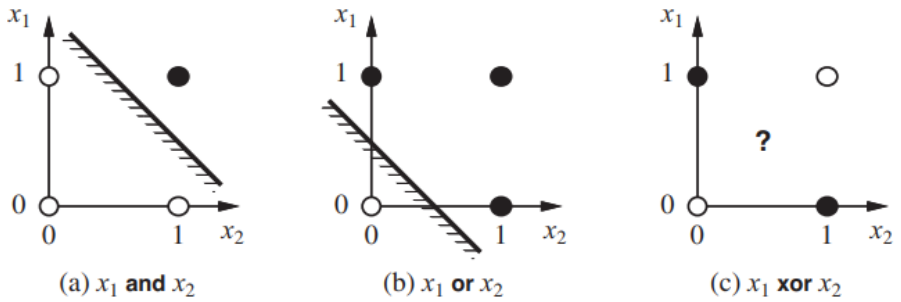


Figure 4.2: Linear separability of different boolean functions using a perceptron, reprinted from Russell and Norvig [39].

on previous inputs. Due to this they are able to support a "short-term memory", meaning their outputs rely on information they have already seen, and not only the intermediate state. Recurrent neural networks will be discussed in Section 4.2.

Neural networks are usually arranged in different *layers*. Units only receive inputs from units of the previous layer, and only pass outputs to units of the next layer. These *multi-layer* neural networks are also referred to by their layer number; a network with one input layer, one hidden layer and one output layer is called a *two-layer neural network* since input layers tend not to be counted for nomenclature. *Hidden layers* refer to layers that are neither input nor output layers.

The potential of neural networks comes from their ability to learn. Provided with large sets of data (for example images and their classification), these networks are able to learn to correctly predict the class of an image, the next character in a sequence, the course of a stock and many more, depending on the context.

One important property to note is the ability of single neurons to model functions. These units (also referred to as *perceptrons*) are only able to learn *linearly separable* functions. The property of linear separability usually applies to two distinct sets of points. In a two-dimensional space these sets of points are linearly separable if and only if there is at least one line with all of the points of one set on one side, and all of the points of the other set on the other side. Figure 4.2 clarifies this concept. Black dots represent an evaluation of the function of 1, white dots an evaluation of 0. The y-axis represents the value of x_1, the x-axis the value of x_2. The boolean `and` function and the boolean `or` function are clearly linearly separable; the black and white dots can be grouped and separated by the given line. The boolean `xor` function is not linearly separable, and thus cannot be represented by a single perceptron.

A way to work around this restriction is to use multi-layered neural networks. A network with a single (sufficiently sized) hidden layer is capable of representing any continuous function with arbitrary precision. In order to represent discontinuous functions at least two hidden layers are needed.

In order to train multi-layered neural networks the output at each neuron in the output layer is broken down into a function of all the preceding weights, neurons and inputs. In order to reduce the loss of a network the weights of the network are then adjusted using *gradient descent*. A more in-depth explanation of the learning algorithms is given by Russell and Norvig [39]. For this thesis the following high-level explanation is sufficient. Using the current state of the model, it is used to make a prediction on an input dataset. The result is then compared to the expected value, and the difference between those two is then used to update the model's weights. This calculation of the error gradient is only an estimate, based on the data seen. Processing data in larger batches and having the network update its weights only after it has seen an entire batch can improve the accuracy of this estimate. The *batch size* refers to the amount of data fed in a single step to the model. Configurations that use the amount of different datasets as the batch size are called *batch gradient descent*. Those that use a batch size of 1 are called *stochastic gradient descent*, while a batch size in between these values is referred to as *minibatch gradient descent*. Reasons for batch size often come down to physical limitations, like the memory of the GPU used for training the network [7].

Unless when dealing with very large datasets, the data is usually fed multiple times to the network. An entire cycle of all the data used in a dataset is referred to as an *epoch*. A problem that can occur with using a high number of epochs is called *overfitting*. Generally networks are trained on one set of data, but are then expected to perform well on a different set that they have never seen. This is referred to as *generalization*. Having a network perform very well on a training set but with a high error on a test or validation set often indicates overfitting; the network has memorized the training set rather than learning the general rules behind it. The opposite of this effect is called *underfitting*, the network has not seen enough data to properly approximate a function. With the increasing size of datasets over time the amount of epochs has reduced, sometimes the entire dataset is not even fed to the network once [21].

4.2 Recurrent Neural Networks

Unlike traditional neural networks that are generally intended to be used on single instances of data, *recurrent neural networks* (RNNs) are well-fitted to process sequential data [21]. They are used for appliances like text generation, language translation, speech recognition and many more.

In the mentioned case of input in the form of sequences, the input can be described using $\overline{x}_1 \ldots \overline{x}_n$, where $\overline{x}_t$ is the input at time step t. For example, when fed a text sequence the input at a time step t could have the form of a *one-hot encoding* [1].

One-hot encoding is a way of representing information using only binary data. It consist of a vector of the size n, where n equals the amount of distinct words in the lexicon describing the data. Every component in the vector except for the one describing the word is set to 0, the one corresponding to the word is set to 1. For the mapping of which word corresponds to which position in the vector an unambiguous, collision resistant

Table 4.1: Exemplary one-hot encoding of words *a, b, c* in the sequence $\{a, b, c\}$.

Word	Bit "a"	Bit "b"	Bit "c"
a	1	0	0
b	0	1	0
c	0	0	1

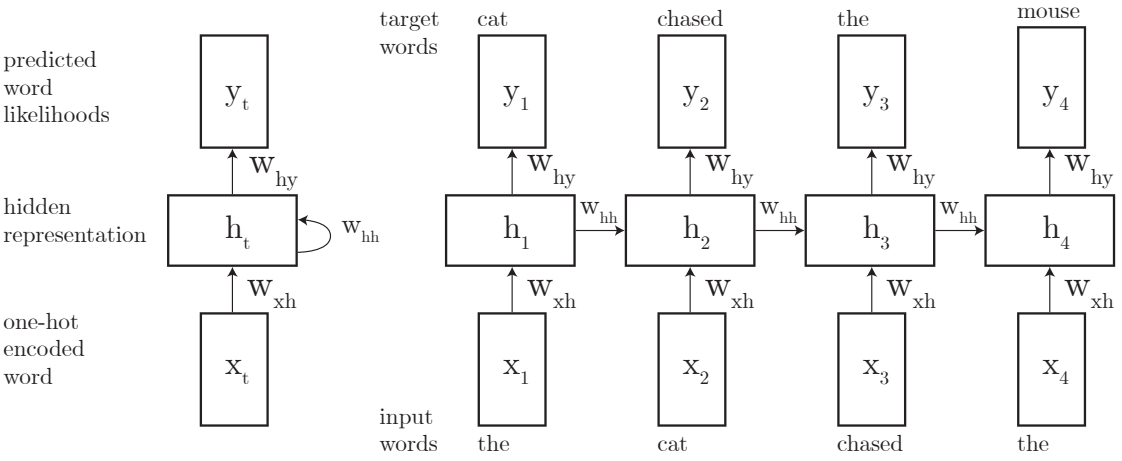


Figure 4.3: Architecture of a recurrent neural network and a corresponding time-layered representation, recreated from Aggarwal [1].

metric hast to be used, for example the position of words in the lexicon. Take for instance the sequence $\{a, b, c\}$ of size $n = 3$. Table 4.1 shows a possible one-hot encoding of the three words for the given sequence.

A core part of recurrent neural networks is the fact that connections from a layer to a previous or the same layer are possible, creating loops. For example, the hidden layer could have a connection to itself, which causes the layer to change after each $\overline{x}_t$. Due to this fact recurrent neural networks are able to output information after each time step, rather than only at the end of a sequence. These outputs are then fed into the network again, making its current state dependent on previous ones. This makes the network able to "remember" sequences, in a sense that an output for an input $\overline{x}_t$ is not only dependent on the input itself, but also the preceding inputs [1].

Figure 4.3 shows the architecture of an example recurrent neural network, its hidden layer has a connection to itself. In order to better understand the inner workings of the network, the figure shows a time-layered representation similar to feed-forward networks on its right hand side. Here, the dependence of previous layers on the ones preceding them is clearly recognizable: layer h_2 receives input from h_1 in addition to the input from x_2.

Although these networks are able to output information depending on previous states,

they are not without flaws. Networks of this form have trouble when it comes to long-term dependencies, connecting information across big temporal gaps.

Take for instance this example adapted from the website of Olah [34]: When asked to generate the next word in the sequence "I grew up in Austria. [...] I speak fluent *German*" (where *German* is the expected output and the text represented by the ellipsis is sufficiently long) a usual RNN will probably be unable to generate the correct output.

Long Short Term Memory networks (LSTMs), first introduced by Hochreiter and Schmidhuber [25], are capable of overcoming these deficits. LSTMs are a kind of *gated* RNNs, which are based on the idea of models with paths through time. Unlike previously introduced RNNs that accumulate information over time and never let it go, LSTMs are able to *forget* some of that information. For example, a network might have to analyse a sequence made of sub-sequences. A gated RNN will be able to learn when to forget about the old state [21].

4.3 Sequence-to-Sequence Models

Sequence-to-sequence neural networks are a relatively new neural network architecture, originally introduced in 2014 by Sutskever et al. [41]. Instead of the one-to-one or many-to-one generation provided by respectively basic neural networks and recurrent neural networks, they are capable of many-to-many translation. They map a sequence of one language or space to a sequence of another language or space, not necessarily of the same length [1]. Sequence-to-sequence (often stylized as *seq2seq*) models are used in fields such as speech recognition, machine translation and question answering [21].

Sequence-to-sequence models are composed of a combination of two different recurrent models, often using LSTM units. One of these networks acts as an *encoder* (also referred to as *reader*), processing an input and generating a context representing this input, often a function of its final hidden state. The second part of the network acts as a *decoder* (also referred to as *writer*), that expects a fixed-length vector representing the encoder's hidden state as input. Using this input it then generates an output based on the encoded state [21].

A great advantage of this model over others is the fact that the input and output sequence can be of different lengths. A commonly used technique for seq2seq models is to insert start and end words respectively to the beginning and end of the input sequence. The decoder model is primed with the start word upon receiving the encoder's hidden state, and tasked to generate words until a desired length is acquired or the end word is generated.

Due to its recency many improvements of the architecture are still developed and found. A possible way of enhancing a sequence-to-sequence is using an *attention* mechanism, such as the one proposed by Bahndanau et al. [3] in 2014. In addition, they proposed making the output vector of the encoder of variable length, in order to work around the limitation of a fixed-length vector when it comes to excessively long input sequences.

CHAPTER 5

Algorithmic Composition Approach

This chapter will discuss the structure and functionality of our tool PAUL. The source code of it can be found at the following repository on GitHub:

`https://github.com/FelixSchoen/BachelorsThesis`.

The following list provides an overview of the project's dependencies:

- Python 3.7.7;[1]
- MIDI 1.0;[2]
- Mido 1.2.9;[3]
- Tensorflow 2.1;[4]
- CUDA 10.1;[5]
- cuDNN 7.6.5;[6]

The rest of this chapter is structured as follows: Section 5.1 covers the general ideas behind the project, Section 5.2 goes over the work done to convert and process MIDI files in order to use them in the neural network, and Section 5.3 covers the process of training the networks in order to generate new pieces and the creation of those.

[1] `https://www.python.org/`
[2] `https://www.midi.org/`
[3] `https://pypi.org/project/mido/`
[4] `https://www.tensorflow.org/`
[5] `https://developer.nvidia.com/cuda-zone`
[6] `https://developer.nvidia.com/cudnn`

5.1 Concept

As stated in Section 1, the aim of PAUL is to create musical pieces of special nature. The output consists of MIDI files containing two tracks: one track representing the treble staff of a piano piece, the other one the bass staff. Furthermore, these two tracks should be dependent on each other, meaning a connection regarding melody or rhythm should exist, in order to provide listeners or players with the feeling of a complete work, more than incoherent random musical phrases.

In addition to the mentioned properties, the possibility of defining a complexity grade is provided by PAUL. Specifying one of the complexity ratings `EASY`, `MEDIUM`, or `HARD` for each of the tracks (not necessarily the same for both) should result in pieces that conform to the complexity rating as specified in Section 5.2.6.

The consideration behind these measures is to eventually use the system in order to create simple musical phrases, used for sight-reading education. Such an educational program would use the generator PAUL in order to dynamically create new pieces based on the skill level of the pupil. A streak of correctly played notes would result in an increase in difficulty level, successive errors in playing in a lowered difficulty.

Although the granularity of the complexity rating (consisting of three distinct levels) is rather low, other adjustments can be made. It can be argued that using a key of a high distance to `C major` or `A minor` on the circle of fifths is more difficult to play than a key with a low distance, due to the fact that the number of accidentals to consider is higher. A possible way of incorporating this would be to automatically retrieve the key of the generated sequence and transpose it in such a way that the resulting difficulty better fits the desired amount.

In order to generate the pieces two different kinds of neural networks will be employed: a recursive neural network using LSTM units in order to independently generate messages for the right-hand track, and a seq2seq model that takes as input the generated track and outputs a dependent left-hand track.

For each of the models, three different sets of trained weights will be stored, corresponding to the specified complexity ratings. In order to generate a new piece, the required weights of the given complexity will be loaded.

The networks are trained on a pre-existing corpus of (mostly) classical piano pieces. These pieces are readily available on the online database of

`piano-midi.de`.

A more in-depth description of the format and nature of the pieces will be given in Section 5.2.1. In order to provide the networks with enough data, the pieces are split up into bars, which will then be individually judged regarding complexity, independently for the right-hand and the left-hand track. This is done in order to ensure an (approximate) equality of the amount of data available for each complexity class. Due to the nature of

classical piano music, and the fact that the overall complexity of a piece can be assumed not to be significantly lower than its hardest part, an over-representation of `HARD` pieces would ensure.

5.2 Data Preprocessing

In order to obtain a basis for training the neural networks, MIDI representations of a large number of works by famous composers provided by different online databases could be used. For instance,

`kunstderfuge.com`

contains a very large database with MIDI files of many compositions of composers like Mozart, Beethoven, Bach, and many more. However, a major drawback of the database is the fact that its data is not created according to a specified standard; many contributors can provide MIDI data, and no coherent shape or form can be assumed. Due to the fact that the network needs a specific format (separate tracks for the right and the left hand, quantized notes and same-length tracks) this and many other databases cannot be used.

On the other hand,

`Piano-midi.de`

provides a regulated database of MIDI files created by a single contributor. These files are exclusively sequenced and not live recordings, which is essential for the quantization process—otherwise, quantization of the tracks cannot be done consistently. Furthermore, the files include several separate tracks corresponding to either the treble staff, the bass staff, or the pedal. The latter will not be used for our purposes.

A collection of different pieces by five different composers was acquired from the above dataset and is used for the training process of the neural networks. Table 5.1 lists all the used pieces sorted by composer and opus number. Each movement or number of an opus is contained in its own MIDI file. Due to batch sizes or time signature constraints not every bar of every piece listed—or even every piece itself—is guaranteed to be used in the training of the neural networks.

5.2.1 Representation

Neural networks cannot accept traditional MIDI data as input, thus a conversion from the MIDI file format to one usable by the input layers of the networks has to be done. In order to represent MIDI files, a representation inspired by Payne's Clara [38] and general MIDI conventions was chosen for the *relative* representation of MIDI. Musical pieces are hereby represented using three different *Message Types*: *play*, *stop* and *wait*.

Table 5.1: MIDI files used as training set for the neural networks. Files retrieved from Krueger's database [29].

Composer	Opus	Movement or Number
Beethoven	Opus 10	Movement 1-3
	Opus 13	Movement 1-3
	Opus 22	Movement 1-4
	Opus 27	Movement 1-3
	Opus 53	Movement 1-3
	Opus 57	Movement 1-3
	Opus 81	Movement 1-3
	Opus 90	Movement 1-2
	Opus 106	Movement 1, 3-4
Chopin	Opus 7	Number 1-2
	Opus 10	Number 1, 5, 12
	Opus 18	Number 1
	Opus 25	Number 1-4, 11-12
	Opus 28	Number 1-24
	Opus 31	Number 1
	Opus 35	Number 1-4
	Opus 31	Number 1
	Opus 53	Number 1
	Opus 66	Number 1
Mozart	Opus 311	Movement 1-3
	Opus 330	Movement 1-3
	Opus 331	Movement 1-3
	Opus 332	Movement 1-3
	Opus 333	Movement 1-3
	Opus 545	Movement 1-3
	Opus 570	Movement 1-3
Rachmaninoff	Opus 3	Movement 1
	Opus 23	Number 2-3, 5, 7
	Opus 32	Number 1, 13
	Opus 33	Number 5-6, 8
Schubert	Opus 15	Movement 1-4
	Opus 53	Movement 1-4
	Opus 15	Movement 1-4
	Opus 90	Number 1-4
	Opus 94	Number 1-4
	Opus 142	Number 1-4
	Opus 143	Movement 1-3
	Opus 960	Movement 1-4

Table 5.2: Internal elements and their representation used in the program.

Message Type	Range	Textual Representation	Numeral Representation
Padding	n/a	n/a	0
Play	21-108	p21-p108	1-88
Stop	21-108	s21-s108	89-176
Wait	1-24	w1-w24	177-200
Start	n/a	m0	201
End	n/a	m1	202

Table 5.2 shows the different types of *Elements* in the relative representation, a combination of a message type and a numerical value representing either a piano key, the amount of ticks to wait, or an identifier. The textual representation is simply used for readability and feedback, the numeral representation for input for the neural networks. The program uses 88 different notes in order to model the entire range of the piano, and thus makes use of 88 play and stop messages. A play message signals the start of a note, the corresponding stop message the end.

The program allows for note and rest values of down to thirty-second triplets. In order to allow for both thirty-second notes and thirty-second triplets, a PPQN value of 24 was chosen, meaning that a quarter note lasts 24 ticks. This defines the length of thirty-second triplets and thirty-second notes with 2 and 3 ticks respectively. In order not to dilute the vocabulary, the longest wait value was set to 24 ticks, one quarter rest. Consecutive rests are simply consolidated when converting to MIDI, and split up when converting from MIDI. In order to be able to support every combination of thirty-second notes and -triplets all values in the range of 1 to 24 were used for wait messages. Note values are simply constructed using a combination of play, wait, and stop messages, rests using only wait messages.

The padding, start, and end messages are used exclusively for the neural networks. The padding message is used in order to provide equally sized sequences within batches. The start and end messages are used in the translation of treble sequences to bass sequences by the seq2seq model.

To illustrate this representation, the first bar of the first movement of Beethoven's Opus 27 (represented in Figure 3.2) was converted from MIDI. The conversion resulted in the following string:

```
p56 w8 s56 p61 w8 s61 p64 w8 p56 s64 w8 s56 p61 w8 s61 p64 w8
   p56 s64 w8 s56 p61 w8 s61 p64 w8 p56 s64 w8 s56 p61 w8 s61
   p64 w8 s64
```

Importing a MIDI file consists several steps of preprocessing. The processing pipeline consists of the following steps:

1. conversion;
2. quantization;
3. splitting into equal time signatures;
4. adjustment;
5. splitting into bars;
6. complexity assessment; and
7. consolidation of adjacent complexity classes.

These processing steps will be covered in the next sections.

5.2.2 Conversion

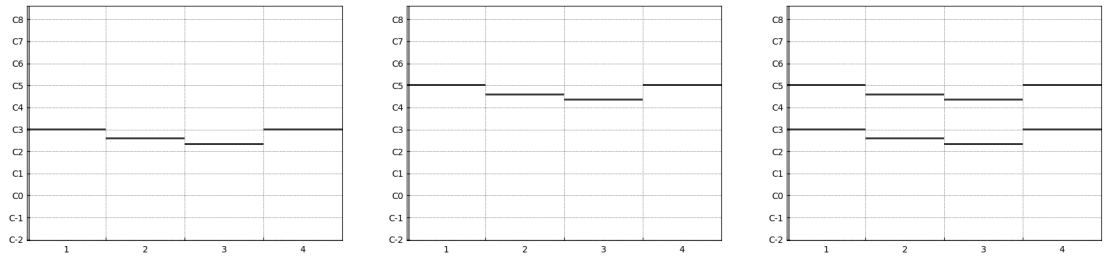
In order to import a MIDI file, it is opened using the Mido Python library. Every track of the file is converted to a `Sequence` object, consisting of a time signature, a name and a list of elements. While iterating over the messages in the MIDI track a wait buffer keeps track of the time between the last play or stop message. Upon encountering such a message the wait buffer is converted to a number of wait messages (due to the fact that wait messages can last for a maximum of 24 ticks), and the original message is appended.

The resulting sequences are analysed regarding their names; all sequences with the word "right" and "left" in their name (not including track names like "copyright") are sorted into the respective lists of tracks for a staff. Tracks of the same hand are then converted to an absolute representation and merged. This absolute representation consists of pairs of play or stop messages and their absolute point in time given in ticks. Merging results in the overlapping of two tracks, combining their notes. Figure 5.1 shows a visual representation of such a process. The sequence depicted in Figure 5.1c results from the combination of the tracks in Figure 5.1a and Figure 5.1b.

5.2.3 Quantization

As stated in Section 3.2, *quantization* refers to the process of fitting the start and end of every note and rest perfectly to a fixed-size grid, in this case a combination of thirty-second notes and thirty-second triplets. This has to be done in order to ensure that splitting along bars for example—if a note is played slightly earlier than the start of a bar it would be assigned to the preceding bar—, the recognition of musical patterns or other processes work as intended.

In order to quantize a sequence, it is first converted to an absolute representation. During iteration over all the elements of the sequence, the value representing the point in time of a message is quantized using the algorithm depicted in Algorithm 5.1. It keeps track of the amount of triplets and normal notes that could have passed or have passed up to



(a) Example track 1. (b) Example track 2. (c) Result of merging tracks 1 and 2.

Figure 5.1: Visual representation of the merging process of tracks, created using Pypianoroll.

Algorithm 5.1: Quantization of notes.

Input: Absolute point in time of an event **wait**
Output: Quantized absolute point in time of an event

```
1  minimumWait = timeNormal * ⌊wait / timeNormal⌋;
2  stepsNormal = ⌊wait / timeNormal⌋;
3  distance = (2 - stepsNormal) * timeNormal - (2 - stepsNormal) * timeTriplet;
4  remainder = timeNormal - distance;
5  if minimumWait + distance > wait then
6  |  if minimumWait + distance / 2 ≥ wait then
7  |  |  return ⌊minimumWait⌋
8  |  else
9  |  |  return ⌊minimumWait + distance⌋
10 |  end
11 else
12 |  if minimumWait + distance + remainder / 2 > wait then
13 |  |  return ⌊minimumWait + distance⌋
14 |  else
15 |  |  return ⌊minimumWait + timeNormal⌋
16 |  end
17 end
```

this point in time, and on that basis calculates the nearest possible value of the message. The messages with the quantized values are inserted into a new absolute sequence which is then converted back to a relative one.

Algorithm 5.2: Split sequence along time value.

Input: The original sequence **sequence**, the point to split at **capacity**
Output: Two sequences resulting from the split of the original sequence at the given point

```
1  initialize currentSequence;
2  while element to process exist do
3      if should use carry elements then
4          element = carryElement;
5      else
6          element = normalElement;
7      end
8      switch element.messageType do
9          case MessageType.play do
10             if element fits in capacity then
11                 add element to currentSequence;
12             else
13                 add element to carry queue;
14             end
15         end
16         case MessageType.stop do
17             add element to currentSequence;
18         end
19         case MessageType.wait do
20             if element fits entirely in capacity then
21                 add element to currentSequence;
22             else
23                 add fitting elements to currentSequence;
24                 add rest to carry queue;
25                 currentSequence = initialize Sequence;
26             end
27         end
28     end
29 end
30 return first Sequence, second Sequence
```

5.2.4 Splitting into Equal Time Signatures

Musical pieces do not have to have one consistent time signature over the course of the entire composition. It is not unusual for composers to change the time signature, even a few times over. Due to the fact that the program relies on splitting compositions into its bars, and these bars rely heavily on time signature, it is important to split a MIDI file

into parts of different time signatures as well.

In order to do so, the absolute timings of time signature changes are extracted. Signature changes are encoded as MIDI messages and can be converted in similar fashion to normal play, stop and wait messages. These timings are then used to iteratively split the sequence into two parts according to the algorithm shown in Algorithm 5.2.

Splitting into bars works analogously, here the split algorithm gets applied iteratively as long as the second sequence is not empty. The capacity is defined by the time signature of the bar.

5.2.5 Adjustment

This relatively simple step assures that the sequence represents a valid MIDI file. It makes a few essential adjustments that will be covered in this section.

First off, it ensures that no two consecutive play messages (without stop messages in between) can exist, removing doubled messages. In a similar style it removes stop messages that precede play messages, or can't be assigned to any messages at all.

Furthermore, in this step wait messages will be consolidated and split up in accordance to the in Section 5.2.1 defined maximum wait size. After this procedure, consecutive wait messages are falling monotonously and of maximum size of 24.

In the next step, empty bars will be padded with wait messages in order to fill them. This ensures that when consolidating bars the spacings of notes will be preserved.

Lastly all messages occurring at the same time step will be sorted by their numeral representation. This ensures that pattern recognition will not be thrown off by internal representation of concurrent events.

5.2.6 Complexity Assessment

In this step, each bar is assigned a complexity value from the set `{EASY, MEDIUM, HARD}`. Important to note is the fact that complexity is judged twice: once for the right hand and once for the left one. These judgements are not dependent on each other.

Complexity assessment is done on the basis of four different metrics, which will be discussed in the next sections. These metrics are based on personal experience (13 years of piano and music theory lessons) and private communication with Prof. Wolfgang Schmidtmayr from the University of Music and Performing Arts Vienna on 10th March 2020. These are in order of importance for the assessment:

- the average time of note and rest values;
- the pattern of the element;
- the amount of concurrently played notes; and

Table 5.3: Amount of complexity classes in the used corpus.

Track	Complexity	Amount	Proportion (rel. to Track)
Right Hand	Easy	2263	29.6%
	Medium	3752	49.1%
	Hard	1630	21.3%
Left Hand	Easy	2810	36.8%
	Medium	3883	50.8%
	Hard	952	12.5%

- the amount of note classes.

Table 5.3 shows the amount of each complexity class per track in the corpus of pieces in the training corpus. Bars of the complexity rating `MEDIUM` make for about 50% of all the bars, the other two classes for the remaining half. It is interesting to note that the amount of bars rated `HARD` is lower for the bass track than the treble one. In piano music the bass track is often used in order to accompany the melody played in the lead one, thus it is often less complex.

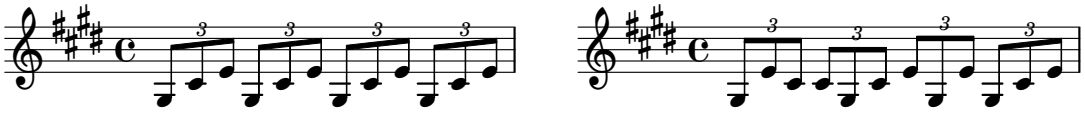
After judging their complexity, adjacent bars are then consolidated in groups of a maximum length of 4. This is done in order to provide the neural networks with lengthier examples. The assumption was, that if only ever fed single bars the networks would not be able to learn longer-term structures. A maximum size of 4 was chosen in order not to isolate certain examples, stray the dataset and artificially increase the need to pad certain sequences.

Average Note and Rest Values

This metric is based upon the average time of wait messages in a bar. A lower average wait time implies lower note and rest values, which in turn imply a more complex piece. In this assessment adjacent rests are consolidated, in order not to artificially lower the average amount.

Pieces with lower note and rest values are arguably harder to play, due to the fact that more notes have to be read in the same amount of time, and they have to be played faster.

Average wait times of greater or equal to 18 (which implies a majority of quarter notes) are evaluated with `EASY`. An average wait time of greater than 12 ticks (the value of eight notes and rests) are evaluated with `MEDIUM`, values lower than these with `HARD`.



(a) Easy pattern. (b) More complex pattern.

Figure 5.2: Two bars consisting of the same notes in a different arrangement in order to illustrate the influence of patterns on complexity, created using LilyPond.

Note Patterns

Patterns and repetitions of musical elements can make playing a score considerably easier. Take for instance the two bars represented in Figure 5.2. Both of these bars consist of the same notes, an extract of Beethoven's Opus 27 No. 2, first movement, arranged in two different ways: while Figure 5.2a is a one-to-one copy of the mentioned piece, Figure 5.2b consist of randomly swapped positions of the elements. Even to the untrained eye, it will be obvious that the second piece is much more difficult.

This program uses regular expressions in order to detect patterns in sequences, and judges their complexity based on these findings. Algorithm 5.3 roughly sketches the methods used to find patterns and calculate complexity. It tries to find the pattern that maximizes coverage in a sequence, without containing a pattern itself. Based on the length of an instance on the pattern and the amount of repetitions of it, the algorithm returns a complexity rating.

Concurrent Notes

Playing several notes at the same point in time does not result in a lower average note or rest value, but does increase the complexity of the piece. Reading multiple notes at one time requires recognition of usual patterns like chords, and the sight reading of more notes in general.

This metric calculates the average amount of notes played between wait and stop messages. Based on this number it returns a complexity rating of `EASY` if the average is less than or equal to 2, `MEDIUM` if it is less than or equal to 3.5, and `HARD` otherwise.

Note Classes

Although the sequence represented in Figure 5.2b is definitely harder to play than the one represented in Figure 5.2a, it still consists of the same notes, and more experienced players will have no difficulties with it. Had the pitches of the sequence changed (in a way that would not result in an easier pattern) the previous metrics would not have estimated it to be more complex. This is why the amount of different note classes played has to be considered as well.

This metric does not consider `C3` and `C4` for instance to be the same note. A value of 4 or less note classes in a bar is assessed as `EASY`, 8 or less as `MEDIUM` and everything

Algorithm 5.3: Find patterns in sequence.

Input: A sequence in text representation **sequence**
Output: Complexity rating of the sequence regarding its pattern

```
1  initialize pattern, patternLength, patternAmount, result;
2  while sequence contains pattern do
3      while sequence contains pattern do
4          while sequence contains pattern do
5              foundPattern = find pattern;
6              if foundPattern covers more than result and foundPattern contains no
                pattern then
7                  result = foundPattern;
8              end
9              patternAmount +=1;
10         end
11         patternLength += 1;
12         patternAmount = 1;
13     end
14     remove result from sequence;
15 end
16 return complexity based on result
```

above that as `HARD`.

5.3 Neural Network Training and Generation

As stated previously, two different kinds of neural networks, each with three different sets of weights, will be created: a recurrent network using LSTMs in order to create the treble or lead sequences, and a seq2seq model in order to generate the bass or accompanying sequences.

Both networks were created in Python using the Tensorflow and Keras APIs, and trained on either an NVIDIA GeForce GTX 1060[7] (6GB model) or an NVIDIA Tesla K80[8].

5.3.1 Treble Sequences Generation

A three-hidden-layer architecture, consisting of an embedding layer, three LSTM layers with dropout layers prepended after each, and a dense output layer. Furthermore, a vocabulary size of 201 was chosen, representing the 200 elements and a padding word. After embedding an input using 32 dimensions, it gets passed through the hidden layers,

[7] `https://www.nvidia.com/en-sg/geforce/products/10series/geforce-gtx-1060/`.
[8] `https://www.nvidia.com/en-gb/data-center/tesla-k80/`.

each with a size of 1024 nodes per layer. A dropout of 20% is applied after each LSTM layer, in order to combat overfitting.

The input data is split into two different lists, consisting of input and target values. The network is then trained to predict the next word in a sequence. In order to be able to use batches, the input data had to be padded for all the entries to be of same length. Batch sizes were chosen in accordance to GPU memory available; since `EASY` sequences usually encompass fewer messages a batch size of 64 was chosen, each subsequent increase in complexity results in a reduction of the batch size by half.

After training the network, using categorical sampling of the predictions—each output has a chance to be drawn proportional to its confidence level—new sequences can be generated using a primer of one or more start messages. Training this network took approximately 4 to 6 hours per complexity rating, dependent on the size and batch size of the dataset. Doubling the amount of layers and nodes—like in the implementation of the seq2seq model—results in exponentially higher training durations and memory requirements, which could not be managed by the locally available NVIDIA GeForce GTX 1060.

5.3.2 Bass Sequences Generation

An encoder-decoder or sequence-to-sequence model was employed to manage the task of bass sequences generation. In similar fashion to the conventional model, both the encoder and decoder consist of an embedding layer with 32 dimensions, and three LSTM layers with 1024 nodes and a dropout of 20% each. A dense output layer is prepended to the decoder model, in order to generate sequences.

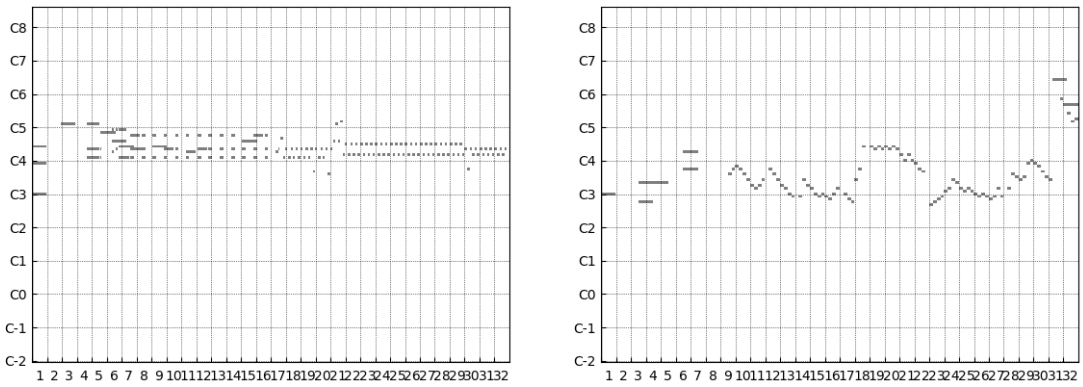
In contrast to the model used for treble sequences, the seq2seq model uses a larger vocabulary size of 203, in order to include the additional words for `start` and `end`.

Due to the much larger network size, training this model demanded significantly more resources compared to the traditional one. Training the network on the local graphics card was only possible on the `EASY` rating with a batch size of 8, a quarter of the batch size of the treble model. Higher complexity ratings had to be trained on a virtual machine using the Microsoft Azure[9] service. This way half of an NVIDIA Tesla K80, this half possessing 12 GB of GDDR5 memory, could be utilized.

Training the `MEDIUM` model for one epoch took approximately 4 hours. It was trained for around 40 hours, resulting in 10 completed epochs.

The generation of sequences works analogously to treble sequences, with the exception that bass sequences need a full treble sequence as input, instead of a primer consisting of possibly few words.

[9] `https://azure.microsoft.com/en-us/`.



(a) Medium 01 Instance.

(b) Medium 02 Instance.

Figure 5.3: Results of a generation of complexity `MEDIUM` primed with the input 128, represented here by a `C3`.

5.4 Evaluation of the Results

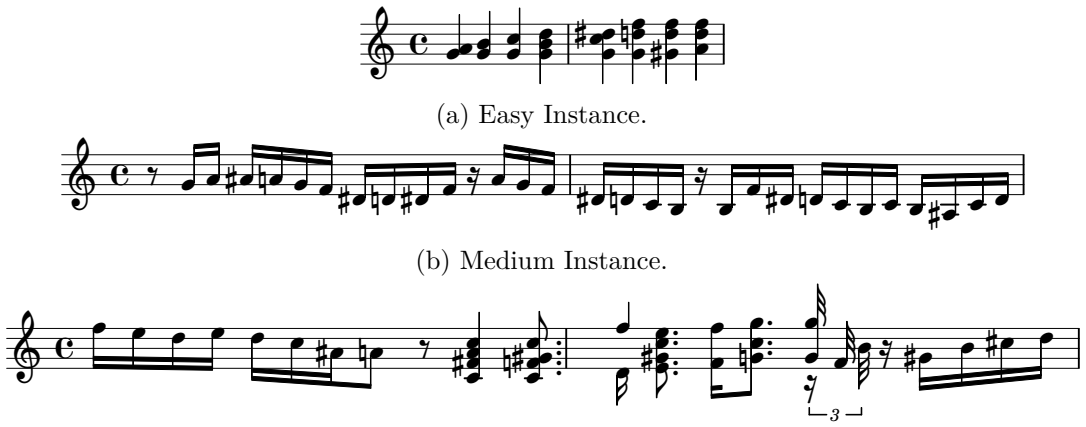
First of all, a clear difference in quality between treble and bass tracks can be observed.

The generated treble sequences are of good, sometimes even great quality. Figure 5.3 shows two different examples generated with the treble model using a `MEDIUM` complexity rating. Audio files generated from these MIDI files can be found at the GitHub repository

`https://github.com/FelixSchoen/BachelorsThesis/tree/master/samples`.

Figure 5.4 shows the musical notation of two bars from example instances generated for each complexity. The generated MIDI files were not transposed, and no accidental-reducing key was chosen. As a result the excerpts may appear more difficult (regarding conventional standards) than if these measures had been taken. PAUL does not consider accidentals regarding complexity. Apart from the aforementioned metric a clear correlation between primed complexity and complexity regarding sight reading can be observed; the `EASY` instance only contains quarter notes and slowly changing chords, the `MEDIUM` instance contains notes of smaller values, and the `HARD` instance additionally contains triplets and polyphony.

One weakness of the network lies in stopping some previously played notes. Sometimes, a few notes drag on until the end of the generated piece, cluttering visual representation. In order to combat this phenomenon, a limit to maximum note length of 24 ticks was introduced. A cut-off function is applied to the generated pieces in order to improve visual representation.



(a) Easy Instance.

(b) Medium Instance.

(c) Hard Instance.

Figure 5.4: Two bars from example instances of different complexities, created using LilyPond.

Table 5.4: Amount of complexity classes generated when primed with a specific class, measured on 80 bars.

Primed Complexity	Easy	Medium	Hard
Easy	60 (75%)	19 (23.75%)	1 (1.25%)
Medium	9 (11.25%)	52 (65%)	19 (23.75%)
Hard	2 (2.5%)	11 (13.75%)	67 (83.75%)

Interesting results can be observed regarding the complexity rating. Table 5.4 shows the results of generating 80 bars per complexity class and judging these bars regarding their complexity, in order to find out if the network really is capable of generating pieces that belong to a given complexity.

A clear trend regarding primed complexity and output complexities is visible. Both `EASY` and `HARD` complexities generate their respective complexity about three quarters of the time, and almost never complexities on the other end of the spectrum. `MEDIUM` complexity generation is a bit more scattered. About two thirds of the time the same complexity is generated, but almost every fourth time a `HARD` bar is generated. In order to combat misaligned complexity the bar can either be generated anew, or as described in earlier chapters the scale can be adjusted in such a way that the complexity is shifted.

Independently of whether or not the used metrics accurately describe the "real" complexity of pieces, the networks are capable of generating pieces conforming to the implemented measurements.

Unfortunately, the current results produced by the bass track generator are nowhere near the quality of the treble ones. Although a network with three layers for each the encoder

and decoder, consisting of 512 nodes each, is capable of generating messages, these do not translate to MIDI well. Although a clear relationship of playing one note, stopping it and then playing another one can be observed, almost no wait messages are being generated, resulting in the generated tracks having no temporal structure. Due to this fact, the evaluations done for the treble sequences cannot be conducted for the bass ones.

A reason for the observed underperformance could be wrong hyperparameters such as a too small network size or a too low amount of epochs. Training the seq2seq model for one epoch on the `MEDIUM` dataset took almost 7 hours.

Due to the built framework regarding musical sequences, adapting the treble network to generate both treble and bass sequences in one track would be a quite manageable task. Merging the treble and bass sequences as described in Section 5.2.2 and then training the treble network on the resulting sequences could result in sequences containing both treble and bass parts.

CHAPTER 6

Conclusion

In this thesis, a composer, PAUL, capable of generating music based on a primer and complexity rating input was created. PAUL works on the basis of preprocessed MIDI files converted to an internal format, representing musical sequences.

Preprocessing of the data consists of several different steps, ensuring that the resulting sequences can be fed to the neural networks. These steps encompass converting MIDI files retrieved from a database to the internal format, merging different tracks that correspond to treble and bass tracks into a single respective track, quantizing and splitting the sequences into parts of equal time signatures, and assessing the complexity for all the different bars.

Using this preprocessed dataset, two different types of neural networks with three different sets of weights were trained. A general recurrent neural network consisting of three layers of LSTM units was trained to generate treble sequences, and a sequence-to-sequence model consisting of three layers of LSTM units per encoder and decoder for a sum of six layers was trained in order to generate bass sequences. The three different sets of weights correspond to the three different complexity ratings used for complexity assessment.

Sequences generated from the treble generator are of good to great quality. The results are characterized by their good temporal structure, harmonics and musical patterns. Furthermore, a clear correlation between specifying a given complexity and the complexity of the generated sequences can be observed.

At the moment the bass generator is unable to produce valid musical sequences. Although a general coherence in the play and stop messages can be observed, they lack the needed wait messages to generate temporal structure. Increasing training duration or network size could improve this behaviour.

Algorithmic composition and neural networks in general are fields that still undergo steady improvements. As stated in previous chapters, using sequence-to-sequence models,

for example, is a relatively new technique [41] that is still being improved upon.

A possible way of further improving these models is to use the *attention* mechanic [42]. MuseNet [36], for example, makes use of such a technique. This mechanism allows networks not to focus on the entire input sequence at the same time but only parts of it [21].

Future iterations of the generators introduced in this thesis could implement such an attention mechanic, in order to improve upon the sequence-to-sequence models. This way, a better translation from treble to bass sequences could be achieved, and the current restrictions of the model could be lifted. Such a mechanism could also be applied to the treble generator, in similar fashion as MuseNet.

Although the results generated by the treble composer are pleasing, the output generated by the current sequence-to-sequence model is sub par. In subsequent work this model could be improved upon using the aforementioned techniques. Additionally, a project of bigger scope could make use of higher performance GPUs, lifting some of the current hardware restrictions. This way, a bigger model size could be employed for the sequence-to-sequence model, and the hyperparameters could be better tuned due to the shorter training timespan.

In order to train more elaborate networks, the hurdles of this project regarding hardware and time limitations will have to be overcome. As graphics card become more advanced with time, training these networks will be more efficient as well. NVIDIA's GeForce RTX 2080 Ti[1] nearly doubles the amount of memory, having 11 GB of GDDR6. With cards like these, it will be possible to train more elaborate networks.

[1] https://www.nvidia.com/de-at/geforce/graphics-cards/rtx-2080-ti/.

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning.* Springer, Cham, Switzerland, 2018.

[2] MIDI Manufacturers Association. `https://www.midi.org`. Accessed: 2020-04-12.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *Proceedings of the Third International Conference on Learning Representations* (*ICLR 2015*), 2015.

[4] Eleonora Bilotta, Eduardo Reck Miranda, Pietro S. Pantano, and Peter M. Todd. Artificial life models for musical applications: Workshop report. *Artif. Life*, 8(1):83–86, 2002.

[5] 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.

[6] Brian Boone and Marc Schonbrun. *Music Theory 101: From Keys and Scales to Rhythm and Melody, an Essential Primer on the Basics of Music Theory.* Adams Media, New York, 2017.

[7] Jason Brownlee. *Machine Learning Mastery.* `machinelearningmastery.com`, 2019.

[8] Anthony Richard Burton and Tanya Vladimirova. Generation of musical sequences with genetic techniques. *Comput. Music. J.*, 23(4):59–73, 1999.

[9] David Cope. Computer modeling of musical intelligence in emi. *Computer Music Journal*, 16(2):69–83, 1992.

[10] 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 Thirty-Second AAAI Conference on Artificial Intelligence* (*AAAI 2018*), pages 34–41. AAAI Press, 2018.

[11] Hao-Wen Dong, Wen-Yi Hsiao, and Yi-Hsuan Yang. Pypianoroll: Open source python package for handling multitrack pianorolls. In *Late-Breaking Demos of the 19th International Society for Music Information Retrieval Conference* (*ISMIR 2018*), 2018.

[12] Arne Eigenfeldt and Philippe Pasquier. Realtime generation of harmonic progressions using constrained markov selection. In Dan Ventura, Alison Pease, Rafael Pérez y Pérez, Graeme Ritchie, and Tony Veale, editors, *Proceedings of the First International Conference on Computational Creativity* (*ICCC 2010*), pages 16–25. computationalcreativity.net, 2010.

[13] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, editor, *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, Berlin, 2009.

[14] Flavio Everardo and Antonio Aguilera. Armin: Automatic trance music composition using answer set programming. *Fundamenta Informaticae*, 113:79–96, 01 2011.

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

[16] 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, San Rafael, CA, 2012.

[17] 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.

[18] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the the 5th International Conference and Symposium on Logic Programming* (*ICLP/SLP 1988*), pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[19] Jon Gillick, Kevin Tang, and Robert M. Keller. Machine learning of jazz grammars. *Comput. Music. J.*, 34(3):56–66, 2010.

[20] Perry Goldstein. *Rudiments of Music: A Concise Guide to Music Theory*. Kendall Hunt, Dubuque, 2018.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, Cambridge, Massachusetts, 2016.

[22] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Y. Bengio. Generative adversarial networks. *Advances in Neural Information Processing Systems*, 3, 06 2014.

[23] Björn Gottstein. Gottfried Michael Koenig: Die Logik der Maschine. In Björn Gottstein, editor, *Musik als Ars Scientia. Die Edgard-Varèse-Gastprofessoren des DAAD an der TU Berlin 2000–2006*. Pfau Verlag, Saarbrücken, 2006.

[24] Suzana Herculano-Houzel. The human brain in numbers: A linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, 3:31, 2009.

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

[26] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, and Douglas Eck. Music Transformer: Generating music with long-term structure. *CoRR*, abs/1809.04281, 2018.

[27] MIDI Manufacturers Association Incorporated. *The Complete MIDI 1.0 Detailed Specification*. The MIDI Manufacturers Association, Los Angeles, third edition, 2006.

[28] Kris Makoto Kitani and Hideki Koike. Improvgenerator: Online grammatical induction for on-the-fly improvisation accompaniment. In *10th International Conference on New Interfaces for Musical Expression, NIME 2010, Sydney, Australia, June 15-18, 2010*, pages 469–472. nime.org, 2010.

[29] Bernd Krueger. Classical Piano MIDI Page. `http://www.piano-midi.de`. Accessed: 2020-04-12.

[30] Andrey Andreyevich Markov. Extension of the law of large numbers to quantities, depending on each other (1906). reprint. *Journal Électronique d'Histoire des Probabilités et de la Statistique [electronic only]*, 2(1b):Article 10, 12 p., electronic only–Article 10, 12 p., electronic only, 2006.

[31] Sam McGuire. *Modern MIDI: Sequencing and Performing Using Traditional and Mobile Tools*. Routledge, London, second edition, 2019.

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

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

[34] Christopher Olah. Understanding LSTM Networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs`, 2015. Accessed: 2020-04-12.

[35] Harry Ferdinand Olson. *Music, physics and engineering*. Dover Publications, New York, 1967.

[36] OpenAI. MuseNet. `https://openai.com/blog/musenet`. Accessed: 2020-04-19.

[37] Sarah Opolka, Philipp Obermeier, and Torsten Schaub. Automatic genre-dependent composition using answer set programming. In Thecla Schiphorst and Philippe Pasquier, editors, *Proceedings of the Twenty-First International Symposium on Electronic Art* (*ISEA 2015*), pages 627–632, Brighton, UK, 2015. ISEA International.

[38] Christine Payne. Clara: Generating Polyphonic and Multi-Instrument Music Using an AWD-LSTM Architecture. `http://www.christinemcleavey.com/files/clara-musical-lstm.pdf`, 2018.

[39] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education Limited, Harlow, third edition, 2016.

[40] Ian Simon and Sageev Oore. Performance RNN: Generating Music with Expressive Timing and Dynamics. Magenta Blog, `https://magenta.tensorflow.org/performance-rnn`, 2017.

[41] 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.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Proceedings of the 30th Annual Conference on Neural Information Processing Systems* (*NIPS 2017*), pages 5998–6008, 2017.

[43] Joe Wolfe. Note Names, MIDI Numbers and Frequencies. `https://newt.phys.unsw.edu.au/jw/notes.html`. Accessed: 2020-04-12.