Generating Musical Compositions through a Data-Driven Approach along with Static Implementations of Theoretical Principles

Daniel Jiang

Publisher: Dean Prof. Dr. Sascha Alda

Hochschule Bonn-Rhein-Sieg – University of Applied Sciences, Department of Computer Science

Sankt Augustin, Germany

December 2022

Technical Report 04-2022

Hochschule Bonn-Rhein-Sieg
University of Applied Sciences

ISSN 1869-5272

ISBN 978-3-96043-103-9

This work was supervised by	Wolfgang Heiden Ernst Kruijff
Co-Author	Wolfgang Heiden

Abstract

In the field of automatic music generation, one of the greatest challenges is the consistent generation of pieces continuously perceived positively by the majority of the audience since there is no objective method to determine the quality of a musical composition. However, composing principles, which have been refined for millennia, have shaped the core characteristics of today's music. A hybrid music generation system, mlmusic, that incorporates various static, music-theory-based methods, as well as data-driven, subsystems, is implemented to automatically generate pieces considered acceptable by the average listener. Initially, a MIDI dataset, consisting of over 100 hand-picked pieces of various styles and complexities, is analysed using basic music theory principles, and the abstracted information is fed into explicitly constrained LSTM networks. For chord progressions, each individual network is specifically trained on a given sequence length, while phrases are created by consecutively predicting the notes' offset, pitch and duration. Using these outputs as a composition's foundation, additional musical elements, along with constrained recurrent rhythmic and tonal patterns, are statically generated. Although no survey regarding the pieces' reception could be carried out, the successful generation of numerous compositions of varying complexities suggests that the integration of these fundamentally distinctive approaches might lead to success in other branches.

Digital Object Identifier https://doi.org/10.18418/978-3-96043-103-9

Contents

Chapter 1

Introduction

Having existed since prehistoric times, music has incessantly occurred in every known human culture as a form of entertainment. In the mid-2010s, with other entertainment sectors growing in the past few years, the music industry's revenue has shrunk significantly compared to its peak in the early 2000's due to digitisation and convergence of production and distribution systems. Nevertheless, with streaming services having become a trend, the music sector has once again grown after its previous plummet [Loz20]. Nowadays, in the development of digital entertainment such as movie productions and video games, utilising music for various purposes has become the norm in the those industries, additionally to music itself being a separate sector on the market, implying that the production, as well as the consumption, is steadily increasing.

Since the beginning of recorded history, albeit the contentious origins of music, there has never existed an objective way to determine whether a musical composition sounds "good" due to this being a subjective matter that each individual has to decide for oneself. However, this specific question has been topic of several debates throughout history and disputants have yet to come to a conclusion that the majority will agree with. Even so, the musical style, which heavily varies depending on the culture, society, time period and geographical location, is of highest importance whilst attempting to objectively classify a piece.

The main objective of this study is the exploration of a combined approach - making use of both data-driven methods as well as theoretical principles - to implement an application that is capable of generating an instrumental piece that is considered satisfactory by the average listener in terms of melody, rhythm and harmony. The piece itself is to be presented as a *Musical Instrument Digital Interface* (MIDI) file - a standardised format in modern music production-, utilising datasets of various compositions as a statistical base for a machine learning approach as well as static implementations of theoretical principles based on musical characteristics of mainstream music of the past millennium. This project does not aim to further improve additional characteristics like dynamics and timbre of the composition, as doing this would require a tool more powerful than MIDI due to its internal functionality and much more time than given with the time frame of a bachelor's thesis.

To set a minimum goal for the project, the most basic composition that will be considered a piece shall be a melody with several clear, repeating patterns accompanied by a chord progression that harmonise with each other by Western standards [Ric98]. The piece itself is to be generated using a combined approach from basic machine learning models along with implementations of theoretical musical principles. Moreover, the application has to be accessible and partially configurable from either a command line interface or a graphical user interface.

Furthermore, additional features of the application could be supplementary sub-melodies, further accompanying elements and a more advanced instrumentation. Should the remaining time be sufficient after implementing those, creating, or rather improving, the application's graphical user interface, exporting artefacts and adjusting the code to suit a higher configurability are options worth considering.

Chapter 2

General Background

Nowadays, mainstream music is considerably influenced by earlier music, such as music from the Classical (1750s - 1820s), Baroque (1600s - 1750s) and Romantic (1820s - 1900s) period, thus creating certain guidelines of which characteristics in a composition will sound acceptable to the average listener. Despite music having evolved significantly in the past century, most of the compositions from that time period are still based on the same characteristics (e.g. melody, harmony, rhythm, and timbre) as the ones from the Renaissance, for example.

Although numerous endeavours to define music have been made, none of the definitions has been officially accepted, implying that technically anything containing any sounds can be considered music[1], which is why there can only be principles, but no rules for composing music [SSS67]. Nevertheless, the result of the musical evolution is that most people have adjusted their ears to certain intervals in-between frequencies, which implies that harmonic rules could be derived according to that.

Notwithstanding the fact that this proves the possibility of utilising these mainstream characteristics to compose a piece that will be considered harmonically pleasing to the ear, it is difficult to statically generate compositions that are not completely monotonous as a result of the brain's demand for something more compelling [Cly13]. On top of that, there is only a finite amount of possible compositions of a given duration - assuming that these do not exploit the utilisation of endless tuples, time signatures or orchestrations. An alternative would be the use of high level probabilistic rules, which can subsequently be applied on parameter learning algorithms, as well [SD10].

While multiple systems, which make use of different kinds of *Artificial Neural*

[1]Since music is a form of art, even complete silence can be interpreted as music, e.g. *4′33″ (1952)* by *John Cage (1912 - 1992)*, which consists of four minutes and 33 seconds of absolute silence.

Networks (ANN) to compose polyphonic music, have been successfully implemented [MMJ19; LWZM15; HHC18; HC16], purely using machine learning as an approach has its disadvantages, as well. Considering the existence of difficulties on a musical level while attempting to interpret a composition, these problems are also being translated onto an implementational level. Due to the nature of music not having clear rules, a chord, for example, could technically be interpreted in various ways. Granted that only few of those chord interpretations would make sense, having more than one option leaves room for multiple interpretations of the current phrase, potentially changing the entire structure of a piece [SSS67]. Thus suggesting, that the main computational challenges of chord recognition are the ambiguity and contextual dependency, which are both proportional to the harmonic complexity [SR08]. This is why there is no reliable algorithm to determine which chord exists at what time in a more complex MIDI file (e.g. one that contains more than a single chord per measure, extended chords, non chord tones and/or interactions between multiple parts). Hence implying that - to properly train a model to interpret and generate more complex compositions - one would have to feed it immense amounts of data, so the correlations and patterns between every single note can be computed, instead of larger elements like chords. Neither the acquisition and preparation of the training data nor the time being used to train the models are realistic in this case. Therefore, a proper chord recognition algorithm is crucial to this approach.

As for the musical interface, MIDI offers enough functionalities that satisfy the requirements for reading, editing and writing data, therefore it has been established as a technical standard for now. It makes use of the *chromatic scale* [For13] - which has mostly been used in popular music around the globe since the early modern European periods - by mapping the scale's frequencies onto notes - represented as integers. A MIDI file itself contains event messages, that describe certain musical elements. In the majority of MIDI pieces, most of these are *Note On* and *Note Off* events that conjointly represent a note being played, which eventually have to be processed to derive larger elements, such as chords.

Chapter 3

Related Work

Regarding data-driven approaches for music generation problems, it is crucial to understand the syntactic rules within a composition. Additionally to improving the synthesis, classification and information extraction, this knowledge can be taught and implemented by the application of machine learning [Dan00]. For automatic music generation problems, popular ANN choices include *Convolutional Neural Networks* (CNN) [LB+95] and *Recurrent Neural Networks* (RNN) [RHW86].

A *Long Short-Term Memory* (LSTM) [HS97] network is an artificial RNN architecture used in the field of deep learning, which is able to process sequences of data. This type of ANN is used by Mangal et al. [MMJ19] in a successful attempt at creating a system that automatically generates MIDI music. Similarly, Lyu et al. [LWZM15] integrate a LSTM network with a *Restricted Boltzmann Machine* (RBM) [Smo86] for high dimensional data modelling, with polyphonic music generation as field of application. As last example, the variation by Huang et al. [HHC18] is a model that combines a CNN with a LSTM network by establishing convolution layers to extract features of the musical score matrix, which has previously been converted from a MIDI file.

A different kind of approach is the implementation of probabilistic rules, as Sneyers and De Schreye [SD10] have done. In their paper, they make use of the probabilistic logic language's ability to express statistical and relational information to create a new system for automatic music generation by modelling fragments of musical compositions using high level probabilistic rules. The system allows parameter training from arbitrary examples as well as previously generated pieces.

Contrary to that, Herremans and Chew present a system capable of automatic music generation with recurrent pattern constraints and tension profiles that enables the ability to generate music according to any given context [HC16]. This system implements a *variable neighbourhood search* [MH97] optimisation algorithm using both hard and soft constraints in its generation process to allow for a successful composition.

Disregarding the prior composing problem itself, generating accompaniment parts for a present piece is achievable using different techniques, too. The work of Crestel and Esling introduces the first system performing automatic orchestration from a real-time piano input [CE16], by learning the underlying regularities existing between piano scores and their orchestrations by well-known composers. To accomplish that, a class of statistical inference models based on a RBM is investigated.

Lastly, referring to the previously mentioned chord estimation problem, two ways to interpret chords within a piece are a CNN based deep feature extractor trained to estimate chords of music audio recordings, as proposed by Wu and Li [WL18] and Scholz and Ramalho's *Complex Chords Nutting* (COCHONUT) system, which uses contextual harmonic information to solve ambiguous interpretations [SR08] in MIDI data. The latter study shows results indicating that the use of decision theory, optimisation, pattern matching and rule-based recognition respectively enhance the results, implying that the used approach is applicable on MIDI pieces of higher complexity, as well.

Similar to the previously mentioned works, LSTM networks will be implemented to predict on sequential music data, since the findings of said studies indicate a practical application of this specific type of neural network. As for the chord recognition and orchestration problems, static approaches based on methods utilising music theory will be applied in order to reduce the problems by abstracting fundamental information.

Chapter 4

Prerequisites

Prior to the implementation of the ANN, a static analysis will be performed on the MIDI data to create the network's input data. To ensure the analysis's comprehensibility, a rudimentary understanding of music theory concepts crucial for this study has to be established. Furthermore, basic concepts of ANNs will be introduced before the in-depth presentation of either approach, too, since they are fundamental to the understanding of the ensuing chapters of this thesis.

4.1 Theoretical Principles

The theoretical principles of musical composition are a set of guidelines and practices of basic music theory concepts that are essential to the comprehension of the music creation process.

4.1.1 Definitions

Since any arbitrary sound contains a certain amount of oscillating waves at a fixed frequency, the membranes inside the human ear will perceive it differently, thus resulting in what is known as *pitch*, one of the most basic fundamentals in music theory. It is used in the *chromatic scale*, a set of twelve specific pitches, or *notes*. In 1979, Forte described the chromatic scale as a series of *half steps* - the smallest interval of the scale - which comprises all the pitches of our equal-tempered system [For13]. In Western music, usually, a *diatonic* scale is used. It uses seven notes of the chromatic scale and the intervals between notes consist of either a *whole step* (two semitones) or a *half step* (one semitone). The series *2-2-1-2-2-2-1* or *full-full-half-full-full-full-half* describes a *major* scale's intervals between its notes. In a diatonic scale, all notes are numbered by their amount of steps from the first to seventh *degree*.

A musical *piece* is either a vocal or instrumental composition, which usually consists of multiple sections, that are formally structured into arrangements of musical units of rhythm, melody, and/or harmony that show repetition or variation with a given orchestration [Tit09]. This definition contains three key concepts, which can be interpreted as follows: In music theory, *rhythm* refers to any regular recurring motion or a movement marked by the regulated succession of strong and weak elements, or of opposite or different conditions. A *melody* is a linear succession of musical tones - a combination of pitch and rhythm - that is perceived as a single entity. *Harmony* is a perceptual property of music that consists of concurrently occurring pitches; the sound of two or more notes heard simultaneously. Furthermore, the *orchestration* of a piece refers to the used combination of instruments or parts inside the composition [Toc77; Ric98; Ape69].

In Western music, *chords* - a set of simultaneously played notes, usually from the chromatic scale - have been the most widely used elements to create harmony. Depending on the interval (e.g. an *octave* is an interval that consists of 12 semitones) between its notes, different chord types result, giving each chord its own characteristics, allowing more complexity, whereas the chord's *root note* - the note, on which the remaining notes of the chord are built - determines its name. Arranged into order, a set of chords becomes a *chord progression*, the foundation of Western popular music on which melody and rhythm are built [Toc77; Ric98].

Since chord progressions are the foundation of most music, they direct the flow of the piece and its overall perception, invoking the audience's emotions associated with the chords [YC11; Cly13]. Additionally, the function of chord progression is the establishment of a *key*, which determines the *scale* - specifically ordered notes of the chromatic scale - that forms the composition's basis. In the context of the key, each chord has its own function. *Resolution* is one of the most important functions: Usually, a chord progression starts by building up tension with each chord change, and follows up by decreasing said tension, until it *resolves* [Toc77; Ric98].

To enhance the readability of the piece's musical notation, it is common practice for the composer to include *key signatures* and *clefs*. Both are mandatory symbols to identify notes within any piece: The key signature implies the scale, while the clef indicates the range in which the notes lie.

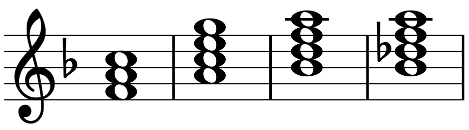


Figure 4.1: Chord Progression F-Am^7-B♭M^7-B♭m^{M7}.

The example in Figure 4.1 is a non-resolving chord progression in the key of F written in musical notation utilising the *treble clef*. It consists of four chords: F major, A minor seventh, B-flat major seventh and B-flat minor major seventh, which are usually denoted as F, Am^7, B♭M^7 and B♭m$^{\mathrm{M7}}$, respectively. Based on the given key signature, the scale can be derived, which allows the reader to calculate the amount of semitones between each chord's notes, consequently determining the chord type.

4.1.2 Context

Put into context of the study, the above defined concepts have to be modelled as data structures and implemented as either hard or soft constraints in the analysis as well as in the generation process. The data models will be implemented as conventional classes in any object-oriented programming language, while representing any notes as numbers. Constraints will be used in the chord estimation algorithm and the ANN's element prediction.

Lastly, the importance of using MIDI for all music handling needs to be manifested. The A note above *middle C* or C_4 - the fourth C key from left on a standard 88-key piano keyboard - (hence also known as A_4) is usually set at a frequency of 440 Hz, since it has been established as the pitch standard. It is known as *A440* or *Stuttgart pitch* [Ape69] and used in the specification of the *MIDI Tuning Standard* (MTS) [Ass21]. The main reason MIDI is used as the music interface for this study is not solely its popularity, but its straightforward internal pitch-to-note mapping, which simplifies the conventionally logarithmic frequency scale to a linear scale of integers: The MTS defines its data value d as

$$d = 69 + 12 \times log_2 \left(\frac{f}{440\text{ Hz}} \right) \tag{4.1}$$

with f being the frequency. The quantity $log_2\left(\frac{f}{440\text{ Hz}}\right)$ represents the number of octaves above A_4, which can be multiplied by 12 to calculate the number of semitones above that frequency. Adding 69 - the MIDI note number of A_4 - equals the number of semitones above the C five octaves below C_4. To visualise the equation's application, two examples - of which both utilise the A440 standard for the substituted frequencies - are shown below in Equations 4.2 and 4.3:

$$A_4 = 69 = 69 + 12 \times log_2 \left(\frac{440\text{ Hz}}{440\text{ Hz}} \right) \tag{4.2}$$

$$C_4 = 60 \approx 60.000028765 = 69 + 12 \times log_2 \left(\frac{261.626\text{ Hz}}{440\text{ Hz}} \right) \tag{4.3}$$

4.2 Artificial Neural Network

The modelling of feedforward artificial neural network[2] computing systems has been originally inspired by biological neural systems (neural circuits). Nonetheless, as modern machine learning research is driven by mathematical and engineering principles, ANNs are consequently described as, inter alia, "function approximation machines that are designed to achieve statistical generalisation" [HGBC18].

4.2.1 Formal Definition

Neural networks [HGBC18; LWG22] can be defined by the mapping $y = f(x; \theta)$, where x denotes the input and θ the parameters that are required to optimise the approximation to an underlying function f^* of the training data[3]. f is usually modelled by a composition of subsequent functions, in which each one represents a *layer*. The dependencies of the layers on one another can be illustrated by a directed acyclic graph that represents the network itself. During the network's training process, the examples are assigned a *ground-truth* - an ideal expected result that has been manually classified - label $y \approx f^*(x)$ and the output layer has to return a value that minimises the error at every training point. Neither the behaviour (functions) nor the output of the layers in between is specified, implying that the network has to learn the utilisation of these layers to implement the best possible approximation.

When it comes to computing the values inside the hidden layers, a predefined non-linear *activation function* has to be chosen in order for neurons to perform linear regression or classification. On account of the activation function's non-linearity - since linear activation functions would not alter any data -, any continuous function can be theoretically approximated up to an arbitrary tolerance $\epsilon > 0$ by a neural network with one hidden layer and a sufficiently large number of neurons. One popular activation function is *Rectified Linear Unit* (ReLU), defined by $g(x) = max(0, x)$, where g denotes the function and x the input of the artificial neuron.

In order to measure the output quality, a *loss function* is used to penalise the deviation of the prediction from the target. Therefore, the goal is to minimise the result of this function. Often - depending on the loss -, the network parameters need to be adjusted. The amount and direction are specified by the network's gradients ∇f which are computed by the loss through the network - a process also known as *backpropagation*.

[2] *Feedforward artificial neural networks* will simply be referred to as *neural networks* from here on.

[3] The learning of its parameters is equivalent to evaluating f on the provided training data and comparing it to f^*.

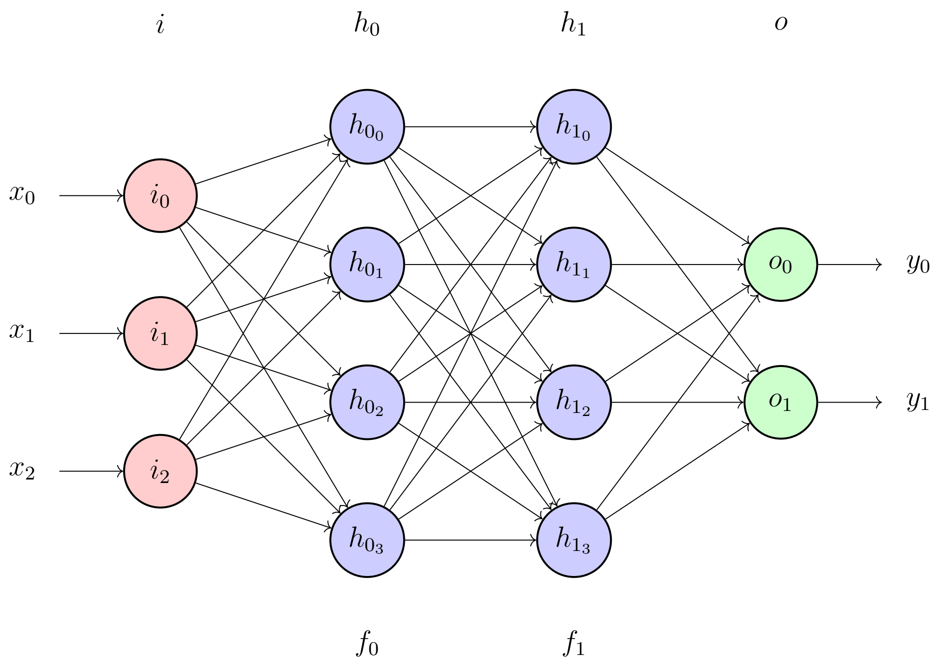


Figure 4.2: Artificial neural network architecture with three layers. The input layer is not counted. i, h_0, h_1 and o denote the input, first hidden, second hidden and output layer, respectively. The nodes are named after their layer joint with individual indices. x and y represent the input and output, while each f describes the hidden layer's function.

To roughly visualise an ANN's architecture, Figure 4.2 illustrates a simple feed-forward neural network consisting of three layers: The input layer (marked red) - which is not counted -, two hidden layers (marked blue) and the output layer (marked green). In this example, each circle serves as a node, which represents a neuron, while the arrows represent the connection from each neuron's output to the inputs of the consecutive neurons in the next layer. The input data x is fed into the network using the input layer i and the hidden layers h_0 and h_1 learn the parameters θ using functions f_0 and f_1, respectively. y denotes the final output - the product of the output layer o.

4.2.2 Long Short-Term Memory Network

Long short-term memory networks [HS97] are a special kind of recurrent neural network, which - unlike generic neural networks - considers not only the current input example, but takes the previously perceived information into account instead. It is dedicated to performing efficiently on sequences consisting of values of variable length, where the length represents the period of information flow. Under the assumption that learning specific parameters for an input feature is useful at one time step, the ability to recognise this feature is implied to be of equal use at other time steps, since it can be avoided to learn the parameters from anew.

In theory, any function that is computable by a Turing machine can also be approximated by a two-layer RNN, while in practice, however, they are not capable to learn long-term dependencies due to vanishing and exploding gradients [BJZP20] during backpropagation[4] [Hoc91]. LSTMs were developed to address this exact problem by Hochreiter in 1991 [Hoc91] and later proposed by him and Schmidhuber in 1997 [HS97]. LSTM units have mechanisms to control the flow of information through each unit, which includes deciding on whether to keep, alter or discard information. As a consequence of this design, LSTMs can handle vanishing gradients far better than regular RNNs. Figure 4.3 illustrates and describes an LSTM cell in more detail.

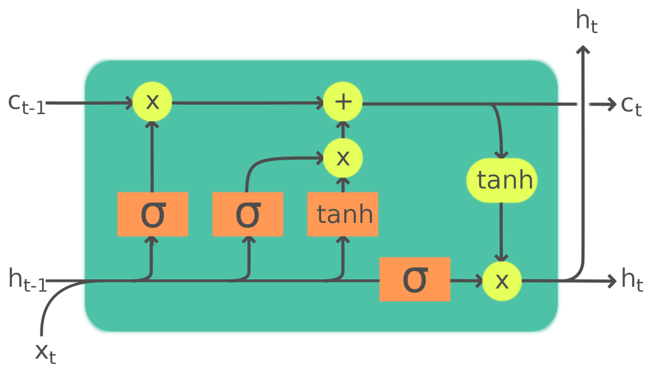


Figure 4.3: Illustration by Chevalier [Che18] of an LSTM cell C_t at the time step t - which replaces the hidden units of a traditional RNN - used to process data sequentially and retain long-term memory. Inside the unit, $+$ and $\times$ are point-wise operations whereas σ and $tanh$ denote *sigmoid* and *tanh* activation layers that act as gates controlling which information to keep, alter or discard. After it is decided which information can pass through, the former cell state C_{t-1} is updated. The current output is h_t and the previous output is h_{t-1}.

[4]A common problem when using gradient-based learning methods and backpropagation, is that the gradients might shrink to a point where the network's weights are effectively prevented from changing their values. At the same time, the opposite - infinitely increasing gradients - leads to loss of input information, as well.

Chapter 5

Static Approach

5.1 Overview

Starting with the initial input files in MIDI format, its data will first be transformed into custom data models for further analyses using third party libraries for MIDI processing. A chord progression is then identified by applying several different strategies that are also used by musicians for manually interpreting pieces, as well. For this process - the chord estimation -, it is crucial that the majority of the interpreted chords is correct. Since the chord progressions are the foundation of most pieces, its accuracy is essential to the succeeding interpretation of phrases.

After a successful chord estimation, the program will proceed with the residual element extraction - a process that uses both the progressions and any existing MIDI parts to save elements. This includes the extraction of independent phrases which can be seen as melodies, and phrases that serve as either pitched or unpitched accompaniments. To differentiate between these phrase types, a classification criteria has to be defined and implemented.

The purpose behind this design is the partially static reusability of the extracted elements for the generation of original pieces: Given any progression as context, the accompaniments' patterns - notes relative to each chord - can be applied to create an accompanying part inside a piece, while the chord progressions and melodies will be used to train the system's neural networks in order to generate original elements. The unpitched accompaniments will be saved and sampled for "new" drumlines[5]. Figure 5.1 illustrates the complete design.

[5]The resulting drumlines only have the potential to be original due to the fact that they consist of randomly combined patterns from the datasets' pieces.

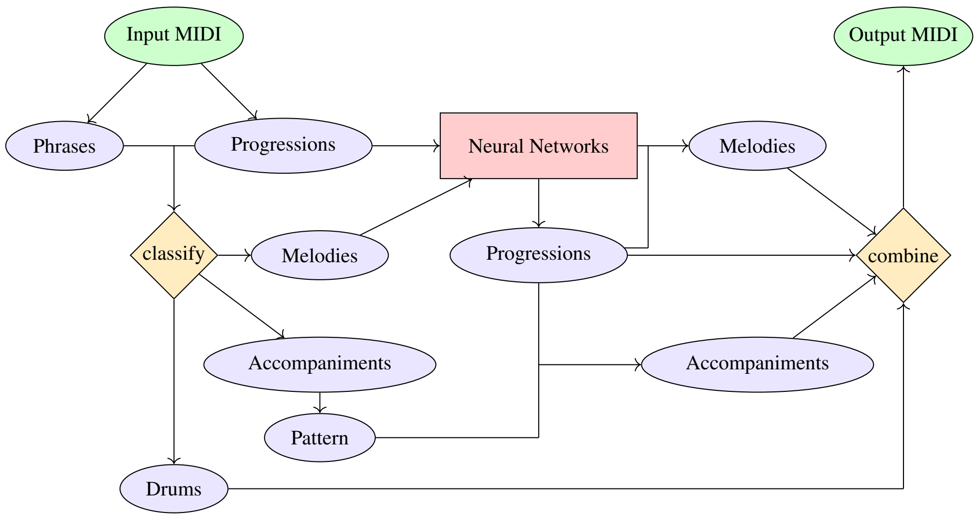


Figure 5.1: Rough design of the system's architecture. The subsystem containing the neural networks to produce the progressions and melodies has been simplified and will be described in detail in Chapter 6.

5.2 Difficulties

The main computational and implementational challenge is the accuracy maximisation of the implemented chord estimation algorithm due to the high heterogeneity of the dataset. Depending on the piece and its style, genre, composer, transcriber and/or arranger, a phrase's underlying chords - provided the section has any - can be indicated in numerous ways using notes only. Furthermore, if said accuracy turned out to be insufficient, the static extraction of other musical elements could be drastically hindered as a direct consequence. This would affect the input (and therefore output) of the neural networks, as well.

Additional potential difficulties could result from a poorly designed project architecture and implementation of certain data models.

5.3 Analysis

As mentioned before, the initial analysis consists of three main parts: MIDI processing, chord estimation and element extraction. As for the first part, a library has to be found that supports all necessities. At the same time, the criteria that a MIDI file has to meet for it to continue be analysed has to be specified to avoid unforeseen complications. The chord estimation method - which will be the system's most complex algorithm -, however, involves a much more intricate procedure as it will incorporate multiple strategies to determine a chord at a given time. Although each of these strategies has to produce effective results on its own, they will all be ultimately merged to find a value that is deemed the most probable given its context (preceding and ensuing data). Lastly, provided that the estimated chord progression is fairly accurate, the piece's phrases will be able to be investigated and subsequently classified, which eventually leads to the sample's following processing method.

5.3.1 Analysis Criteria

To limit this project's scope, the analysis will only be performed on MIDI files that fit the *General MIDI* (GM or GM1) standard - a standardised specification for MIDI instruments. The pieces' time signature must be 4/4 (as it is the most common one in modern music) as the neural network predicting the progressions will be solely trained for this time signature. Additionally, the MIDI tracks will have to contain at least one *Note On* and *Note Off* event each since tracks with control events only do not contribute any harmonic information[6].

Consequently, the generated compositions will not include time signatures other than 4/4 due to the neural network specifications and both tempo changes or dynamic changes will be excluded as well, since the project does not aim to improve these characteristics as stated in Chapter 1.

5.3.2 Chord Estimation

When it comes to chord estimation, various strategies have to be implemented, as previously mentioned. Each individually yielded result - which contains chord possibilities at a given time - is then evaluated and conclusively merged.

5.3.2.1 Estimation Strategies

The following chord estimation strategies have to be implemented in order for a successful consecutive merge of all possibilities. An example piece will be provided in Figure 5.2, which will be used to demonstrate each strategy's outcome.

All Possibilities The initial transformation of the MIDI data. It does not discard any parts, but attempts to fix[7] any potential faulty elements caused by MIDI processing errors. Since volume, tempo and orchestration are irrelevant for the

[6]Of course changes in a piece's key signature - which are *Controller* events - add context for the harmony, however, these are conventionally included in tracks containing the notes.

[7]In this context, *fixing* implies *differently interpreting* data during transformation *in a constrained manner*.

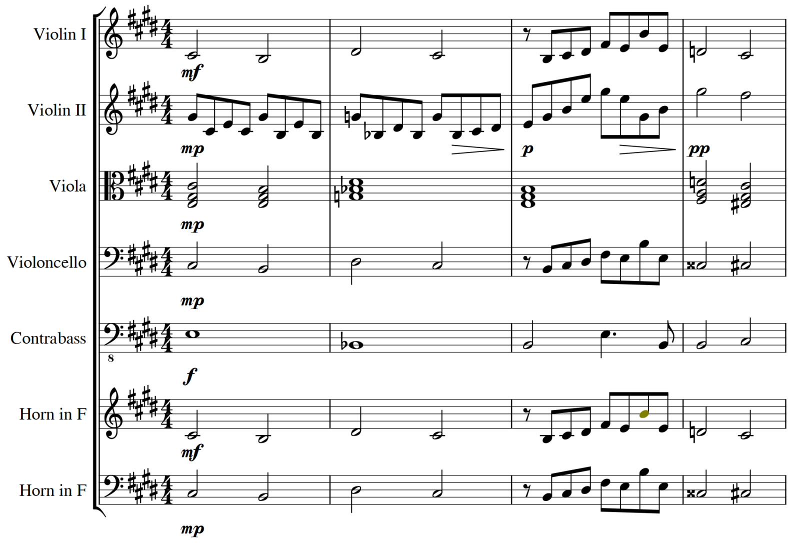


Figure 5.2: Original four measures of an arrangement based on *It's just a burning memory (2016)* by *The Caretaker*.

chord estimation, the respective information is excluded. This strategy can also be used to verify the authenticity of the processed data by comparing it to the original. Its output will be used by the other strategies. Given the example piece from Figure 5.2, ideally, the same MIDI (except for the volume, tempo and instruments) would simultaneously be this strategy's output.

Most Chords This strategy only accepts coherent sections of a MIDI track with the most amount of notes per chord in each section. Figure 5.3 shows the strategy's output MIDI with Figure 5.2 as input.

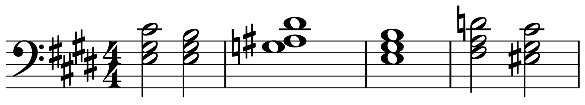


Figure 5.3: The output produced by the *Most Chords per Part* Strategy converted back into MIDI.

Potential Bassline Although not every piece has a bassline, one will be created, if possible, as seen in Figure 5.4 (resulting output of Figure 5.2). With a bassline, the likelihood of any chord possibility that contains the bassline's notes increases, since basslines are a type of accompaniment, which are usually built using a chord progression.

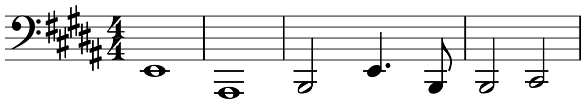


Figure 5.4: The output produced by the *Potential Bassline* Strategy converted back into MIDI.

Triad Recognition Since most chords' structures are built on top of *triads* - a set of three notes stacked vertically in *thirds* (intervals between notes of either three (minor third) or four (major third) semitones) -, this strategy will find these patterns (either in chords or consecutive notes) inside the MIDI tracks' phrases. In Figure 5.5, the triads recognised in Figure 5.2 by the strategy are shown.

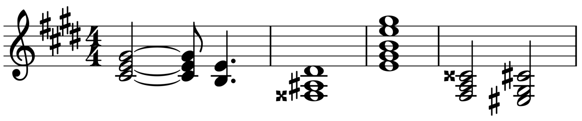


Figure 5.5: The output produced by the *Triad Recognition* Strategy converted back into MIDI.

Chordification "*Chordify* is a... word that we created in MUSIC21 for the process of making chords out of non-chords... [by] reducing a complex score with multiple parts to a succession of chords in one part that represents everything that is happening in the score." (Cuthbert et al., 2006 - 2021) [CAHO21]. By chordifying the score, its current total harmony (including notes outside the composer/arranger/transcriber's intended chord) at a given time can be calculated. The MUSIC21 library offers a method that *chordifies* the MIDI stream, with Figure 5.6 representing the result of the method's invocation on the given example piece from Figure 5.2.

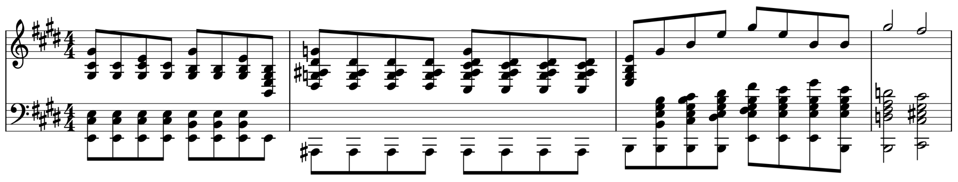


Figure 5.6: The output produced by the *Chordification* Strategy converted back into MIDI.

5.3.2.2 Merging Possibilities

Although merging all possibilities can be considered a strategy, one key difference does exist: It requires a successful application of all other strategies first. Assuming the requirement is fulfilled, this algorithm will iterate over the *Most Chords*, *Potential Bassline* and *Triad Recognition* strategies' outputs and find all notes' respective starts (time of the *Note On* event) to check whether any results for the current start are viable. If that is the case, the possibility that seems most plausible will be picked, else the original transformed information - output of the *All Possibility* strategy - will be used to create chords from a given section.

5.3.3 Element Extraction

Next to the chord progression, additional information has to be extracted, too. For this, the elements in question are the phrases within the MIDI tracks. There are three kind of phrases in total, including chord-independent phrases (melodies), chord-dependent phrases (pitched accompaniments) and percussive phrases (unpitched accompaniments).

Before proceeding with the classification of phrases, the detected progressions have to be validated. Using a checksum, the duration of a section - which can be calculated by the amount of included measures - will be compared to the progression's affiliated rhythm to eliminate the chance of mistimed elements, which would be fatal.

To classify a phrase, excerpts of the phrase in a given time window have to be compared to the chord from the estimated progression at the same time. If all notes from the phrase occur in their respective chord, the phrase will be classified as (pitched) accompaniment, else as a melody. Disregarding the phrase itself, if the part (MIDI track) containing the phrase has a percussive instrument mapped as its MIDI instrument, it will be considered an unpitched accompaniment. All information will be saved for further analysis.

5.4 Component Generation

Since the neural network will generate the progressions and melodies, any remaining elements based on them could be statically created and included in a piece afterwards. The static elements that will be generated in this project are counterpoint melodies, pitched accompaniments, unpitched accompaniments and key signature changes.

5.4.1 Counterpoint

In music, *Counterpoint* is the relationship between multiple simultaneous phrases which are harmonically interdependent yet independent in rhythm and melodic contour [Lai08]. It focuses on melodic interaction and is used to designate a phrase or an entire composition [Toc77; SD01].

There are several "species" of counterpoint with each increasing the complexity. For this study, the first (and most basic) species of counterpoint - known as *1:1 Counterpoint* - will be implemented, since its applications have lasted from medieval times to now, proving its importance. Additionally, many classical composers, such as Bach, Schubert, Salieri, Haydn, Mozart and Beethoven, were not just taught this exact composing method, but also made use of species counterpoint [Wil20].

In Figure 5.7, a well known counterpoint application is shown: The blue notes represent a *Cantus Firmus* (simple, pre-existing melody that forms the basis of a polyphonic composition) that has originally been written by Salieri as a counterpoint exercise, while the purple ones form the melody that his student Schubert came up with as a solution.

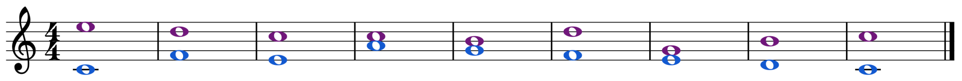


Figure 5.7: *Cantus Firmus* (blue) written by *Antonio Salieri (1750 - 1825)* [CF20] and *Counterpoint Melody* (purple) applied by *Franz Schubert (1797 - 1828).*

Based on Schubert's style, a static algorithm will be implemented to create counterpoint melodies for existing melodies.

5.4.2 Pitched Accompaniment

When saving the accompaniment information at the end of the element extraction, the pitch (MIDI data value from 0 - 127) itself will not be saved, but its relation to the chord's triad instead. Inside a triad, the default three notes - which are the *root*, *third* (*major* and *minor*, with a distance to the root consisting of three and four semitones) and fifth (*diminished*, *perfect* and *augmented*, with a distance to the root consisting of six, seven and eight semitones) - will be classified by mapping $0 \mapsto r$, $3 \mapsto t$, $4 \mapsto t$, $6 \mapsto f$, $7 \mapsto f$ and $8 \mapsto f$. Thus, a function $m : X \rightarrow Y,\ x \mapsto y$ can be defined with the domain $X = \{0, 3, 4, 6, 7, 8\}$ with $x \in X$ and the codomain $Y = \{r, t, f\}$ with $y \in Y$. Since the patterns' notes have a relation to each other to differentiate between their octaves, the octave o, in which the note $n \in \{0, ..., 127\}$ appears, will be calculated using $o = \lfloor \frac{n}{12} \rfloor$, too.

Lastly, to account for the phrase's transposition and its relation to the current chord, the key signature's relative pitch k (C$\flat \mapsto 11$, C$\natural \mapsto 0$, C$\sharp \mapsto 1$, D$\flat \mapsto 1$, D$\natural \mapsto 2$, D$\sharp \mapsto 3$, etc.) and the chord's root pitch c have to be subtracted from the note. Provided that each pitch lies within a chord's triad (which is the prerequisite that has to be fulfilled by the phrase classification), this leaves us with a final mapping $(y, o) = a(n, k, c)$, as seen in Equation 5.1.

$$a(n, k, c) = (m((n - c) \bmod 12), \left\lfloor \frac{n - k - c}{12} \right\rfloor) = (y, o) \tag{5.1}$$

5.4.3 Unpitched Accompaniment

Since the unpitched accompaniments (drums) are created by randomly picking a sample of n patterns, the patterns can then simply be converted back into MIDI without any further modification.

5.4.4 Key Signature Change

Upcoming key signature changes are usually hinted by the chord progression, although that is often not the case. Since the neural network's progressions will be generated knowing only whether the progression should resolve, meaningfully indicating any imminent key signature changes will not be possible. Instead, these key changes should only occur if it is manually configured by the user.

5.5 Drawbacks

Using this design, one major drawback appears to be the potential bottleneck that is being created using the chord estimation algorithm. All subsequent algorithms depend on a successful, accurate chord estimation. Although a non-perfect result is expected and accounted for[8] - if the recognised progressions were exceedingly divergent from the actual (unspecified) progressions[9], those inaccuracies could result in repeated occurrences of false phrase classifications, which would then imply a poor element extraction and succeeding component generation. The neural networks' precision will be affected by inferior data, as well.

[8]A perfect result is not possible due to the non-deterministic nature of chord interpretations.
[9]This has to be manually assessed by the author.

Chapter 6

Data-Driven Approach

6.1 Overview

As previously pointed out, the usage of LSTMs is an established approach in the context of automated music generation due to the architecture's ability to process sequential data and its high performance [MMJ19; LWZM15; HHC18]. Therefore, it will be implemented for the prediction of all musical elements. The system itself will consist of two subsystems - one for predicting chord progressions[10], the other one for predicting phrases[11].

The progression network is the first network that needs to be implemented, since the phrases will be based on the progressions. To maximise its precision, a progression of n chords will require $n-1$ different models, of which each will specialise on predicting on a sequence of $i \in \{1, ..., n-1\}$ preceding chords. The first chord, which will be required as the input of the first model, will be randomly chosen from a set of predefined chords. Any subsequent inputs will be the prior model's output. This subsystem will predict tuples (r, t), which represent a chord's root note and chord type.

As for the phrase network, only three models are needed, since creating $n-1$ new models for specific phrase lengths would not be pragmatic due to their sizes[12]. Additionally, an average phrase contains more notes than the average amount of chords in an arbitrary progression, thus implying that a higher sequence length is required. Therefore, the models will specialise on predicting a note's offset o, pitch p and duration d, respectively, and produce single-value outputs with

[10]This subsystem will be simply referred to as *progression network*.

[11]This subsystem will be simply referred to as either *phrase network* or *melody network*.

[12]The total information stored in all phrases far exceeds the progressions' information in terms of size. This implies a distinctly higher computation time.

input sequences of a fixed length m. A random excerpt from all phrases, whose respective chord progression's last chord matches the generated progression's first chord, will be chosen as the initial input. The order of each model's invocation will be essential for this subsystem to work properly, since they are dependent on each other's information. The pitch itself depends on the note's time, since the timing specifies a note's function (e.g. whether a *non-chord tone* - pitch outside the current chord - shall be used) according to music composing principles. Only then - when a note's pitch is known -, a sensible duration value can be assigned. Summarised, an offset to the previous note has to be determined first, and by using its value, a pitch can be predicted, followed by the note's duration. Looping this procedure (until it covers the entire progression), a phrase can be generated note by note, invoking three consecutive predictions at a time.

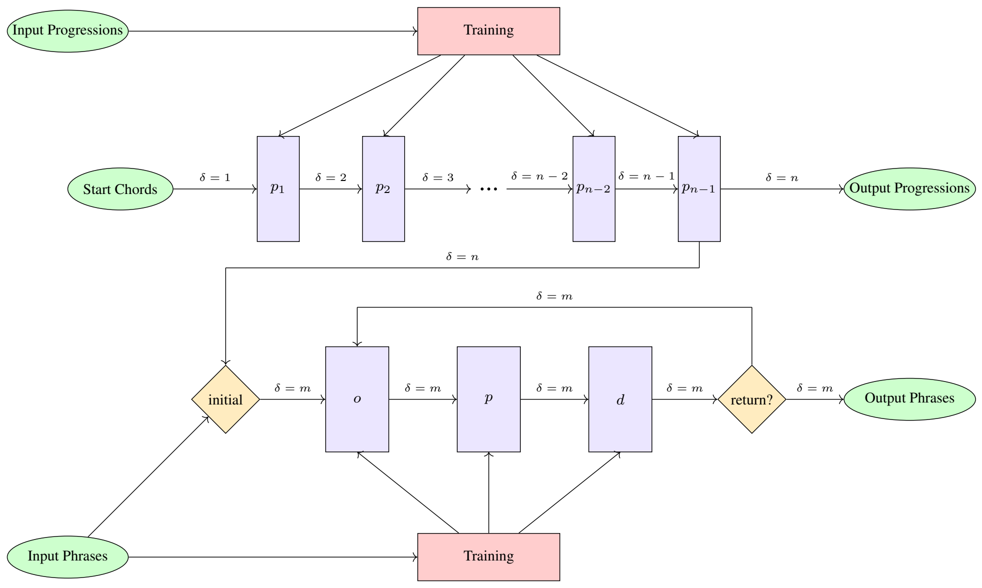


Figure 6.1: Rough design of the progression network and phrase network's individual architectures and their interrelation inside the data-driven subsystem. The initial inputs (progressions and phrases) originate from the static analyses. δ denotes each sequence's length, n the progression's maximum sequence length and m the phrase's fixed sequence length. p_1 to p_{n-1} represent the $n-1$ progression network's subnetworks, o, p and d represent the phrase network's subnetworks responsible for predicting a note's offset, pitch and duration.

6.2 Difficulties

When working with any kind of artificial neural network, the configuration for the training process, as well as the preceding data preparation is a major challenge for the user. Even though every configured compiling network will produce results eventually, the produced data will often not be usable, due to a lack of understanding the network's internal layers' functionality, the misjudged relevance of any kind of specific information from the dataset, incorrectly formatted (transposed) input and/or output matrices, falsely interpreted output matrices, incompatible data formats, faulty data inside the the training data, false or insufficient classifications or poorly fitting[13] the network during training. All of these issues must be regarded during implementation.

Since this project aims to make use of music theory principles, every single network has to implement individual constraints while evaluating its prediction matrices. Although raw, unaltered results could technically be used, there is no limit on the quantity of constraints that can be implemented either. Finding an optimal amount will be an additional difficulty.

6.3 Training Data

Since ANNs require intensive training, a dataset comprising independent music information with ground-truth labels is a necessary requirement for this phase. Considering the availability and accessibility, MIDI will be chosen as the system's initial input format. On the other hand, the final network input will be part of the generated artefacts that are created by the system following the analysis of the provided dataset's MIDI data and converted into a numeric format that is readable by the neural network.

To ensure the highest possible heterogeneity of the input data, a new dataset of over 100 MIDI files will be put together for this project. Most of the files are transcriptions or arrangements of various popular pieces from different genres throughout the past century, although the dataset contains original classical pieces - that have been converted to MIDI -, as well. As for the origin of these files - approximately half of files have been contributed from various users of the sheet music sharing site Musescore.com [BV22], which uses a Creative Commons license [Com22], while the other half consists of transcriptions/arrangements produced by the author.

[13]*Fitting* means adjusting the network's performance to work well on both data it was trained on and data it was not trained on. Two common phenomenons are *overfitting* and *underfitting*.

6.4 Correlations

Word2vec [Ron14] is a natural language processing technique that - as the name implies - maps words onto vectors to detect any semantic similarity by using multiple neural network models for word associations from a text corpus. Additionally to the main machine learning systems, Word2vec will be incorporated as an alternate element substitution method. Making use of these semantic similarities, any musical element such as a chord is able to be substituted in a similar manner as synonyms in a sentence, provided the musical context has been converted into a consistent textual format.

6.5 Component Generation

As stated earlier, all musical elements generated using this approach will be predicted by the data-driven subsystem. The progression network generates a chord progression by predicting chords, while the phrase network generates a phrase by predicting notes.

6.5.1 Chord Prediction

The concept of chords is one of the most fundamental ones, meaning that chords will become one of the most important data models of this project. The model's two most important fields are its root (integer) and its chord type (enum), of which the latter's type is incomprehensible for a neural network[14]. To circumvent this issue, a new notation will be introduced. Similar to *Roman numerals*[15], the chord itself can be represented using numbers relative to the scale only (which reduces transpositions to simply shifting the notes by the key's value). In this case, a tuple (r, t), where r denotes not the scale's degree, but the semitone interval to the scale's value instead, while t refers to a predefined chord type (e.g. major $\mapsto 0$, minor $\mapsto 1$, diminished $\mapsto 2$, etc, as defined in Appendix A). For instance, in the key of *C*, the chord progression F-G^7-Em^7-Am is denoted as IV-V^7-iii^7-iv using Roman numerals, while a tuple of tuples $((5,0),(7,5),(4,4),(9,1))$ represents the same using the introduced notation. Converted into this notation, the chord progressions (lists of these tuples) will be fed into the networks, while - with each

[14]Neural networks usually only work with numbers.

[15]One widely used chord representation consists of (primarily) Roman numerals (e.g. I, ii, iii, IV, ...). Most commonly, they denote the chord whose root note is also the current scale's degree (e.g. IV denotes a chord whose root note is the fourth note in the scale). Using this notation, uppercase Roman numerals represent a major chord, while lowercase Roman numerals represent minor ones.

iteration -, a chord of a predefined duration will be predicted as a tuple to generate a new rhythmic (due to the additional duration) chord progression.

The LSTM networks' outputs will consist of three-dimensional matrices sized $x_i \times 2 \times c$ (with x being the total amount of provided progression samples for the sequence length $i \in \{1, ..., n-1\}$ and c being the amount of different classes, which is equal to $max\{p, 12\}$ with p representing the amount of predefined ChordType enums, while 12 is derived from the amount of possible root pitches in a chromatic scale), that represents the total probabilities of each class being predicted. Constraints inside the prediction can be differentiated into hard constraints and soft constraints. While the hard constraints directly prohibits or determines a prediction, a soft constraint will only alter the probabilities by a given factor $\lambda > 0$. The input matrices size, however, will be $x_i \times i \times 3$. To take falsely identified chords (which have a higher probability of having a short duration[16]) into account while training, additional context will be added to the input tuples in form of a third value that represents the chord's duration d, making them triplets (r, t, d).

6.5.2 Note Prediction

Similar to chord progressions, each phrase will also be transformed into a readable format for the networks. Each note will be represented as a triplet (o, p, d) (offset, pitch and duration), meaning that phrases - which contain lists of notes - will be represented as sets consisting of tuples.

In this subsystem, all three networks will each predict a value contributing to the triplet. The network outputs matrices will be sized $y \times c_i$ (with y being the total amount of provided phrase samples and c_i being the amount of different classes per network $i \in \{0, 1, 2\}$). The input matrices will include additional contextual information, too, as the progression networks' inputs did. Their sizes will be $y \times m \times a_i$, where a_i represents the respective network's amount of parameters (size of n-tuple). Using Δ to symbolise the current prediction[17] - when it comes to the offset network, its input's septuple will consist of the previous note's offset $o_{\Delta-1}$, pitch $p_{\Delta-1}$ and duration $d_{\Delta-1}$, the previous chord $(r_{\Delta-1}, t_{\Delta-1})$ and the current chord (r_Δ, t_Δ), flattened to $(o_{\Delta-1}, p_{\Delta-1}, d_{\Delta-1}, r_{\Delta-1}, t_{\Delta-1}, r_\Delta, t_\Delta)$. Its output will be the current note's offset o_Δ. Using this output, the pitch network will take in a quadruple $(o_\Delta, p_{\Delta-1}, r_\Delta, t_\Delta)$ to compute the current pitch p_Δ. Lastly, the duration network will use the preceding information to calculate d_Δ - the current note's duration. Its input value is the quintuple $(o_\Delta, p_\Delta, p_{\Delta-1}, r_\Delta, t_\Delta)$. After all three models have predicted a value, the triplet is appended to the phrase. With

[16]Most chords of shorter duration are the consequence of the chordification process, which will only be used if the preceding strategy merges did not yield anything of value.

[17]This implies that Δ is incremented with each iteration.

the iteration completed, Δ is incremented and the network input is adjusted - after appending the triplet to the input, the first item has to be removed in order to retain the fixed sequence length of m items.

6.6 Drawbacks

Depending on the quantity of the input data, the amount of time required for training the networks may rapidly surge. Considering the project's experimental nature, countless (yet final) configurations of the network have to be tested (which implies training the network) in an attempt to optimise the results. Given the thesis's time frame and the project's scope, only a fraction of all possible configurations can be tested, implying that the optimisation attempt has to be halted while still in its early stages.

Chapter 7

Implementation

Prior to the implementation, a programming language has to be chosen. For this project, PYTHON 3.8 [Fou22], an interpreted, high-level, general-purpose programming language, will be used, as its object-orientation approach and garbage collection, together with its vast area of application, fulfil the minimum requirements of a project of this kind. Although comparably slower than languages closer to the operating system (e.g. C [KR88]), internally, the language itself, as well as a significant amount of popular libraries, are implemented in C to ensure the highest possible performance for their respective tasks. Most importantly, however, Python is especially heavily used in scientific fields due to its wide range of available libraries (for both standardised and specific purposes), such as NUMPY, PANDAS or SCIKIT-LEARN, just to name a few. Especially MIDI handling and ANN interfaces have been taken into account, for which multiple options are offered as Python libraries.

7.1 MIDI Handling

MUSIC21 [CAHO21] is a Python library for computer-aided musicology, developed by Cuthbert et al. and licensed under the *BSD license* [Ini22]. It is employed first in the static analysis to transform a stream of MIDI data into custom data models, which will then be used to determine a chord progression. Afterwards, only its MUSIC21.KEY.KEY class is used to compute relative keys.

While handling the MIDI data, a MIDIDATA (see Appendix B) object is created for each MIDI file. Before being passed on to the following analyses, the analysis criteria are verified immediately once the constructor has extracted the file's most important meta data.

7.2 Data Models

As mentioned before, specific classes have been implemented as an attempt to model all relevant kinds of data in order to capitalise on the context that is provided by each model. They are grouped into three categories: "analysis", "basic" and "composition", with a respective module each. The ANALYSIS module contains models useful to the analysis only. Its models do not have any additional application. BASIC consists of all most fundamental musical elements, which have all been previously mentioned. These models hold the highest importance since they provide the neural networks' data with context and thus they are tested using unit tests, too. Lastly, COMPOSITION holds models which are used to create a piece. Most models (from all three modules) have additional functions (logic), which are used ubiquitously, beside just accessing their fields.

```
model/
|-- analysis/
|    |-- chord_accompaniment.py
|    |-- chord_possibilty.py
|
|-- basic/
|    |-- chord.py
|    |-- chord_progression.py
|    |-- key.py
|    |-- resolution.py
|    |-- rhythm.py
|    |-- rhythmic_chord_progression.py
|    |-- scale.py
|
|-- composition/
|    |-- drums.py
|    |-- instrument.py
|    |-- part.py
|    |-- phrase.py
|    |-- piece.py
|    |-- section.py
|
```

Figure 7.1: SRC.MODEL module's structure. Each file's name represents its (main) content. The ANALYSIS module's content is used during analysis only and does not represent any elements directly derived from music theory, contrary to BASIC and COMPOSITION.

Briefly summarised, a PIECE contains SECTION objects[18], which consists of PART objects. They on the other hands hold a PHRASE and an INSTRUMENT. In order to stay within a thesis's scope, none of the data models will be described any further. A list of all implemented models can be found in Figure 7.1.

[18]Sections are essentially excerpts of a piece in a musical context, e.g. a chorus or a verse.

7.3 Information Extraction

7.3.1 Chord Estimation Algorithm

Since multiple different strategies are used to find a chord progression, the *strategy pattern* (or *policy pattern*)[19] can be implemented for this algorithm's architecture. Additional to the strategies previously mentioned in Chapter 5.3.2.1 (*All Possibilities*, *Most Chords*, *Potential Bassline*, *Triad Recognition*, *Chordification*), the drum (unpitched accompaniment) extraction will be implemented as a strategy, as well. Each strategy has additional strategy-specific functions next to the overridden parent class ESTIMATIONSTRATEGY's abstract functions and all except for DRUMEXTRACTIONSTRATEGY return a list of CHORDPOSSIBILITY - a data model from the ANALYSIS module - objects. MOSTCHORDSSTRATEGY, BASSLINESTRATEGY and TRIADSTRATEGY invoke the inherited, protected function _GET_COVERED_POSSIBILITIES_(SELF, ALL_POSSIBILITIES: LIST, STRATEGY_FUNCTION, RELATE, CONSTRAINT) using self-evaluating CHORDPOSSIBILITY functions and operators in order to fit the best possible CHORDPOSSIBILITY objects into their list, which will eventually be returned. MIDI results of each strategy that can be verified are generated with the parent class's function CREATE_MIDI(SELF, COVERED: LIST, NAME: STR, CHANNEL=0). For a full UML diagram, see Appendix B.

Once run through the algorithm, a RHYTMICCHORDPROGRESSION object, which consists of a RHYTHM and CHORDPROGRESSION (a list of CHORD objects with a corresponding KEY), is created and returned. Its numerical notation, which is readable by the neural network, is saved into a CSV file.

7.3.2 Phrase Classification

Making use of Python's dictionaries[20], both CHORDPOSSIBILITY and CHORDACCOMPANIMENT save their respective notes with this data structure: The key marks the time in which a note appears, while the value contains either its pitch or duration. Furthermore, CHORDACCOMPANIMENT contains a field in which the current chord is saved. The reason for that is its creation process: After a rhythmic (timed) chord progression has been estimated, each phrase will be divided at each chord change, leaving only excerpts of phrases for each associated chord. Using

[19]When implementing a *strategy pattern* as a system's behavioural software design pattern, the algorithm associated with each strategy is selected during runtime.

[20]A dictionary is the Python equivalent of a hash table.

this method, the pitches are easily compared to the chord's *triad pitches*[21] in order to classify a phrase as either melody or accompaniment. As for the unpitched accompaniments, all percussion instruments are exclusively mapped on MIDI channel 10, implying that filters can be utilised during the MIDI file-to-stream transformation.

7.3.3 Additional Element Extraction

Using the chord progression and its classification, a phrase can be extracted and saved. When it comes to melodies, quintuples (similar to the notation in Chapter 6.5.2) consisting of $(o_\Delta, \{p_{1_\Delta}, ..., p_{n_\Delta}\}, d_\Delta, (r_\Delta, t_\Delta), (r_{\Delta+1}, r_{\Delta+1}))$ with Δ denoting the current index, and p_1 to p_n representing all current pitches (n is mostly equal to 1), will be saved in a list for each part's phrase. Accompaniments will be saved as lists of tuples, whose values are calculated using $a(n,k,c) = (m((n-c) \mod 12), \lfloor \frac{n-k-c}{12} \rfloor) = (y, o)$ as described in Chapter 5.4.2, while each drumline is saved as a dictionary - which technically is a surjective map -

$$\begin{aligned} d : &\{\Delta_1, ..., \Delta_i\} \to \{\{p_{1_{\Delta_1}}, ..., p_{n_{\Delta_1}}\}, ..., \{p_{1_{\Delta_i}}, ..., p_{n_{\Delta_i}}\}\}, \\ &\Delta \mapsto \{p_{1_\Delta}, ..., p_{n_\Delta}\} \end{aligned} \tag{7.1}$$

where each key Δ represents the time and p_1 to p_n denote the current pitches at time Δ. All elements are stored in CSV files.

7.4 Element Prediction

Keras [Cho22] is an open-source Python library that provides an interface for ANNs such as the *TensorFlow* [AAB+15] library and implements common building blocks used in neural networks. Due to its user-friendliness, modularity and extensibility, the API will be utilised to implement the neural networks essential to this project.

7.4.1 Data Preparation

To start things off, the input and output data has to be computed. By iterating over the previously extracted information, the input n-tuples and their corresponding output n-tuples have to be mapped onto each other and all matrices have to be transformed into the desired dimensions. In addition, the input is normalised

[21]Generally speaking, this refers to the first three pitches in a chord. Since all implemented chords are triads or extended chords built on triads, this property (triad pitches) can always be accessed.

in order to obtain a mean close to 0, while the output is *one-hot encoded*[22] for categorisation.

7.4.2 Network Composition

Since the progression network, as well as the phrase network, predict on sequences, their respective compositions resemble each other immensely. Thus, a generic neural network is created that enables small variations by using a control parameter. Figure 7.2 illustrates both networks side by side.

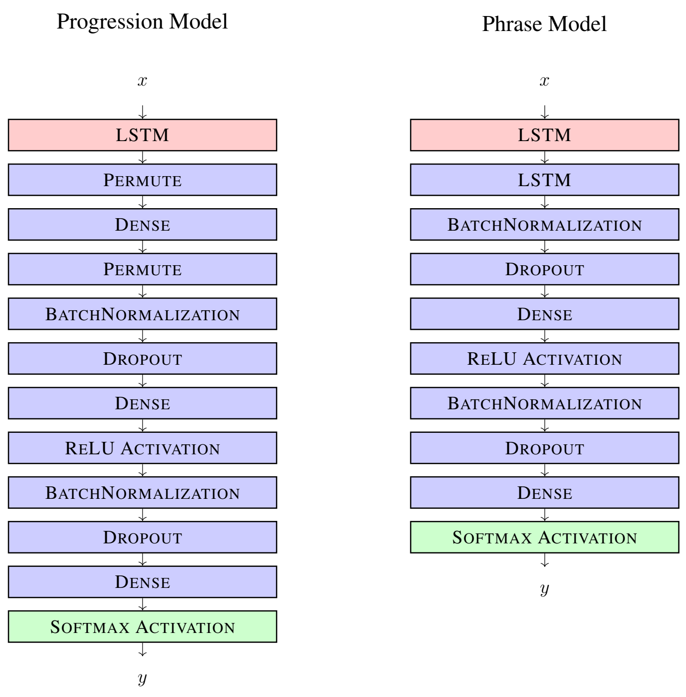


Figure 7.2: The respective model of the progression network and phrase network. The initial layer used for the input is marked red, while hidden layers are marked blue. Green represents each network's final, output layer.

As input, both implement LSTM layers that return the full sequence of hidden states $\{h_0, ..., h_n\}$, as their utilisation has been established previously. Due to the matrices of each network being of a different size, the progression network's matrix has to be reshaped with two PERMUTE-layers and an additional DENSE

[22] An one-hot is a group of bits consisting of only one single high (1) bit, while the rest remains low (0).

layer (a deeply connected neural network layer) in order to fit the successive layer's input, while a second LSTM layer returning the hidden state h_n at the final time step is required for the phrase model in order to match the progression model's return type. BATCHNORMALIZATION is then applied for both to keep the mean output close to 0 and the output standard deviation close to 1[23], followed by a DROPOUT (layer which randomly sets input units to 0 with a specified frequency) and DENSE layer. Purpose of the dropout application is to avoid overfitting, while the dense connection is used with a parameter that defines a particular amount of interconnected nodes. After a RELU ACTIVATION layer, the BATCHNORMALIZATION-DROPOUT-DENSE-pattern is applied once more with the same purpose as before. Finally, a SOFTMAX ACTIVATION layer - a separate layer that implements an activation function which, in this case, is defined by the *softmax function* (normalised exponential function) σ - is used to convert the network's internal vector z - which consists of K arbitrary numbers at this time - into a vector that represents a probability distribution, where each value lies in the interval $(0, 1)$ and the total sum of all values is 1. The standard softmax function $\sigma : \mathbb{R}^K \rightarrow (0, 1)^K$ is defined as

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{7.2}$$

with $i \in \{1, ..., K\}$ and $z = (z_i, ..., z_K) \in \mathbb{R}^K$.

When compiling the model, a loss function (function to calculate the loss through the network) has to be defined. For this, *categorical crossentropy* - which is defined in 7.3 ($\hat{y}_i$ denotes the the i-th scalar value in the model output, y_i the target value, and n the number of scalar values) - is chosen, due to its application in multi-class classification tasks. As for the optimisation algorithm - *RMSprop*[24] is selected.

$$loss = \sum_{n=1}^{n} y_i \times log(\hat{y}_i) \tag{7.3}$$

To measure both the loss and accuracy, a portion of the dataset is set aside to validate the performance of the model. This validation set is then used to compute those two metrics anew. Using this method, each model will have four metrics (loss, accuracy, validation loss and validation accuracy) that describe its performance.

[23]Since neural networks require their values to be normalised, using batch normalisation is common practice.

[24]The *RMSprop* algorithm is a popular adaptive optimisation algorithm designed for neural networks introduced by Hinton in 2012 [HSS12].

7.4.3 Training Configuration

To allow for a more individually configurable network, the training process is parameterised through a configuration file (described in detail in Chapter 7.6.3). For each network type (progression and phrase), a set of five parameters is specified: Firstly, TRAIN declares, whether a previously trained (implying that the model's weights have been saved) network has to be retrained from scratch or whether training shall continue. Allowed values can be picked from the set {"NEITHER", "RETRAIN", "CONTINUE"}. Additionally, the progression network's maximum sequence length n and the phrase network's fixed sequence length m (as described in Chapter 6) can be specified using the network's respective SEQUENCE_LENGTH parameter, as well as the amount of epochs (training iterations over the provided training set) using EPOCHS (both require integer values). The batch size, which is the number of samples that will be simultaneously propagated through the network until every sample has been propagated, is configurable using BATCH_SIZE, too. Ultimately, the user can specify the SAVE_FREQUENCY, which determines how often the *weights* (a type of learned parameter) of a model shall be saved, in epochs.

When it comes to training (TRAIN), "NEITHER" is set as the default value, as the training process is rather time consuming. The progression network's SEQUENCE_LENGTH is set to 16 as default, since most progressions consist of a lesser amount of chords, while the phrase's default value is 64, which would be an estimated average amount of notes in a phrase that covers a 16-chord-long chord progression. Considering the time factor again, the progression and phrase networks' respective EPOCHS value is set to 40 and 10 (the latter network is slower due to its larger size). Since both networks process data which - with the provided dataset - consist of hundreds to thousands of samples, both BATCH_SIZE default values are set to 64 - a relatively small value that guarantees a faster training session and requires less memory in exchange for less accurate estimates of the network's loss gradient. Lastly, due to insufficient physical storage, a frequency of 5 and 2 are specified respectively for the progression and melody network.

7.4.4 Chord Prediction

Since the last layer implements a softmax activation function, the final, predicted $1 \times 2 \times c$ ($c = max(p, 12)$ where p represents the CHORDTYPE enum amount) sized matrix will contain each class's probability. The first row - which represents the root note r - possesses only 12 valid values, as there are only 12 notes in a chromatic scale that spans over an interval of an octave. In the second row, the probability of a chord type t is computed. Put together, a chord tuple (r, t) results. Since the correlation of the two vectors is not given, constraints will be applied to estimate a chord that succeeds the progression, while each respective duration is determined using a set of predefined chord lengths. Using the total statistics - converted into relative probabilities P - of all chords in the provided dataset, a new, relative probability is calculated by multiplying each permutation's product $P(r_c) \times P(t_c)$ by $p_{rt} \in P$. At the same time, a *debuff table* - which alters a chord's probability based on the preceding progression - is created as a soft constraint, while hard constraints are set by prohibiting certain chords in order to circumvent too many repeating chords. Furthermore, real-time conditional, hard-coded chord substitutions are randomly utilised, as well as semantic substitutions using Word2vec after a progression is created. The initial chord essential to commence the progression generation is randomly picked from the set $\{(0,0),(5,0),(7,0),(9,1)\}$, which is equivalent to $\{\text{I}, \text{IV}, \text{V}, \text{vi}\}$ in Roman numerals.

7.4.5 Note Prediction

To generate a triplet (o, p, d), the respective networks' outputs have to be evaluated successively. Since the final prediction outputs' matrices are all sized $1 \times c_i$ (with c_i equalling the amount of different classes per network), the prediction itself is an one-dimensional list consisting of probabilities calculated by the output layer using the softmax activation function. The highest result will be picked without any alterations in the probabilities, unlike the method used by the progression network. Therefore, heavy constraints are applied on top of the prediction result. In an attempt to make the phrase *coherent*, all three values are potentially adjusted. Pitches are linked and corrected based on the previous and the current note and chord. Contrary to the progression network, an original phrase cannot be created using an initial, singular triplet, as this would require $(m-1) \times 3$ networks in total for a sequence length of m. Due to that fact, an input phrase of the fixed length m is randomly picked from the network input, whose last note's affiliated chord matches the current chord of the generated chord progression. Since the phrase is not expected to be completed due to it being chosen arbitrarily, the triplet's values are altered at random, as well. By adding uncertainty, originality of a piece results. To balance out the induced unreliability of the note, the prediction results require theory based adjustments in form of additional constraints. The methods applied are based on music principles (utilising the context of the key, chords, progression, borrowed and actual scale, rhythm and resolution) and will not be elaborated any further, as they exceed the scope of the thesis.

7.5 Static Element Generation

As stated in Chapter 5.1, the reason for transforming and analysing the initial data this thoroughly is the musical context, that is not just fed into the networks, but also later generated with each element predicted by the neural networks. By creating a foundation consisting of chord progressions and melodies using machine learning, this allows for statically computable components afterwards, too.

7.5.1 Counterpoint

Without going into detail about the principles used by Schubert on 1:1 Counterpoint, summarised, harmonic intervals in phrases, classified as *consonance* and *dissonance* are used to generate a new phrase on top of an existing one. A semitone interval is calculated by the counterpoint melody's relationship to the original one by making use of the current (actual or borrowed) degree of the scale. In addition, the algorithm differentiates between *perfect* (1st, 5th and 8th degree) and *imperfect* (3rd, 6th and 10th degree) consonances. It is randomly chosen whether a counterpoint melody lays above or below (in terms of pitch) the original phrase. The only constraint is, that it has to fit the range of the instrument that the phrase is assigned to.

7.5.2 Pitched Accompaniments

Since the pitched accompaniments' pitches are saved as tuples (y, o), these can be converted back into actual pitches when a chord is given. For example, $(t, 5)$ would represent the chord's *third* (essentially equal to the second note of a triad) in the fifth octave. Given an *A major* chord, its triad's pitches are calculated by adding a root $r_A = 9$ to each pitch specified in the associated CHORDTYPE enum, resulting in `[P +` r_A `FOR P IN [0, 4, 7]]` $\mapsto$ `[9, 13, 16]`. Now, by picking the chord's third's integer value (the second value of the triad pitches) 13, the amount of octaves o (multiplied by 12) has to be added, which leaves the final result at 73. Using a pattern consisting of these tuples together with a duration and a start, entire accompaniment phrases can be computed.

7.5.3 Unpitched Accompaniments

Unpitched accompaniments' notes have been directly saved inside patterns using their pitch's MIDI value, implying that these values can be used once again. When creating a piece, the previously saved patterns are then sampled (either randomly or manually) to produce a drumline.

7.5.4 Key Signature Change

As mentioned, key signature changes are manually added inside a piece. Although a key is randomly chosen for a piece, the key itself can be picked anew or a key signature change event can be added by providing a transposition amount (in semitones). Internally, all phrases inside a piece are saved in the key of *C*, while a section possesses a field TRANSPOSE_AMOUNT, which is allowed to be altered by the user and accessed during the MIDI creation to transpose each phrase.

7.6 Miscellanea

7.6.1 Path Management

All relative paths concerning the location of files are specified in the top level package and therefore instantly loaded. The project's root's path is then attached in front of each relative path to guarantee the usages of full paths for file accesses.

7.6.2 User Interface

The user interface is designed as user-friendly as possible. Since the given time did not allow for a graphical interface, a textual one is implemented, in which default PRINT() invocations are replaced by calling a custom method. This implementation allows for extensibility during printing, as well as improved console outputs. Considering the time required for both the analysis and training process (which could potentially take multiple days depending on the dataset and configuration), each output includes a timestamp. To improve readability, the verbosity is configurable and all prints are colour coded. The user inputs (choices and text) are handled using the same methods, as well.

7.6.3 Model Weights

When it comes to saving "training checkpoints", the frequency is specified in epochs and the model's weights are saved as HDF5 (Hierarchical Data Format version 5) files, which are later imported. Additional information, such as the weight's loss, accuracy or amount of undergone epochs, is stored in the file's name. While loading, each file is converted into a WEIGHTDATA object, which allows for specific filtering. Depending on the configuration, the weight used for the network is loaded from the file that contains either the best out of all, latest out of all or best out of the most recent available (with the network compatible) weights. The score s used to determine the best weight[25] is calculated as seen in Equation 7.4, where a denotes the training accuracy, l the training loss, a_{val} the validation accuracy and l_{val} the validation loss. This formula emphasises a high accuracy and a low loss,

[25]Since none of the original four metrics has a higher importance compared to the rest, all values have to be taken into account.

while moderately minimising the difference between the validation values and the training values.

$$s = \left(\frac{a}{l} + \frac{a_{val}}{l_{val}}\right) \times \left(1 - min\left\{\frac{\sqrt{|a - a_{val}| + |l - l_{val}|}}{4}, 1\right\}\right) \tag{7.4}$$

Additionally, if neither of the two values used to pick a weight is enabled, the weight can be picked manually.

7.6.4 JSON Configuration

In an attempt to design the system as configurable as possible, two configuration files are provided.

The CONFIG.JSON specifies the training configuration of both neural network systems as described in Chapter 7.4.2, while also providing general settings, such as LOAD (boolean; states whether the configuration file is to be used. If FALSE, the default configuration is applied.), VERBOSITY (integer $v \in \{1, 2, 3\}$; specifies printed level of verbosity.), LATEST and BEST (both boolean; together, they determine which weight is loaded.), OVERWRITE (boolean; states whether to overwrite existing extracted information or to skip the corresponding MIDI file during analysis.) and RETRY (boolean; during the analysis, a CSV file is created that states its success (evaluated using the progressions' checksums). If an analysis was either unsuccessful or semi-successful, the analysis can be attempted again.). While loading, the validity of each value is verified, since the values - which receive global access - have to be applied before the remaining initialisation of the system. The default configuration can be found in Appendix A.

TEMPLATES.JSON defines the user's configured piece and section templates that can be used for automatically composing entire pieces. Whether the configuration is semantically valid is not checked during initialisation, since the usage of templates is optional. If a runtime exception caused by a syntax error occurs while using invalid values from a template, the default templates, which are stored separately, are loaded.

Chapter 8

Results

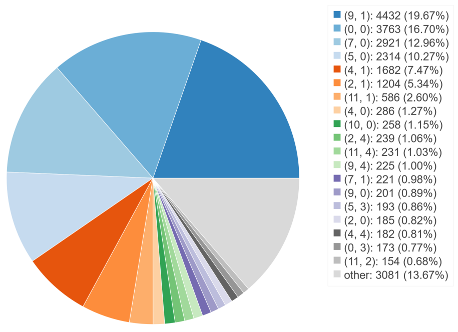
8.1 Information Extraction

Given the provided dataset, out of 122 MIDI files, one could not be analysed due to it not meeting the specified requirements. The extracted information - consisting of the piece's chord progressions, melodies, pitched accompaniments and unpitched accompaniments - of the remaining files was transformed into a format essential to the ensuing program execution and stored in the respective CSV file for further usage. Due to rounding errors, timings involving fractions (e.g. *tuplets* in phrases) are converted into FRACTION objects, which are not deserializable. The amount to be discarded eventually is quantified by the percentage of not deserializable elements as presented in Table 8.1. As expected, only a negligible quantity of the data is disposed of. Additionally, each strategy used in the chord estimation algorithm saved its result as MIDI without any complications.

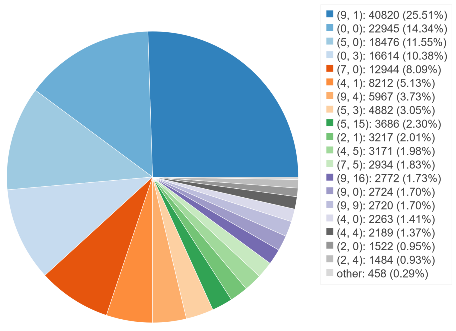
Type	**Total**	**Not Deserializable**	**Not Deserializable Percentage**
Progressions	197	0	0%
Melodies	4139	678	16.38%
Accompaniments	23349	617	2.64%
Percussion	10642	330	3.1%

Table 8.1: Serialization and deserialization Statistic.

To verify the viability of the generated elements, both chord progressions and phrases are compared to the data yielded from the preceding extraction. The com-



(a) Dataset



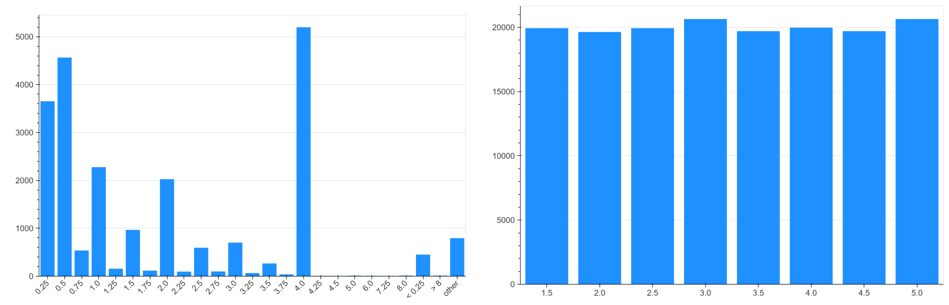
(b) Generated

Figure 8.1: Comparison of chord occurrences as pie charts. Although all chords are included in the charts, only the 19 most frequent ones are represented using a distinct colour since it is theoretically possible to represent 348 different tuples using the 12 potential root notes from the chromatic scale and the 29 defined chord type enums. The remaining tuples are summed up and classified as "other" - the 20th class that is found at the bottom of the legends.

parison is carried out using 10000 generated progressions consisting of 16 chords each and 1000 short phrases, which are built on *4-chord progressions*[26] sampled from the previously generated 16-chord progressions. Although differences are expected in the comparison, few major resemblances are anticipated, as well. Depending on the similarities found in the data, the potential findings would imply whether the system has successfully generated its compositions in a manner similar to the dataset's MIDI files.

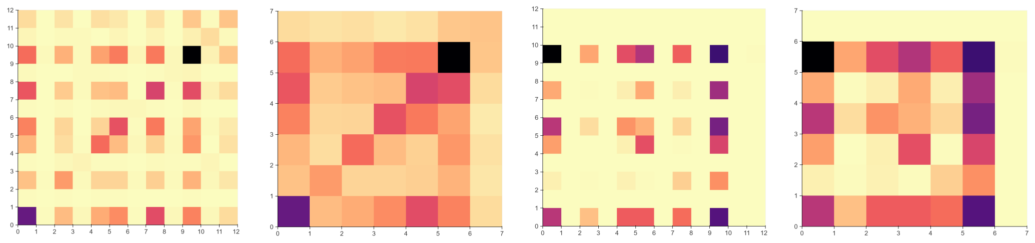
The two pie charts in Figure 8.1 show all chords, which were recognised by the chord estimation algorithm, using the tuple notation (r, t) as introduced in Chapter 6.5.1. With this data, a comparison is made between the chords used in the dataset's progression (Figure 8.1a) and the generated chord progressions (Figure 8.1b). As for each chord's duration, Figure 8.2 illustrates the total count of each occurred duration in comparison, as well. Lastly, Figure 8.3 shows the chord changes based on a chord's root note, disregarding the chord type, as a two-dimensional depth map. All 12 half steps of a chromatic scale are shown in Figures 8.3a and 8.3c, while only the 7 degrees of the diatonic scale are included in Figures 8.3b and 8.3d. By comparing the dataset's and the generated progressions' charts, all previously mentioned illustrations show various levels of deviation from each other.

[26]An *n-chord progression* is a chord progression consisting of n chords.



(a) Dataset **(b)** Generated

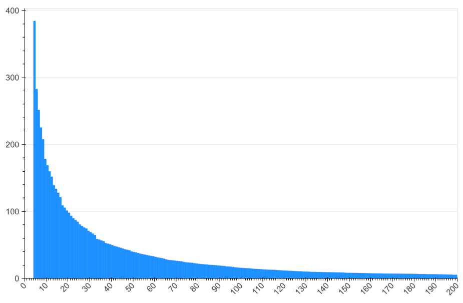
Figure 8.2: Comparison of chord durations as bar charts. All lengths that either contain fractions or have been rounded faultily are summed up and labelled as "other". The chords' durations are found on the x-axis, while the respective amount is represented by the y-axis.



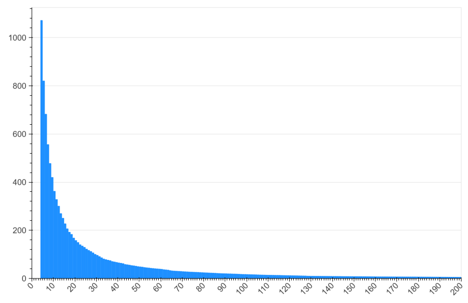
(a) Dataset Chromatic **(b)** Dataset Diatonic **(c)** Generated Chromatic **(d)** Generated Diatonic

Figure 8.3: Relative depth map of the root notes from each chord change $r_\Delta \mapsto r_{\Delta+1}$. The current chords' root notes r_Δ are represented on the x-axis, while the succeeding root notes can be found on the y-axis. It is differentiated between root notes that lie within the diatonic (seven degrees resulting in a 7×7 map) or chromatic (twelve degrees resulting in a 12×12 map) scale. Inside the maps, darker colour implies more frequent chord change occurrences, while the amount of occurrences are all relative to the most frequent chord change of each map. This is seen at $9 \mapsto 9$ and $5 \mapsto 5$ for the dataset's maps and $0 \mapsto 9$ and $0 \mapsto 5$ for the generated progressions' maps, which mark the respective maxima at a relative 100% as seen in the provided colour scale.

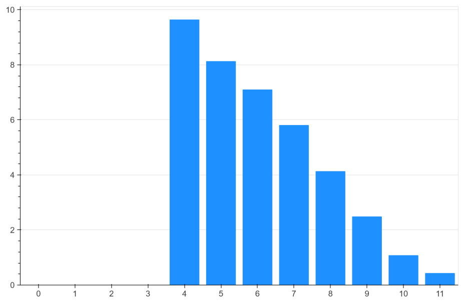
Similarly, the phrases will be reduced to their core elements, as well, in order to verify if any repeating rhythmic and tonal patterns exist. Here, a pattern is defined as a sequence that is longer than three notes. For the rhythmic parts, the offset to the preceding note and the own duration is utilised, while the relevant information for the tonal patterns is the pitches. The results are illustrated in comparison in Figure 8.4. It is immediately noticeable that considerable differences in terms of pattern length and occurrence amount exist when comparing the phrases.



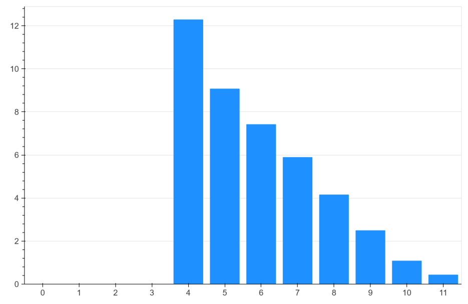
(a) Dataset Tonal Patterns



(b) Dataset Rhythmic Patterns



(c) Generated Tonal Patterns



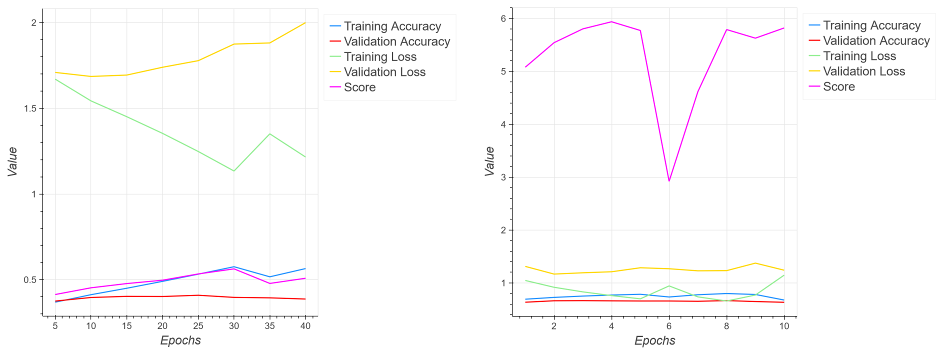
(d) Generated Rhythmic Patterns

Figure 8.4: The average amount of occurrences of repeated tonal and rhythmic patterns detected in the phrases. Here, the x-axis marks the length of any arbitrary pattern that was found at least twice inside a phrase, while the y-axis represents the counted occurrences associated with said pattern. Since a pattern is defined as a sequence of at least three notes, bars only exist for $x > 3$.

Judging by the extracted information, barely any major similarities exist. Except for the chords themselves - whose generated data overlaps noticeably with the dataset -, only minor resemblances are found in the chord progressions. While the dataset's durations vary immensely, the generated durations are evenly distributed. Furthermore, when it comes to chord changes, the maxima are located at different spots, as well. Lastly, the generated phrases contain fewer patterns - both tonal and rhythmic - of shorter lengths compared to the dataset's phrases. Although these results seem to suggest that the system's compositions lack resemblance, this does not necessarily imply an unsuccessful generation. Neither the neural networks nor their constraints have produced any futile data. The source of this apparent mediocrity and the implications of these findings are discussed in Chapter 9.

8.2 Element Prediction

Depending on the number of MIDI files that are fed into the system, the required time for the training process increases. Given the dataset, training an epoch takes approximately one minute for the progression network, while 30 minutes are required for the phrase network using the default configuration. The training and validation loss and accuracy throughout the networks are shown in Figures 8.5a and 8.5b. In order to illustrate the values' relationships to each other, the score, which is calculated using Equation 7.4, is included, as well.



(a) Progression Network **(b)** Phrase Network

Figure 8.5: Trends of all utilised metrics during training. For the progression network, the mean is calculated from all weights of the same epoch, disregarding the sequence length $i \in \{1, ..., n-1\}$. As for the phrase network, all three values from their respective subsystems (offset, pitch and duration) are averaged. In Chapter 9, the significance of these metrics and their consequences, as well as the abnormalities seen in the score, are explained.

Table 8.2 presents further useful statistics in comparison. The implication of the data is discussed and put into context in Chapter 9.

8.3 Static Element Generation

All additional elements are statically generated using the patterns that have been saved previously in combination with the neural network's prediction results. The system allows for manually added key signature changes. Furthermore the user interface enables the interactive alteration of the current piece. An initial key, tempo and name is automatically picked, however, it is possible to overwrite this data manually. Editing specific sections and parts is allowed, as well. Playback

Property	Dataset Progressions	Generated Progressions	Dataset Phrases	Generated Phrases
Total Amount of Sequences	197 Progressions	10000 Progressions	3461 Phrases	1000 Phrases
Average Sequence Duration	49.0534 Measures	13.028 Measures	11.3077 Measures	3.0379 Measures
Total Amount of Items in Sequences	22531 Chords	160000 Chords	177374 Notes	14341 Notes
Average Amount of Items in Sequences	114.3706 Chords	16 Chords	51.2493 Notes	14.3410 Notes
Minimum Amount of Items in Sequences	1 Chord	16 Chords	1 Note	3 Notes
Maximum Amount Items in Sequences	1880 Chords	16 Chords	1234 Notes	25 Notes
Total Amount of Chords with Diatonic Root	21513 Chords	159992 Chords		
Total Amount of Chords with likely Duration	21289 Chords	160000 Chords		
Average Amount of Rhythmic Patterns			57.2011 Patterns	3.5653 Patterns
Average Amount of Tonal Patterns			34.3735 Patterns	3.2305 Patterns

Table 8.2: Additional Statistics. Given the 4/4 Time Signature, one Measure has the duration of one whole note (semibreve).

of either the entire piece or a particular part is also implemented. Essentially, the static generation of elements proceeds as planned. In Appendix C, two generated examples are shown, which make use of the majority of implemented features.

Chapter 9

Discussion

In this study, a combined approach, which uses both data-driven methods and theoretical principles, has been explored by successfully implementing an application capable of generating instrumental pieces. By analysing a dataset of various MIDI files, musical elements are either directly extracted or transformed into inputs and outputs for artificial neural network subsystems. After learning and predicting a composition's fundamental elements, additional components are statically generated on top of those.

Considering the provided dataset's size and versatility, its included information is considered sufficient for this project. Since the MIDI file that has been left unprocessed has an internal time signature of 3/4, this confirms the successful application of a filter based on this specific analysis criteria. Based on the lost information during deserialization (Table 8.1), the total resulting 4.23% of not deserializable elements is not significant enough to affect this study in terms of quantity. Therefore, the static generation procedure of any arbitrary elements that make use of the extracted patterns remains completely unchanged. However, the quality of the chord progression identification impacts not only the progression network's input but also the subsequent phrase classification. It hence affects the melodies that are fed into the phrase network. Since the data is highly contextual, this amplifies the consequences.

When it comes to occurrences of chords, the dataset (Figure 8.1a) shows a higher versatility compared to the generated progressions (Figure 8.1b). In addition, a disproportional ratio of chords per progression per duration is derived from the values of Table 8.2. According to the chord estimation algorithm, the dataset's progressions' durations are only approximately four times as long as the generated ones. In contrast, the average amount of chords is more than seven times as high in comparison, hence implying that, on average, an additional 75% of chords are identified given any chord progression of any duration. Assuming that chord durations shorter than an eighth note (quaver), longer than a double whole note (breve) or part of a fraction are wrongly identified chords[27], this implies that at least 6% of all extracted progressions in Figure 8.2a are faulty, while, in reality, the chord progressions from the dataset contain fewer chords of higher durations. On the other hand, the generated progressions' chord durations seen in Figure 8.2b are very evenly distributed since each duration is statically computed, contrary to the predicted matrices that are transformed into (r, t) tuples. Granting that the chord recognition's accuracy - although not verified - falls short in comparison to Scholz and Ramalho's system [SR08], the yielded results are nonetheless usable for the subsequent processing. In addition, the majority of the input files' complexity is much higher due to the given orchestration.

The depth maps from Figure 8.3 imply the same as the chord occurrences, as the root notes of the dataset's chords occur outside the diatonic scale, which is not the case for the generated progressions. In addition to existing inaccuracies in the chord estimation, this can be explained by the implemented constraints for the generated chords. Since static successions and substitutions are utilised, the amount of probable elements and their successions is limited. Furthermore, the maxima at $0 \mapsto 0$ and $9 \mapsto 9$ in all maps are caused by the nature of mainstream music, in which progressions usually resolve in their scale's first degree, while the scale itself is either a major or minor one. The $0 \mapsto 9$ and $9 \mapsto 0$ successions in the generated progressions are the consequence of the constraints, where the tuples $(0, 0)$ and $(9, 1)$ are part of the potential initial chord set. Since the chord type is discarded in the map and over 50% of all generated chords' roots are either 0 or 9, these peaks can be traced directly to the chord occurrences.

In comparison to the dataset's phrases, the repeated patterns found in the generated phrases are quite rare, as seen in Figure 8.4. Due to the identified progressions and phrases being of a greater length, the likelihood of repeating patterns increases. Additionally, pieces will oftentimes consist of a structure that involves repetitions of entire sections. Although the chord estimation algorithm does split pieces into excerpts, it will not do so if all phrases inside a section are considered completely coherent. Therefore, when it comes to the dataset, all patterns with a length of l (with) inside a section's phrase of length p_s are counted, as well, while the generated pieces are all individually generated without any relation to each other. Since the implemented piece generation creates a piece section by section, the repetitions inside a full piece would increase linearly if the same pattern detection algorithm was applied. The reason for not evaluating entire pieces is the amount of time it would require to generate enough data, which is impossible due to the limited time frame.

[27]Although these chord lengths are used in compositions, too, in the majority of modern music, they are rather unlikely.

Although the predictions are altered using music theory based constraints, the networks' training metrics illustrated in Figures 8.5a and 8.5b have to be taken into account, as well. During the first 30 epochs, the progression networks' average score steadily increases until a sudden peak in the training loss causes a short plummet in the score. Generally, both the training accuracy and the validation loss seem to climb throughout the epochs, while the validation accuracy stays unchanged, indicating potential improvements for future epochs. The values of phrase network, however, seem to converge around a certain point depending on the metric. Although the network's values barely change, the score abruptly plunges at the sixth epoch, implying the existence of divergences from the values' means for each separate network. The exact cause is the difference in value between the offset network's and the other two networks' metrics, which can not be seen in Figure 8.5b since only the mean of all three values is used. Since the majority of the offset outputs is 0 due to the lack of pauses in-between notes inside phrases, the offset network would achieve relatively high scores by predicting 0 only. This does not apply to the other two networks, which is why their scores are comparably low. Under the assumption that the offset network's scores are much greater than the pitch and duration networks' scores, any irregularities found in the offset network's values will cause major abnormalities in the mean. This is exactly the case for the drop at epoch 6. Even though the decrease of the offset network's score lies within the expected margin of error, its impact on the mean is enormous since the other two scores are considerably lower. These findings suggest that the networks can be optimised by using individually built and adjusted neural networks for the phrase network's subnetworks to further tweak the system. The influence of a single network on the mean can be reduced in addition by introducing a threshold in Equation 7.4. Furthermore, the chord estimation algorithm's accuracy - which was the foreseen bottleneck explained in Chapter 5.5 - has to be improved in order to provide all networks with better, more accurate input data. When it comes to network fitting, slight underfitting is implied by the metrics, also suggesting that the training inputs and outputs require adjustments. Again, this was anticipated due to the experimental nature of the project and its time frame as described in Chapter 6.6. However, the utilisation of LSTM layers has evidently shown its benefits, similar to the previous results of Mangal et al. [MMJ19] and Lyu et al. [LWZM15]. Additionally, the LSTMs' input data contains a high level of context - computed by the preceding analysis - that allows for contextual outputs that are eventually capitalised on during the static element generation. Although the differences in the MIDI results would be barely noticeable due to the fact that the current system relies on constraints to filter out unlikely predictions, a more precise input would allow for relaxations in the implemented constraints, which directly translate to a higher amount of unaltered progressions and melodies.

Lastly, the static approach to generating orchestration is distinctly different from Herremans and Chew's [HC16] or Crestel and Esling's [CE16]. However, all statically generated elements fulfil their purpose since any pitched accompaniment is always relative to a chord, implying that the included pitches will not alter the existing harmony itself. Since the data fed into the neural networks is highly contextual, this leads to outputs that can be used in a musical context, as well. Thus, the subsequent component generation, which makes use of this context, is easily employed. Furthermore, the unpitched accompaniments only serve as underlying percussion, which is needed to add an additional sense of rhythm to the pieces. Therefore, the static element generation is considered a success.

To stay within the limits of this study's scope, the generated pieces were not included in a survey. Thus it could not be verified whether the compositions are considered satisfactory by the average listener. To establish this assertion, an additional, unbiased study is needed. Nevertheless, the creation of the fundamental chord progressions itself guarantees a certain amount of harmony, while melodic and rhythmic patterns are found throughout the generated pieces, as well.

Chapter 10

Conclusion

By implementing a system that generates original musical compositions, the findings of this thesis have indicated that the combined approach using machine learning and static implementations of theoretical principles does have potential in the practical field of music generation. The implemented system consists of two main subsystems that continuously interact with each other in order to achieve the mutual goal of automatic music generation. In the beginning, the static, theory based system provides the data-driven system with the input data that is required for its neural networks by extracting and transforming the information relevant in a musical context from the MIDI dataset. Using the received information, the networks are able to learn the elements' interrelation inside the sequences. Later on, after the data-driven system has generated the fundamentals of a piece, the static system creates additional elements on top of that foundation.

Using various methods based on music theory concepts, a highly complex, static chord estimation algorithm is implemented in the form of a strategy pattern that attempts to identify chord progressions applied throughout a MIDI file. The result is then applied to distinguish between chord-dependent and chord-independent phrases in order to classify these for further processing. While both the chord progressions and melodies are subsequently fed into LSTM networks, the accompaniments, however, are directly stored with the purpose of being sampled into a generated piece afterwards. The artificial neural network systems implemented can be separated into two subsystems as follows: First, a progression network, which consists of $n-1$ networks specialising in predicting chord tuples (r, t) on a preceding sequence of length $i \in \{1, ..., n-1\}$ in order to create a n-chord progression. And second, a phrase network that makes use of the three main properties of a note to create a network for the offset, pitch and duration, respectively, essentially successively creating note triplets (o, p, d) on sequences of a fixed length m. Both systems rely on static constraints that alter the probabilities of their prediction matrices. With the underlying fundamentals generated, additional elements, such as counterpoint melodies, accompaniments and key signature changes, are statically computed. A piece is built using sections, which are either manually created by the user or automatically generated using a previously configured template.

In addition to this work's successful attempt in automatic music creation, generally implementing an abstract combination of a static, theory based and a data-driven approach might lead to evidence supporting the application of this proposition in other branches, as well. Lastly, the usage of LSTMs has proven itself to be a viable approach in the context of automated music generation once again [MMJ19; LWZM15; HHC18], while the statically constrained, theory based approach confirms its existing potential for chord recognition problems [SR08; WL18], as well as music generation tasks [HC16; CE16].

For further reading, Hochreiter's works [HS97; Hoc91] describe dynamic neural networks focusing on LSTMs - along with their origin and application - in detail. Furthermore, the work of Schoenberg et al. [SSS67] provides a vast collection of music theory for an in-depth explanation and an extensive understanding of all concepts and principles mentioned in this study.

Chapter 11

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/ (visited on 05/07/2022) (cited on page 30).

[Ape69] Willi Apel. *Harvard Dictionary of Music*. Harvard University Press, 1969. ISBN: 9780674375017 (cited on pages 8, 9).

[Ass21] The MIDI Association. *Official MIDI Specifications*. 2021. URL: https://www.midi.org/specifications/midi1-specifications/midi-1-addenda/midi-tuning-updated (visited on 02/26/2022) (cited on page 9).

[BJZP20] Sunitha Basodi, Chunyan Ji, Haiping Zhang, and Yi Pan. "Gradient amplification: An efficient way to train deep neural networks". *Big Data Mining and Analytics* 3.3 (2020), pp. 196–207. DOI: 10.26599/BDMA.2020.9020004 (cited on page 12).

[BV22] MuseScore BV. *Musescore.com*. 2022. URL: https://musescore.com/ (visited on 04/10/2022) (cited on page 23).

[CAHO21] Michael Cuthbert, Christopher Ariza, Benjamin Hogue, and Josiah Wolf Oberholtzer. *music21: a Toolkit for Computer-Aided Musicology*. 2021. URL: https://web.mit.edu/music21/ (visited on 04/14/2022) (cited on pages 17, 27).

[CE16] Léopold Crestel and Philippe Esling. "Live orchestral piano, a system for real-time orchestral music generation". *arXiv preprint arXiv: 1609.01203* (2016) (cited on pages 6, 48, 50).

[CF20] *53 Cantus Firmi by famous musicians*. 2020. URL: https://48a396c9-e039-4167-9469-820fc5572658.filesusr.com/ugd/cf79fa_272174fb4d9f4e24aa9ce71afc6dcb91.pdf (visited on 05/04/2022) (cited on page 19).

[Che18] Guillaume Chevalier. "LARNN: linear attention recurrent neural network". *arXiv preprint arXiv:1808.05578* (2018) (cited on page 12).

[Cho22] François Chollet. *Keras: the Python deep learning API*. 2022. URL: https://keras.io/ (visited on 04/20/2022) (cited on page 30).

[Cly13] Manfred Clynes. *Music, mind, and brain: The neuropsychology of music*. Springer Science & Business Media, 2013 (cited on pages 3, 8).

[Com22] Creative Commons. *About the Licenses - Creative Commons*. 2022. URL: https://creativecommons.org/licenses/ (visited on 04/10/2022) (cited on page 23).

[Dan00] Roger B Dannenberg. "Artificial intelligence, machine learning, and music understanding". In: *Proceedings of the 2000 Brazilian symposium on computer music: arquivos do simpsio brasileiro de computao musical (SBCM)*. 2000 (cited on page 5).

[For13] Allen Forte. *Tonal Harmony*. 3rd ed. Holt, Rinehart, and Wilson, 2013. ISBN: 0-03-020756-8 (cited on pages 4, 7).

[Fou22] Python Software Foundation. *Python Release Python 3.8.0*. 2022. URL: https://www.python.org/downloads/release/python-380/ (visited on 04/18/2022) (cited on page 27).

[HC16] Dorien Herremans and Elaine Chew. "MorpheuS: Automatic music generation with recurrent pattern constraints and tension profiles". In: *Proceedings of IEEE TENCON,-2016 IEEE Region 10 Conference*. IEEE. 2016, pp. 282–285 (cited on pages 4, 5, 48, 50).

[HGBC18] Jeff Heaton, Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. 2018 (cited on page 10).

[HHC18] Yongjie Huang, Xiaofeng Huang, and Qiakai Cai. "Music Generation Based on Convolution-LSTM." *Comput. Inf. Sci.* 11.3 (2018), pp. 50–56 (cited on pages 4, 5, 21, 50).

[Hoc91] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". *Diploma, Technische Universität München* 91.1 (1991) (cited on pages 12, 50).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667 (cited on pages 5, 11, 12, 50).

[HSS12] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent". *Cited on* 14.8 (2012), p. 2 (cited on page 32).

[Ini22] Open Source Initiative. *The 3-Clause BSD License | Open Source Initiative*. 2022. URL: https://opensource.org/licenses/BSD-3-Clause (visited on 04/10/2022) (cited on page 27).

[KR88] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988 (cited on page 27).

[Lai08] Steven Geoffrey Laitz. *The complete musician: An integrated approach to tonal theory, analysis, and listening*. Vol. 1. Oxford University Press, USA, 2008 (cited on page 19).

[LB+95] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995 (cited on page 5).

[Loz20] Josko Lozic. "The revenue recovery of the music industry as a result of revenue growth from streaming". *Economic and Social Development: Book of Proceedings* (2020), pp. 203–214 (cited on page 1).

[LWG22] Fei-Fei Li, Jiajun Wu, and Ruohan Gao. *CS231n: Deep Learning for Computer Vision*. 2022. URL: http://cs231n.stanford.edu/ (visited on 04/12/2022) (cited on page 10).

[LWZM15] Qi Lyu, Zhiyong Wu, Jun Zhu, and Helen Meng. "Modelling high-dimensional sequences with lstm-rtrbm: Application to polyphonic music generation". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015 (cited on pages 4, 5, 21, 47, 50).

[MH97] N. Mladenović and P. Hansen. “Variable neighborhood search”. *Computers & Operations Research* 24.11 (1997), pp. 1097–1100. ISSN: 0305-0548. DOI: https://doi.org/10.1016/S0305-0548(97)00031-2. URL: https://www.sciencedirect.com/science/article/pii/S0305054897000312 (cited on page 5).

[MMJ19] Sanidhya Mangal, Rahul Modak, and Poorva Joshi. “LSTM based music generation system”. *arXiv preprint arXiv:1908.01080* (2019) (cited on pages 4, 5, 21, 47, 50).

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. *nature* 323.6088 (1986), pp. 533–536 (cited on page 5).

[Ric98] Alan Rich. “Harmony”. *Encyclopædia Britannica 15th Edition* 18 (1998) (cited on pages 2, 8).

[Ron14] Xin Rong. “word2vec parameter learning explained”. *arXiv preprint arXiv:1411.2738* (2014) (cited on page 24).

[SD01] Klaus-Jürgen SACHS and Carl DAHLHAUS. *Counterpoint, Grove Music Online*. 2001 (cited on page 19).

[SD10] Jon Sneyers and Danny De Schreye. “APOPCALEAPS: Automatic music generation with CHRiSM”. In: *Proceedings of 22nd Benelux Conference on Artificial Intelligence (BNAIC’10)*. 2010, pp. 1–8 (cited on pages 3, 5).

[Smo86] Paul Smolensky. *Information processing in dynamical systems: Foundations of harmony theory*. Tech. rep. Colorado Univ at Boulder Dept of Computer Science, 1986 (cited on page 5).

[SR08] Ricardo Scholz and Geber Ramalho. “COCHONUT: Recognizing Complex Chords from MIDI Guitar Sequences.” In: Sept. 2008 (cited on pages 4, 6, 46, 50).

[SSS67] Arnold Schoenberg, Leonard Stein, and Gerald Strang. *Fundamentals of musical composition*. Faber & Faber London, 1967 (cited on pages 3, 4, 50).

[Tit09] Jeff Todd Titon. *Worlds of Music: An Introduction to the Music of the World’s Peoples*. 5th ed. Schirmer Books, 2009. ISBN: 978-0534595395 (cited on page 8).

[Toc77] Ernst Toch. *The shaping forces in music: an inquiry into the nature of harmony, melody, counterpoint, form*. Courier Corporation, 1977 (cited on pages 8, 19).

[Wil20] Victoria Williams. *Counterpoint*. 2020. URL: https://www.mymusictheory.com/learn-music-theory/reference/579-counterpoint (visited on 05/04/2022) (cited on page 19).

[WL18] Yiming Wu and Wei Li. “Automatic audio chord recognition with MIDI-trained deep feature and BLSTM-CRF sequence decoding model”. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 27.2 (2018), pp. 355–366 (cited on pages 6, 50).

[YC11] Yi-Hsuan Yang and Homer H Chen. *Music emotion recognition*. CRC Press, 2011 (cited on page 8).

Appendix A

Code Excerpts

```
class ChordType(Enum):
    MAJ = [0, 4, 7], ['Major'], '', 0
    MIN = [0, 3, 7], ['Minor'], 'm', 1
    DIM = [0, 3, 6], ['Diminished'], 'o', 2
    MAJ7 = [0, 4, 7, 11], ['Major Seventh'], 'M7', 3
    MIN7 = [0, 3, 7, 10], ['Minor Seventh'], 'm7', 4
    MAJMIN7 = [0, 4, 7, 10], ['Major Minor Seventh', 'Dominant Seventh'], '7', 5
    DIM7 = [0, 3, 6, 9], ['Diminished Seventh'], 'o7', 6
    HDIM7 = [0, 3, 6, 10], ['Half-Diminished Seventh', 'Minor Seventh Flat Five'], '7b5', 7
    MINMAJ7 = [0, 3, 7, 11], ['Minor Major Seventh '], 'm/M7', 8
    SUS4 = [0, 5, 7], ['Suspended Fourth'], 'sus4', 9
    SUS2 = [0, 2, 7], ['Suspended Second'], 'sus2', 10
    AUG = [0, 4, 8], ['Augmented Triad', 'Augmented Fifth'], '+', 11
    ADD9 = [0, 2, 4, 7], ['Added Ninth'], 'add9', 12
    ADD4 = [0, 4, 5, 7], ['Added Fourth'], 'add4', 13
    MINADD4 = [0, 3, 5, 7], ['Added Fourth'], 'madd4', 14
    MAJ6 = [0, 4, 7, 9], ['Major Sixth'], '6', 15
    MIN6 = [0, 3, 7, 9], ['Minor Sixth'], 'm6', 16
    AUG7 = [0, 4, 8, 11], ['Augmented Major Seventh', 'Augmented Seventh', 'Major Seventh Sharp Five'], '7#5
        ', 17
    DMAJ9 = [0, 4, 7, 10, 14], ['Dominant Ninth', 'Dominant Major Ninth'], '9', 18
    DMIN9 = [0, 4, 7, 10, 13], ['Dominant Minor Ninth'], '7b9', 19
    MIN9 = [0, 3, 7, 10, 14], ['Minor Ninth'], 'm9', 20
    MAJ9 = [0, 4, 7, 11, 14], ['Major Ninth'], 'M9', 21
    P69 = [0, 4, 7, 9, 14], ['Six Add Nine'], '6/9', 22
    M6SUS4 = [0, 5, 7, 9], ['Sixth Suspended Fourth'], '6sus4', 23
    M6SUS2 = [0, 2, 7, 9], ['Sixth Suspended Second'], '6sus2', 24
    MAJ11 = [0, 4, 7, 11, 14, 17], ['Major Eleventh'], 'M11', 25
    MIN11 = [0, 3, 7, 10, 14, 17], ['Minor Eleventh'], 'm11', 26
    D11 = [0, 4, 7, 10, 14, 17], ['Dominant Eleventh'], '11', 27
    LYD = [0, 4, 7, 11, 18], ['Lydian', 'Major Seventh Sharp Eleventh'], '#11', 28

    def __init__(self, pitches: list, names: list, abbreviation: str, ctid: int):
        self._pitches_ = pitches
        self._names_ = names
        self._abbreviation_ = abbreviation
        self._ctid_ = ctid
```

Listing A.1: All defined chord type enums so far (new ones are easily added). Partial code of SRC.MODEL.BASIC.CHORD.CHORDTYPE. Excerpt shows enums and class' constructor.

```
{
  "general": {
    "load": true,
    "verbosity": 2,
    "latest": false,
    "best": true,
    "overwrite": false,
    "retry": false
  },
  "chord_progression": {
    "train": "neither",
    "sequence_length": 16,
    "epochs": 40,
    "batch_size": 64,
    "save_frequency": 5
  },
  "melody": {
    "train": "neither",
    "sequence_length": 64,
    "epochs": 10,
    "batch_size": 64,
    "save_frequency": 2
  }
}
```

Listing A.2: Default configuration of the system. Configuration values are validated upon loading.

Appendix B

UML-Diagrams

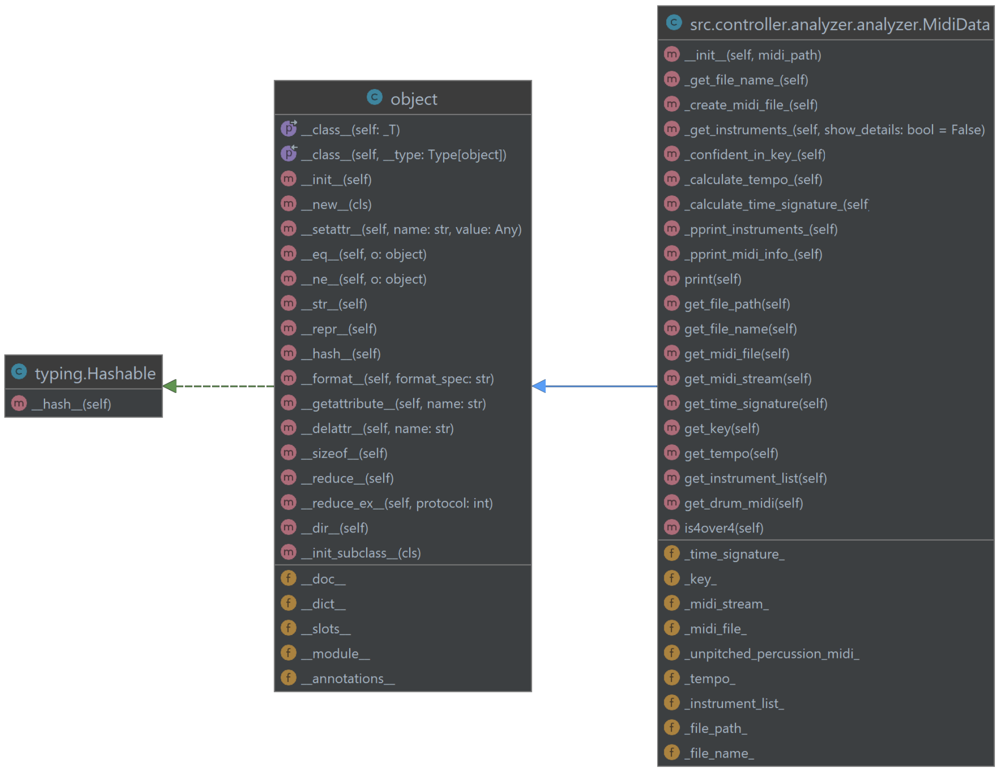


Figure B.1: Initial MIDI data Handling upon data transformation. Full UML-Diagram of class SRC.CONTROLLER.ANALYZER.ANALYZER.MIDIDATA.

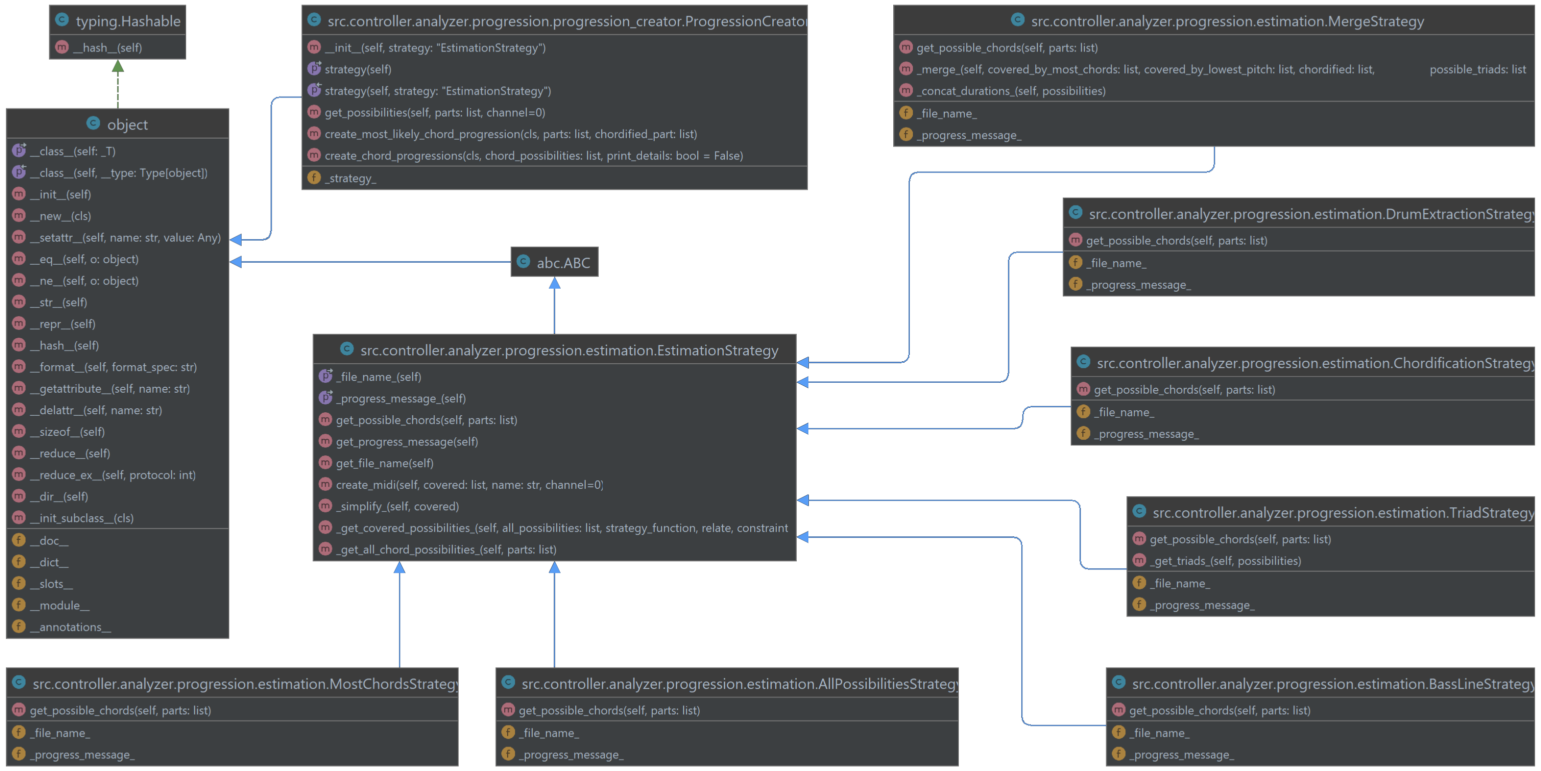

typing.Hashable
__hash__(self)
object
__class__(self: _T)
__class__(self, __type: Type[object])
__init__(self)
__new__(cls)
__setattr__(self, name: str, value: Any)
__eq__(self, o: object)
__ne__(self, o: object)
__str__(self)
__repr__(self)
__hash__(self)
__format__(self, format_spec: str)
__getattribute__(self, name: str)
__delattr__(self, name: str)
__sizeof__(self)
__reduce__(self)
__reduce_ex__(self, protocol: int)
__dir__(self)
__init_subclass__(cls)
__doc__
__dict__
__slots__
__module__
__annotations__
src.controller.analyzer.progression.progression_creator.ProgressionCreator
__init__(self, strategy: "EstimationStrategy")
strategy(self)
strategy(self, strategy: "EstimationStrategy")
get_possibilities(self, parts: list, channel=0)
create_most_likely_chord_progression(cls, parts: list, chordified_part: list)
create_chord_progressions(cls, chord_possibilities: list, print_details: bool = False)
strategy
abc.ABC
src.controller.analyzer.progression.estimation.EstimationStrategy
_file_name_(self)
_progress_message_(self)
get_possible_chords(self, parts: list)
get_progress_message(self)
get_file_name(self)
create_midi(self, covered: list, name: str, channel=0)
simplify(self, covered)
_get_covered_possibilities_(self, all_possibilities: list, strategy_function, relate, constraint
_get_all_chord_possibilities_(self, parts: list)
src.controller.analyzer.progression.estimation.MergeStrategy
get_possible_chords(self, parts: list)
merge(self, covered_by_most_chords: list, covered_by_lowest_pitch: list, chordified: list, possible_triads: list
_concat_durations_(self, possibilities)
_file_name_
_progress_message_
src.controller.analyzer.progression.estimation.DrumExtractionStrategy
get_possible_chords(self, parts: list)
_file_name_
_progress_message_
src.controller.analyzer.progression.estimation.ChordificationStrategy
get_possible_chords(self, parts: list)
_file_name_
_progress_message_
src.controller.analyzer.progression.estimation.TriadStrategy
get_possible_chords(self, parts: list)
_get_triads_(self, possibilities)
_file_name_
_progress_message_
src.controller.analyzer.progression.estimation.MostChordsStrategy
get_possible_chords(self, parts: list)
_file_name_
_progress_message_
src.controller.analyzer.progression.estimation.AllPossibilitiesStrategy
get_possible_chords(self, parts: list)
_file_name_
_progress_message_
src.controller.analyzer.progression.estimation.BassLineStrategy
get_possible_chords(self, parts: list)
_file_name_
_progress_message_

Figure B.2: The Chord Estimation Algorithm as UML-Diagram.

Appendix C

Generated Examples

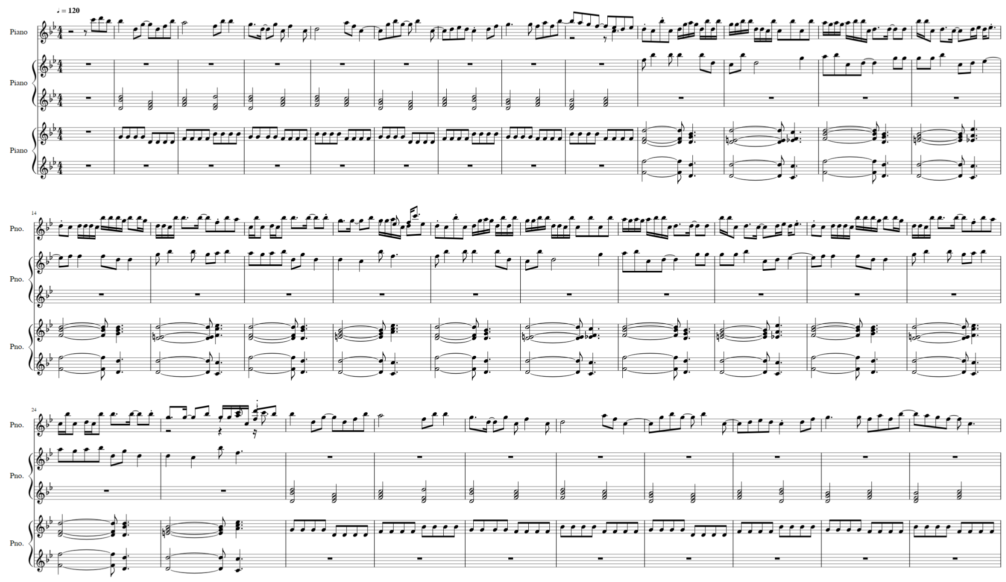


Figure C.1: Template used: PIANO. Full piece.

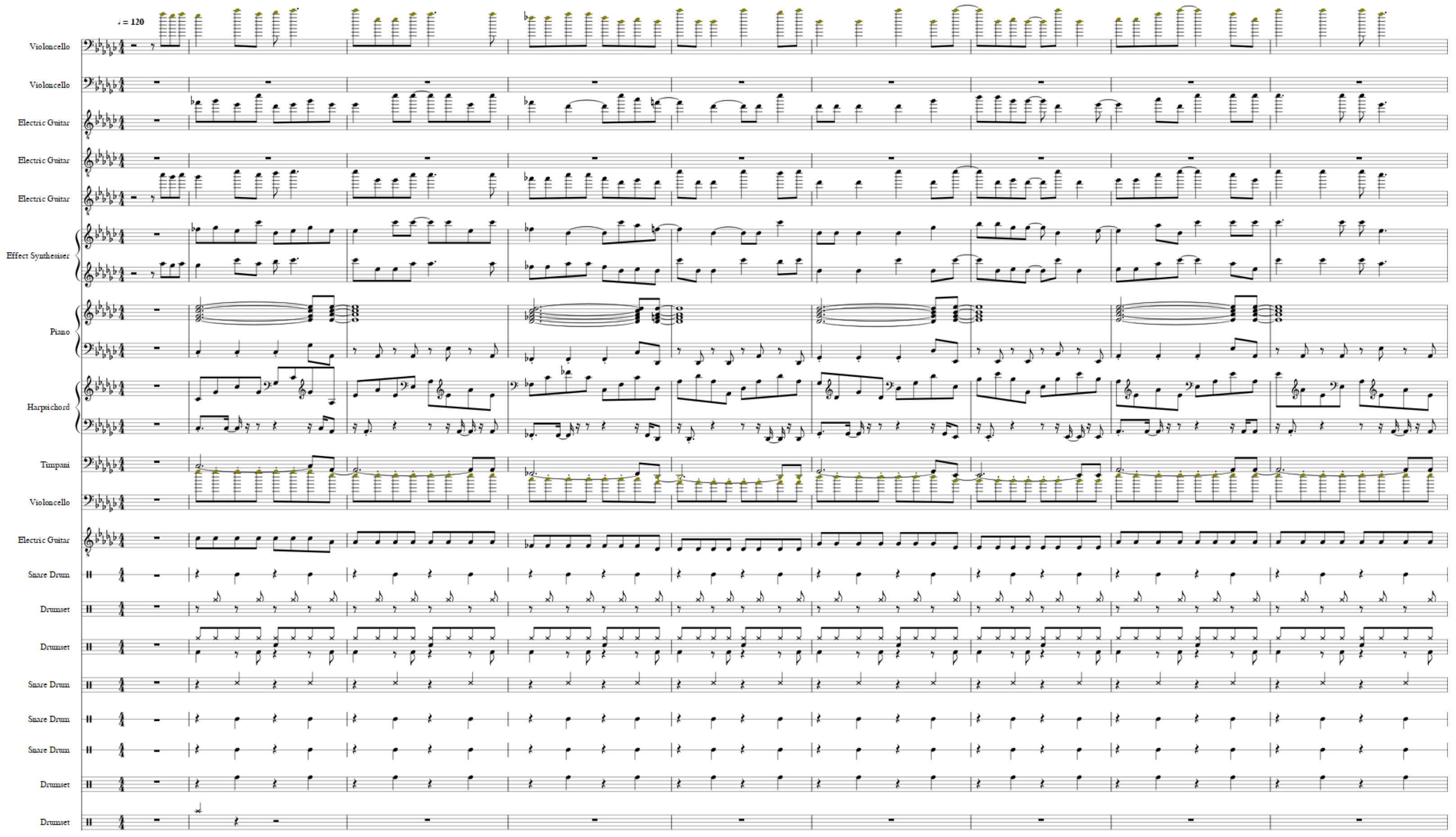


Figure C.2: Template used: SHORT_SONG. Only the first half of the first section is included in this screenshot.