Generating Structured Music Using Artificial Intelligence

Final Report of Bachelor Thesis

Submitted by Tim Wedde

In fulfillment of the requirements for the degree
Bachelor of Information and Communication Technology
To be awarded by
Fontys Hogeschool Techniek en Logistiek

Venlo, NL - June 11, 2018

WORD COUNT: 13.400

NAME OF STUDENT: Tim Wedde

COURSE: Informatics - Software Engineering
PERIOD: February - June 2018

COMPANY NAME: Genzai B.V.

NON-DISCLOSURE AGREEMENT: No

Generating Structured Music Using Artificial Intelligence: Final Report of Bachelor Thesis, © Tim Wedde, June 11, 2018

SUMMARY

This thesis is concerned with the computational generation of musical pieces, utilising concepts from the area of Artificial Intelligence. The main focus lies on the finding of a solution to the problem of bringing long-term, high-level structure to high-dimensional, sequential streams of data, into which music can be encoded, while also replicating stylistic information of a specific genre of music, in this case classical carnival music.
To achieve this, feasible approaches are selected from the current state-of-the-art within the field and combined into a software package that allows for the generation of structured musical pieces containing multiple instruments and distinct sections within the generated output approximating common song structures. The completed solution is able to generate structured songs conditioned on an underlying chord progression while replicating multiple instruments.

All code artifacts and samples are available under the following URL
https://github.com/timwedde/ai-music-generation

STATEMENT OF AUTHENTICITY

I, the undersigned, hereby certify that I have compiled and written this document and the underlying work / pieces of work without assistance from anyone except the specifically assigned academic supervisor. This work is solely my own, and I am solely responsible for the content, organization, and making of this document and the underlying work / pieces of work.

I hereby acknowledge that I have read the instructions for preparation and submission of documents / pieces of work provided by my course / my academic institution, and I understand that this document and the underlying pieces of work will not be accepted for evaluation or for the award of academic credits if it is determined that they have not been prepared in compliance with those instructions and this statement of authenticity.

I further certify that I did not commit plagiarism, did neither take over nor paraphrase (digital or printed, translated or original) material (e.g. ideas, data, pieces of text, figures, diagrams, tables, recordings, videos, code, ...) produced by others without correct and complete citation and correct and complete reference of the source(s). I understand that this document and the underlying work / pieces of work will not be accepted for evaluation or for the award of academic credits if it is determined that they embody plagiarism.

Venlo, NL - June 11, 2018

TIM WEDDE ( )

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CODE SNIPPETS

ACRONYMS

AI Artificial Intelligence

ANN Artificial Neural Network

ASF Apache Software Foundation

AWS Amazon Web Services

BPM Beats Per Minute

CC Control Change

CI Computational Intelligence

CPU Central Processing Unit

CSV Comma-separated value

DAW Digital Audio Workstation

DNN Deep Neural Network

FSF Free Software Foundation

GCP Google Cloud Platform

GPU Graphics Processing Unit

GUI Graphical User Interface

LSTM Long Short-Term Memory

LVK Limburgs Vastelaovesleedjes Konkoer

MIDI Musical Instrument Digital Interface

MIR Music Information Retrieval

ML Machine Learning

PoC Proof of Concept

PPQ Pulses Per Quarter Note

RBM Restricted Boltzmann Machine

RNN Recurrent Neural Network

TUI Text-based User Interface

UI User Interface

VM Virtual Machine

GLOSSARY

MUSICAL TERMINOLOGY

Bar	A segment of time consisting of a number of beats, as determined by the meter.
Beat	The fundamental unit of time used to measure progression of time within a musical piece.
Chorus, Verse	Commonly used to denote repeating, alternating sections within a musical piece.
Chord	The sounding of multiple notes at the same time.
Corpus	A collection of musical pieces.
Harmony	The vertical aspect of music, in contrast to the horizontal melodic line progression. The composition of sounds at the same timestep to form chords and intervals.
Key	The group of pitches, or scale, that forms the basis of a musical piece.
Lick	A stock pattern or phrase consisting of a short series of notes used in solos and melodic lines or accompaniment.
Meter	Synonymous to the time signature of a piece, defines recurring patterns, e.g. beats and bars.
Mode	A type of musical scale coupled with a set of characteristic melodic behaviors.
Monophony	A simple line of individual notes.
Motif	A pattern in a melody that repeats multiple times.
Musical Piece	An original composition, either a song or instrumental segment, specifically the structure thereof.
Note	A specific pitch or frequency emitted by any instrument.
Polyphony	Two or more simultaneous lines of independent melody.
Scale	Any set of musical notes ordered by fundamental frequency or pitch.
Voice	A single strand or melody of music within a larger ensemble or a polyphonic musical piece.
Voice Leading	The linear progression of melodic lines (voices) through time and their interaction with one another to create harmonies, according to the principles of common-practice harmony and counterpoint.

1 INTRODUCTION

This thesis, in the following sections, will describe the process and the results of the graduation project executed at *Genzai B.V.*, a company specialized on delivering solutions incorporating Artificial Intelligence (AI) to deliver insights into clients' data, as well as creating additional business value by implementing custom solutions for extended data analytics.
The below will describe the basic context of the project, its motivation as well as giving a quick overview of the problem domain and the content of the chapters following it.

1.1 BACKGROUND

This thesis documents the process of the creation of the final software product, as well the various decisions that have been made during the execution of the project. It serves as an overview of this project and as a repository of additional information about the research area the project is situated in in general.
In addition, it can be used to recreate or continue the project, either by independent researchers or by Genzai, after this thesis has concluded.

1.2 THE COMPANY - GENZAI B.V.

Genzai is a relatively new consulting company founded in 2016 providing services in the realm of AI to solve various problems within the business environment of other companies. Sectors include supply chain management as well as retail, public and agrofood, among others. It consists of four employees, including the CEO Roy Lenders, and is based within the Manufactuur, a space where multiple other innovative startups are also housed.

Figure 1: Logo of the company *Genzai B.V.*

Current projects include stock market analysis and prediction of mid- to long-term trends within it, supply chain management for various clients as well as root-cause analysis to determine inefficiencies within the internal processes of a client's large helpdesk office.

1.3 THE TASK - MUSIC GENERATION

To make itself more well-known in the realm of AI, Genzai wants to execute a high-profile project involving artificial music generation, with the overall goal being to participate in and possibly win the Limburgs Vastelaovesleedjes Konkoer (LVK) 2019 (a competition for carnival songs within the region of Limburg) with such a generated piece. Additionally, Genzai hopes that research into the area of time-series data prediction, pattern recognition and reconstruction can serve as a supplement to another currently active project concerned with long-term stock-market trend prediction.
To achieve this goal, a project is to be executed that should determine the feasibility of and possibly find a solution to the problem of generating pleasant-sounding music (mainly carnival music) in order to enable the artificially generated songs to be performed by actual musicians of the field. The main problem lies in the finding of a fitting approach to achieve this idea using techniques from the realm of Artificial Intelligence, and subsequently the implementation of a Proof of Concept (PoC) application.
Thus, the overall focus of this thesis in specific is not directed at the generation of individual melodies with short- to mid-term structure but rather towards the combination of various research areas and disciplines to eventually form a complete musical piece, a "song".

1.4 CONTEXT

This project follows in the footsteps of a multitude of similar ventures, a large amount of which have sprung up within the last decade with the advent of easily accessible Machine Learning (ML) frameworks such as TensorFlow[1] or Theano[2], but the origins and predecessors of which date back almost 30 years, coming close to the advent of computing itself, with even one of the first computers, the ILLIAC I being used for this purpose [HI92].
A multitude of approaches ranging from algorithmic, rule-bound composition to autodidactic neural networks have been tried with varying degrees of success, but up to this point, no solution has been found that would enable the generation of believably authentic and complete musical pieces indistinguishable to human assessors without additional post-processing and human intervention.
Music generation has become a focus of researchers in the past decades due to its relatively complete and vastly expansive documentation across a large timespan, with records of musical pieces reaching back as far 100 AD with the first recorded piece of music, "Seikilos epitaph" (first described in [Win29]), as well as the digital availability of transcribed pieces in a multitude of formats. Additionally, in contrast to other artistic areas (e.g. painting, writing, acting), music exposes the most rigid and well-defined ruleset of any of the disciplines, from rules codifying the relationship between individual notes up to regulating the compositional structure of entire pieces.

1 https://tensorflow.org
2 https://github.com/Theano/Theano

These rules and conventions contribute to the reduction of the solution space for musical pieces, since only a small subset of all possible combinations of notes and compositions thereof achieve "musicality", this referring to the pleasantness of the music to the human ear.
Because music consists of a large amount of parameters, including pitch, rythm, melody, harmony, composition, time and key signatures, used instruments and much more, possibilities for creating novel musical pieces are virtually endless and thus make it necessary for projects attempting an algorithmic approximation of human compositions to restrict the search space to a subset of these parameters, leading to a fractured scientific landscape, with projects focusing on small individual parts of the whole (e.g. chord prediction, monophonic melody or drum pattern generation).
The aim of this project thusly is to combine results within these different areas into a larger whole, which is eventually able to create structured musical pieces.

1.5 STRUCTURE OF THE THESIS

The thesis following this introductory section is split into five subsequent chapters approximating the development process of the software solution and the research preceding it. A more detailed description of the project can be found in Chapter 2. Following this is a look at and deeper research into possible approaches to solving the formulated problem (Chapter 3), with a selection and further description of the final course of action that was taken following suit in Chapter 4, which explains the final architecture and design decisions while detailing interesting aspects of the eventual implementation. This is supplemented by Appendix B, which details how the acquired data was pre- and post processed.
The project is concluded in the last two chapters, within which the final output of the project is presented, analyzed according to several quality criteria and compared to results of similar projects (Chapter 5) and finally the project as a whole reflected upon (Chapter 6).
An introductory overview of useful domain knowledge concerning Artificial Intelligence and its inner workings can be found in Appendix A. Subsequently, Appendix B elaborates on how data was prepared for use in this project and analyses how it is made up, to provide a baseline of information about the foundation of the trained models. Appendix C contains additional information, figures and images used within this thesis, as well as two sections expanding in more detail on basic concepts of music theory and the MIDI standard respectively, to enable deeper understanding of them in relation to this research objective.

2 THE PROJECT

Since little knowledge related to artifical music composition exists within the company, the main purpose of this project is to assess the feasibility of achieving the set goal and subsequently finding a way to achieve it as far as it has been determined to be possible, providing a PoC implementation along the way which can then be used by the company to expand upon utilising the knowledge generated in this project.
For this purpose, the thesis should provide introductory information on generative AI, especially when concerned with structure and patterns in time-series data and an architecture should be proposed that attains the end goal of generating structured, musical pieces in a specific style.

2.1 REQUIREMENTS

After consultation with the CEO and initiator of the project, Roy Lenders, and a detailed look at the situation, the below requirements were determined.
The project should...

- use MIDI files, to enable the easier collection of data and provide a common input and output format that can be used for and by a variety of applications
- generate music in a specific style (genre of music), to fit the intended use-case of producing songs that could possibly be utilised for participating in the LVK
- generate music in a structure similar to common song structures (e.g. make a distinction between Verse and Chorus)
- generate music that makes use of multiple instruments

However it is not required to...

- surpass the current state of the art in the area of music generation systems
- find or create a new approach to generating music
- emulate all facets of a complete musical piece (only the most important features should be emulated)
- conform to all theoretical aspects of a style or genre from a music theory point of view

2.1.1 *Stakeholder & Risk Analysis*

Given that this project is relatively free-form, due to its nature being that of a feasibility study, no outside dependencies exist as the project is entirely self-contained. Thus, there are no risks outside of the ordinary that need to be factored in besides the usual risks of the project being delayed because of unforeseeable changes in schedule or delays during implementation. Similarly, the roster of stakeholders is very small:

1. Roy Lenders as the project initiator
2. Frans Pollux as an artist who would possibly perform a generated song live, as well as domain expert on carnival music
3. External personnel from other companies as interested spectators with possible specialised knowledge in related technical fields

The list of stakeholders is ordered in descending order of influence over and interest in the project.

2.2 APPROACH

The area of artificial music composition utilising AI in its current form is a relatively niche field of study as a whole when compared to other areas such as image recognition and processing. Under the additional consideration that little knowledge about it exists at Genzai, the first step towards the end goal is the creation of a knowledge resource detailing the important concepts involved in it as well as a general overview of this area of scientific inquiry. This is intended to provide a baseline of knowledge and thusly aid in the understanding of the final product when the handover occurs.
In addition, two other topics will be covered on a high level as they are important to the execution of this project: Music Theory and the MIDI file format. The former is especially important in determining what a "style" of music (sometimes referred to as "genre") encompasses, which is valuable to know to be able to emulate its likeness in the final output and check the resulting samples for during the analysis phase. Given that the project is set to use MIDI files as its primary source of data as well as its output format, they will have to be understood to be able to process them into a format admissible for training. This is especially important in order to be able to allow for the recreation and continuation of this project in the future, especially with different or modified training data.
Following this, a survey of available and subsequently an evaluation and comparison of possible approaches to solving this problem will be executed to determine the best approach to take within the context of the goal of generating a structured and musical song.
Based on the selected approach, an architecture for the software will be devised and implemented, the process of which will be documented and important parts of it highlighted. For the parts of the project involving ML, a *training*

plan will be created that describes the data used as well as how it was processed to produce the final result, to enable the recreation and modification of the models as well as the training data at a later stage.
Once the PoC is completed, the final output will be assessed and compared to similar software in this area to determine the overall quality and success of the project.

The final output of this project encompassess:

- A PoC application that is able to generate structured music approximating common song structures in a specific style
- All datasets that were used for the training of involved ML parts
- All trained models used within the final application
- Scripts and documentation describing the pre- and post-processing of the datasets to allow for recreation and ajdustment of the experiment and the datasets themselves
- A small selection of generated songs from the PoC application for demonstration purposes

2.3 PLANNING

Table 3 details, in chronological order, the individual tasks that have been identified and have to be successfully executed to achieve the goal of this project. A Gantt chart mapping of these tasks over the timespan of this project can be found in Figure 23. Both figures are available in Appendix C.
This project combines two approaches to managing project time that are commonly found in the area of software engineering: The Waterfall model for sequential tasks and the Agile methodology for rapidly adapting parts of the project. On a high level, the project is structured using the Waterfall model, since the overall time alotted is fixed and thus all high-level tasks have to be fitted into this timeframe.
For parts involving research (which runs in parallell with the reporting tasks `D2` and `D3`), an adapted version of the Agile methodology will be applied, meaning that within the planned timeframe, multiple smaller sprints (less than one week in length) will be executed. This approach was taken as the research parts will be focused on the selection of an overall approach to take and thus may stray wildly between possible solutions before converging on the final chosen approach.

3 EXPLORING THE PROBLEM SPACE

Finding a workable approach to the formulated problem is quite difficult in an area within which development is rapidly ongoing and no optimal solution has yet been discovered. As such, this chapter will consist of research into different approaches to similar problems that are currently state-of-the-art. It will do this by posing a hierarchical list of research-objectives related to sub-problems at a lower level of abstraction, which will build upon each other to eventually determine the best approach to each subdomain. The results will then be combined to solve the overarching problem of structured music generation in the context of this project.

3.1 METHODOLOGY

The main acquisition procedure for information regarding this topic will be literature research, in combination with the empirical evaluation of existing projects that provide code which can be run and the output tested.
Literature will mostly be acquired digitally, as little physical publications exist on this topic and are hard to come by. As the area of artificial music generation is still quite new and research is continuously ongoing, most information reviewed here will be primary literature in the form of research papers and similar publications, with publications of a secondary nature used to supplement the main part of the research and provide further directions to look into. There will also be a small amount of grey literature included due to the fact that new information and approaches are constantly being released, such that some papers are recent enough to not have undergone peer-review and publishing yet, even though they may contain relevant information.
The sources of information will mainly consist of commonly known scientific platforms, such as Google Scholar[1], ResearchGate[2], arXiv[3] and the Fontys-provided search engine[4] as well as private repositories of individual researchers if their information can not be found on any of the aforementioned platforms. To acquire fitting literature, these platforms will be searched with combinations of the keywords and phrases shown in Figure 2, primarily within the areas of Artificial Intelligence and Machine Learning. The keywords were determined by a preliminary look into the mentioned areas and a subsequent gathering of common keywords from abstracts of papers roughly fitting the premise of the project.

1 https://scholar.google.com
2 https://www.researchgate.net
3 https://arxiv.org
4 http://biep.nu

```
Music Generation, LSTM, Time-Series, Conditional Generation, Drums,
Melody, Harmony, Algorithmic Composition, Generative Model of Music,
Machine Learning, Deep Learning, (Deep) Recurrent Neural Networks,
Feed-Forward Neural Networks, Autoencoder, Backpropagation, Grammar
```

Figure 2: Keyword-Cloud for gathering research information

To answer the research questions defined below, the following process will be applied:

1. Gather information (e.g. papers)
2. Apply constraints to reduce the search space
3. Create a short summary of each individual piece of information selected
4. Relate the information pieces to research questions
5. Answer the research questions by providing an overview of the found approaches and relate them to each other

The constraints that apply to this research are defined in Section 3.2 below. Once all research questions are answered, an approach will be synthesized that combines parts of these results to form an approach to solving the overarching problem within the specified requirements.

3.2 CONSTRAINTS

To restrict the search space of this research and thus the amount of information having to be evaluated, as well as ensuring only relevant information is more closely researched, requirements for the solution were instated that researched approaches will have to fit at least in parts to be considered for this project:

- Trainable on data that can be obtained from MIDI files
- Produces output that can be transformed into MIDI files
- Adaptable to different styles of music
- Can be used to generate either drum or melody tracks
- Necessary training data can be created from the data available to this project

The goal is to produce, for every sub-problem, a short list of available approaches that solve the given problem. In a second step, they are then evaluated in combination with each other to determine which path to take for the next chapter. The sub-problems will build upon each other, starting with base assumptions and growing in abstraction.

3.3 GENERATING MELODIES

A melody forms one of the most foundational compositional components of a musical piece, at its most basic simply consisting of a sequence of notes. In the context of composition, a higher-level structure is added, meaning a recurring arrangement of notes (a "Motif"), which is effectively a pattern in a sequence that repeats over time. This is especially important in music as the human brain has evolved to be tuned for pattern recognition of many kinds, which helps the brain infer structure and meaning from the sounds it perceives.
Thus, to emulate and eventually generate a melody, the generating component needs to possess knowledge of previous events, it has to know about time in the context of notes following after each other. During the advent of computer-generated music, this was often times achieved algorithmically, by encoding rules and structures of music theory into the program in the form of grammars (a more recent example of this being [QH13]). The concept of a musical grammar was in large parts inspired by the field of language processing, within which grammar governs how words are composed to form sentences, a concept similar to the composition of a musical piece in that smaller modules are composited into a larger whole based on specific rules. However this approach is tedious and inflexible, as a large number of constraints have to manually be encoded into a format understandable to a computer and are quite rigid in their application and the output they produce.
In the area of ML, Recurrent Neural Networks (RNNs) are an architectural model that integrates time as a feature the network can take into account. The most common form of such a network is the Long Short-Term Memory (LSTM) network (first proposed by [HS97]), which supplements the neurons that make up the network with an additional small bit of memory, enabling the network to recall previous events, even over longer periods of time. The focus here lies on the assumption that, given enough sample data, an LSTM network will, with sufficient training, take on similar properties as to what musical grammars would look like, saved in the state of the trained network and able to be replicated in slight variation from that state.
While earlier work in this area has focused on statistical models such as Markov Chains [DE10] and their combination with various optimisation algorithms and extended strategies [Her+15], over the past years approaches utilising methods of ML have started seeing success.
Most often encountered are methods utilising RNNs, often times in the form of LSTMs [Col+16] but also using Restricted Boltzmann Machines (RBMs) [BBV12], which take inspiration from Markov Models. Research in this area has found LSTMs to be a good fit for melody generation (e.g. in the evaluation done by [ES02]), as such networks are able to more successfully reproduce structure and style of their training data when compared to other approaches.
The representation of the training data varies wildly, even within the singular category of LSTMs, however the most common approach is to model sequences of notes as "words", inspired by natural language processing, which LSTM networks are known for performing well at [CFS16]. This produces approaches such as [Shi+17], which compare favorably with most other data designs. They

represent a note as a word expressing its four main features: position, pitch, length and velocity.
In a recently executed taxonomy of a multitude of music generation systems [HCC17] a general trend towards methods of ML is shown, especially when concerned with using LSTMs and MIDI data, which are the most common network structures and data types respectively.
Surveying the most successful approaches to date (based on comparisons contained within the initial paper proposing each architecture) which come with a reference implementation, a short-list of feasible approaches can be generated:

- Biaxial RNN [Joh17]
- JamBot [Bru+17]
- MidiRNN[5] (created by Brannon Dorsey)
- MelodyRNN (Google Magenta Project[6])
- PerformanceRNN (Google Magenta Project)
- PolyphonyRNN (Google Magenta Project)
- MusicVAE (Google Magenta Project, see [REE17])

The best performing approaches appear to all be based on LSTMs using a word-wise representation for individual notes and perform best when generating melodies in the range of several bars (4 - 16), as stated in their respective papers as well as based on an evaluation of provided sample output.
To assess generation quality, the reference implementation of each approach was trained on the aggregated dataset for this thesis using the default values for configuration, and the output compared. Quality factors that were assessed are:

- Adherence to Scale / Key
- Strength of Motif (repeating patterns in the generated sequence)
- Simplicity of Setup (time needed to setup and configure before training)
- Training Time (shorter is better)
- Little Repetition outside of Motif (excessive repetition of singular notes)
- Little Noise (dissonant, doubled, misplaced, open-ended or too many notes)
- Few stretches of Silence (long periods without any notes being played)

Each criterion operates on a scale of points ranging from zero to ten, multiplied by the given weight to assign different importance to several features. The sum of all criteria per model forms the final score. All criteria are expressed as positives, meaning that only addition is required and higher point scores

5 https://github.com/brannondorsey/midi-rnn
6 https://github.com/tensorflow/magenta

are generally better. Adherence to scale and motif are weighted the highest because stylistic replication is wanted and they form the largest contributors to generating "pleasant sounding" melodies, as will be discussed in more detail in a subsequent section.
The score for the time criterion is calculated inversely, starting out with ten points and deducting points if the time taken exceeds a certain threshold. The maximum runtime is set to 120 minutes, after which a run will be aborted if it did not finish. A model finishes early either by completing within its specific configuration parameters or by starting to achieve worse results (e.g. overfitting). The following formula is used to calculate the time score:

$$\text{score} = 10 - ((\text{minutes}/120) * 10)$$

All models were configured to run on the Graphics Processing Unit (GPU) of the testing machine (see Section 4.5.1 for its technical specifications) to achieve a significant overall speedup (in comparison to local execution) and emulate conditions similar to what the model would be expected to run under should it be chosen to be utilised for this project.

Criterion	Biaxial	JamBot	MidiRNN	MelodyRNN	Perf. RNN	Poly. RNN	MusicVAE	Weight
Setup	2	8	8	7	8	8	8	2
Time	-	8.3	-	5.8	-	4.2	3.3	3
Scale/Key	-	0	-	8	-	2	8	5
Motif	-	0	-	6	-	1	0	4
Repetition	-	10	-	6	-	10	10	1
Noise	-	0	-	10	-	3	7	3
Silence	-	10	-	10	-	8	5	2
	4	70.9	16	151.4	16	77.6	106.9	200

Table 1: Comparison of different music generation systems

Table 1 details the results of the model evaluation. It is important to note that several models did not finish the evaluation. Specifically, the Biaxial model turned out to be using the Theano ML framework, which is a currently unmaintained library and incompatible with the current versions of the `CUDA` and `cuDNN` libraries required to access the GPU, thus failing to start the training process. MidiRNN implements an extremely inefficient preprocessing step, which failed to complete in the allotted timeframe before even starting the training process, leading to its disqualification. Also failing within the preprocessing step, PerformanceRNN started ballooning the dataset to over 20GB in size, filling up the available space of the test machine and forcing the premature

abortion of the evaluation process. The other models completed the evaluation successfully, though they vary quite significantly in their score.
The best-performing model was determined to be MelodyRNN, based on the grounds that it excelled in the musical aspects of scale and motif-adherence with little repetition outside of the motif while exposing no additional noise or prolonged periods of silence. In contrast, the JamBot model generated excessive amounts of noise, as shown in Figure 3, with many notes layering atop each other. Such output leads to a very chaotic and unstructured sound, which in fact was quite difficult to even extract a melody from during listening runs.

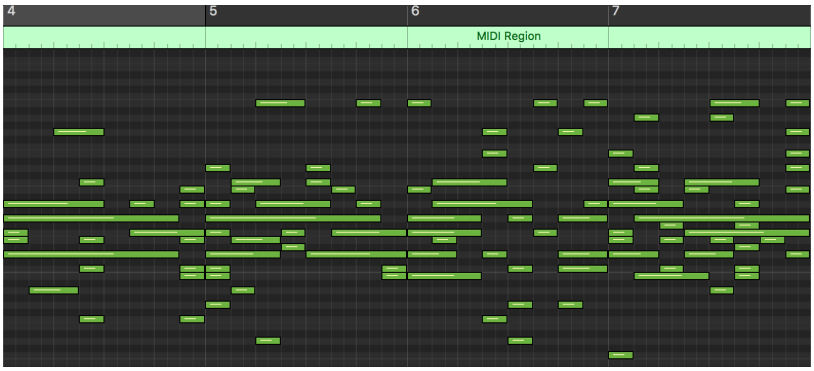

Figure 3: Noise in a generated MIDI file

PolyphonyRNN espoused noise in the same vein as the JamBot model, albeit in lesser quantities. It also showd slightly more structure, leading to a negligibly better score.
MusicVAE and MelodyRNN generated better results than the other models, especially in regards to scale adherence. Figure 4 shows the notes of the major scale in the leftmost part of the pianoroll over the two octaves C2 and C3. All keys were highlighted that were played at least once within this segment. As is clearly visible, they all conform to notes on the shown scale scale, showing that the model was able to successfully learn some harmonic relationships between different note pitches.

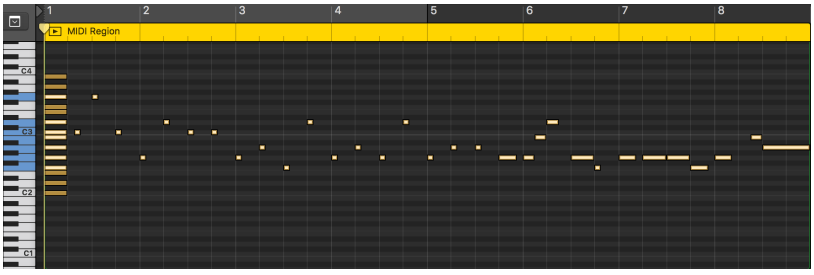

Figure 4: Adherence to the C-Major scale in a generated MIDI file

However, only MelodyRNN managed to also generate notes approximating a motif, and then repeating it more than once. As can be seen in Figure 5, a similar arrangement of notes is repeated twice, with slight variations and a different transition after each repetition, approximating some of the structures also found in songs composed by humans.

Based on the quality of the generated output and its ability to generate the most prominent motifs in particular, MelodyRNN was thus chosen as the model to base the generation of melodies on.

Figure 5: Repetition of a Motif (highlighted in green) in a generated MIDI file

3.4 REPLICATING STYLISTIC CUES IN MELODIES

A "style" of music generally refers to various features within a musical piece that commonly appear within one subset of music but are significantly less common within others. Pieces that exhibit similar features are grouped into a "style" of of music, and generally exhibit a similar overall sound. Common features that determine a style of music are:

Tempo	Chords	Instrument
Key	Scale	Pacing
Mode	Motif	Rythm

Table 2: Overview of stylistic factors within music compositions

In an attempt to automatically classify musical style [WS94], motif was determined to be an important factor, which may be closely connected to how humans distinguish different musical styles as well, given that we have a tendency to classify things based on patterns we encounter. [DZX18] meanwhile identify multiple levels of style within music, in contrast to the more extensively researched area of artistic style replication and transfer in images, and note that many different interpretations of style exist within the field of music generation due to its breadth and complexity.
Given that this thesis relies on MIDI data, which is inherently more malleable than audio data, tempo, key and instrumentation will be excluded from the defintion of "style" used in this thesis as they can be changed after the fact without affecting the other features (see Section C.3 for more detailed information about the MIDI standard). Focus will mainly be laid upon repeating motifs within generated segments as well as overall pacing and rythm, which generally have the largest impact on how music is perceived.

Approaches utilising ML have been shown to be able to replicate musical style simply by being trained on a sufficiently large corpus of stylistically consistent data [TZG17], taking on the correct features for rythm, scale and motif. Because of the way neural networks operate, a slightly more faithful representation can be achieved by slightly overfitting the network to the training data, however care has to be taken to not make the trained network plagiarise when overfit by a larger margin.
As such, if the proposed network architecture for a specfic model was not designed explicitly with enhanced style replication in mind, one has to rely on the models innate ability to learn such cues from the data it is provided with during training.

3.5 CONTROLLING THE GENERATION PROCESS

When making use of a neural network, control can be exerted at two stages in the process, once at the training stage, by modifying the hypermarameters and the training data and at the generation stage, by supplying different starting data and by tweaking the previously encoded parameters. To this end, several models implement conditional generation, which either refers to the implementation of some additional network architecture that is used to condition the main network during training or the supplementing of the input vector with additional features, which can then be used to steer the generation process later on (e.g. [Shi+17], who condition on song segments).
Common data used for conditional generation of musical sequences is higher-level, abstract information such as song segmentation (e.g. delimiting verse and chorus) or chord progressions, which can enable the model to generate specific motifs for some parts or chord progressions but not others, as dictated by the training data (as seen in [TZG17]). The drawback with this method is that to exert more control over the generated output, more feature-data has to be provided during training, which in some cases might be difficult to come by, depending on the source of the data.
In the case of this thesis, the available dataset was scraped from the internet and thus does not provide any additional metadata that could be used for such conditioning, save for heuristically extracting it from the available MIDI files, if at all possible.
The only other way to affect the output of a model is to utilise post-processing methods which can operate on any kind of MIDI sequence. Most of these methods are independent from the method used to generate the initial sequence and are mostly algorithmic in nature. For MIDI sequences especially, a large amount of audio plugins exist that provide a multitude of transformations of such sequences (e.g. transposition, channel splitting or merging).
However there also exist some specialised solutions, e.g. [Jaq+17], who propose an adversarial network that can be used to improve the originally trained network after the initial training has completed, using refinement methods incorporating aspects of encoded music theory.
As such, the only way achieve more control over the generated melodies is to either make use of algorithmic post-processing or to utilise a model that was

designed with a specific condition from the start. Feasible models that exist at the current moment are:

- MusicVAE
- JamBot

Because MusicVAE has to be trained on lead sheets for chord conditioning, it is not possible to use it in this project, since only MIDI data is available, which does not contain the required information. JamBot heuristically extracts these features from MIDI files, which is not very accurate and would need more training data to produce good results. As such, the aforementioned features will have to be added during the post-processing phase.

3.6 GENERATING A "SONG"

Within music, several layers of abstraction exist on which structure can be found. This was highlighted quite fittingly in [HCC17], who describe three layers of abstraction within the compositional process (see Figure 6 for a graphical representation):

- Physical: The actual physical frequency of a note that is emitted by an instrument.
- Local Composition: The rythm, melody and motifs contained within several bars of music.
- Full Composition: The composition of multiple distinct parts from the previous layer into a bigger whole. This is often referred to as "song-structure".

In its most simplistic form, a composition is often described by denoting parts by letters of the alphabet, creating a structure such as `AABA`, with distinct letters denoting different parts of the song. The previous example is a commonly used structure found in many songs of american pop culture.
Figure 6 highlights two important areas where structure has to be created, once on the bar-to-bar level (within melodies) and overarchingly within the composition of the melodies into an entire piece. Given that LSTM networks in their current capacity excel at generating structure within melodies, but fall short over longer periods of time if not provided with manual hints of the intended structure (producing meandering melodies without direction), long-term structure generation will have to be supplemented by another approach. To this end, both algorithmic and AI-inspired approaches are possible. As discussed in Section 3.5, some models are built to factor in features like segmentation data, enabling them to replicate melodies that were found to be more common for one part in a song than another, however such options are severely limited and require such data to be available in the first place. Especially the task of extracting segmentation data is a hard problem a lot of time has been dedicated to in the Music Information Retrieval (MIR) area of research, and existing approaches fall short of human-provided segmentation data by a large

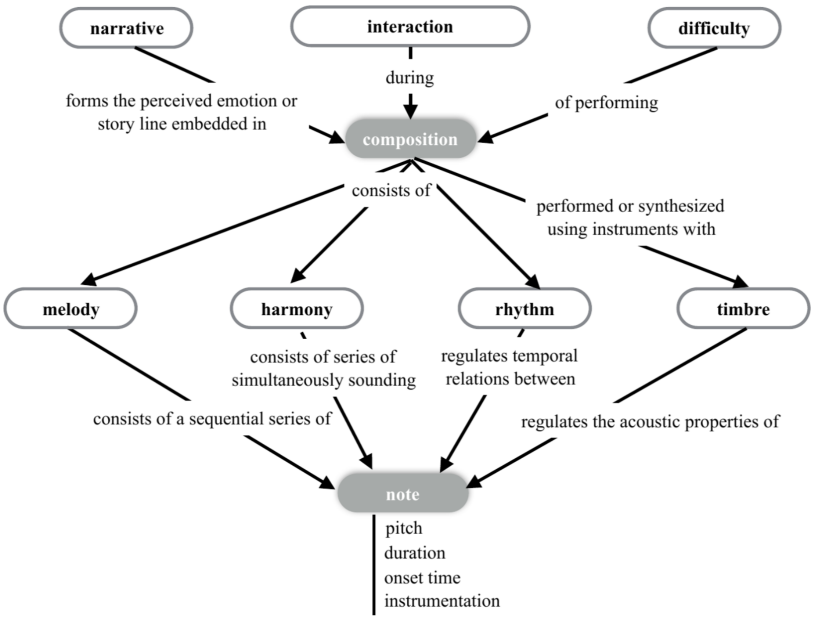

Figure 6: Concept map for an automatic music generation system (graphic created by [HCC17])

margin (as shown in an evaluation of human vs. algorithmic performance by [Ehm+11]).
Another option would be to introduce a second layer of abstraction, similar to Figure 6, which is simply concerned with the generation and composition of multiple melodies that are provided by the layer below. The way in which melodies are composited could be determined either by a secondary model that is able to generate song structures, or via an algorithmic solution similar to a Markov Model. However the combinations of song structures that are commonly used are a rather small subset of all possible permutations, so manually inputting a specific song structure to base generation on could also be a viable avenue.

3.7 INTEGRATING MULTIPLE INSTRUMENTS

If multiple instruments are to play together at the same time, they have to share a common baseline of information that enables them to sound pleasing to the human ear when played at once (see Section C.2 for more detailed information). To achieve a harmonic sound, several key pieces of information are required:

- Mode
- Key Signature
- Time Signature
- Scale (dependent on Key and Mode)

The way in which to integrate multiple instruments with each other differs depending on the solution used to generate the melodies. Since most models

only predict one line of melody per run, generating multiple lines and merging them together will only result in a pleasant sounding mix if the generating model has learned to conform to the aforementioned properties and can faithfully replicate them in a similar manner in repeated runs.
Should this not be the case, another option is to post-process the output after the tracks for all required instruments have been generated, transposing all emitted notes to the same key to ensure harmonic integrity.
In a second step, instruments playing at the same time may interact with each other. For example, a drum track might accentuate at all timesteps where another instrument plays a certain note. This kind of integration between instruments closely mirrors what might appear in a "Jam-Session" between multiple musicians, but also what is intentionally brought about by a composer when arranging multiple pieces for a song.
The only model with a refernece implementation currently able to generate multiple lines of instruments at the same time is MusicVAE [REE17], which can generate melody, bass and drum lines in conjunction with each other. This method allows the model to learn dependencies between different instruments, which not only enables it to create a pleasant-sounding mix but further allows for the interaction between different instruments (e.g. coordinated pauses and accented drums or melody). [Mak+17] propose a similar approach, with a stronger focus on bass and drum interaction.

3.8 PUTTING IT TOGETHER

Due to the fact that no approach exists that fulfills all the requirements of this project at once, the eventual solution will have to be composited from several parts. As such, the previously found architectures will have to be cross-evaluated and compared with one another to find an arrangement of approaches that works well together and achieves the overall goal. The eventual aim of this solution is to generate a "song", referring to a musical piece with several high-level segments that repeat while incorporating multiple instruments.
As shown in Section 3.3 and Section 3.4, it is possible to generate pleasant sounding melodies for individual instruments for medium length timespans in the 4 - 16 bar range. However utilising the same approach for generating melodies with clearly distinguishable segments for a song of several minutes will likely not produce good results (see Section 3.6). Thus the high-level structure has to come from a different source.
Given that the scraped dataset does not make available data on high-level structure, and it is exceedingly difficult and error-prone to extract it heuristically (see Section 3.5), it can not be generated via a ML model. If manually-annotated data would be available (e.g. at a later point in time or from a different dataset), such a model could be trained and this part of the input replaced by automatic generation from said model. Given that the variations would be limited to a small set, a simple markov model or RBM would suffice.

At this point in the generation process, multiple melodies can be generated and arranged together, however they have a high chance of sounding displeasing because of harmonic interference (dissonance).
This problem can be addressed by a few models that are able to generate multiple instruments at the same time, which then fit together naturally because they were generated from the same probability distribution at the same timestep. However the results vary wildly, and thus such models will not be used for this project as they produce inferior results when compared on the level of individual instruments (see Section 3.7). Because this problem can not be solved at the generation level, it has to be addressed in a post processing step, which should shift all occuring notes onto a valid subset of possible notes determined by the chord progression.
Similar to the high-level structure, no data on chord progressions is available from the scraped dataset, meaning that they will have to be manually inputted as long as such data is lacking. Should data for this become available, a similar solution as was proposed for the structural problem can be applied, in that such progressions could be generated by a markov model or RBM trained on the chord data.
Under consideration of the above, the MelodyRNN model was chosen for generating the melodies of the lead and bass lines. Since an equivalent model exists for drum lines (DrumsRNN), which slightly changes the data representation for training but otherwise functions in the same way as MelodyRNN, it was selected for the drums line. A beneficial factor to choosing these models is the fact that they are provided by the Magenta project, which provides an ecosystem of supporting libraries and individual classes around the selected as well as several additional models, making them easier to work with.

As such what the eventual software package must do is the following:

- Be provided with the high-level structure for a song: chords and segments
- Per segment, generate the required instrument lines
- Condition generated notes on the chord progression
- Output the generated music to either a synthesizer or a file
- Utilise the “MelodyRNN” and “DrumsRNN” models from the Google Magenta project
- Generate three instrument lines: Melody, Bass, Drums

These findings will have to be taken into account when moving into the implementation phase in the following chapter.

4 ARCHITECTURE & IMPLEMENTATION

Based upon the findings in the previous chapter, an application architecture will be devised that achieves the previously determined goals within the specified requirements. Additionally, this chapter will detail the creation of the software package, elaborating on specific code snippets for complex parts and highlighting the most important sections within the codebase. It will also detail the training process.

4.1 SETUP & ENVIRONMENT

Due to the fact that all of the models appearing in this thesis are implemented in Python[1] (provided source code was available), it was chosen as the main language for implementation of the rest of the system for a maximum of interoperability. Python is a flexible, interpreted programming language with support for many different styles of programming, as well as being easy to learn and use, which has led to its widespread adoption within the scientific community. Because of the latter, many packages for scientific computing are available (e.g. SciPy[2]), simplifying development in these areas and enabling rapid development of ML-related programs.
In conjunction with this, since models from the Magenta[3] project were chosen for the generative part of the system, TensorFlow[4] is implicitly used as the backing framework for the implementation of these models.
Given that very little user input or interaction is needed for the basic operation of the program, a complete Graphical User Interface (GUI) was foregone in favor of a terminal-based interface utilising the Urwid[5] library for providing a simple TUI instead.
Because ML models require large amounts of computational resources that could not be provided locally, Google Cloud Platform (GCP) was chosen as a provider for remote computational capability for training the selected models. Resources provisioned on GCP are relatively cheap, in addition to Google providing a 300$ starting credit upon first registration, which made GCP an attractive and eventually the final choice for this project against its contender Amazon Web Services (AWS).

1 https://python.org
2 https://scipy.org
3 https://magenta.tensorflow.org
4 https://tensorflow.org
5 http://urwid.org

4.2 CHOSEN APPROACH

Before the implementation of the generative framework begins, the individual models will be trained to a point of satisfying performance (see Section 4.5 for detailed information on training and evaluation of the selected models). Once an acceptable performance is reached, the trained models will be exported as `GeneratorBundle` files, a custom TensorFlow format for storing the weights of a network along with some metadata describing the hyperparameters used for training and some general information about the network architecture, enabling another instance of TensorFlow to load the model back into memory and provide a programmatic `SequenceGenerator` interface for it.
Once the bundle files are available, implementation of the generation framework can begin, to enable the loading of all three models and the generation of MIDI data from them. The basis of the application is built upon another part of the Magenta project, the `midi_interface` module, which exposes a `MidiInteraction` class with several capabilities that ease implementation of the requirements. Specifically, it is capable of loading a `GeneratorBundle` and generating music from it in an interactive, semi-real-time way. The original intent of this class was to provide call-and-response interactivity between a human and a Magenta model. It enables the input of MIDI Control Change (CC) events and can generate output based on those events, enabling human and machine to take turns playing to each other. This specific functionality is not required for this project, however the basic features of this class were used as a baseline for the implementation. An additional advantage to using this class as a foundation is the fact that it exposes I/O in the form of virtual MIDI ports, which make it very easy to integrate this application into an existing workflow and various professional applications, such as Digital Audio Workstations (DAWs), plugins and MIDI-controlled or -emitting hardware, which can work in real-time via transport over these ports.
As such the final task is to extend the `MidiInteraction` class to accept multiple models and integrate high-level structure. Supplementing this, a post-processing chain for chord conditioning and recording to disk is to be implemented via additional software modules.

4.3 SOFTWARE ARCHITECTURE

The following details the basic architecture of the final application. Figure 7 shows a high-level overview of the packages and classes that exist within this project, as well as how they are composited. All main functionality was encapsulated in the `ComposerManager` class, which is administrated by the `TerminalGUI` class which provides the TUI. This design is frontend-agnostic, meaning that the TUI could be replaced with another GUI framework without impeding the functionality of the base application.
The `ComposerManager` class is the central hub of the application that is responsible for compositing various smaller parts together and to administrate their configuration and lifecycle from start to finish. Specifically, it contains the entire signal chain that determines how the MIDI signal is routed from the gen-

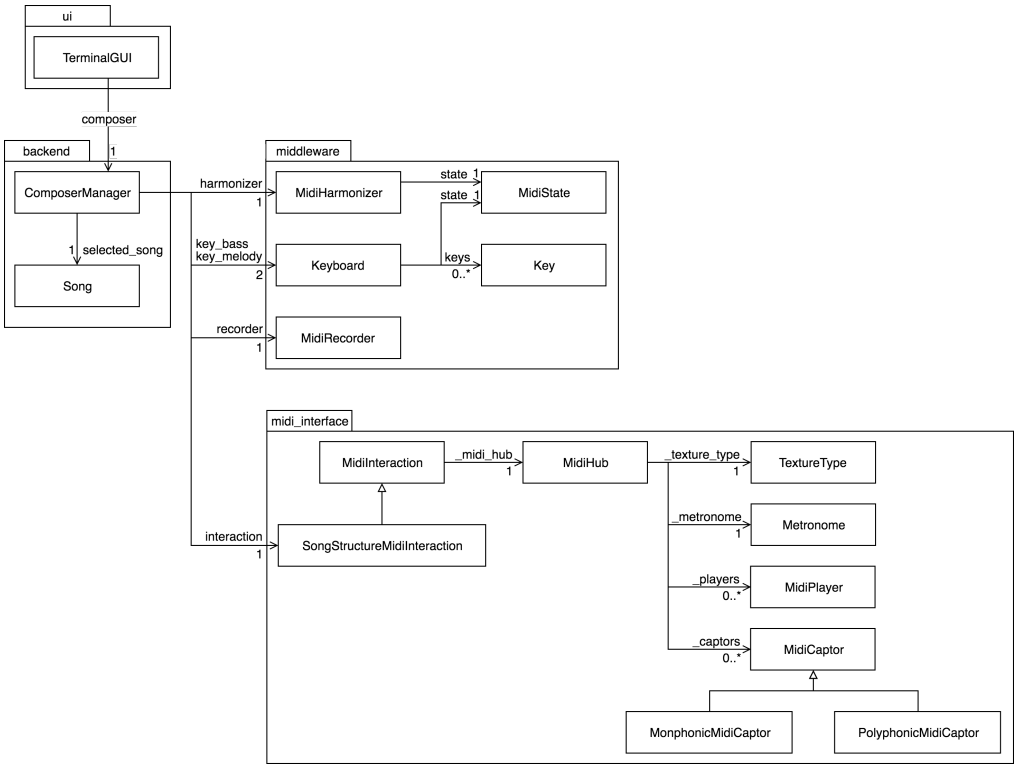

Figure 7: Class diagram detailing the application architecture

erator to the output (see Figure 8). The middleware class `Harmonizer` exposes a callback which sends MIDI events to a virtual keyboard, which itself is used in the TUI to display the currently playing notes. At the same time it relays incoming messages to the `Recorder`, which saves them to disk in the correct format while also relaying the messages to an external synthesizer to produce actual sound.

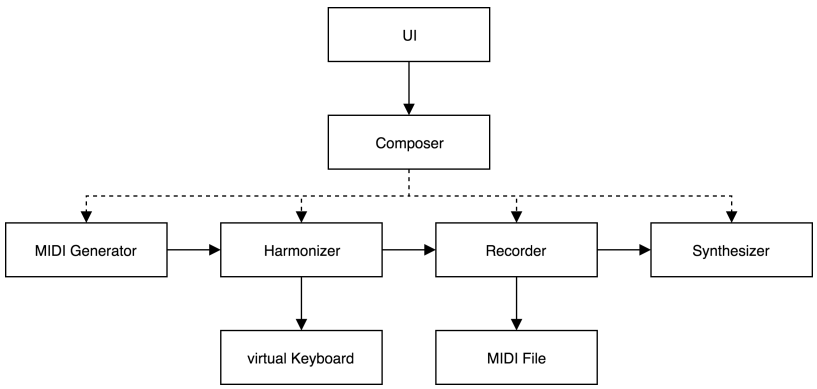

Figure 8: Schematic detailing the routing of MIDI signals through the application

The aforementioned architecture was chosen because while during generation time the signal flow is quite static (in that it follows a specific path of operations), during intialisation time several pieces of information have to be prepared and distributed to the correct objects, most notably the information about song structure and chord progressions, as well as dependencies

for the main objects. Thus the `ComposerManager` was created to administrate the already-created signal-flow within the program.
While the classes within the `middleware` module are standalone, without extra dependencies, the `SongStructureMidiInteraction` class inherits from the base-class `MidiInteraction`, which is a composite class of several smaller pieces that enable the functionality initially discussed in Section 4.2.

4.4 PROGRAM FLOW

The application is mainly event-driven, as actions occur mostly in response to MIDI events being emitted. For the sake of modularity and proper seperation of concerns, the different operations within the signal-chain were split into threads. These threads do not interact with each other directly save for being administrated by the `ComposerManager`, instead they each open a MIDI port for input and output and react to events on those ports. As such, each thread is self-contained and stateless, making threading very easy to accomplish.
The program exposes nine threads with differing responsibilities:

1. Main Application Manager, User Interface (UI)
2. `SongStructureInteraction`
3. `Harmonizer`
4. `Recorder`
5. `MidiCaptor`
6. `MidiPlayer` x 4 - (Melody, Bass, Drums, Chords)

Since these threads are predominantly stateless and self-contained, their internal flow will be described independently of each other. For the purpose of brevity, only the two most important threads will be elaborated on here.

4.4.1 *Main Application Manager, User Interface*

This thread supplies the entrypoint of the application. It is responsible for setting up the UI and responding to events emitted by it, to dispatch them to the correct threads if necessary. Figure 9 shows the general flow of the application in the style of a sequence diagram.
On startup, during the initialisation step, a `ComposerManager` instance is created, which is the main coordinator for the different threads and the signal flow. The manager loads the pre-trained models from the `models/` directory on initialisation and makes them accessible as `SequenceGenerators` which are then passed to the actual generating interaction as needed to prevent loading them more than once.
Directly afterwards, the `songs/` directory is scanned for available song definitions (described in more detail in Section 4.6.2) so they can be shown and selected in the interface.

Once initialisation completes, the main thread sleeps until an event is emitted from the UI, which it then dispatches to the `ComposerManager`. The main events are `start()` and `stop()`, which govern the state of the generative part (whether notes are emitted or not).
Once the `start()` signal is given, the manager initialises a `SongStructureMidiInteraction` with a song given by the UI, a `MidiHarmonizer` and a `MidiRecorder` with proper MIDI I/O port configurations such that signals flow according to Figure 8. The manager also registers a callback on the Harmonizer, which emits an event every time a note is relayed, which in turn is used to keep track of currently active notes for the purpose of showing them on a virtual keyboard (the resulting graphic can be seen in Figure 11). After this, the MidiInteraction takes over and starts generating notes.
If the `stop()` signal is given, either from the user quitting through the UI, aborting via `SIGINT` or the application finishing its run to the end of the defined song, a termination event is sent to all active threads, which terminate as soon as their current iteration completes. The `ComposerManager` deletes all stopped threads and dereferences them for garbage collection, as stopped threads can not be restarted. Upon receiving a `start()` signal, the threads always newly initialised.

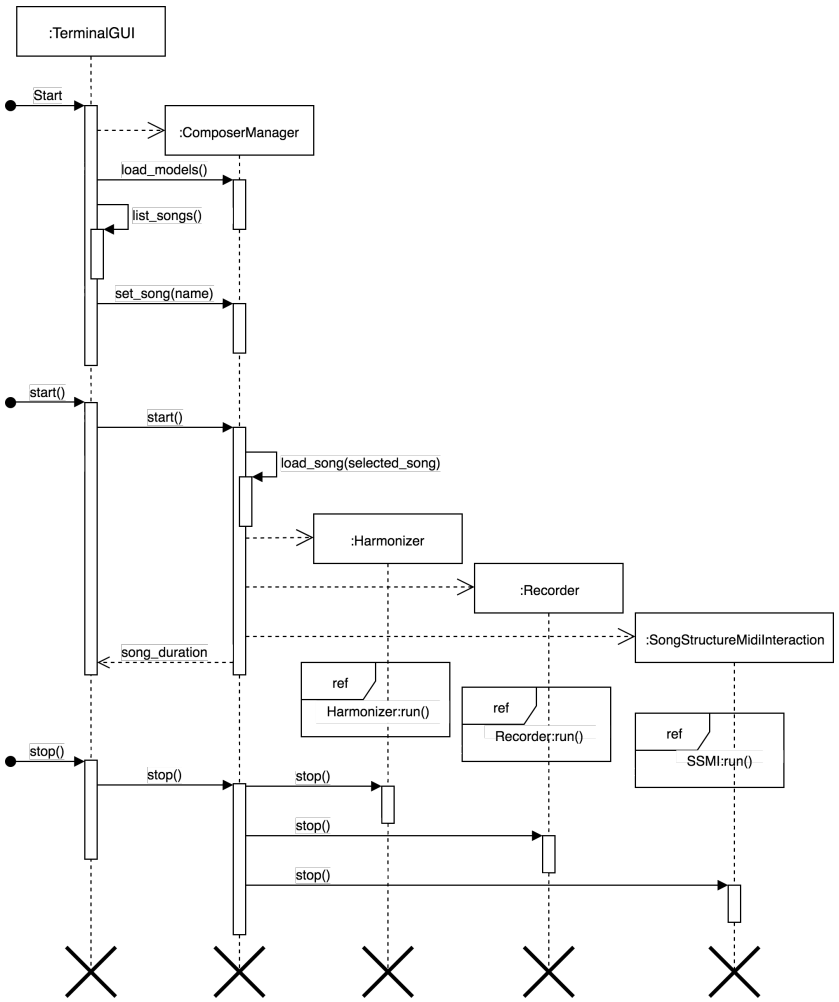

Figure 9: Application flow of the main thread

4.4.2 *The MIDI Generator*

The class `SongStructureMidiInteraction` is based on the `MidiInteraction` class originally provided by the Magenta project, however it replaces most of the implementation with custom code tailored for the specific purpose of generating segmented melodies, each of a few bars in length. Playback of generated MIDI events is achieved through a `MidiHub`, a Magenta-provided class that manages any number of `MidiPlayers`, which themselves are capable of emitting such events to a virtual or hardware-provided MIDI port on a specific channel (to easily separate the events downstream). The hub is instantiated at initialisation time and persists until the thread terminates.
The thread itself runs in a loop, firing once for each MIDI tick. On every iteration, the current part of the song is calculated (in bars). If a new part is reached, new instrument lines are generated and sent to the `MidiPlayer` instances for the respective instrument. Additionally, all generated lines are cached so they can be recalled should the same part of a song be encountered again. Should this be the case, the lines are retrieved from the cache instead of being newly generated.
This loop continues until the end of the song is reached, at which point the thread terminates, dereferencing the `MidiHub`, which automatically stops all `MidiPlayers` upon being garbage collected.

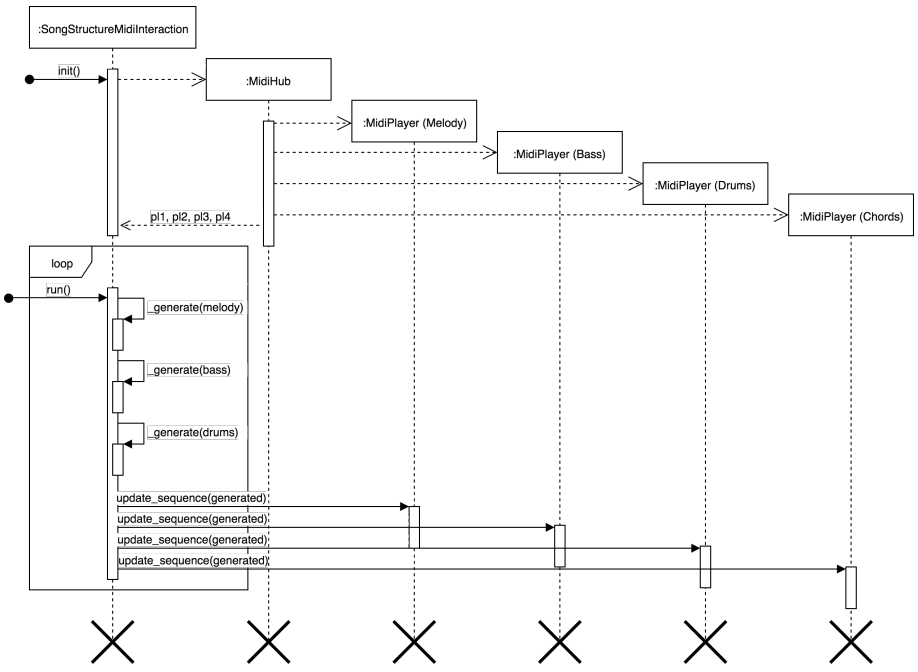

Figure 10: Application flow of the `SongStructureMidiInteraction` class

4.5 TRAINING PLAN

To enable training of the chosen models, the datset (see Appendix B for the intial creation process) has to be converted into a format the models can understand using a two-step process. Initially, all MIDI files are converted into

a special, optimised container format (`tfrecord`), which is a Magenta-specific representation of MIDI files that is easier to work with internally (see Snippet 1).

```
convert_dir_to_note_sequences \
    --input_dir=melody/ \
    --output_file=melody.tfrecord
```

Code Snippet 1: Command for converting MIDI files into a `tfrecord` container

In a second step, this container has to be converted to a special sub-format precisely tailored for one specific model, meaning that two variants have to be created, one for the two `MelodyRNN` instances and one for the `DrumsRNN` instance. The models provide a small script for converting a `tfrecord` container into the required format, which makes the process very simple and executable in one command, as can be seen in Snippet 2.

```
drums_rnn_create_dataset \
    --config=drum_kit \
    --input=drums.tfrecord \
    --output_dir=drums/ \
    --eval_ratio=0.2
```

Code Snippet 2: Command for converting a `tfrecord` container into the required sub-format for the `DrumsRNN` model

4.5.1 *Configuration*

To run these models, a Virtual Machine (VM) was provisioned in GCP to provide a consistent and powerful base for execution. Since both, Magenta and TensorFlow provide the ability to execute on a GPU, which is much faster than a typical CPU, a NVIDIA Tesla GPU was provisioned as the main processor. The specific configuration was set as follows:

- Debian 9
- 4 CPU Cores (Broadwell XEON)
- 16GB RAM
- 1 NVIDIA Tesla P100, 16GB VRAM

Because GPUs are limited via quotas on GCP, they have to be manually requested and the quota increase confirmed by a Google employee. Because of this only one GPU was available for training, but for larger workloads additional quota increases could be requested. Additionally, the following software packages were installed as dependencies for the required software:

- NVIDIA Linux drivers (`390.30_x64`)
- NVIDIA CUDA (`7.5.17`)

- NVIDIA cuDNN (`7.1.2.21-1`)
- Python (`3.5.3`)
- magenta-gpu (`0.3.5`)
- tensorflow-gpu (`1.6.0`)

4.5.2 *Data*

Datasets per model were split 80% - 20% for training and evaluation respectively. The raw dataset contains roughly 1000 MIDI tracks of varying lengths, of instruments found to be playing relevant notes to either melody, drum or bass parts. Appendix B contains more details on how the collected data was prepared for training and what data was used for each model.

4.5.3 *Training*

Models were trained sequentially, one at a time, since they will make use of as many resources as are available. As such, one training job was executed in tandem with an evaluation job, which is similar to the former but uses a held-back dataset for testing and does not modify the weights of the model (see Snippet 3 for an example of the used commands). A compilation of all the specific commands used for the different training runs can be found in Section C.4.

```
drums_rnn_train \
    --config=drum_kit \
    --run_dir=run/ \
    --sequence_example_file=drums.tfrecord \
    --hparams="batch_size=64,rnn_layer_sizes=[256,256]" \
    --num_training_steps=2000

drums_rnn_train
    --config=drum_kit
    --run_dir=run/
    --sequence_example_file=drums.tfrecord
    --hparams="batch_size=64,rnn_layer_sizes=[256,256]"
    --num_training_steps=2000
    --eval
```

Code Snippet 3: Commands for training and evaluating the `DrumsRNN` model

Monitoring of the training progress was achieved via TensorBoard, a standalone supplementary application that works in conjunction with TensorFlow to provide a web-based GUI for viewing statistics about the current run of the model (executed via `tensorboard -logdir run/`).
All models were run for a maximum of 2000 steps, each taking about 40 minutes per run at a rate of slightly less than 1 step per second. The runtime of

course is dependent on the configuration of the model, e.g. a larger network size would take a larger amount of time. In this configuration, all models were run with a two-layer configuration of 256 nodes each, with a batch size of 64 elements per step. The batch size had to be reduced from the default of 256 elements because not enough VRAM was available on the GPU to be able to stem this amount of data. Models were prematurely stopped if overfitting was observed (evaluation and training graphs starting to diverge).

4.6 IMPLEMENTATION DETAILS

The models used in this project are custom implementations based on an underlying basic LSTM created in TensorFlow. The Magenta project has expanded upon the basic architecture by introducing tweaks specific to music reproduction, specifically "lookback" functionality. This method augments the input data of the network with events not immediately preceding the current one, but from one or two bars ago. Additionally, the position within the bar is passed. This enables the model to more easily recall events from within a range of 2 bars before the current note, allowing it to more easily repeat structures it has generated before.

4.6.1 *Licensing*

Because two parts of the created software package are supporting libraries created by someone else, under different licensing conditions, it was necessary to take precautionary steps in order to comply with the license requirements of these packages.
Mingus[6], a Python-package specialising on programmatic access to chord- and music-theory, was chosen to provide support for parsing and modifying chord symbols, specifically for the chord progressions that have to be provided by the user. Unfortunately it is not being maintained since 2015 and does not provide support for Python 3, the target platform of the main application. Thus, some changes in the source code were necessary to port the package to the new version of Python. Since Mingus is GPL-licensed, it requires the distribution of the original source code as well as the documentation of any changes made to said code.
In a similar vein, the `midi_interface` module of the Magenta project, which supplies the base class for MIDI-emitting generators of trained TensorFlow models, is distributed under the Apache license, which, similar to GPL, requires the documentation of changes to the original source code. The Apache Software Foundation (ASF) and the Free Software Foundation (FSF) state that the Apache v2 and GPLv3 licenses are compatible[7], provided the packages that use them in combination are redistributed under the GPL v3 license[8].
Thus, to comply with the license requirements, the original files of each package were committed first and then the changes were applied in separate com-

6 https://github.com/bspaans/python-mingus
7 https://www.gnu.org/licenses/license-list.html#apache2
8 http://www.apache.org/licenses/GPL-compatibility.html

mits so that they can be seen clearly. Additionally, each package was put into a distinct folder with their own license included, to correctly delimit the proper licensing areas. The project itself is distributed under the GPLv3 license, as required by the usage of the Mingus package.

4.6.2 *Song Structure File Format*

Since song-level structure has to be provided from an external source, the application needs a way of ingesting such information in a format that is both easy to parse as well as easy to generate and read. The format has to be able to convey two pieces of information, namely information about song segmentation and information about chord progressions. As such, a simple CSV-inspired format was chosen. Snippet 4 shows a simple example of the final structure.

```
1 INTRO, I, I, V, IV
2 CHORUS, I, V, IV, V
3 VERSE, I, IV, V, IV
4 CHORUS
5 VERSE
6 OUTRO, V, V, IV, I
```

Code Snippet 4: Structure of the `.sng` file format

In this format, each line defines one segment of a song. The first value is an identifier which denotes distinct parts of a song. This is used for caching previously generated instrument lines, which are recalled when this identifier is encountered again. The values after this identifier are chords in the roman numeral format (`I - VII`). Each chord is played for one bar before switching to the next one. If the list of chords for a segment is exhausted, the next segment is played. The amount of chords is not limited. If a segment does not have any chords listed, they will be inferred from an earlier definition with the same identifier. This reduces redundancy in the notation, but because of this, the first identifier encountered always has to list a chord progression.

4.6.3 *Harmonising Multiple Instruments*

As determined in Section 3.7, since harmonisation can not occur at generation time, it has to be added at a later stage. The `MidiHarmonizer` class fulfills this purpose. It is entirely event-driven, as it responds to individual MIDI messages that are received via a set of MIDI ports. Events are filtered by channel and type, relaying all events that do not have to be modified directly to the output. All events that arrive on the melody or bass channels (1 and 2 by default) and are of the type `note_on` or `note_off` will have reharmonisation applied to them.
For this reharmonisation to take place, a chord is required to harmonise the other notes on. Because the `SongStructureMidiInteraction` emits chords per bar, the Harmonizer can listen on an additional channel (3 by default) for those chords and utilise them for the reharmonisation process.

The basic premise of harmonisation in this context is that there exist two types of notes that can be played: safe notes and unsafe notes. The former is a subset of all possible notes that sound pleasant with the current chord, while the latter is the inverse of that. Once a set of safe notes has been determined, any possible note can be transposed into this set to enable harmonic interplay between multiple instrument lines.
Music theory as well as logic dictate that every note within a chord is harmonically safe to play, given that it is already part of the chord. Hence, to determine a rudimentary set of safe notes, the current chord is normalised to the lowest possible position, which is the chord being played in octave zero. From there, it is transposed up to all possible octaves. All the individual notes of the chord in each octave are then aggregated into a sorted set and form the most basic set of safe notes (see Snippet 5).

```
octaves = list(range(0, 127, 12))

lowest, count = min(chord), -1
while lowest >= 0:
    count += 1
    lowest -= 12

mpd_ovr_rng = [[e - (12 * count) + octave for e in chord]
                      for octave in octaves]
```

Code Snippet 5: Generation of the set of transposed chords

At this point, a secondary problem surfaces, since it may be the case that multiple half-steps lie between one safe note and the next. Simply transposing incoming events to the next note would result in several different keys mapping to the same note, destroying the melodic movement in the original instrument line. Because of this, it is preferable that every possible movement of any size results in a movement within the set of safe notes.
To achieve this, the set of safe notes is split at the center octave where chords normally take place, which is the fourth octave in the case of this project. This creates two subset of safe notes. The set with the safe notes of the lower half is destined for movement in a negative direction, while the other set is set for positive movement. Any movement (given as an integer denoting a half-step, either positive or negative in the valid note range -127 - 126), can now be used as an index into either of these sets, depending on whether it is positive or negative. As such, any movement now results in a movement within the set of safe notes. In addition to the chord notes, the notes of the major scale are also added to these sub-sets. Snippet 6 shows the slicing and addition of major-scale notes of the note sets.
To calculate the movement of the instrument line, a center octave is set where melodic lines normally take place, which is the eighth octave. The C of that octave is used as the centerpoint for the instrument line. The difference is calculated by subtracting the key value of that note from the actual key value of the incoming event, resulting in a positive or negative difference from the

```
middle_octave_chords = 4
middle_octave_melody = 8

negative = SortedSet([e for l in mpd_ovr_rng[:middle_octave_chords]
                      for e in l])
negative.update([e for l in major_notes[:middle_octave_chords]
                 for e in l])
positive = SortedSet([e for l in mpd_ovr_rng[middle_octave_chords:]
                      for e in l])
positive.update([e for l in major_notes[middle_octave_chords:]
                 for e in l])
```

Code Snippet 6: Splitting of safe notes into positive and negative movement sub-sets

centerpoint. Snippet 7 shows the process of calculating the difference and determining the new key value for the event. The value is clamped twice, as it can over- or undershoot the valid range at two points in this process.

```
diff = note - octaves[middle_octave_melody]
diff = max(-len(negative), min(diff, len(positive) - 1))

if diff < 0:
    note = negative[len(negative) + diff]
else:
    note = positive[diff]

note = max(0, min(note, 127))
```

Code Snippet 7: Difference calculation, clamping and note harmonisation of a MIDI event

4.6.4 *Recording to Disk*

Due to the structure of the application, MIDI events are simply relayed through a chain of objects that apply different processing steps to them if applicable, ending in the sending of the event to the last output port, which by default is a synthesizer. In this project specifically, `fluidsynth`[9] was chosen as it is the most well-known synthesizer, has cross-platform support and allows for the use of different sound-fonts (sound libraries with different sound samples). In this case, the "General User" font by Christian Collins[10] was used for synthesizing the real-time output. To persist the generated songs to disk, a middleware plugin is required that is able to save MIDI events that pass through to disk. This functionality is provided by the `MidiRecorder` class.

9 http://fluidsynth.org
10 http://schristiancollins.com/generaluser.php

The Recorder uses the `mido`[11] package to transform MIDI events into their binary representation. Similar to the Harmonizer, it listens for events on an input port. Once an event arrives, a copy of it is sorted into one of four tracks (a list of MIDI events), corresponding to the channels emitted by the generator at the beginning of the chain. The original event is relayed to the output port without any modification. Once the application terminates, the thread shuts down and - upon stopping - writes the saved tracks out to disk in their binary format, adding some metadata to the start of the file to form a proper MIDI file.
During development of the application, a problem was discovered related to the timing of relayed MIDI events. As it turns out, sending such an event through a virtual MIDI port (either input or output), removes the timing information, which is vital in being able to properly place the events in a MIDI file, as otherwise all events end up at the same timestep. To resolve this, the Recorder re-times incoming events itself, with a resolution of at most 1/100th of second. Since absolute time is not needed, the first run of the `time()` method is cached and all following measurements are interpreted as relative to the first measurement.
Because MIDI files represent time as ticks instead of seconds or a similar unit, the relative time has to be appropriately converted. Timing in MIDI files is dependent on the tempo of the song (Beats Per Minute (BPM)) and the Pulses Per Quarter Note (PPQ). The latter defines the resolution of the file, similar to the sample rate in raw audio files like the `MP3` format, which in combination with the BPM results in the actual time value. The conversion is handled by a `mido`-provided function for converting seconds to ticks.

```
1  with localcontext() as ctx:
2      ctx.rounding = ROUND_DOWN
3      tm = float(Decimal(time()).quantize(Decimal("0.001")))
4      if not self.first_time:
5          self.first_time = tm
6          tm = 0
7      else:
8          tm -= self.first_time
9      tk = int(second2tick(tm, 480, bpm2tempo(120)))
10     msg.time = tk
```

Code Snippet 8: Restoration of the time attribute of incoming MIDI messages

During the actual saving of the file, all times are converted to delta-timings, which refers to translating the absolute timings (relative to the start of the recording) that were initially saved in Snippet 8 into relative timings between individual notes. This allows for easier modification of the MIDI file as tempo changes can simply add or remove specific amounts of time to or from the deltas of each note (see Snippet 9) instead of having to recalculate the time for each note individually.

11 https://github.com/olemb/mido

```
midi_file = MidiFile()
for track in self.tracks:
    t = MidiTrack(_to_reltime(track))
    midi_file.tracks.append(t)
midi_file.save("recording.mid")
```

Code Snippet 9: Conversion and flushing of in-memory tracks of MIDI events

4.6.5 *User Interface*

The user interface is a relatively simplistic frontend since very little user interaction is required. However it does contain several convenience functions that will be explained below.
Due to the GUI being a low priority feature, it was decided that a simple TUI would suffice. A terminal-centric, text-based interface is very portable, uses little screenspace and is efficient. Additionally, it is simple to construct and use. Figure 11 shows the main (and only) screen of the application, which contains all the elements required for controlling the application.

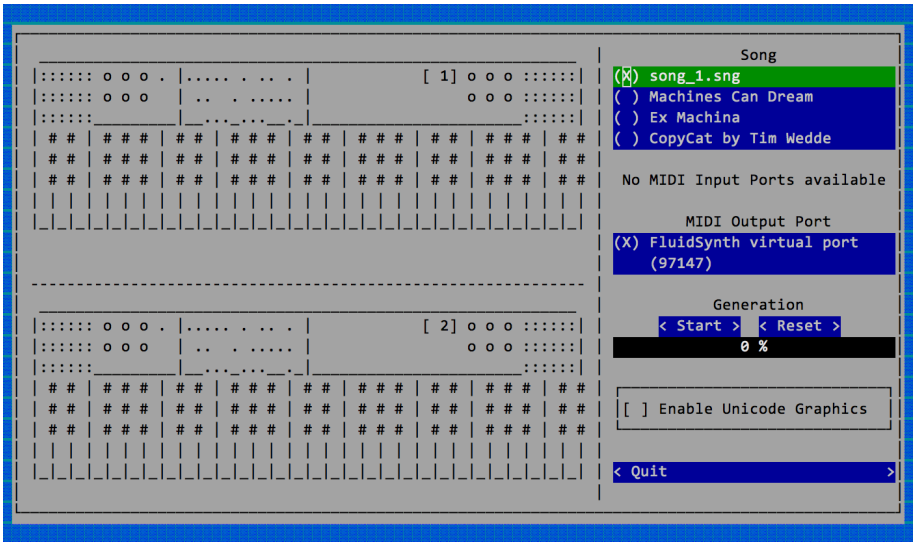

Figure 11: TUI of the software package

The application window is split into two panes. The left two-thirds of the screen are split horizontally and provide a view of two virtual keyboards which update in real-time to display the notes currently being played by the melody and bass lines.
The right-hand third of the screen contains the control elements stacked vertically on top of each other. The top-down design follows the steps required to configure the application in that initially, a song must be chosen to be played, after which MIDI in- and output ports can be selected if available. It is required that at least one MIDI output port is available. After that, the generation can be started, stopped and reset via the buttons below the I/O section. A progress bar is provided to show the estimated percentage of the song that is completed at any given time.

Below the progress bar a checkbox controls whether unicode graphics should be used. This setting takes effect for the progress bar and supplements the `ASCII` rendering of moving elements in the main window with characters of the `UTF-8` character set, e.g. half- and quarter-bars for the progress bar, provided the current terminal supports this extended character set.
The application can be quit via the "Quit" button, or alternatively by hitting "Q". It also responds to `SIGINT` signals (as emitted by e.g. `Ctrl+C`) with a graceful shutdown.

4.6.6 *Documentation & Code Quality*

To ensure a baseline consistency of the codebase, all code adheres to the `PEP8` styleguide for Python[12]. For this purpose, the utility program `pylint`[13] was used to detect violations of this styleguide. Pylint is a linting program that analyses source code for adherence to specific stylistic guidelines as well as analysing program flow for obvious errors or possible bugs. As an exception to the styleguide, the maximum line length was increased from 80 to 120 characters, since this makes some specific parts of this project more readable.
Documentation was provided via Docstrings[14] at the module, class and function/method level. Such docstrings can be read by several documentation generators as well as providing a simple documentation right in the source code.
To supplement the development experience, all relevant parts of the project log information on various levels of severity (from `INFO` to `FATAL`), which is intended to help with debugging as well as development of new features. During normal operation, the application logs data to the file `output.log`.

4.7 ADDITIONAL SOFTWARE

In addition to the main application, two other software packages were developed, specifically intended for use in the the precursory data-processing step: `pydicsv`[15] and `banana-split`[16]. These modules are concerned with the processing of MIDI and CSV files, which were the formats chosen for processing the scraped dataset.
Because MIDI files are a binary format, they are hard to work with if no abstraction is provided. Since the dataset had to be analysed and files modified before they would eventually be used for training, an intermediary format was chosen that was easier to work with. The choice fell on CSV due to the fact that only simple modifications and tasks of data-extraction would be required and thus larger Python libraries providing higher levels of abstraction would be too large and unwieldy to work with, in terms of performance and the amount of upfront learning involved in learning to work with them. The most prominent library in this regard is music21[17], a collection of music-theory related classes

12 https://python.org/dev/peps/pep-0008
13 https://pylint.org
14 https://python.org/dev/peps/pep-0257
15 https://github.com/timwedde/pydicsv
16 https://github.com/timwedde/banana-split
17 http://web.mit.edu/music21

and functions that mostly operates on MIDI files but can also work with other formats. Appendix B provides more detailed information about how data was processed with these tools.

4.7.1 *pydicsv*

This tool is direct Python-port of the `midicsv` and `csvmidi` tools[18] courtesy of John Walker, which serve the purpose of converting MIDI files into their respective CSV representation and vice versa. It was used at the beginning and end of the data processing step. Figure 12 shows the structure of the produced files. Each line describes exactly one complete event, with the first three fields (track number, timestep and event name) fixed and the rest of the values dependent on the type of event being described. The format itself was created specifically for this program, but is easy to implement as it effectively translates the already existing bytes of information into a more human-readable form without changing the makeup of the file or its contents. These properties enable seamless and lossless conversion from and to the MIDI file format.

```
0, 0, Header, 1, 8, 240
1, 0, Start_track
1, 0, Time_signature, 4, 2, 24, 8
1, 0, Tempo, 705882
2, 0, Start_track
2, 0, Title_t, "zang"
2, 0, Program_c, 0, 3
2, 0, Control_c, 0, 7, 100
2, 9840, Note_on_c, 0, 63, 80
2, 10032, Note_off_c, 0, 63, 64
8, 95616, End_track
0, 0, End_of_file
```

Figure 12: Excerpt of a MIDI file converted to intermediary CSV format

4.7.2 *banana-split*

This custom script is able to extract MIDI events from CSV-MIDI files, split by channel and track, and was used to create the base dataset which was then modified further by a custom cleanup script (see Section B.3). It works by first extracting channels, which correspond to instruments, and subsequently tracks, which each denote a singular instrument of a specific type, but might be aggregated in one channel. This makes is easier to access individual instruments such that they can be easily modified, searched for and individual instrument lines extracted, as well as allowing the assembly of custom, reduced or enhanced MIDI files for training

18 https://fourmilab.ch/webtools/midicsv

5 RESULTS

This chapter will showcase the generated output and compare it to other, similar software, as well as analyzing it on an individual basis for the adherence to the initial requirements.

5.1 ACQUISITION OF OUTPUT AND EVALUATION METHODOLOGY

The application was tested in version `0.4.3`[1], which is the latest version at the time of this writing and also contains the generated samples analysed later in this chapter.
To acquire a set of samples for analysis, six musical pieces were generated. For this, three song templates were created with differing chord progressions and two samples were generated for each template.
The generated samples will be examined for three criteria:

STRUCTURE : Repetition of whole sections according to the given structure.

MOTIF : Repetition of small sections common in carnival music.

SCALE/KEY : Adherence to one scale and key.

Analysis of the samples will be achieved audiovisually, via listening to the pieces as well as by visual analysis of the MIDI events. The software used for visualisation of the files is Logic Pro X[2] (version `10.4.1`).

5.2 EXAMINATION OF THE RESULTS

Each generated MIDI file consists of four tracks in the order defined below. A visualisation of one of the generated samples with the tracks organised in the same order can be found in Figure 13.

1. Drums
2. Melody (`Program No.57` - Trumpet)
3. Bass (`Program No.68` - Baritone Sax)
4. Chords (`Program No.01` - Acoustic Grand Piano)

Figure 14 shows a side-by-side view of the same segment generated by two different runs of the same song template. While the chord progression remains the same (lowest instrument line), the other instruments vary, showing that varying melodies are being generated even when all parameters remain the

1 https://github.com/timwedde/composer/releases/tag/0.4.3
2 https://www.apple.com/logic-pro

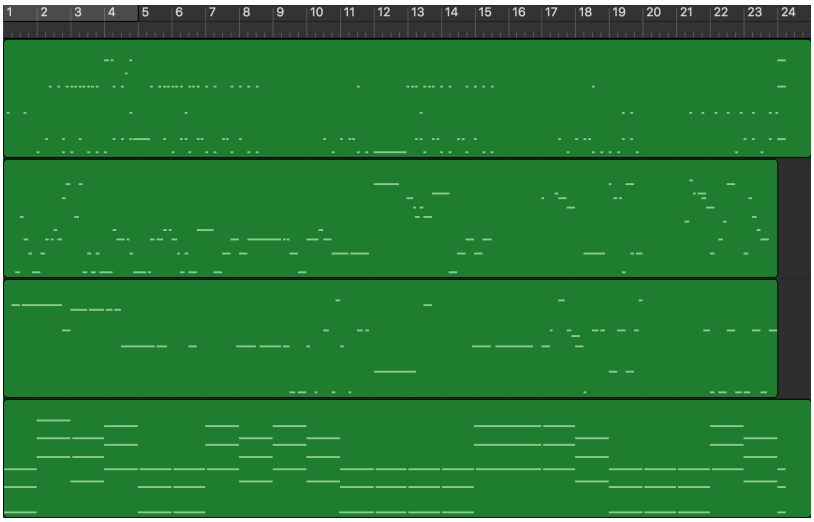

Figure 13: Visualisation of generated output

same between runs. It is noteworthy that the bass line (second from the bottom) exhibits a pattern that is commonly found in carnival music (as well as several other more simplistic genres), in that it simply alternates between two notes. This is especially visible in the left-hand-side, while the right-hand-side shows a slightly more complex pattern with the same basis.

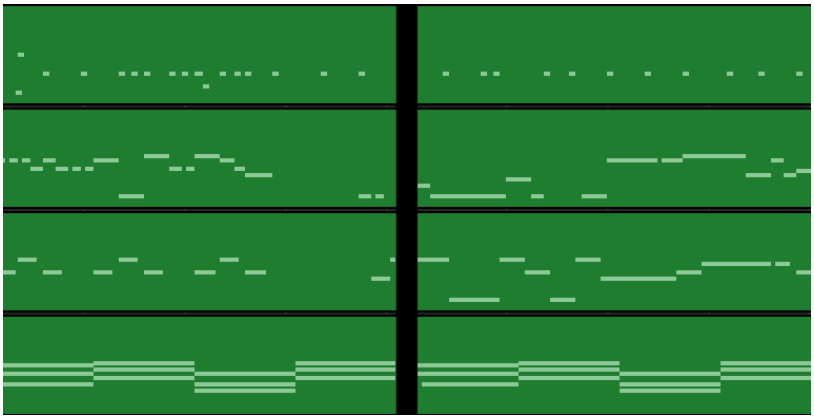

Figure 14: Comparison of the same segment in the same song template, two different generation runs

On the other hand, the drum lines indicate a pattern of minimalism, in that most generated samples contain large timespans of only the hi-hat being used. However the model appears to be able to generate more complex patterns as well (see Figure 15) which are rythmically consistent, evident by the equal amounts of space between the MIDI events. In general, the model seems to favor the hi-hat and the bass drum, which is in accordance with the general simplistic makeup of carnival music.
The melody lines consistently exhibit typical patterns of rising and falling melody (by playing scale notes up or down), which can also be found in the source data (see Figure 16). Additionally, the melody lines contain repeating notes with different durations, mimicking what in the original song would be

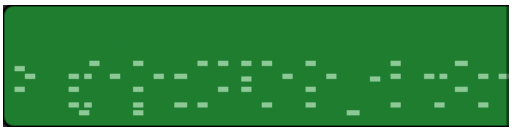

Figure 15: Example of a more complex drum pattern

the voice line of the singer, creating a sing-along line of melody. The duration of inidvidual notes is more varied than with the bass, adding to the dynamic nature of the melody line.

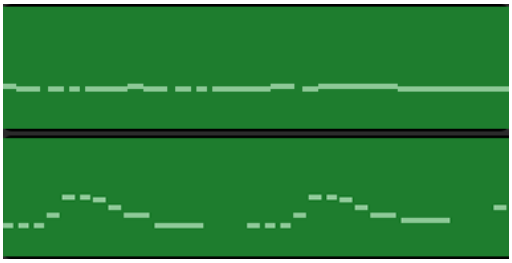

Figure 16: Example of a melody line following the singer (top) and a descending pattern (bottom), from the original dataset

Overall, the software package fulfills the initially given requirements in that it is capable of producing harmonically consistent music containing multiple instruments. However, due to the lack of information about high-level structure during the training step, it was not possible to infuse a sense of purposeful composition into the generated pieces. So even though they technically adhere to the requirement of generating music containing different segments, there is no musical variance between these segments, making them hard to tell apart. So while the output comes in the form of a basic song, it is still very easy to tell it apart from a human-composed piece. Because of this, acceptance testing with multiple test-subjects was foregone, seeing as the output is very obviously distinct from human-composed songs.

5.3 OTHER SOFTWARE

It was discovered that within approximately the same timeframe as this project, a very similar attempt (in purpose rather than execution) at generating music comprised of several instruments was made in the form of MuseGAN[3] [Don+18], although their project focuses more on multiple instruments and less on long-term structure, as their output only consists of four bars per generation. No singular project, as far as is known at the point of writing this thesis, is aiming at music generation in the realm of songs possibly several minutes long, with the closest thing released being [Sim+18], which allows for automatic chord conditioning and thus extends the approach utilised here by integrating this conditioning into a ML model.

3 https://salu133445.github.io/musegan

6 CONCLUSION

This thesis has examined the feasibility of and the various possible approaches to generating musical pieces that consist of multiple instruments and several high-level segments (e.g. Chorus or Verse). Along the way, it was proven that such a thing is indeed possible within a limited scope and a PoC implementation was provided demonstrating as much. This chapter will reflect on the obtained results, assess the success of the project in general and present opportunities for future work.

6.1 PROJECT REFLECTION

Over the course of the previous chapters (specifically Chapter 3), it was shown that the compound problem of generating full musical pieces is very new and seldomly researched. Since the area of applying ML to artistic disciplines is quite new (in its current form), most work in this area is foundational in nature, meaning that the focus lies heavily on small sub-problems such as isolated melody generation or chord prediction, with little time expended to combine these different efforts. In fact, the only other projects in this area that could be found at the time of this writing are MusicVAE [REE17] and MuseGAN [Don+18], although even these projects focus more on integrating different instruments at generation time rather than generating longer sequences of music. Just a few days before the finalisation of this thesis, a modification of MusicVAE was released [Sim+18] that was focused in a very similar direction as this project and appears to be the first published research on the topic of generating multi-instrument music of considerable length, proving that this problem is indeed on the cutting edge of this area of scientific inquiry.
The quality of the output of this project was also heavily affacted by the lack of data that could have been used to enhance the generated output, most notably information about different segments within a song, which would have enabled conditional generation for the segments given during generation time. Due to this research area being relatively new, few pre-aggregated datasets exists, which means that a large part of any project executed within this area is data collection and pre-processing. Given a larger timeframe, it could have been possible to acquire or produce more featureful datasets to enhance the generated output.

6.2 OUTPUT

As described in Section 2.2, this project has several outputs defined, all of which were produced successfully. The main result is the PoC implementation (named `composer`), which is responsible for automatically generating new musical pieces. To provide an easier starting point for subsequent (or possibly

even unrelated) research, the dataset used for training the different models is also published in the aggregate repository[1], which centralizes all outputs in one place. The trained models are included in the PoC implementation and are explicitly linked to in Section C.5.
In addition to this, the execution of this project saw the creation of two supplementary tools with broader applicability (described in more detail in Section 4.7), which were also made publicly available.
Additionally, the songs generated for analysis in this thesis are shipped with the latest version of `composer` at the time of this writing (version `0.4.3`[2]).

6.3 FUTURE OPPORTUNITIES

Given that this project has been executed in an area where little to no research effort has yet been invested, there are quite a few opportunities to improve this project as well as topcis to investigate more closely, possibly in a derivative work.
To improve the quality of the models currently in use, one could utilise transfer learning to build on top of existing models that were trained on larger datasets, as is the case with the Google-provided example models. However, since these are incompatible with recent versions of TensorFlow and Magenta, a larger and more general model could be trained and then retrained with smaller, style-specific datasets to achieve a better end-result.
Generally, a larger set of training data would be beneficial to this project, to allow for longer training and more diverse evaluation datasets, which is likely to increase performance of the models. In combination, provided more computational capability could be acquired, larger model sizes could be used to allow the model more room for learning.
From the perspective of the software package, it would be possible to replace the models currently in use by something else, should it provide superior output (e.g. the aforementioned MuseGAN by [Don+18]). Alternatively, effort could be focused on enabling segment-conditioned generation of melodies in the current model, provided a fittingly-annotated dataset were available. The creation of such a dataset would be beneficial to a multitude of projects within this area.

1 https://github.com/timwedde/ai-music-generation
2 https://github.com/timwedde/composer/releases/tag/0.4.3

REFERENCES

[BBV12] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent.
"Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription."
In: *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*.
Vol. 2.
June 2012.

[Bru+17] Gino Brunner et al.
"JamBot: Music Theory Aware Chord Based Generation of Polyphonic Music with LSTMs."
In: *Computing Research Repository* abs/1711.07682 (2017).
arXiv: 1711.07682.
URL: http://arxiv.org/abs/1711.07682.

[CFS16] Keunwoo Choi, George Fazekas, and Mark Sandler.
"Text-based LSTM networks for Automatic Music Composition."
In: *Proceedings of the 1st Conference on Computer Simulation of Musical Creativity*.
Apr. 2016.
arXiv: 1604.05358.
URL: http://arxiv.org/abs/1604.05358.

[Col+16] Florian Colombo et al.
Algorithmic Composition of Melodies with Deep Recurrent Neural Networks.
June 2016.
DOI: 10.13140/RG.2.1.2436.5683.
URL: http://dx.doi.org/10.13140/RG.2.1.2436.5683.

[DE10] Stephen Davismoon and John Eccles.
"Combining Musical Constraints with Markov Transition Probabilities to Improve the Generation of Creative Musical Structures."
In: *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part II*.
EvoCOMNET'10.
Berlin, Heidelberg: Springer-Verlag, 2010,
Pp. 361–370.
ISBN: 3-642-12241-8.
DOI: 10.1007/978-3-642-12242-2_37.
URL: http://dx.doi.org/10.1007/978-3-642-12242-2_37.

[Don+18] Hao-Wen Dong et al.
"MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment."

In: *Proceedings of the 32nd Association for the Advancement of Artificial Intelligence Conference*.
Vol. 32.
Feb. 2018.
arXiv: 1709.06298.
URL: http://arxiv.org/abs/1709.06298.

[DZX18] Shuqi Dai, Zheng Zhang, and Gus Xia.
"Music Style Transfer Issues: A Position Paper."
In: *arXiv Computing Research Repository*.
Mar. 2018.
arXiv: 1803.06841.
URL: http://arxiv.org/abs/1803.06841.

[Ehm+11] Andreas F. Ehmann et al.
"Music structure segmentation algorithm evaluation: Expanding on MIREX 2010 analyses and datasets."
In: *Proceedings of the 12th International Society for Music Information Retrieval Conference*.
Oct. 2011,
Pp. 561–566.
ISBN: 0-615-54865-2.

[ES02] Douglas Eck and Jürgen Schmidhuber.
"Finding temporal structure in Music: Blues improvisation with LSTM Recurrent Networks."
In: *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*.
Feb. 2002,
Pp. 747–756.
ISBN: 0-7803-7616-1.
DOI: 10.1109/NNSP.2002.1030094.
URL: http://dx.doi.org/10.1109/NNSP.2002.1030094.

[HCC17] Dorien Herremans, Ching-Hua Chuan, and Elaine Chew.
"A Functional Taxonomy of Music Generation Systems."
In: *ACM Computing Survey* 50.5 (Sept. 2017), 69:1–69:30.
ISSN: 0360-0300.
DOI: 10.1145/3108242.
URL: http://doi.acm.org/10.1145/3108242.

[Her+15] Dorien Herremans et al.
"Generating music with an optimization algorithm using a Markov based objective function."
In: *ORBEL29, Belgian Conference on Operations Research* (2015).

[HI92] Lejaren A. Hiller and Leonard Isaacson.
In: *Machine Models of Music*.
Ed. by Stephan M. Schwanauer and David A. Levitt.
Cambridge, MA, USA: MIT Press, 1992.
Chap. Musical Composition with a High-speed Digital Computer, pp. 9–21.

ISBN: 0-262-19319-1.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber.
"Long Short-Term Memory."
In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780.
ISSN: 0899-7667.
DOI: 10.1162/neco.1997.9.8.1735.
URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[Jaq+17] Natasha Jaques et al.
"Tuning Recurrent Neural Networks With Reinforcement Learning."
In: *Proceedings of the 2017 International Conference on Learning Representations*.
Apr. 2017.
arXiv: 1611.02796v9.
URL: http://arxiv.org/abs/1611.02796v9.

[Joh17] Daniel D. Johnson.
"Generating Polyphonic Music Using Tied Parallel Networks."
In: *Computational Intelligence in Music, Sound, Art and Design*.
Ed. by João Orreia, Vic Ciesielski, and Antonios Liapis.
Cham: Springer International Publishing, 2017,
Pp. 128–143.
ISBN: 3-319-55750-5.

[Mak+17] Dimos Makris et al.
"Combining LSTM and Feed Forward Neural Networks for Conditional Rhythm Composition."
In: *Engineering Applications of Neural Networks*.
Ed. by Giacomo Boracchi et al.
Springer International Publishing, 2017,
Pp. 570–582.
ISBN: 978-3-319-65172-9.

[MP43] Warren S. McCulloch and Walter Pitts.
"A logical calculus of the ideas immanent in nervous activity."
In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133.
ISSN: 1522-9602.
DOI: 10.1007/BF02478259.
URL: https://doi.org/10.1007/BF02478259.

[QH13] Donya Quick and Paul Hudak.
"A Temporal Generative Graph Grammar for Harmonic and Metrical Structure."
In: *Proceedings of the 2013 International Computer Music Conference*.
Sept. 2013,
Pp. 177–184.

[Raf16] Colin Raffel.
"Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching."

PhD thesis. Columbia University, 2016.

[REE17] Adam Roberts, Jesse Engel, and Douglas Eck.
"Hierarchical Variational Autoencoders for Music."
In: *Proceedings of the 31st Conference on Neural Information Processing Systems*.
Dec. 2017.

[Shi+17] Andrew Shin et al.
"Melody Generation for Pop Music via Word Representation of Musical Properties."
Note: This paper's submission was withdrawn from the 2018 International Conference on Learning Representations. Its purpose here is simply to showcase a different implementation of a specific network architecture, not substantiate any claims or portray facts cited from it.
Oct. 2017.

[Sim+18] Ian Simon et al.
Learning a Latent Space of Multitrack Measures.
June 2018.
arXiv: 1806.00195.
URL: http://arxiv.org/abs/1806.00195.

[TZG17] Yifei Teng, An Zhao, and Camille Goudeseune.
"Generating Nontrivial Melodies for Music as a Service."
In: *Proceedings of the 2017 International Society for Music Information Retrieval Conference*.
Vol. 18.
Oct. 2017,
Pp. 657–663.

[Win29] Reginald P. Winnington-Ingram.
"Ancient Greek Music: A Survey."
In: *Music and Letters* 10.4 (1929), pp. 326–345.
DOI: 10.1093/ml/X.4.326.
URL: http://dx.doi.org/10.1093/ml/X.4.326.

[WS94] Martin D. Westhead and Alan Smaill.
"Automatic Characterisation of Musical Style."
In: *Music Education: An Artificial Intelligence Approach*.
Ed. by Matt Smith, Alan Smaill, and Geraint A. Wiggins.
London: Springer London, Jan. 1994,
Pp. 157–170.
ISBN: 1-4471-3571-7.
DOI: 10.1007/978-1-4471-3571-5_10.
URL: https://doi.org/10.1007/978-1-4471-3571-5_10.

[Zho15] S. Kevin Zhou.
Medical Image Recognition, Segmentation and Parsing.
Academic Press, 2015.
ISBN: 0-12-802581-6.
URL: https://www.sciencedirect.com/science/book/9780128025819.

Appendix

A PRIMER ON ARTIFICIAL INTELLIGENCE

To provide a common baseline of knowledge for the main part of the thesis, this chapter will give a small overview of the general concepts of AI and explain the basic technicalities required to understand its operating principles.

A.1 WHAT IT IS

AI in its most general form is a term used to refer to any form of intelligence exhibited by an object or machine that is not human or generally biologically-based (e.g. animals), wherein the term "intelligence" means general intelligence and refers to the purposeful taking of actions, the application of logic, the capability of understanding a concept as well as learning, planning and problem solving within some capacity.
In the area of computer science specifically, the term Computational Intelligence (CI) exists, which refers to a subset of the scope of AI in that it is concerned with the ability of a computer to learn a specific task or concept from data, without the goal being explicitly described beforehand. To achieve this it often mirrors concepts found in nature, which are able to cope with situations in which a computer would normally be unable to produce correct results. A current example of tasks that exhibit excessive complexity or uncertainty are those related to self-driving cars, which have to cope with many variables and a constantly-changing environment they often times can not perceive entirely at any given time. CI, in contrast to AI, excludes methods of hard computation, meaning any task or operation that can be translated into binary logic and be computed by an algorithm, it instead focuses on stochastics and fuzzy (many-valued) logic.

A.2 MACHINE LEARNING

ML utilizes concepts from the area of CI to enable computer systems to learn to execute a task or operation without being explicitly programmed to do so, but instead based on the "learning" of the task from sample data, leading to the creation of a stochastic model tuned for a specific task. It is largely based in the areas of statistics and pattern recognition and is mostly concerned with the finding of predictive algorithms and patterns in large amounts of data as well as analysis of such to gain insights into complex structures that would otherwise be difficult to understand.
Machine Learning, in large parts, relies on the availability of large sample-sizes, which subsequently have to be processed to enable learning, leading to massive resource usage and a strong reliance on large amounts of data. As such performance of these stochastic models is often strongly correlated with the quality of the input data as well as the computational power expended

during the "training" phase of such a model. These requirements are often the reason an algorithmic, concrete approach is favored when applicable, if it produces similar results.
Advantages to this approach lie in its ability to comprehend complex, possibly faulty or incomplete data structures as well as potentially being able to discover hidden relations between such data and extrapolate the results into a conceptual understanding of the data which can then be used to deal with unknown scenarios at a later stage.

A.3 TYPES OF MACHINE LEARNING

Many different approaches exist for ML, many of which stem from the statistical science fields, such as modeling probabilities using various forms of tree structures, cluster analysis or utilising genetic algorithms. The areas mostly talked about when discussing Machine Learning, as well as the specialisation of this thesis, are more related to the approaches concerning Artificial Neural Networks (ANNs) and Deep Learning.
Such systems stem from the concept of "Connectionism", which represents a set of approaches with the common goal of trying to emulate and model mental processes of often times biological inspirations, such as the brains of animals or even humans, in an attempt to capture similar behavioral patterns or abilities. This is often accomplished by modeling networks of simple units, often times referred to as "artificial neurons", similar to structures as they would occur in actual brains. With correct training of these models, emergent behaviour can be observed that can be shaped to a specific task with the adjustment of the learning process.
Deep Learning enhances this concept by proposing the integration of multiple such models, often called "layers" of neurons, which are "stacked" and often times sequentially connected, the output of one layer feeding into the input of the next one. Such networks, with sufficient "depth", are able to emulate more complex tasks similar to what a biological brain might be able to produce, especially regarding tasks such as image recognition, the understanding of natural language or playing games such as Chess or Go, which exhibit extraordinary complexity, making them unsuitable for algorithmic computation.

A.4 HOW IT WORKS

Neural Networks are based on connecting a large amount of individually simple parts with each other in different configurations, producing a network of nodes which, as a whole, is capable of exhibiting emergent behaviour. The artificial neurons used in such networks are modeled as individual units with one or multiple inputs and one output, which are connected by one of several mathematical functions that are applied to transform and reduce the input into an activation - the level which is eventually emitted at the output of each neuron. Sigmoidal functions are most commonly used, but other functions may be better suited for specific purposes.

The function defines which output value to emit, given an input value, and if the output value lies above a certain threshold the neuron "fires", activating the neurons connected to its outputs (thus the function is sometimes called "activation function").

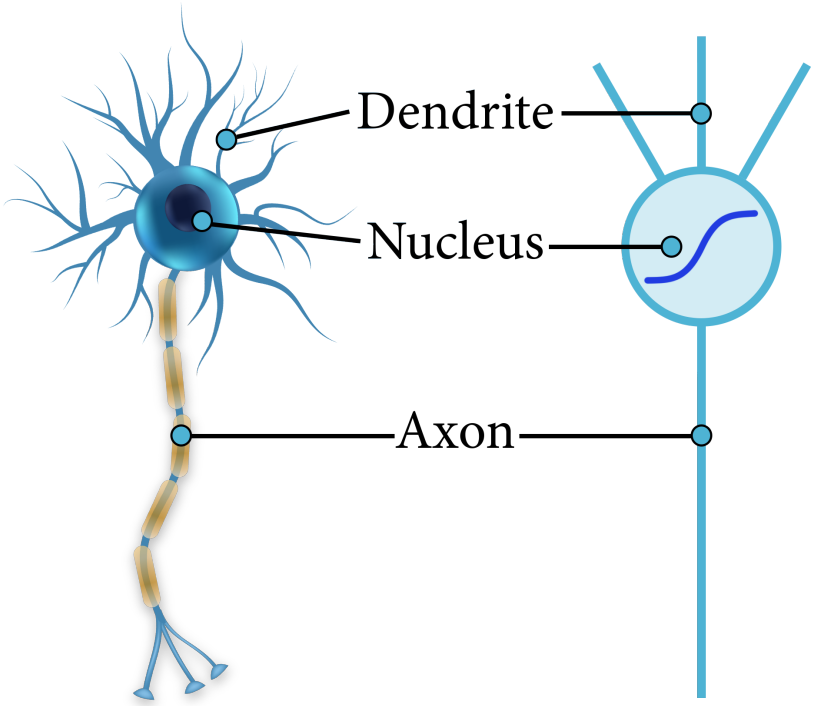

Figure 17: Biological vs Artificial Neuron (biological neuron graphic created by *Freepik*, https://freepik.com; Online, accessed 2018-05-12)

Figure 17 shows the similarities between the biological inspiration and the artificial recreation for such a neuron. The Dendrite, Nucleus and Axon translate into the Inputs, Activation Function and Output respectively.
An individual neuron functions similarly to a logic gate on a regular processor, only that instead of a value range of two (`0` and `1`) it possesses an infinite range of decimal numbers that can be emitted for different inputs. As such it is theoretically possible to build logic gates from artificial neurons, a concept which is called "threshold logic". In fact, the first artificial neuron was labeled "Threshold Logic Unit" by [MP43].
Overall however, "programming" a neural network is very different from writing regular code, which can automatically be translated into the correct values for the low-level logic gates. Instead of manually defining the values for each neuron while taking into account their interplay with every other (transitively) connected neuron, the process of programming has been replaced by the concept of "training".
Training in this case refers to the continual adjustment of the values of individual neurons until the network as a whole exhibits the desired behaviour in some capacity. Training happens by transforming input data into a format the network can understand (multidmensional matrices of numbers), which are then "pushed through" the network by inputting the numbers into the first set of neurons, modifying the subsequently connected neurons as the previous

ones fire, biasing them in a specific direction. This type of network is called *feed-forward* network.
All neurons with the same connection depth to previous neurons are said to be on a "layer" (see Figure 18). Stacking multiple layers expands the network, making it more expensive to train but also able to store more information. Networks with multiple layers are also sometimes referred to as Deep Neural Networks (DNNs).

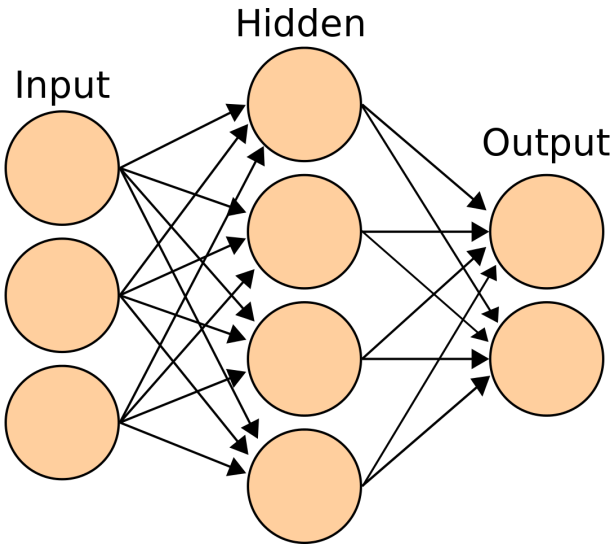

Figure 18: A simple Feed-Forward Neural Network (image created by *Wikipedia Contributors*, https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg; Online, accessed 2018-05-03)

To arrive at the actual value to adjust a neuron by, the output of the last layer of the network is compared to the expected output, and the difference between these is used to tweak the values of the neurons in the network to nudge them closer to being able to replicate the expected output data. This method of applying the computed difference is called *backpropagation*. In many models, a *learning rate* exists which governs how drastic the change of the applied difference is, as applying the entire difference can in some cases be too big of a step towards a particular direction, leading to rapid overfitting (a state within which a network produces the same outcomes for almost all inputs).
The most common way to train a network is gradient descent, which uses a loss function to apply backpropagation to the model. Such a function determines, for every output of the network, how "good" it is, this referring to a custom implementation of criteria as supplied by the programmer, which are then factored into the process of calculating the deviation from the expected output.

A.5 GENERATIVE AI

The output of a neural network comes in the form of a probability distribution over the range of possible values the network was trained to recognize.
The difference between classification and generation networks stems from the specific distribution they learn. Classification models learn a conditional prob-

ability distribution $p(y|x)$ which allows them to classify a given value x into a category y. Generative models in comparison learn the multivariate probability distribution $p(x, y)$, which gives the probability of a specific combination of x and y, for every possible permutation.
One of the advantages of generative models is that the multivariate probability distribution can be transformed into a conditional probability distribution later on. The biggest difference however is that the multivariate probability distribution learns a more abstract concept, in that it makes assumptions about the nature of how the data is structured, and will assume a shape that will attempt to closely emulate the original structure. To generate values from such a distribution, several methods can be applied, the most common of which are:

- Gibbs Sampling
- Argmax Function
- Softmax Function
- Random Sampling
- Beam Search
- Greedy Search

These methods detail different ways of how to interpret the resulting probability distribution that is generated when a set of inputs is given to the model.

A.6 APPLICATION AREAS

AI (and ML in particular) have found a large amount of use cases within various fields. Many recommendation systems try to maximize their usefulness by learning to predict what its users want to see, Netflix, YouTube and Amazon being on the forefront of this area of research in the corporate space. Commonly known examples of the broad applicability of similar technology can be seen in products such as "Google Home" or "Amazon Echo".
The medical field is seeing an increasing interest in applying techniques of image recognition to medical images as produced by X-Ray or Magnetic Resonance Imaging machines to recognize a multitude of diseases and anatomical issues [Zho15].
Out of all its application areas, the field of self-driving cars is probably the most prominent, popularized by companies such as Tesla, Google and Uber and many car manufacturers who are now starting to venture into this area as well.
Much of the initial mainstream-interest in AI stemmed from the creative fields, most prominently the artistic ones, as researchers started experimenting with style transfers between images, as well as the recreation and classification of objects within an image, providing the foundation for the application of similar and enhanced concepts in many other fields requiring the actual understanding and conceptualization of images.

B

DATA REPRESENTATION & PROCESSING

This appendix will explain how the data used within this project was acquired and how it was processed for use within the application. Additionally, a closer look will be taken at the gathered collection of files to determine selection criteria that will later be used to create the training data.

B.1 GATHERING THE DATA

Data was obtained from the dutch website "Limburg Zingt"[1], which archives carnival songs from the Limburg region and surrounding places. A site-crawler was created to extract all songs available as MIDI files from the website. The crawler was implemented using `Scrapy`[2], a Python framework for quickly creating web crawlers, and is available in the project's aggregate repository[3].

B.2 DATA ANALYSIS

Execution of the crawling process led to the gathering of a of 892 MIDI files, with one file failing to download due to file corruption, leading to a total of 891 usable files.
A large portion (roughly 48%) of the collection of files (henceforth "the corpus") appears to lack key information entirely, as shown by the label "empty" (which was substituted in case of missing data) in Figure 19, making the results incorporating key signature less reliable due to the large amount of dark data, which is missing explicitly tagged pieces of information. As such, correlations are hard to identify.
The following results will be compared in parts to an analysis done on a larger set of songs, the "Lakh MIDI Dataset" [Raf16], which will be used as a median for key and time signatures to provide a frame of reference seeing as it is based upon a compilation of songs spread over many genres and types of music, providing a good baseline for further analysis.
Focusing on the songs containing key information, all songs appear to be written in a major mode, with the key of C making up the largest share by far. The next largest share is the key of G major after which the other keys are mostly neglibile in frequency of occurence. This somewhat coincides with the results of Raffel's analysis, with a large majority of all modes being major. Noticable is the lack of diversity in keys overall, generally being restricted to C, G and D major.
Roughly 50% of all songs are in standard 4/4 time (see Figure 20), with the less common signatures 2/4 and 3/4 each making up about 20%, which is

1 http://www.limburgzingt.nl
2 https://scrapy.org
3 https://github.com/timwedde/ai-music-generation

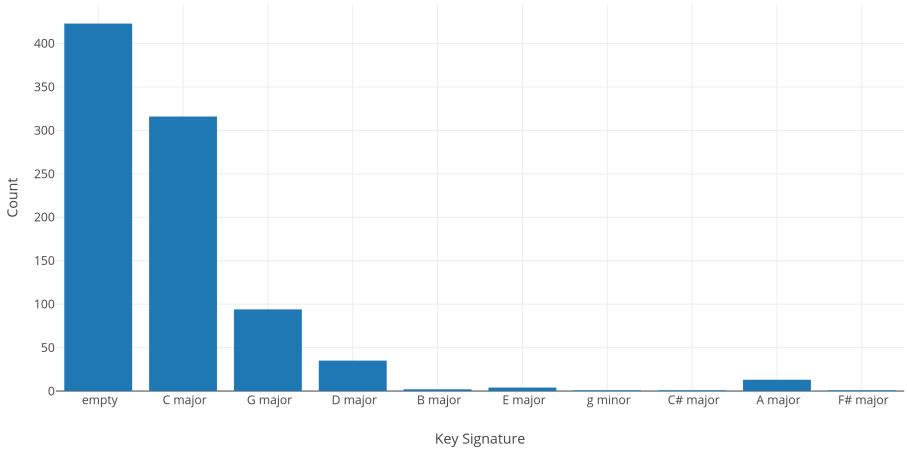

Figure 19: Key Signature Distribution

similar to Raffel's analysis, which in turn mirrors common pop music trends. In contrast to their results however, this dataset appears to contain slightly more songs in the odd meter 6/8 (about 9%).

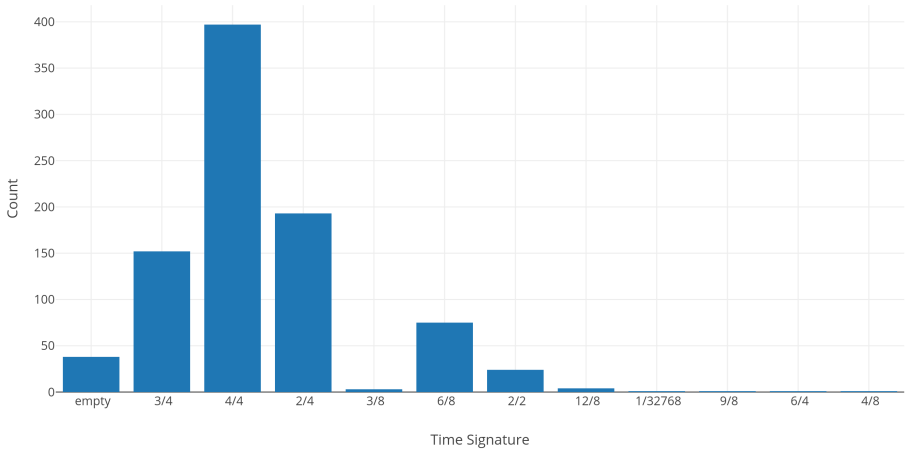

Figure 20: Time Signature Distribution

Taking a look at Figure 21, tempos lie largely in the range of 100 to 130 BPM, as is common for most songs, with a spike at 120 BPM, which is the most common tempo across any genre. In comparison to Raffel's analysis however, where tempos almost assume the shape of the standard deviation, the range of tempos is much smaller, as most carnival songs appear to favor medium to quick pacing.
Combining tempo and time signature into a two-dimensional heatmap (see Figure 21), one can see that most time signatures cluster around the 100 to 130 BPM region, with the 3/4 signature being an outlier, finding its largest

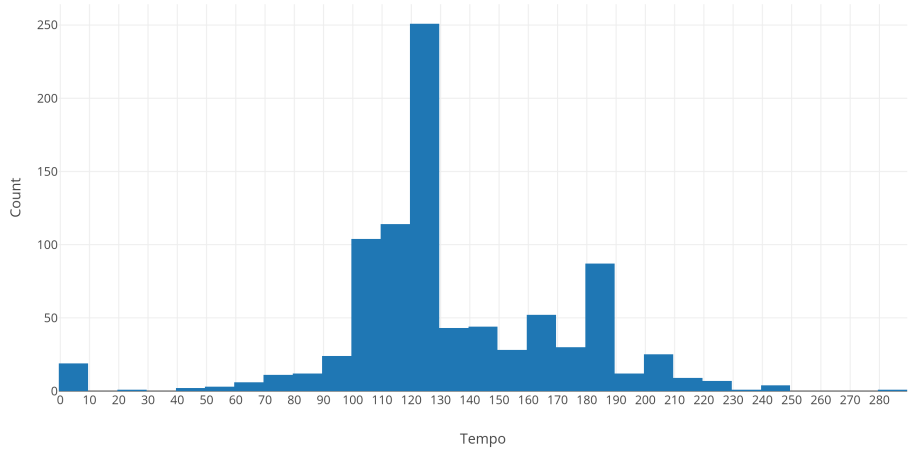

Figure 21: Tempo Distribution

accumulation close to the 180 to 200 BPM range. Given that the amount of samples for this range is the second-largest cluster of tempos, it is reasonable to assume that a correlation exists between the non-standard 3/4 meter and a faster pace. This might be due to stylistic cues specific to this genre of music, the assumption of which will have to be verified by manually auditing several samples of this data region.

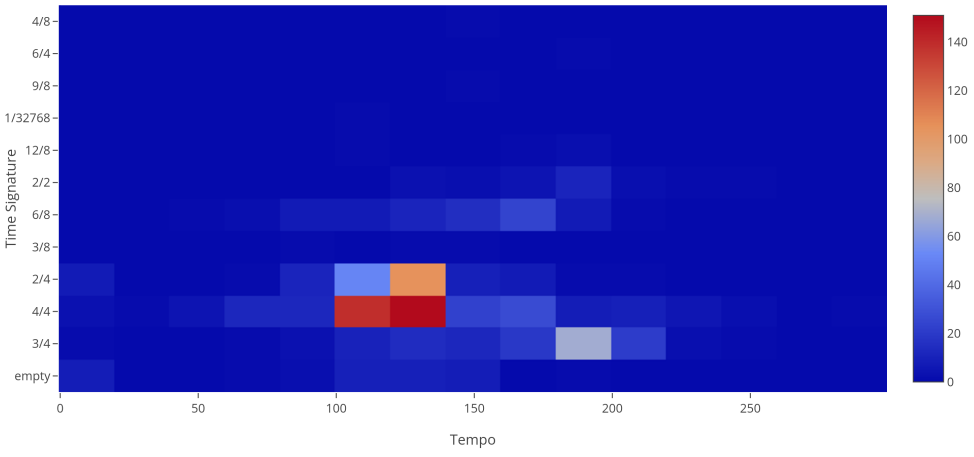

Figure 22: Tempo in Relation to Time Signature

B.3 DATA PREPARATION

Seeing as the chosen MelodyRNN model generates individual lines of instrumentation, the input data has to be prepared fittingly by filtering out any notes

not belonging to either melody, drum or bass lines, such that the model does not have to deal with any "noise" stemming from unfitting instrument lines. Due to the MIDI file format being hard to read and slow to parse in bulk, it was decided that an easier to read and modify intermediary format was to be converted to first, based upon which all cleanup actions could then be performed much more efficiently (see Section 4.7 for more details on the software used for this process).

As such, the scraped data was intially run through `banana-split` to obtain the dataset in CSV format. This script nets an output of the converted files split by channel and track, making the filtering of these properties very easy. In a second step, the different instrument lines were extracted. To figure out which tracks or instruments were most relevant to either of the three categories, a sample subset of the original dataset was inspected to generate a list of instrument numbers that were generally most likely to play melody or bass lines. For the melody part, the selection range lies within the set `(0-9, 57-65, 73, 74)` while the bass range is mostly present in the set `(33-41, 44, 59, 68)`. Due to a convention of the MIDI standard, channel `9` is always dedicated to drums, making extraction of the percussive elements very easy. This method of extracting specific parts works well enough that no complex heuristics are required for extraction of said features.

After the melody, bass and drum lines were extract from each file, the resulting cleaned dataset was converted back to the MIDI format and was subsequently converted to a `tfrecord` container (a special format Magenta-based models make use of) by a script provided by the Magenta project (using the commands shown in Snippet 10). In addition to the initial format conversion, the `tfrecord` container file was then converted to a model-specific representation where all required features for each MIDI event were transformed into one-hot vectors and concatenated to produce the final represenation of each individual event, which was then fed into the model.

C

ADDITIONAL INFORMATION

C.1 PROJECT PLAN

No.	Description	Duration
T1	Acquisition of suitable MIDI files to use as training and testing data	10d
T2	Conversion of acquired files into usable format for the software	15d
T3	Set up of Python project and creation of framework for individual modules	5d
T4	Implementation of the generator for overall song structure	10d
T5	Building of ML models and subsequent training on prepared data	15d
T6	Consolidation of output fragments into a complete song (MIDI file)	10d
T7	Post-Processing of completed MIDI file to increase quality	10d
D1	Preparation phase for Project Plan (MS1)	6d
D2	Preparation phase for interim project delivery (MS2)	10d
D3	Preparation phase for final project delivery (MS3)	15d
MS1	Project Plan	1d
MS2	Interim Report & Presentation	1d
MS3	Final Report & Presentation	1d

Table 3: List of planned tasks

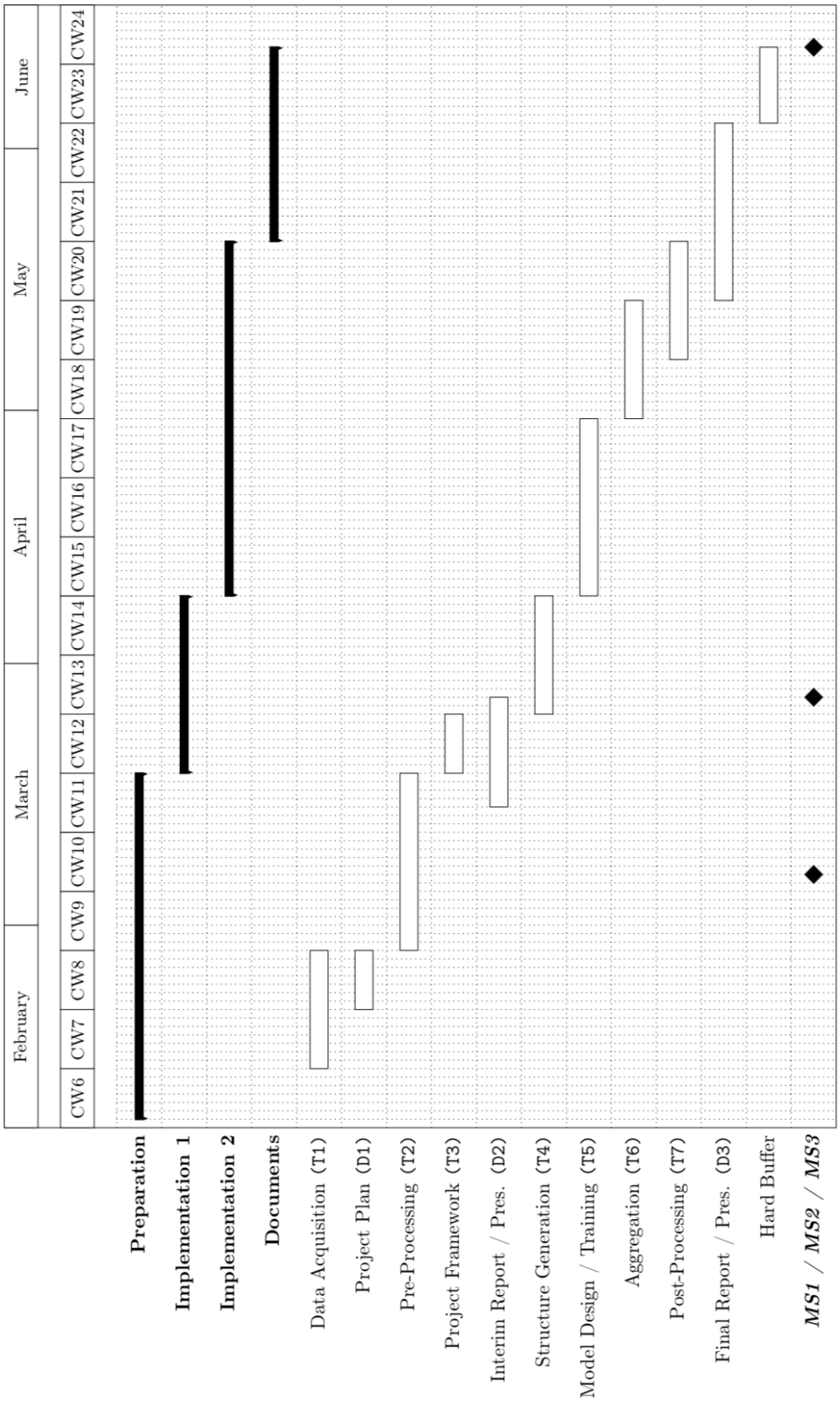

Figure 23: Project Plan

C.2 MUSIC THEORY

This section will give a high-level overview of the general concepts used in music theory as well as lay out some of the terms, relationships and specific concepts related to this thesis. Different views of music theory exist all over the world, but for the purpose of this thesis the main focus will be laid upon its western interpetation and structure.
The foundation of harmony in music lies in the interaction of notes of differing frequencies. At the most basic level, the frequency spectrum is divided into octaves. An octave is the range of frequencies between a starting frequency and another frequency with either half (downwards) or double (upwards) its starting frequency. All frequencies within one octave are mirrored across all other octaves, provided they were derived from the same starting point, which means a note played in one octave is harmonic with the same note played in any other octave.
A scale is a subset of frequencies per octave, separated by arbitrary intervals. Most common is the chromatic scale, which samples twelve points from an octave range and thus extends to all notes nomally found in western music.
Based on this, a chord is formed by a combination of multiple notes from a scale. Multiple chords played sequentially form a chord progression. Several rules exist that govern the harmonic interplay between different chords, most notably the circle of fifths, which provides a set of harmonically coherent chords for a given key and mode by mapping the three aforementioned features into a geometrical space within which such relations become visually obvious.
Chords have an abstract representation called "Roman Numeral Notation", which is key agnostic. It describes the position of a note in the scale, given a key and a mode. Using a key and a roman numeral between `I` an `VII`, one can obtain the actual chord to play. Listing multiple numerals after one another forms a chord progression, which is the backbone of most songs and provides a common harmonic ground for multiple instruments to play together, as it contains information about the key, the mode and the valid notes that can be played (the scale).

C.3 THE MIDI STANDARD

This section will explain some general information about MIDI as well as the actual format itself, to enable better understanding of the parsing and usage of MIDI files in this and other, similar projects.
MIDI, short for Musical Instrument Digital Interface, is a standardised file format (upheld by the MIDI Association[1]) that describes several things needed to enable standardised communication between various instruments and a computer or similar processing device. It exhibits specifications for hardware connectors as well as how communication is handled over these connectors and how such information can be stored. The latter part is the more relevant to this

1 https://www.midi.org

project, as it deals with how MIDI data is stored on disk and how it is supposed to be sent through various devices.
The format is intended for real-time communication, such that live performances become possible (most notably keyboards and synthesizers make use of this technology). It describes an event-driven format in which actions on the hardware controller (e.g. a keyboard) are encoded into events, which are represented on the byte-level and then sent to a receiving entity where they can be suitably processed. Additionally, several meta-events are possible, such as changing the time signature on the receiving end or changing a preset bank on a MIDI processor. To make the protocol extensible, custom data can be encoded as well, to allow instrument- or vendor-specific (possibly ecrypted) events to be sent for processing by proprietary equipment.
To ensure low-latency communication, all events are encoded as three-byte tuples with the exception of the aforementioned vendor-specific messages, which may contain a larger payload. The most common events are `note_on` and `note_off` events, which tell the receiving system when to start or stop playing a specific note. Their binary representation (displayed as hexadecimal values) looks like this: `0x90 0x5A 0x64`.
In some cases, data is split over the data bytes of a message, for example with the `pitch_bend` event (e.g. `0xE0 0x18 0x40`). Here, the value is split into two bytes, ordered with the least-significant bit preceding the most-significant bit. This is due to the fact that the range of the value extends up to 16384. To restore the correct value upon reading the message, the first data-byte has to be bit-shifted left by 7 digits, after which it can be combined with the second data byte via an OR operation. The specifics of how data is encoded for each event are defined in the MIDI standard.
The MIDI file format specifies two additional types of information which have to be included in any file of this format: Header Chunks and Track Chunks. A file has a single header chunk giving some general information about the file, as well as how many tracks it contains. It is followed by several track chunks, which themselves are followed by arbitrarily long lists of MIDI events, the length of which is specified in the track header. All of this is encoded into its respective binary representation and forms a standard-compliant MIDI file.

C.4 COMMANDS USED

The commands listed below show the exact configuration used to convert the initial dataset as well as run the individual models for training and evaluation.

```
convert_dir_to_note_sequences --input_dir=melody/ \
    --output_file=melody.tfrecord
convert_dir_to_note_sequences --input_dir=bass/ \
    --output_file=bass.tfrecord
convert_dir_to_note_sequences --input_dir=drums/ \
    --output_file=drums.tfrecord
```

Code Snippet 10: Commands used for converting MIDI files to `tfrecord` containers

```
drums_rnn_create_dataset --config=drum_kit --eval_ratio=0.2 \
    --input=drums.tfrecord --output_dir=drums_data/ \

drums_rnn_train --config=drum_kit \
    --run_dir=drums_run/ --num_training_steps=2000 \
    -sequence_example_file=drums_data/train.tfrecord \
    --hparams="batch_size=64,rnn_layer_sizes=[256,256]" \
    (--eval) <-- added for evaluation job

drums_rnn_generate --config=drum_kit --save_generator_bundle \
    --run_dir=drums_run/ --bundle_file=drums.mag \
    --hparams="batch_size=64,rnn_layer_sizes=[256,256]" \
```

Code Snippet 11: Commands used for training, evaluating and exporting the `DrumsRNN` model

```
melody_rnn_create_dataset --config=lookback_rnn --eval_ratio=0.2 \
    --input=melody.tfrecord --output_dir=melody_data/ \

melody_rnn_train --config=lookback_rnn \
    --run_dir=melody_run/ --num_training_steps=2000 \
    --sequence_example_file=melody_data/train.tfrecord \
    --hparams="batch_size=64,rnn_layer_sizes=[256,256]" \
    (--eval) <-- added for evaluation job

melody_rnn_generate --config=lookback_rnn --save_generator_bundle \
    --run_dir=melody_run/ --bundle_file=melody.mag \
    --hparams="batch_size=64,rnn_layer_sizes=[256,256]" \
```

Code Snippet 12: Commands used for training, evaluating and exporting the `MelodyRNN` models.

C.5 TRAINED WEIGHTS

The weights of the trained models are available in the repository of this project, but are provided here as direct download links for convenience. The weights were exported with version `1.6.0` of Tensorflow and version `0.3.5` of Magenta. Compatibility for different versions of these packages is not guaranteed.

Melody Model:
https://github.com/timwedde/composer/raw/master/models/melody.mag

Bass Model:
https://github.com/timwedde/composer/raw/master/models/bass.mag

Drums Model:
https://github.com/timwedde/composer/raw/master/models/drums.mag