Simen Kværneng Kaspersen

Combining Deep Learning and Combinatorial Optimisation to Generate Four-Part Chorales

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
July 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Simen Kværneng Kaspersen

Combining Deep Learning and Combinatorial Optimisation to Generate Four-Part Chorales

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
July 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

This thesis describes the implementation of a functional prototype consisting of a recurrent neural network and a branch-and-bound optimiser which together generate original music compliant to the rigid rule-set of the four-part chorale in the style of Johann Sebastian Bach.

The exact nature of music generated by neural networks is troublesome to predict, and by extension to manipulate in any meaningful way between input and output. How well the results adhere to the more abstract aspects of music is hard to control, and often display little to no musical structure or significant creativity. Also, there are few avenues for interactivity and active intervention in the usage of such a "black-box" system. The paper explores means to mitigate this externally, and shows that combinatorial optimisation strategies can be employed *post-generation* to enforce musical constraints and adjust the output in such a way that it adheres to the conventions and expectations of western music theory. The constrained results display significant improvement in functional structure, melodic and harmonic progression and aesthetics compared to those generated by the neural network alone. Although the prototype is not capable of locating optimal transformations from the neural network output within reasonable time, a method for finding low cost chord combinations with valid structure can be used to generate results that demonstrate the applications of the system.

Sammendrag

Denne masteroppgaven beskriver implementasjonen av en funksjonell prototype som består av et rekurrent neuralt nettverk og en branch-and-bound optimeringsalgoritme i konjunksjon for å generere musikk som tilretter seg reglene for firstemmig korallsats i Bach-stil.

Den eksakte naturen til musikk generert av neurale nettverk er vanskelig å forutse, og derfor også å manipulere på noen som helst meningsfull måte mellom input og output. Hvor godt resultatene gjenspeiler de mer abstrakte aspektene av musikk er vanskelig å kontrollere, og ofte viser de lite eller ingen musikalsk struktur eller kreativitet. I tillegg er det få innfallsvinkler for interaksjon og inngripen når man benytter seg av et slikt "svart boks"-system. Denne oppgaven utforsker måter å imøtekomme disse utfordringene på fra utsiden, og viser at kombinatoriske optimeringsstrategier kan benyttes i etterkant til å påføre musikalske rammer og endre output slik at den innfinner seg konvensjonene og forventningene til vestlig musikkteori. De nye resultatene demonstrerer en stor forbedring i funksjonell struktur, melodisk og harmonisk utvikling og estetikk sammenlignet med fra før. Selv om prototypen ikke var i stand til å finne optimale transformasjoner fra neuralnettverkets output innenfor rimelig tid, ble det demonstrert en metode som finner lavkostnadskombinasjoner med gyldig struktur som kan benyttes til å generere resultater som demonstrerer programmets funksjon.

Preface

The following paper is a Master Thesis in Computer Science conducted in the spring semester of 2019. The basis of the thesis is a research project which was carried out during autumn, 2018, and contained a comprehensive literature review of existing work, along with initial ideas of potential future improvements. Significant excerpts of this research project are included in the first chapters of this paper with the purpose of supplying introductory insight into the subject field. For transparency, any such chapters or sections are preceded by a statement. For a complete introduction, the reader is encouraged to read the full research paper[1].

The idea for the thesis was sparked through brainstorming conversation with my supervisor, Prof. Hetland, and is a marriage between my current education as a computer scientist and previous education and experience within music. This, together with an interest in learning more about AI and machine learning systems, led to the conception of the thesis topic which allowed me to both utilise my existing knowledge and learn more about aspects of computer science that I had little prior insight into.

While Prof. Hetland possesses extensive knowledge about a wide variety of CS subjects, the musical aspects of the thesis warranted a secondary source of academic support. As such, Prof. Hetland suggested that I contact Prof. Øyvind Brandtsegg at the Department of Music, NTNU for further guidance. Prof. Brandtsegg showed great interest in the work being conducted and accepted to become a co-supervisor for the project. Together, their shared expertise has proven immensely valuable throughout the thesis development.

This thesis assumes that the reader is familiar with general concepts of both computer science and music theory, although some effort is made to make relatively complex systems within both fields understandable by including some basic theory on each subject. Nonetheless, time constraints and the scope of the thesis prohibits extensive descriptions of every necessary aspect and the reader is advised to self-educate on any central topics that remain unknown or confusing after being mentioned and/or briefly described here. Some sections contain references to external sources of knowledge to simplify this process somewhat, and should be judged on a case-by-case basis depending on the confidence of the reader in the matters that are being discussed.

Spring Semester, 2019

Contents

List of Figures

1 Introduction

“Ever since the digitization of sound became possible, computer generated music has been a widely researched topic. While the computer has become central to the production and distribution of original music, the creative process is still dominated by the human mind. The aesthetics of art have been proven difficult to define quantitatively, due to its highly interpretive nature. Despite this, numerous attempts have been made throughout the history of computers at creating digital systems that are capable of not only imitating human artists, but also generating completely original content. The results of these experiments are diverse, ranging from open-source software to commercially recorded music created by AI. The fact remains, however, that in the wide landscape of music, computer compositions are still far and few between.” [1]

Such were the introductory words which initiated the precursor research into possible avenues for furthering the scientific effort of using digital technology to generate music, with emphasis on deep learning systems. During this time, a large amount of ideas, methodologies and implementations that trace all the way back from the inception of computer science to cutting edge technology today have been explored. Although a decisively successful approach is yet to be defined, one becomes exceedingly aware of the awe-inducing magnitude of time and effort that has both historically been, and is contemporarily being, dedicated by many great scientists to capture the essence of this creative art in digital form. From the pioneering experimentation of Hiller and Issacson [2] at IBM to ongoing projects conducted by industry giants like Google, the goal is still the same; finding a method of consistently and autonomously create aesthetic music by the click of a button and, by extension, defining a conclusive formula for how art is made. While this thesis does not presume to revolutionise the field, it’s contents are written, described and presented with the intention of making *some* contribution to the science of music, however small it may be.

Throughout the following chapters, we first take a retrospective look into the previous discoveries through the reiteration of the literature review, which eventually leads to the domain of one of the hottest topics of computer science today; machine learning. Where 50+ years of specifically tailored algorithms appear to achieve some, but not universal, success, neural networks have proven themselves as a very promising avenue of seemingly "uncapped" potential. While we already know that they are incredibly useful for finding patterns in huge, chaotic systems that are way beyond human comprehension, the limits of its application to music generation is still unknown. It would, after all, be somewhat ironic if this was also a means to achieve greater insight into what is, arguably, one of the most fundamentally human things in the world.

There are however, as the research has revealed, some challenges to their use that are difficult to mitigate due to the hands-off, black-box nature of neural networks. A developer can spend all his time trying to accommodate and improve upon the learning process, but it is oftentimes not feasible to exactly predict the nature of that which is emitted as output. An article published in 2017 by Briot et al. [3] pinpointed four glaring issues that seem to be pervasive among the music generated by deep learning systems alone:

- Difficulty to **control** how well the generated output adheres to the various constraints that comprise a conventional piece of music.
- The challenge of making a neural network learn the concept of musical **structure**, as this is an abstract concept which is not necessarily easy to derive from the mere selection of notes alone.
- Promoting **creativity**, as this is somewhat of a fundamentally contradicting concept to what a NN is usually supposed to do.
- Identifying avenues of **interactivity**, as the black-box nature of deep learning leaves little room for active intervention.

While it is possible to attack these problems internally by using techniques that are still being pioneered by field experts, it is also important to realise that it is not ultimately necessary to achieve the goal by one methodology alone. On the contrary, the output, however imperfect, might be the input to another algorithm which seeks to only tailor and adjust instead of invent and create all by itself. By the use of a multi-layered process, it alleviates much of the burden to achieve every desirable aspect within a single methodology and distributes it among several which might have complementary strengths and weaknesses.

It is therefore the purpose of this thesis to explore an avenue which involves the use of a neural network in conjunction with a *combinatorial optimisation algorithm* to produce original music that belongs to an especially strict musical style; the Bach chorale. While there already exists software that achieves notable success for a similar style by the use of neural networks alone, namely *Deep-Bach* by Hadjeres et al. [4], this would a different approach to the same problem. There is both a musical example uploaded to youtube [5], and a presentation held by Hadjeres on vimeo [6] where he provides great insight and data about the perceived quality to listeners through a survey, but there is some uncertainty of how well the chosen notes and musical structure adheres to the strict rule-sets of the chorale style itself. For instance, one major rule of Bach-style chorales is broken already by the fourth beat of the piece; where all four voices move in the same direction simultaneously. According to Bekkevold [7], this is only allowed during the final cadence and is an illegal motion otherwise. As such, this thesis attempts to implement a system which is guaranteed to generate music that is entirely compliant to the style by using a constraint based optimisation approach to adjust any inaccuracies or illegalities, and ensure the validity of some neural network output throughout the piece.

To set some clear goals, the thesis proposes a set of research questions that are to be answered, and a set of implementation milestones that, if achieved, hopefully results in a functional prototype of software which generates valid and aesthetically pleasing music in the style of a Bach chorale.

1.0.1 Research Questions

1. What are the explicit rules of a Bach chorale?
2. How can the various elements of musical information best be represented as digital data?
3. What optimisation strategies are suitable for this problem. Is it solvable in reasonable time?
4. How is a neural network model designed, implemented, trained and used?
5. How should the chosen optimisation strategy be implemented to achieve desired results?
6. How well does the generated output, if any, adhere to the requirements of the Bach chorale?

While some of these have already been explored by the research project, namely the inner workings of a neural network and its use, there is still some ways to go before a prototype is completed. As for this, the set of goals that are chosen to mark each milestone along the journey is as following:

1. Design and implement a neural network which has the potential of generating some musical output,
2. Train the model and generate music,
3. Identify a suitable optimisation algorithm and make any necessary adjustments or compromises such that it is applicable to the problem space,
4. Implement a system which translates the musical data into manipulable and operable data for the algorithm,
5. Represent and implement the rules of the Bach-style chorale in such a way that it is usable by the algorithm and fulfils the desired purpose,
6. Run the optimiser using the neural network output as input, and compare the results.
7. Rejoice.

2 Background

Note: This chapter is a direct excerpt from the 2018 Autumn research project, and is reiterated here so that the reader can be brought up to speed on the history behind the chosen subject field.

2.1 The origins of Computer Music

In 1950, the CSIR Mark 1 was the first known case of a computer playing a melodic sequence of notes [8]. The Australian machine which would later be redubbed as CSIRAC lacked any sort of visual display or alphanumeric input peripherals, as is typical for early computers. Instead, the raw *sound* of the computer was often used as diagnostic feedback for operators, and it was not uncommon to purposefully program new sounds that would signify the status of operation. As a result, audio research and music generation was a natural domain for early experiments with digital computers. CSIRAC was consequently programmed to play many different melodies from popular music, but the results were unfortunately never recorded. Some years later, effort was made to reconstruct the music created by CSIRAC "within the accuracy of better than one percent of the waveforms that would have been heard at the time the pieces were originally played" [9]. Two samples of the reconstructed music can be found and heard on the webpages of the university of Melbourne.

Contemporarily, progress in modern computing was also made in England and the USA. For the former, the Small Scale Experimental Machine (SSEM) was developed towards the end of the 1940s, and became the world’s first commercially available general-purpose computer under the name Ferranti Mark 1. It was programmed by research physicist Cristopher Strachey to play music merely a year later than its australian cousin, in the form of the british national anthem "God Save the King". Although it was a monumental task to write programs for CSIRAC and Ferranti Mark 1 that produced music, the results were generally regarded as a novelty and not worthy of further investigation in comparison to the more "serious" applications of computers.

The next milestone was when the digital-to-analogue converter (DAC) was used by pioneer scientists Max Mathews and John Pierce to synthesise waveforms by computer at the New Jersey Bell Telephone Laboratories in 1957. Although the research was originally aimed at telephony, Mathews realized the wider potential of the program and invited composer James Tenney to experiment with it for music generation in the early 60s. The precision and control offered by the DAC spurred a different kind of interest from the scientific community in the USA than in other countries, and allowed Mathews et al. to continue their research for decades. This work would eventually culminate as the more advanced MUSIC-N [10] family of programs, defined several industry standards and marked the US as the leading scientific community for computer music. Csound [11], a popular computer music program that was developed in 1985 as a successor to MUSIC, is still used today.

2.1.1 Synthesizers

A possible definition of computer-generated music is "music composed by, or with the extensive aid of, a computer" [12]. Although much of the music today falls within this category, this paper uses it to specifically refer to music where the computer is not only used but also vital for the creative process. Still, this boundary can be somewhat unclear, especially when considering the technological advancements of the 60s and 70s with the invention of the synthesizer. In 1964, Robert Moog debuted the first commercially available (and affordable) synthesizer, which would revolutionise the music industry. The use of the Moog synthesizer was quickly adapted by popular bands of the era, as well as allowing for singular musicians to compose, arrange and record music by themselves. One such artist was Walter Carlos (later Wendy Carlos) who released the album *Switched-On Bach* in 1968, which at the time became one of the highest selling classical albums to date. The recording was time-consuming due to the fact that the synthesizer could only play one note at a time, and was somewhat unreliable in pitch. However, the end result was a major success and can be said to be the point in time where computer music was first commercially relevant [13].

Following the success of the Moog synthesizer and its successor, the MiniMoog, much progress was made in the field of digital sound synthesis and its availability to musicians and producers. Becoming a staple of most contemporary popular groups, original sound created by computers became commonplace within the landscape of music. In 1973, Yamaha entered the synthesizer-business with the Yamaha GX-1, and continued to pioneer the field for many years. The Japanese company licensed the first digital synthesis algorithm, *frequency modulation synthesis* (FM synthesis), and used this technology to develop the first digital synthesizer prototype in 1974. Other major actors in the industry included Roland, Casio and Korg. The development of the Musical Instrument Digital Interface (MIDI) and new and better digital synthesizers continued towards the next millennium until digital software synthesizers marked the next achievement within the field by the turn of the century [14]. Software synthesizers (softsynths) are plugins or standalone software that allow for digital synthesis without needing the dedicated hardware of traditional synthesizers. The software synthesizer often works together with some host application that acts as a single interface for all programs or plugins required to record, arrange, process or generate music. Many different such host applications by various proprietaries are readily available for personal computers, and allow for widespread musical experimentation. The softsynth is also easily maintained and updated by its manufacturer and is highly modular with regards to the many different algorithms for synthesis. It has therefore become integral to generate or reproduce sound, especially to represent the outputs of personal computer programs such as the ones we are investigating in this project.

2.2 Computer as composer

In parallel with the development of commercial music technology for popular use, work continued on the creative aspect of music composition by computer. In the book A Composer's Introduction to Computer Music by William A. S. Buxton [15], he separates the historical use of computers in the compositional process into two groups; programs "which on being initalized, would generate

’musical’ structures without further intervention by the composer (composing programs), and those which serve as ’aids’ to the composer in carrying out lower level compositional tasks (computer aided composition)". In this section we take a closer look at some of the initial experiments that sparked an entire movement of research for many decades to come.

2.2.1 Early stochastic experiments

In general, composing programs are dominated by a divide between seeking to create aesthetic music deterministically and using stochasticity for generation. On one hand, deterministic programs are easier to control and often produce a higher quality of output if one is to critically listen to it. However, as critics may say that a computer simply following deterministic processes is not creating anything original and just reflecting the programmers personal aesthetic tastes, some researchers were quick to adopt stochastic methods as a way to attain some semblance of computational creativity. Also, the purely deterministic approach weighs heavier on both the programmer and the user to correctly ascertain and define all rules and input parameters of the generation process, since a given input will result in the same output. We take a look at some of the earliest experiments that delve into both worlds.

The Illiac Suite

In 1957, professors Lejaren Hiller and Leonard Issacson collaborated to create the ILLIAC I computer which generated what is generally accepted as the first score composed by a computer at the University of Illinois. The composition, named Illiac suite (String Quartet No. 4) [2], is structured as having four movements, each being an experiment using different techniques to demonstrate some compositional motif. The first movement concerns itself with generating a *cantus firmi*, a melody which is supplemented by polyphonic voices. The second is a more advanced four-voiced polyphonic segment with counterpoint, while the third presents a contemporarily modern style with chromatic scales and expressive rhythms. The fourth and final movement uses Markov processes to generate random sequences of notes where each note chosen is chosen from a probability distribution created by previous note(s). As Buxton states, although the basic "model" used by these experiments has many names, e.g. the "Monte Carlo" technique, finite state machine or table driven generation, etc., the underlying principle is simple and can be expressed as a three step process:

- **Initialisation**: First, a table of rules or conditions must be defined to determine *valid*, or *legal*, notes.
- **Generation**: A note is generated.
- **Testing**: The note has its *legality* tested against the preset rules from the initialisation step. A note that conforms to the rules is appended to the score, while one which does not is dropped.

The generation and testing steps are thus repeated a number of iterations until the score is completed.

The results are impressive, and the first two movements clearly conform to classical music of at least some quality, while the third is more modern in style and contains many interesting variations and rhythmic phrases. There is, however, a prominent divide between the first three and the last movement, which is also the piece that is the most interesting for this project's purpose. The fourth movement stands out as the most chaotic, due to the fact that it encompasses stochastic processes and randomness, whereas the former have been generated deterministically.

Örjan Sandred et al. have analysed the 4th movement of the Illiac Suite in the paper "Revisiting the Illiac Suite - a rule based approach to stochastic processes" [16]. The movement was generated in a fixed rhythm, an ostinato eight-note pulse, and the computer was only concerned with the pitches of each note contained within this rhythm. Probability tables were used to hold the vocabulary of melodic intervals, at first set to only repeat the starting note. The tables were then updated every second measure to gradually include intervals of varying size. The most harmonic (octaves, fifths, fourths and major thirds) are introduced first, before moving on through the spectrum of the intervals in the western classical system. The final intervals introduced are the disharmonic intervals of the minor second, major seventh and the tritone. The simpler (more harmonic) intervals would always have a higher probability to occur, but the impact of preceeding intervals on the probability table was also increasing throughout the generation process.

One problem with this implementation is the fact that such a stochastic system only controls a single dimension in the music at a time. In the Illiac Suite, the resulting harmonies of the four randomly generated melodies is not considered, and the results are therefore erratic and unstructured in that regard. Also, the blank initialisation of the probability tables, and their subsequent population of every possible melodic interval results in the piece having no melodic context. For a human listener, it is difficult to find some recurring motif which is a staple of most western music. Sandred et al. state that although Hiller and Isaacson were aware of this, and made several attempts at addressing it, the end result bears little evidence of any such achievements. The paper concludes that while probability distribution is useful in music generation, using it as a part of a constraint based system presents opportunities to better control the overarching structure of the music.

2.2.2 Xenakis and the ST computer

Iannis Xenakis was a Greek-French composer, music theorist, architect and engineer who was an important pioneer within western music in the 20th century after World War II [17]. According to Buxton, his experiments often used mathematical concepts to create music, both with and without computer, such as set theory, game theory and various statistical models.

Xenakis developed a structural theory to music based upon language processing. Instead of focusing on the individual sounds, pitches or notes, he deemed the best building blocks for music to be groups (or "clouds") of sound that function well structurally within a piece. Each cloud could be described by its properties, the "speed, colour, density, and shape", and would define the characteristics of each group. This theory sparked attempts to create a meta-language for music, where a musical cloud or sequence is chosen from a vocabulary of clouds by some probability. The contents of each cloud would also be generated stochastically as a result of the specifications given by the user. Using this theoretical foundation, he implemented the ST computer [18] which was used to generate many compositions, the first being *ST/10-1 080262* [19]. Keller et al. state, ST/10-1 "can be viewed as one among the many possible realisations of a particular compositional model." ST uses a number of parameters, including attack time, instrument selection, pitch, duration, dynamics and glissandi to generate the contents of each cloud and, subsequently, each piece.

2.2.3 Koenig's Projects

While teaching at the university of Utrecht, Gottfried Michael Koenig developed two computer composition programs, succinctly named Project 1 (1964) and Project 2 (1966) [20]. Project 1 is a functional program that uses a specified compositional model to generate music that adheres to the serial techniques (Wikipedia [21]) that were prevalent in the 1950s. The program does not allow much user customization, and Buxton comments that "basically, the same process generates each piece, with only random variation". Serial techniques are by themselves quite deterministic, but some stochasticity is introduced by having a "stockpile" of legal permutations and deciding randomly which ones to use at a given time [22]. The limited functionality could be due to the program being intended for personal use only.

The second program, Project 2, was an attempt to generalise the applications of its predecessor. The user is free to specify the compositional rules of the model, and several musical principles are available to generate. It is more capable of polyphonic considerations, as well as other structural details. The variable input parameters include instruments, rhythm control, different harmony principles, dynamic indications and articulation modes. According to Buxton, the program shows some flexibility and is able to produce music of diverse natures, although it requires accurate specifications from the user. A short description of both projects and their central ideas have been published online [23].

3 Existing Work

Note: This chapter is a direct excerpt from the 2018 Autumn research project, and is reiterated here so that the reader can be brought up to speed on existing work and experiments that have already been conducted within the same domain. It also describes the main motivational focus of the thesis, some of which was mentioned in the introduction.

Moving away from a historical perspective of computer generated music, we now take a closer look at some of the more common practices and limitations found when applying various fields of computer science to the music generation problem. Keeping in mind that the subject has been rigorously researched for roughly 70 years at the conduction of this project, there is simply too much data available to cover it all. A critical selection of relevant methods that have produced results is made.

3.1 Music generation using statistical models

Throughout the majority of its history, stochastic computer music generation has revolved around the use of statistical models. The typical representation of music in this sense consists of a sequence of musical *events*, each assigned with a probability of occurring at a given time. An event might be a simple note with a specific pitch, a chord with a certain dynamic or even independent melodies accompanied by a harmonic progression. Using these events and their corresponding probabilities, a statistical model captures the various recurrences and musical patterns in a specific genre, style or *class* of music [24]. If the model is successfully applied, it should produce music that consists of events that are of high probability within a given class, and avoid those that aren't.

Conklin separates the domain of statistical models for music generation into two subgroups, *context models* and *complex statistical models*. Context models are the most prevalent and simplest models, and include Markov, hidden Markov, n-gram and finite state models. In general, these models have many qualities that are obviously desirable in a music composition process:

- the probability of a given event can be made dependent on previous events occurring,
- they are relatively easy to define and implement,
- utilising efficient data structures, they often allow for real-time processing,
- individual calculations are fast, as they are products of sequential probabilities,
- after the model is complete, the generative process is relatively simple.

As a caveat, a specified context model also come with severe limitations in its applications and creative abilities.

On the other hand, complex statistical models retain many of the positive qualities of context models, but also potentially mitigates some of their limitations at some cost. Such models revolve around the idea that the nearly infinite amounts of possible musical events can be embedded as a finite group of properties less numerous but of similar function. Using *formal grammars* as a way of portraying the different properties of musical events allow for a much more versatile treatment of a given musical phrase, and does not require that it is in the exact form of a pattern already analysed by the model. Although very powerful in concept, these models are naturally more complex to define and instantiate as their name would imply. It also follows that despite the capabilities of context-free grammars, we are yet to define any semblance of a "complete" grammar for music.

For further analysis, Markov models are chosen among the context models, while an interesting application of L-systems for music generation is presented as the complex model of choice.

3.1.1 Markov Models

The use of Markov models for computer composition has been a popular approach to the problem in general, and was one of the first as shown by the Illiac suite. Due to their natural applications for generation by computer, this section shortly describes its function within the field of implementing composing programs and also highlights some of the drawbacks related to using it.

The usage of *Markov Chains* is obvious as it allows for the probability of a specific event at a given time to be context-dependent upon the occurrence of a previous event. McCormack [25] provides a description of a Markov model that only considers the pitch of notes for ease of explanation. In short, the model is represented using a *transition matrix* which holds the probabilities of each event. How the events themselves depend upon each other is related to the order of the model, where models of a larger order have more cross-dependencies within the matrix. The transition tables can be represented as finite-state automata by converting them into *event-relation* diagrams.

Additionally, he also mentions some of the problems related to its use. As the probabilities within the transition tables need to be populated, some music must be fed into the system to avoid purely random values. After the probabilities are populated, the resulting music will only be of a similar style as the input. Also, the size of the transition tables quickly becomes very large as the complexity of the model and length of the music increases, and significant computational effort is required for the program to work in real time. Finally, McCormack asserts that the limitations of any resulting musical piece is a direct consequence of the rigidity of the transition table, providing little versatility or variance. To even consider musical structure requires very complex micromanagement of events and their representation in the model as individual models in multiple layers.

3.1.2 Grammar Based Systems

In 1956, Noam Chomsky formalized the generative grammars for linguistics and computer science while working at the Massachusetts Institute of Technology [26]. Stating that "no finite-state Markov process that produces symbols with transition from state to state can serve as an English grammar", Chomsky defined the *Chomsky Hierarchy* for grammars which classifies different types of grammars as having an increasing strictness to their production rules in correlation to its rank. The types of grammars within the hierarchy are:

- **Type-0: Recursively enumerable grammar** which describes all languages recognisable by a

Turing machine.

- **Type-1: Context-sensitive grammar** which describes all languages recognisable by a linear bounded automation.
- **Type-2: Context-free grammar** which describes all languages recognisable by a non-deterministic pushdown automation.
- **Type-3: Regular grammar** which describes all languages recognisable by a finite state automaton.

Although the type-0 grammar is theoretically more powerful, the higher rank grammars have proven useful within many fields of computer science, including music generation. Following Chomsky's definition, a grammar based approach for implementing composing programs quickly became popular and has since achieved notable success. Music is often (rhetorically) referred to as a language in itself, being syntactically and semantically rich, although its literal meaning is yet to be defined. Chomsky summarises his paper by stating that formal grammars may allow us to "picture a language as having a small, possibly finite kernel of basic sentences with phrase structure ... along with a set of transformations which can be applied to kernel sentences or to earlier transforms to produce new and more complicated sentences from elementary components." If we view music as a language, formal grammars can be applied if the kernel of basic musical sentences and their phrase structure is well-defined.

McCormack mentions work by Buxton et. al, along with Roads and Holtzman's Generative Grammar Definition Language compiler, as examples of software built on Chomsky's hierarchy that is usable for music generation. He also mentions Kevin Jones who has done further research into context-free multidimensional space grammars. These grammars can be used to generate multidimensional geometric shapes that could possibly portray a higher level representation of the different musical attributes.

L-Systems

L-systems were invented by the Hungarian theoretical biologist and botanist Aristid Lindenmayer in 1968, and is a form of string rewriting grammar. Although originally intended as a mathematical model of cell development and plant topology, it was later adapted in several ways by other scientists, most notably by the polish computer scientist Przemysław Prusinkiewicz (differential L-systems). Its applications include visual modelling of plants in computer graphics, general modelling of natural forms, image processing and modelling of human organs.

Chomsky's grammars are sequential in nature, meaning that a sentence is interpreted in-order. A given symbol is only analysed once, and any subsequent symbols do not affect the outcome of an interpretation (no look-ahead). McCormack proposed using Lindenmayer systems to achieve parallel rewriting of sets of elements as an improvement upon this shortcoming. In his own words, "L-systems provide a compact way of representing complex patterns that have some degree of repetition, self-similarly or developmental complexity", which in themselves are key when describing music.

For a more thorough explanation of L-systems the reader is advised to revise McCormacks paper, but a short summary is as follows:

The ordered triplet

$$\mathbf{G} = < \Sigma, P, \alpha >$$

is a DOL-system (deterministic, context-free L grammar), where

- Σ is the alphabet of the system.
- P is the finite set of productions or rules.
- α is an axiom, a non-empty word which is an element in Σ.

If there are no other productions for a letter a in an axiom α, the identity production $a \rightarrow a$ is included by default. The iterative process consists of processing the productions of all letters in a word in parallel, beginning with the axiom. In simple musical terms, one can view each letter as a note, and a word a sequence of notes. By iterative processing (replacing with the new notes defined by the production rules), the resulting string becomes a generated melody.

To introduce some randomness to this process, McCormack suggests using stochastic grammars where a given letter can have several production rules, each with a corresponding probability. A C might become both a G and an A with equal probability (1/2), resulting in more variation in the generated sequence. Moreover, it would allow representation of Markov models within the production rules.

Further improvements include parametric extensions to be associated with any letter in a word to specify volume or dynamics, or mathematical expressions can be used as part of the production rules to introduce deterministic constraints that influence their probabilities. Finally, the DOL-grammars can be extended into hierarchical grammars where elements on the right side of a production can be complete grammars in themselves.

As the paper was published as a proof-of-concept for a L-system composing program, a complete version using this methodology has not been evaluated, but McCormack emphasises its promising potential. Using L-systems, he claims a non-strict improvement upon regular grammar based systems, but admits that it is heavily dependent on user interaction in its current state. He mentions the possibility of applying a "novel variation of the genetic algorithm" to create better results. We will further investigate the applications of genetic algorithms for computer composition later in the chapter. Finally, he concludes that the current state of the method, although promising, does not "exhibit creative judgement on its own".

3.1.3 EMI

David Cope's Experiments in Musical Intelligence (EMI, or EMMY) stands as one of the most prominent and successful examples of composing programs. The experiment was sparked by a writers-block that Cope experienced in the early 1980s, and was initially intended to be a tool for computer aided composition [27]. The idea was to create a program that could capture his own musical style, and generate a suitable continuation of arbitrary length (ranging from a singular note to one or several measures) to sequences of his own devise that he would supply as input. However, a lack of data regarding his own musical style prompted him to create programs that analysed the complete works of great classical composers instead. At first, Cope attempted to create a strict rules-based system for generation, using his own understanding of the composing process as source of inspiration. While the music produced by the program was "basically correct", in his own words, "having an intermediary - myself - form abstract sets of rules for composition seemed artificial and unnecessarily premeditative". Such an approach would also require him to create the rules for every desired musical style or genre separately, which would be excessively time-consuming. This realisation lead to Cope deciding to utilise music stored in databases as source for these rules instead, based upon the idea that each piece of existing music contains an instruction set for creating pieces that have similar structure and style. By deconstructing, analysing and reassembling existing works, original but stylistically recognisable music could be generated. The design and grammar-based underlying algorithm is described in Cope's paper, "Computer Modeling of Musical Intelligence in EMI", published in 1992 [28]. EMI uses a reflexive pattern-matcher in combination with an augmented transition network (ATN), which is a graph structure that uses Markov models to parse and identify regularities of natural languages.

Fig. 1 found in the paper fully describes the design of the program, but a short summary is reiterated here. The existing works are supplied as input in a simple piano-roll format, and are kept unaltered throughout the process for later statistical comparison with the program output as a target criterion. If supplied, an initial configuration is loaded from a variable source file, or is otherwise set to a default value. The pattern matching process looks for what Cope defines as "signatures"; patterns that are repeated or found in two or more of the input works of the same style. Viewing these signatures as highly descriptive of the pieces analysed, they are stored in a dictionary from which the ATN pulls when replicating the music. The signatures are inserted at what Cope asserts as logical, not random, positions during the replication process, dependent on several (unknown) factors specified by the algorithm. The resulting generated piece is then statistically analysed against the original input and is deemed successful if acceptable similarities are found (e.g. percentages of repeated notes). A successful piece is stored, while a failure initiates a new iteration with adjusted variable configurations. A thorough walk-through of the iterative process, as well as some actual coded functions and their uses are included in the source paper, which we will not go into at this time.

Cope, who would later publish a book in 1996 titled "Experiments in Musical Intelligence", continued to work on the development of his program, and the results are impressive. EMI has produced both scores for live performance by orchestra, operas and music that would later be commercially recorded. Music composed by the program can also be found online, exemplified by a Vivaldi-style piece [29] and a Bach style chorale [30]. For an experienced classical listener, it is obvious that these works bear a close resemblance to original works by the composers in question, and serve as notable imitations of real musical genius. However, in an interview with The Guardian, Cope reveals that the creative impact and quality of the music generated by EMI was not universally lauded by the musical establishment [31]. The project was finalised in 2004, and Cope moved on to developing its successor, a program named "Emily Howell". Emily builds upon the algorithm and design established by EMI, but functions closer to the original intent of the project as a program for man-machine collaborative composing. Instead of generating music in the style of a specific composer, Emily suggests phrases that are built from the intimate, compounded knowledge of a wide selection of composers (36 in total), including Cope himself. Consequently, Cope and Emily have also created music together, some of which can be found online [32].

3.2 Genetic Algorithms

Although the idea of borrowing prime principles from nature for algorithm development was already prevalent during the days of Alan Turing [33], genetic algorithms were first made popular during the 1960s. At the forefront of evolutionary programming was the American scientist John Holland who published the defining book on the subject, *Adaptation in Natural and Artificial Systems*, in 1975 based upon a series of experiments conducted by Holland and some of his students at the University of Michigan. Being a strong proponent of using natural phenomena as the foundation for digital problem solving, Holland and Goldberg defined genetic algorithms (GAs) as "probabilistic search procedures designed to work on large spaces involving states that can be represented as strings" [34]. GAs achieve parallelism by using a distributed set of samples from the domain space to generate new strings (samples), and subsequently biases the future populations based upon regularities found. Formalising and utilising a framework called the *Holland Schema Theorem*, programs using genetic algorithms gained a "substantial advantage" over traditional methods when operating in complex spaces. GAs are also very versatile and do not require a specific problem structure or problem specific knowledge to function [35].

Holland has written an introductory tutorial to genetic algorithms which further describes their basic methodology [36]. In short, the genetic algorithm technique was extended into a classifier system which consists of a set of rules which individually perform some action whenever its conditions are satisfied. Each condition and action is represented as a binary string where each bit corresponds to the presence (1) or absence (0) of some (preferably simple) characteristic in the input and output. The bits in the output can be programmed to initiate some action if so desired. A fitness function is chosen to evaluate the resulting strings. The high scoring strings are propagated through the iterative process and combined in various ways while the low scoring are removed completely. In this way, strings that show some improvement or partial solutions are prioritised and combined into potentially even more sophisticated solutions. This ability to explore multple local minima (or maxima) in the problem space simultaneously allows the algorithm to outcompete conventional techniques such as local hill climbing.

Many projects have been conducted on applying genetic algorithms to music generation problems. Among these, Andrew Horner and David E. Goldberg discussed its usage in a computer-assisted composition program. Although not designed to be autonomous, the design illustrates the simple usage of GAs to such an end. By using the concept of *thematic bridging*, which is the transformation of an initial musical pattern into some final pattern by reordering or modifying various events and sequences found within it, the final result consists of a concatenation of partial patterns. By using GAs, the transition rules to be executed are chosen by the algorithm using some associated probability of initial random value. Examples of possible transition rules are presented as deletion, addition, mutation, rotation or exchanging of notes or sequences. During iteration, the algorithm applies a fitness function that continuously adjusts the weights of the operations.

Horner and Goldberg state that the generated results are to them aesthetically pleasing, but that this specific application of GAs also raises many of the issues that are common in other methods of generating music within the same methodology, namely the considerations of the operation set, encoding and fitness functions.

3.2.1 GenJam: real-time generation

In 1994, John A. Biles developed a model based on genetic algorithms that simulates a novice jazz musician learning to improvise jazz solos [37]. Requiring constant user feedback, it demonstrates how GAs are also applicable to real-time music generation. Also, the program is fully functional and has been continuously improved after its initial publication. It has its own website [38] which tracks both upcoming and past live performances of the program at work and other related publications. For the purposes of this project, GenJam serves as a fine case study into the actual realisation of a relatively successful musical composition program. For the interested reader, a TEDx demonstration of GenJam can be found on youtube [39].

Biles opens his paper with re-stating that Genetic Algorithms are well suited for searching within complex problem spaces, which is in theory a perfect match for the task of music composition. A short description of the program design is as follows: The inputs for initialisation for the basic GenJam described in the paper are:

- **A progression file**, which provides tempo and rhythmic style, the number of solo choruses and the chord progression.
- **Midi sequences** for each instrument in the backing track.

The program defines a phrase as a sequence of four measures and builds choruses based upon these two populations (measures and phrases) forming a mutually dependent hierarchy between them. At any point in time, the user can present feedback to the program in the form of letters g (good) and b (bad), or multiples of these for emphasis. This feedback is considered by the fitness function when evaluating a given measure or phrase, and it's resulting values are stored when the solo finishes for later use. As with most machine learning programs, GenJam has different modes of operation: training, breeding and demo. In learning mode, fitness values are updated without genetic operators, while breeding mode considers these as well to generate new mutations. Much like a classifier system, GenJam uses the entire populations when generating solos instead of just those it deems as the "best", with the intention of creating a varied selection of musical ideas to use instead of trying to generate only one "perfect" solo for a given tune. Instead of repeating the exact definitions of the binary strings and their implementations with regards to these populations, the reader is advised to read Biles' paper "GenJam: A Genetic Algorithm for Generating Jazz Solos" for further details. A typical training regimen for GenJam consists of several runs in learning mode before alternating between learning and breeding. Initially, the good results are few and far between, but with frequent and heavy-handed feedback, GenJam eventually produces phrases that "sound good" according to Biles.

In the final part of the paper, some ongoing and potential extensions to GenJam are discussed, most notably the addition of a neural network to serve as a preliminary fitness function for one or both of the populations. Biles concludes that GAs can be a useful tool for "searching a constrained melodic space", and expects many new breakthroughs for such applications in the future. Due to the apparent sophisticated state of GenJam in the 2012 TEDx video mentioned earlier, it would seem that he was correct.

3.3 Deep Learning

With the resurgence of machine learning, experiments regarding the use of deep learning for music generation are numerous. Due to the ever increasing popularity of cutting-edge AI technology, the availability of (free) frameworks and sufficiently powerful hardware, these experiments are not only limited to the academic or scientific communities, but conducted by all manner of technologically curious people. The internet is flush with examples, tutorials, guides or articles regarding music generation by deep learning as such applications have become somewhat of a staple exercise in learning how to implement neural networks in code. Although the scope and intention of these experiments are as wide as they are many, this report only considers a select few of them due to space constraints. This section serves as an introductory glance at how deep learning has been utilised for music generation until now, the level of results that can be expected from a basic implementation and new problems or challenges that have been made apparent. The specifics of what neural networks are, how they work and when to use them is left for the next chapter of the report. In the interim, a neural network can be considered a magic black box that takes some data as input and outputs something else, based upon what the network has previously *learnt* from large amounts of analysed data.

3.3.1 Initial experiments

In 2001, Chen and Miikkulainen [40] at the University of Texas used neural networks to generate simple melodies and a set of rules defined by analysing the music created by the composer Béla Bartók to evaluate the generated result. Recognising that the network needs to be able to retain some sort of persistent memory to keep the history of previous patterns, events and notes, Chen et al. elected to use a simple recurrent network (SRN) as basis for their implementation instead of a standard feed-forward network which only considers the current data at any iteration. The process would then allow for feeding and generating one measure at a time, knowing that the system would remember inputs at previous iterations.

Due to difficulties in the music representation as data, they chose to limit the available rhythmic notations to only five notes; whole, half, quarter, eighth and sixteenth. They also omitted rests, dots or other rhythmical elements, and the range of available pitches was limited by a span of three octaves. The output layer of the network would consist of two arrays, one of them representing a relative pitch (in half-step increments) to the singular input pitch. The network would calculate a probability distribution over each of the nodes (half-step) in the array, and the node with the highest value would win. As an example, if the input pitch was C3 and the output node with the highest value represents "-4", the output pitch would become C3 - 4 half-steps = G#2. If at some point the output pitch extended beyond the available range of pitches, it was shifted an octave down or up depending on which end of the boundary it broke. The other array consisted of five nodes which symbolised the duration of the output note (the five rhythmic notations mentioned earlier). Similarly, the node with the highest score would decide the duration of the note. Some additional considerations were made to ensure that the output would fit neatly into each measure.

To make the neural network "evolve" towards good solutions, a genetic algorithm called SANE was used. The compositional rules were used as fitness functions, each rule associated with a weight, and the weight total signified the quality of the measure. The experiment as a whole showed some promise, chiefly that the melodies generated adhered to different sets of constraint weights, contained variations within each set and that they had some melodic structure and pattern repetition or variation. The system was not without fault, and the authors emphasise the distinct limitations and its overtly simplistic nature, as well as the obvious lack of what they call "global structure". They noticed that although each generated measure could be deemed acceptable in a musical sense ("sound good"), there was no overall organisation and the melodies lacked a global flow. Although the paper concludes that this could possibly be addressed by adding more rules, this challenge would later prove to be a recurrent issue for music generation by deep learning as a whole.

3.3.2 Polyphonic generation using LSTM

In the paper "Deep Learning for Music" [41], Allen Huang and Raymond Wu of the Stanford University present some of their methods used and discoveries made while implementing a music generation network for polyphonic music. More specifically, they discuss how to better represent the musical data and the quality of the generated results compared to the common structures of human compositions. Huang et al. begin by specifying two improvements that are desirable with respect to the previous work done by Chen et al. Firstly, the system should produce music with more complex structure and rhythm, and support the use of all types of notes. The network model should also consider the long-term structure of the music and properly utilise the concepts of melodic *themes* and repetition.

Regarding the data representation, the paper suggests using piano roll data as input to properly represent the notions of concurrency in polyphonic music. This two dimensional data structure consists of a time series which tracks the concurrent notes that are active at any given time-step, in this case by using note IDs. In this experiment, the transformation from MIDI to piano roll is done by sampling the MIDI at every eighth note interval and tracking the active notes at these points in time. Polyphonic triads (three note chords) are represented as tokens whose names were concatenations of the note IDs involved (e.g. "60-64-67" for a C-Major chord). In the case of a time step containing more than three notes, a triad is selected randomly from these to reduce the amount of potential tokens.

The model of the neural network consists of a 2-layered recurrent neural network with Long

Short Term Memory (LSTM) [42] units. LSTM units are designed to keep both long and short term data in memory, effectively remembering values over an arbitrary length of time. To measure the impact of using piano roll representation, mirrored experiments using the raw MIDI data as input are conducted in parallel. An embedding matrix is used to map each token into a vector representation, which is concurrently fed into the LSTM during training. A softmax layer transforms the LSTM output into a normalised probability distribution and the loss is calculated by the cross entropy error between each prediction and the actual note supplied by the next timestep (which is what the network is supposed to predict). Some additional specifications are given for the hyperparameters of the model, but we will omit them for now and revisit their function in the next chapter of the book instead.

When generating music, a short sequence is fed into the LSTM to seed the process, and is from this point on given its own output as a new input token instead of using the real notes from the dataset. In this way, each step of the generated sequence is a prediction made upon the result of previous predictions, propagating into what is a new piece of music. A combination of two sampling schemes are used to select a note from the probability distribution given by the softmax layer; one that simply selects the token with the highest probability and one that selects stochastically from the distribution. For more insight on the experiment process itself, the reader is encouraged to read the article at hand.

To evaluate the results given by the experiments, Huang et al. asked 26 volunteers to judge the quality of 3 samples of music presented to them in a blind experiment. The three samples were:

- A 10 second clip of the generated music from the network using a "Bach only"-subset of the training set along with raw MIDI input representation.
- A 16 second clip of the generated music from a similar experiment being conducted by other researchers (Boulanger-Lewandowski et al., first reference in the paper).
- A 11 second clip of the generated music from the network using the piano-roll representation, trained on the entire dataset.

The feedback received indicate that both experiments produced music of at least similar quality as the second sample, and that the music from the piano-roll experiment performed slightly better than the raw MIDI results. However, the authors also note that the sample size was too small to make any significant statistical conclusions.

Finally, Huang et al. state that a character-level multi-layered LSTM is capable of generating music that is comparable to other prevalent techniques in music generation literature, and that it is possible to achieve some musical structure in the results. They emphasise that work needs to be done to develop better metrics of evaluation in regards to the quality of a generated piece of music.

3.3.3 Inherent challenges and strategies to mitigate them

In 2017, Jean-Pierre Briot and Francois Pachet published an article that defines a set of major challenges inherent in using deep learning for music generation [3], while also suggesting some potential approaches to mitigate these issues. First of all, the paper states that "a direct application of deep learning to generate content rapidly reaches limits as the generated content tends to mimic the training set without exhibiting true creativity". Of course, it is debatable what true creativity entails, especially in the context of music composition, but the point is that a neural network will be completely reliant on the contents of the input data set. Simply put, it will never introduce something that *is not already present in the training data in some way*. It is also hard to directly control the generative process of a deep learning system as the contexts of their inner workings during run-time are notoriously difficult for a human to understand.

Briot et al. begin their analysis by juxtaposing the benefits and challenges of machine learning systems versus other methods. On one hand, as a problem domain becomes increasingly complex it is difficult to manually design complete and rigorous rules or constraints. Learning algorithms are also better at generalisation when introduced to new or unexpected inputs, and are less liable to break in such situations. In comparison to grammar- or rule-based systems, these are major advantages to have in terms of both ease of implementation and versatility. Consequently, the drawbacks of deep learning systems are therefore dominated by things that "manual" systems excel at, and the paper defines these four challenges:

- **Control**: The difficulty of enforcing specified constraints such as tonality, note selection, rhythm, harmonic progression and musical motif.
- **Structure**: Generated music is often without any musical structure, e.g. aimless wandering, no melodic and harmonic phrasing.
- **Creativity**: A perfectly trained neural network will properly predict the training data with high accuracy, and therefore runs the risk of copying or plagiarising the input.
- **Interactivity**: The generation process is fundamentally autonomous with little room for intervention.

After establishing these as focal points to investigate, the article presents some potential strategies on how they can be approached.

Control

As mentioned earlier, deep learning networks are not designed to be controlled, as opposed to previous models explored by this project. The internal nodes of the neural network are of a highly distributed nature, and their individual values (or weights) bear little resemblance to the actual output produced at any point in time. Underlining this black-box quality, Briot et al. state that strategies to enforce some control need to be externally introduced at one of the entry-points of the system, e.g. on the input, output or at initialisation through some internal encapsulation scheme.

One way to do this is by **sampling** from the model by applying some constraints to the output generated and only choosing the solutions that conform to these rules. It is worth noting that this strategy should be used in conjunction with other strategies, as there might not exist any such solution to sample unless the network is somehow encouraged to fulfil these criteria. However, depending on the strictness of the constraints and the nature of the generated output, it might sometimes be feasible to employ such a singular strategy if the probability of producing a valid output "randomly" is high enough.

Another strategy highlighted is **conditioning**. By appending some meta-data (e.g. class labels or tags) to the input and adjusting the architecture to consider these, some parameterised control can be achieved. Several examples of conditioning is mentioned in the article, but one of them is particularly interesting to us. Hadjeres, Pachet and Nielsen [4] have designed a system named "Anticipation-RNN", which consists of two models. The main model (Token-RNN) generates on a iterative note-by-note basis, using the previous output as the next input step. Due to the fact that externally enforcing constraints in between these iterative steps can invalidate the internal memory of the network (the prediction is no longer genuine), a second network (Constraint-RNN) is used as a look-ahead for future constraints. The outputs of the Constraint-RNN is used as input to the Token-RNN and summarises future events which the Token-RNN has not yet seen, essentially supplying some anticipation to the generation process. This architecture has been tested on some of Bach's chorales in the *DeepBach* program, and the authors state that the Anticipation-RNN was capable of correctly applying positional constraints.

The idea of **input manipulation** has been utilised by Google's *DeepDream* experiment, and revolves around changing (manipulating) an initial input incrementally towards some target property or attribute. In the case of imaging, an example is to adjust the properties of the image such that some internal units of the network associated with these visual effects have their activation frequency increased (or even maximised). This strategy has been applied to several different applications, including sound generation.

The next strategy mentioned is to use the outputs of a trained neural network as an objective for a **reinforcement** learning problem, which in very simple terms is a technique for teaching an agent to make a sequence of actions that are as optimal as possible through maximising cumulative rewards. The exact design and its constituents are described in the article and will not be reiterated here, as it is quite complex. However, as the authors mention that its results are convincing, it is a natural candidate for further discussion later in the report and the interested reader is encouraged to familiarise themselves with it.

The last control strategy mentioned is **unit selection**, which is to generate short sequences at a time (for example melodies with length of one measure) and selecting units to concatenate based on semantic relevance and concatenation cost. The goal is to at any time select the best (or at least a good) successor to a unit from a pool of candidates. The semantic relevance is defined as the score given to a potential successor unit by some transitional rules learnt by an LSTM, such as how close the successor sequence is to the predicted "optimal" derived from the parent unit. The concatenation cost is similarly deduced by another network, but only considers the final note of the parent unit and the first note of the successor unit. The suggested architecture consists of two LSTM's and one autoencoder. It is noted that the unit selection strategy introduces new entry-points for control which was previously lacking, and is extendable with additional user-specified criteria.

Structure

The seeming "lack of direction" prevalent in music generated by direct application of deep learning can be approached by strategies already mentioned in the control section, and the authors highlight that both the *reinforcement* and *input manipulation* approach could potentially support the introduction of higher level constraints such as song structure (verse/chorus/bridge-relation in popular music), the different movements within a symphony or global dynamics. The unit selection strategy is also considered as a means to impose structure by introducing several layers of generation, where an abstract structure is incrementally built before populating it with musical units. The supplied example is of MusicVAE, a hierarchical architecture that has layers consisting of a bi-directional encoder and two RNNs in a top-down fashion, whose results show an impressive improvement from flat architectures.

Creativity

When considering both the artistic and the potential economic ramifications of generating music that infringe upon copyrighted material, some effort should be made to make sure the generated content is not overly similar to any of the input works. While this is more easily verified after the generative process is complete, it might be desirable to implement such constraints during run-time. Although they inform of the existence of such functionality which has been used during music generation by Markov chains, Briot et al. are not aware of any "equivalent solutions for deep learning architectures".

Interactivity

The issue of interactivity is an interesting one, and depends heavily on the intent of the system. If the aim is to create a fully autonomous composing program, the lack of interactivity is a mute point and is unimportant. However, if the goal is to create a tool to assist or collaborate with humans, deep learning architectures must be used very carefully and specifically. Neural networks are very dependent on internal integrity throughout the whole training and generative process to produce accurate data, and altering variables or parameters within this period could invalidate the entire result. It is also important to note that even with powerful hardware, the training of neural networks can take a long amount of time, ranging from minutes to weeks depending on the complexity of the problem and quality of training data. Such user-induced errors can require a complete restart and loss of any previous attempt. Though is a poor match with how a human composer would work when writing music, it does not mean that they can't be useful in the process.

The article defines the two most common choices regarding how the network is instantiated that have a significant effect on how generation is done. A **single-step/global** generation produces predictions for all time-steps are produced in one single step by the model, while an **iterative/time-slice** generation iteratively calculates each time-step in a sequential manner. As an alternative strategy, an **incremental variable instantiation** approach is suggested. In short, it consists of using separate networks (RNNs) to keep track of time in *both directions* by having one progress forward in time and the other backwards. This produces one network that summarises past events and one that summarises the future. Together with a regular feed-forward network that tracks notes at the current timestep, their outputs are combined and fed into a final feed-forward network that is capable of reproducing sections of the global result at any point in time. Although this strategy requires a customised training and generation regime that is different from conventional neural networks, it has successfully been implemented as a part of the DeepBach [4] program mentioned previously. A human user could rewrite a small section of the music, and the model would be able to recalculate only the affected parts of the output piece.

Briot et al. conclude their paper with optimism regarding the future of deep learning systems for music generation, but reiterate the "open challenges such as control, structure, creativity and interactivity, that standard techniques do not directly address".

4 Technology

Note: The first sections of this chapter are direct excerpts from the 2018 Autumn research project, and is reiterated here such that the reader can be brought up to speed on the most important concepts that are relevant to this master thesis.

This chapter presents relevant technology for the design and creation of deep learning-based systems for music generation.

4.1 Deep Learning: an introduction

As the main focus of the thesis revolves around the implementation of neural networks and deep learning methodology, this section offer a short introduction on what they are and how they work. First, we take a look at the field of machine learning in general and what kinds of tasks it is suited for, and then move on to the specific aspects of neural networks.

4.1.1 Machine Learning

Machine learning is a subset of the field of artificial intelligence, and is the study of using algorithms and statistical models to not only let computers solve complex problems, but also to learn *how* to solve them [43]. By using a knowledge set called the "training data", machine learning systems can locate and identify patterns found within this raw data and use it to make predictions or decisions without being hard-coded to perform this task. In many cases, developing a tailored algorithm for a very complex problem can be very time-consuming and impractical, and it is in these cases that machine learning systems excel. Also, some problem domains are very unpredictable and chaotic, especially when considering the real world, and can often break hard-coded algorithms when they are presented with unexpected data. By using machine learning methodology, we can leave this exercise to the computer, which is much better at finding patterns in large amount of data than we are, and supply it with the tools to quickly adapt to new discoveries.

Task classification

Typically, machine learning tasks are separated into three categories: **Supervised Learning**, **Unsupervised learning** and **Reinforcement learning**. Supervised learning tasks are the class of tasks where both the input domain and output domain are known to the system. The job of the system is to identify the patterns in the input data that result in the desired output, a definition of the mapping between the two. We further separate supervised learning tasks into two groups, **classification** and **regression** problems. In a classification problem, the output of the system is a group, class or label for which the input belongs. For example, a system could be tasked with recognising images that have dogs in them. In this case, the classification problem has two possible outputs, dog or no dog. Regression problems are the set of problems where the output are real values that can be continuous within a given range. If the system is tasked with predicting the temperature for a given day in the future, this is a regression problem as the output can be any (realistic) value within the range of possible temperatures.

In unsupervised learning, no information on the output domain is supplied and the system is left to its own devices for finding structures and patterns in the data that can be grouped or clustered together. The "unsupervised" term is due to the fact that there is no knowledge of what the correct answer is, and the system will therefore have no means of recognising when it is successful or not. Unsupervised problems are divided into **association** and **clustering** problems. Association problems consist of identifying rules that describe the input data in some way, e.g. that "people who have dark skin also tend to have dark hair". Clustering problems are when the system is tasked with finding data that show similar patterns and grouping these together, like grouping growing stocks found in the stock market.

Finally, reinforcement learning systems consist of an actor trying to achieve some specific goal inside a dynamic environment by performing some set of tasks. The actor is given positive or negative reinforcement for every action taken so that it learns the optimal path of actions inside the environment by trial and error. These systems are used in a wide variety of research, e.g. the realms of game theory, multi-agent systems and optimisation.

4.1.2 Deep Learning and neural networks

Deep learning is a subfield of machine learning and specifies a methodology of solving some of the different problem classes mentioned in the previous section. In this case, whenever it is mentioned we are in fact referring to **neural networks** (NN), structures that have their origins in research conducted in the 1940s. Inspired by the biological sciences, neural networks were modelled to mirror our understanding of how learning happens in the human brain. Although we now know that it isn't an accurate replication of this extremely complex function at any stretch of imagination, it is still quite representative in many ways.

In theory, deep learning systems only require one thing to work; a significant amount of representative data. Generally, more training data equals better results given that the quality of data remains consistent. It makes little sense to use neural networks for small data problems (although not impossible) as the whole process revolves around identifying patterns across large, chaotic data and learning how to predict or classify them. Regardless, there usually exists other methodologies that are far more easy to implement and to use in such cases.

Below follows a quick description of relevant concepts in the realm of neural networks, and seeks to serve as a rudimentary introduction to those who have little or no previous knowledge on the subject. As a word of warning, the reader is advised to keep in mind that neural networks are functionally quite complex and that this research project does not presume to reinvent the wheel by presenting all the calculations involved in mathematical terms. **Many important facets of neural networks are omitted**. Little insight is offered into the actual use (training, evaluation and generation of output), as thorough descriptions are space-consuming and can be found elsewhere. Once again, Goodfellow et al. [43] is suggested as a great source of information regarding deep learning systems in general.

The feed-forward neural network

The simplest of neural networks, the feed-forward NN, consists of distributed units called neurons that are grouped in layers. We differentiate between the **visible** layers and the **hidden** layers. In the example given by Figure 1, there are two visible layers (input and output) with 6 neurons each, and one hidden layer with four neurons. Most commonly, the network is fully connected between layers, however there exists network designs which are not. While anyone interfacing with the network can see the contents of each neuron in the visible layers, the hidden layers are not meant for broadcasting as they only exist to supply the network with internal states.

As the name implies, information is only propagated in one direction along each connection within a feed-forward network. When information is allowed to flow sideways or backwards, they are called **recurrent neural networks**. The purpose of this kind of basic network is simply to approximate some function that maps all inputs X to desired outputs Y, or $y = f(x)$ in mathematical terms. By including more than one hidden layer, we can view $f(x)$ as a chain of functions whose length is equal to the number of layers, e.g. $f(x) = f^3(f^2(f^1(x)))$ for a three layer network. Although one could initially think that more is always better, adding more hidden layers than are required to accurately represent the target function is computationally costly. As each neuron is only capable of holding singular values, the number of nodes in the visible layers is often decided by the format of the input data and the desired output data. Each hidden layer may however contain an arbitrary amount of nodes depending on what the purpose of a given task is. Again, an increase in neurons per layer affects performance, so developers generally strive to keep the number as low as possible while still producing accurate output.

To understand how the network learns to approximate $f(x)$, we need to understand **linear-** and **logistic regression**. Linear regression is a statistical method of modelling the relationships between a dependant variable and one or more independent variables. Simply put, if we have a space populated by points, linear regression produces the line that has the least cumulative distance to each point, effectively minimising this error. As the line is continuous, there is nothing stopping it from having any real value at a given point in time, and if its gradient is anything but zero its values will at some point become infinite at both ends. As computers have a better time representing discrete values than real ones, logistic regression is introduced. It has the property of being bounded by some values, namely those defined by an **activation function**. As an example among many, the **sigmoid** function

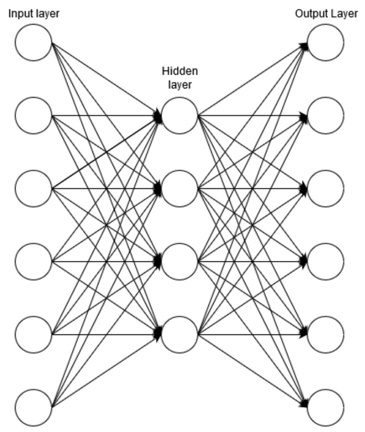


Figure 1: The layout of a basic feed-forward neural network with one hidden layer

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

is frequently used in neural networks, and has boundaries of 0 and 1, respectively. We can use logistic regression for classification problems, by trying to locate a line that accurately separates the classes in space. As the output of a classification is necessarily discrete, the values produced by the regression must also be discrete. We can visualise the results of a binary classification problem using logistic regression as a straight line also, but now the line marks the threshold where the two classes are (probably) separated, as seen in Figure 2.

But suppose our problem space is populated by classes that aren't linearly separable. While straight line shown in the figure does a fairly good job of separating the blue and red points, but it is obvious that this is impossible using only a straight line. What we can do is have each neuron in the network create its own regression line and then combine them. The network can also use adjustable parameters such as **weights** and *biases* to identify which patterns in the data that are more or less useful for this classification. At first, these parameters are randomly assigned, but as the network trains it will gradually figure out which combinations of weights and biases that produce better results. Eventually, the function $f(x)$ may look something like in Figure 3. The process of propagating the values from the input, through the calculations in the hidden layers and to the output layer is called the **forward pass**.

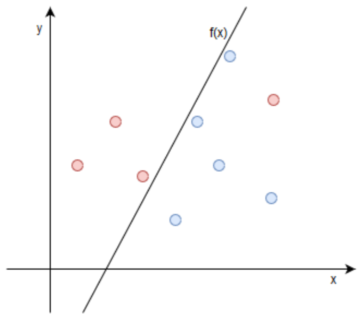


Figure 2: A logistic regression line that tries to separate red from blue

Now, we introduce the concept of **error**. In the case of a supervised problem we already know what the correct result should be, and we define an **error function** which is used by the neural network to calculate how much **loss** there is between the predicted output and the real answer. There are many different ways of calculating the loss, and each have their own merits and drawbacks usually dependent on data format and task. One of the simplest error functions is the **Mean Squared Error**

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2$$

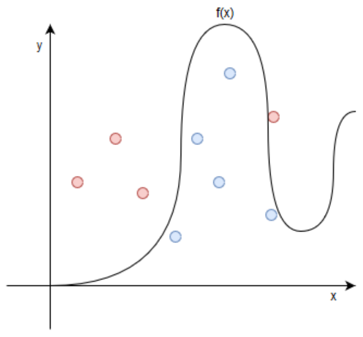


Figure 3: The resulting threshold of combined logistic regressions after training

where n is the number of output nodes, y_i is the target value and $\hat{y}_i$ is the output produced by the network. MSE is well suited for regression problems as it just calculates the average of the sum of the distances from every output value to target value. Other popular loss functions include variations of **Cross Entropy error** for classification problems and the **L1 Hinge error** function for embedding problems. For unsupervised problems, other means of calculating the error are used as the target values are unknown.

As the network wants to minimise this error, an **optimisation algorithm** is used. There are many different optimisation algorithms to choose from, but common among them is the use of **gradient descent** or some variation of it. As the final output is dependent on the adjustable weights and biases W, the network calculates the impact of each of these parameters on the gradients of the error function. By calculating the partial derivative of the error function E with respect to these adjustable parameters $\frac{\delta E}{\delta W}$, the network can find the **gradient vector field** and use it to shift weights towards values that produce smaller errors. Good optimisation algorithms have mechanisms to avoid getting stuck in local minima, e.g. by introducing some randomness to the process. Goodfellow et al. describe many different such strategies in their book in chapter 8.

The last step of the process is to track backwards through the layers and calculate the gradients of the weights in every neuron all the way to those in the first hidden layer. This is called **back-propagation**. Naturally, the depth (number of layers) of the network has a severe impact on the computational power and time required to do this. At any given iteration, we refer to the values currently held by all the internal weights in the system as the *hidden state* of the network. This hidden state will (hopefully) become increasingly more representative of the true function that transforms the input data into the desired output result as it trains.

Together, these steps form the basis of the neural network functionality that allows us to accurately predict a large amount of real world phenomena. One complete iteration on the training set is called an **epoch**, and the amount of epochs required to properly train varies greatly depending on the task, data and model. Since back-propagation is especially computationally costly, it is also often beneficial to divide the training set into **batches** that are collectively processed before calculating any gradients. This often speeds up the process significantly and has even been shown to potentially increase the accuracy of the network in comparison to back-propagating for every single input.

Training, validation and testing

We can typically divide the use of neural networks into two separate processes; **training** and **solving**. The steps mentioned in the previous section are all a part of the training process, where we supply the training data, forward it through the network and back-propagate to evaluate and adjust. To state the obvious, an untrained network is useless as it has had no time to learn how to solve the problem. After sufficient training, it is trivial to produce a solution as it is just a matter of presenting the **test data** and collect the answer from the output layer. When doing this we no longer concern ourselves with optimisation and back-propagation as the network should already have internal states that are representative enough to produce a correct solution, and the network is said to be in **evaluation mode**.

Since training is the arduous process of the two, almost all strategies related to improving network performance are applied there. Being a very sensitive process, some concern themselves only with manipulating the input training data, while others supervise and adjust internal weights and values during and after calculation. As imbalanced data can have a negative effect on results, we can apply **normalisation** to potentially reduce this impact. Other common problems are **under-** and **overfitting**. Underfitting is when the network simply does not learn to model the training data or apply these generalisations to new data. Since underfitting is symptomatic of the model itself being unsuitable for the task at hand, the simplest solution is often just to reevaluate the model or try other machine learning algorithms.

Overfitting is a much more common issue when dealing with network optimisation. A model overfits when it learns the training data "too well", meaning that any uncharacteristic noise in the training set causes negative performance when the network is presented with completely new data. Although it is very difficult to eliminate overfitting completely, increasing the amount of training data, applying **regularisation** and preventing overtraining are common techniques that can help mitigate the problem. Some examples of regularisation are to add a cost term to the loss function which penalises large weight values (**L1/L2 regularisation**) or to randomly select some neurons each iteration and completely remove them and their incoming and outgoing connections (**dropout**) from consideration when doing the forward and backward pass. This randomised removal has been shown to be a very effective tool to improve network performance when overfitting occurs.

As for overtraining, there are techniques that allow the network to detect this by itself. One of them is to remove some of the data in the training set and insert it into a **validation** set instead, effectively creating a representative test set that we already know the answer to. While training, the system can regularly self-evaluate by attempting to solve the validation data in evaluation mode. While the training loss does not reveal overfitting, the validation loss will begin to increase when this limit is reached as the prediction made upon this new data will become worse. This lets the system to know that it will probably not find a better state by training any further and allows it to terminate. This strategy is called **early stopping**.

4.1.3 Recurrent neural networks

When considering the task at hand, it has already been clarified in the last sections of chapter 3 that feed-forward networks alone are not well suited for generating music. Recurrent neural networks (RNNs) have the ability to retain the history of previous inputs in a sequence through time, and are therefore the pivotal class of neural networks for this project.

The basic gist of an RNN is that, given a sequence of arbitrary length as a vector, the **state** h_t of the hidden layer at time step t is forwarded to the hidden state h_{t+1} at the next time step as demonstrated by Figure 4. Goodfellow et al. describe the transferred hidden state as a "lossy summary of the task-relevant aspects of the past sequence of inputs up to *t*" due to the mapping of an arbitrary length sequence to a fixed length vector h_t. Depending on how the training of the network is performed, some aspects of the state might be selectively prioritised or forgotten to increase its precision.

One of the most prevalent problems that face RNNs is the case of vanishing or exploding gradients due to the reapplication of the same functions multiple times every time step across long sequences. A common cause is when back-propagating by chain-rule, the outputs given by activation functions create many multiplications of very low values before reaching the front layers. When gradients become too small, the training rate slows down or might even stop entirely. Also, in the

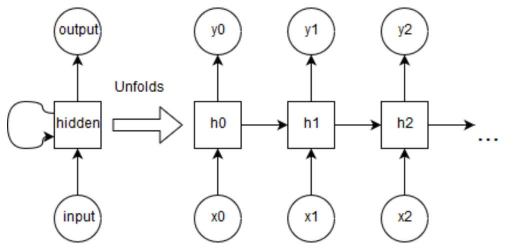


Figure 4: An unfolded RNN

case of weights, it is trivial to realise that as t increases, w^t rapidly decays or grows for very small or large values, respectively. It is also the case that these two problems together affect the networks ability to represent both short and long-term dependencies concurrently in traditional RNNs.

Generation

As RNNs often are generative systems, i.e. that they seek to reproduce something of similar form as the input instead of just discriminating (e.g. classifying) it, there are there are some additional things to consider during design and use. A typical method of generating new values, is to train it as usual and then seed it with the beginning of some sequence that we wish to predict in evaluation mode. For every subsequent step, we insert its own prediction as the next input in a continuation of this seed. As an example, if we wish to predict time-steps 6 and above of a specific sequence, we use time-steps 1 through 5 as initial seed, and then give it its own predicted time-step 6 as the next input and so on.

To speed up training, **teacher forcing** can be used. This means that instead of using the models own prediction from time-step t as input at time-step $t+1$, we feed it inputs from the real sequence. The strategy can be viewed as a way of forcing the network quickly back in line when it is getting off track, as further predictions based upon faulty initial values are less valuable than ensuring that the model trains on actual relations at every time step. However, if the goal is to generate long sequences, the system might run the risk of getting "stuck" if it has never before trained by using its own predictions. One way to deal with this is to gradually utilise teacher forcing less and less during training as its effects are most prominent in the early stages when predictions are very poor. Such a scaling will let the network learn quickly in the beginning while also practising how to recuperate from a bad prediction later on. Depending on the task at hand, it is worth experimenting with such strategies to see what produces the best results.

4.1.4 The Long Short-Term Memory unit

In 1997, Sepp Hochreiter and Jürgen Schmidhuber presented the Long Short-Term Memory (LSTM) unit which avoids the vanishing and nearly eliminates the exploding gradient problems [42]. As the name suggests, the unit is capable of forming both short and long-term dependencies and has become a staple in RNNs tasked with time series predictions or sequence-to-sequence translations. A very good (and pedagogic) description of the inner workings of the unit can be found on this website [44] written by Chris Olah who is currently a research scientist at Google Brain. As stated by Goodfellow et al., other equivalent units have been designed following the invention of the LSTM, perhaps most notably the Gated Recurrent Unit (GRU). As research has found that neither architecture clearly outperforms the other over a wide range of tasks, it is an arbitrary task to chose between them. In practice, they both function similarly inside a network with only minor differences in how they are used.

4.2 Neural Network frameworks

There exists frameworks that allow for the implementation of neural networks in many different programming languages, but a cursory exploration online suggests that Python is the most widely used and supported for AI programming. Some of the most popular choices are listed in this article [45] written by Jeff Hale at *towardsdatascience.com*. As stated in the article, the power ranking is a compilation of the implied usage, interest and popularity gathered from 11 different data sources. The top three contenders are briefly presented below.

4.2.1 TensorFlow

Tensorflow [46] is supported by Google and is an "open source software library for high performance numerical computation". Initially released in november 2015, it is used by many big actors in the industry as the framework of choice, including IBM, Nvidia, AMD, Intel and Google. The framework is highly customizable and allows for deployment across many different platforms and architectures for computation. Google have even developed an application-specific integrated circuit called the Tensor Processing Unit (TPU) with synergy with Tensorflow in mind. Although it has a reputation for being relatively difficult to learn, Tensorflow was for a long time considered to be the state-of-the-art framework for implementing deep learning networks demonstrated by the fact that it is a common choice among the various experiments already analysed in the previous chapter. In recent years, however, the gap has been somewhat bridged by the development or improvement of other libraries, and novices are frequently encouraged to try one of the other, more simple options.

4.2.2 Keras

Whereas Tensorflow is a complex library that caters to the low-level optimisation of neural networks, Keras [47] is intended to be a high-level networks API "designed for human beings, not machines". It is currently only second to Tensorflow in Hale's power ranking, which he attributes to its ease of use. It is worth noting that Keras is not a stand-alone framework, but is an API that runs on top of other frameworks. Although Keras (march 2015) preceeds Tensorflow by roughly half a year, the former has since been adapted to use the latter as its back-end. Keras' official website presents itself as the best candidate "for easy and fast prototyping (through user friendliness, modularity, and extensibility)" of convolutional and recurrent networks or a combination of the two.

4.2.3 PyTorch

Third on the list is PyTorch [48], which is a stand-alone framework and is a relatively new addition to the field. It first saw daylight in October 2016, and is primarily a result of the work done by the AI research group at Facebook. Being placed somewhat between its two contenders in complexity, PyTorch is quickly becoming a strong contender to especially Tensorflow as a framework with more advanced capabilities. According to Hale, PyTorch even allows some customisation that TensorFlow does not, and gains a lot of traction by having a strong synergy with the NumPy library which is a popular favourite for scientific programming in Python. Due to its adaptability and ease of use, along with the fact that it is newer, it is not necessarily so far behind TensorFlow in practice as Hale's ranking might suggest.

Note: This concludes the excerpt from the research project, and the rest from here on out is original work for the thesis.

4.3 Languages

4.3.1 Deep Learning

As the previous section implied, all choices are valid for implementing neural networks, and it is difficult to say which is objectively the best. Although Tensorflow might appear as the most decisively powerful out of the three, it also comes with added complexity which might be problematic in a schedule already starved for time. Keras is on the other end of the scale, with extraordinary simplicity which would streamline the implementation process. However, it has been decided that the thesis will attempt to strike a middle ground by the use of PyTorch as the framework of choice. This also implies that python [49] will be used to implement the module. While both the implementation of a neural network and a specialised optimiser will be a completely new experience, the availability of good framework makes it a safe choice for at least one of the parts. Previous experience in working with python is also decisive for making this choice.

4.3.2 Optimiser

Although python is an intuitive and simple language to use, it has certain shortcomings with regards to performance when compared to many other languages. For a wide variety of applications this does not matter much, as its simplicity of syntax and dynamic typing significantly increases ease of use and streamlines the development process without having to worry about the added complexity of lower level, but faster, languages such as C. After all, if all that is required is to perform some simple calculations, the time spent on juggling memory allocation or declaring variables and types might dwarf any possible gains when it comes to execution time. In such cases, "interpreted" languages such as Python have an edge, as it requires no discrete compilation stage and just translates the written code into *bytecode* on the fly. Although it does what it can to optimise the execution within the narrow scope it is given, this does not compare well to "compiled" languages for more complex or extensive calculation. To paint a picture, NASA modeling guru [50] supplies some benchmarks on the efficiency of python versus many other popular languages for a wide variety of tasks. It is clear that in general, python performs from a little bit, to much, worse for a variety of tasks.

The cases of optimisation and high performance computing (**HPC**) are such situations, and the difference can be huge between languages that are "interpreted" and those that are more rigorously compiled. This term must be used with some care, as strictly speaking all higher level programming languages are compiled in some way, but in this case we are referring to languages who utilise compilers that consider the entire program for optimisation instead of just parts of it at a time. By identifying the most commonly used or demanding portions of the program, it may very well end up restructuring it entirely behind the scenes to accommodate as fast an execution as possible of the sections that are most responsible for increased total run-time.

Compilers come in many forms, but can generally be either classified as just-in-time (**JIT**) (those used to implement Java, C, etc.) or ahead-of-time (**AOT**) (used for C, C++, ...). AOT compiled languages are usually considered the most complex and least dynamic of all languages. In the case of C, it forces the developer to consider things that would elsewhere be handled such as typing, memory management, variable declarations and pointers. While ultimately, if implemented perfectly, these languages usually perform the best for HPC as they leave little room for error or inaccuracies in either compiling or execution, JIT compiled languages sacrifice some of this rigorousness for the luxury of not having to go through a separate compilation stage or making the developer maintain complete control over aspects that might be unnecessary to consider for a given task. Among these, Java [51] is perhaps the most popular and is a "general purpose, concurrent, strongly typed, class-based object-oriented language". The reasons for this are numerous, but among these is its platform independence, long life-time in the industry and being specifically tailored for object oriented programming (**OOP**) which is a very common programming paradigm to use. However, it can be said that Java is a very *verbose* language, which means that creating a program in Java often requires a lot of written code to make it functional. This, combined with the fact that it is *not as fast* as C/C++ and *not as simple* as python means that one of these is usually rather used by the scientific community depending on the requirements of the task.

Regardless of the language chosen to use for the optimiser, it should be judged with regards to the requirements of the thesis; efficiency for the optimisation algorithm, access to frameworks and libraries that may streamline its implementation and relative ease of use as it needs to be implemented within a short span of time.

Julia

Interestingly, a "new kid on the block" called Julia[52] has been under steady development since its inception in 2009, and has since then experienced considerable growth in popularity for scientific programming. Julia is a general-purpose, dynamic programming language which is specifically tailored for computational science such as numerical analysis. It is also highly versatile, including use of, or support for:

- *parametric polymorphism*, which means that it maintains type safety, but programs can be written as if both strictly or dynamically typed.
- *multiple dispatch*, which means that functions and methods may have shared names but dif-

ferent functionality depending on parameter and argument specifications.
- *parallel* and *distributed* computing if a task is better completed across multiple threads or processes.
- uses a JIT type compilation method, allowing for immediate execution without need for an independent compilation stage.
- memory *garbage collection*, meaning that it is unnecessary to manually free objects or variables from memory after use.
- highly *modular*, with significant support for many scientific and optimisation libraries or frameworks.
- a centralised *package manager*, similar to python's *pip*, which makes the fetching and installation of new libraries simple and fast.
- seamless support for directly calling C and Fortran code without intermediate libraries. In addition, libraries exist for calling python code.
- full integration with *jupyter notebook* [53], which provides a singular platform to work on the neural network code (python) and the optimiser in Julia in the same workspace.

While potentially not as powerful as C, it appears significantly more "pythonic", which is a bonus as it will be easier to switch between working on the two modules. Coincidentally, Julia is one of the languages compared in modeling guru's benchmarking where it performs generally well across a multitude of tasks. While using Julia would be a completely new experience and imposes some additional challenge to the implementation process, there can be little doubt that the language is well suited for this specific application.

5 Music Theory

5.1 Introduction

This section contains an introduction to various fundamental elements (*rudiments*) of western music theory (**WMT**). It is intended for the reader who is not already well versed in the ins and outs of basic music theory as it portrays the problem space for the work conducted in this thesis. It is also written with the purpose of presenting the nature of a specific traditional musical style, called the *Bach chorale*, which is the style that the generative system seeks to imitate. Many of the challenges encountered in implementing such a system are directly derived from its musical requirements, as the thesis explores how these rules can be represented and enforced in a digital environment.

Instead of continuous citing of sources for every statement about the basics of music, the first sections of this chapter cites only one book [54] as source material, which is available for free online for all Norwegian IP-addresses through the National Library of Norway. For any other nationality or language preference, equivalent source material is abundantly available online or in other books on the same subject.

The exact definition of music theory is a widely discussed topic, and can range from more restrictive definitions such as "how a specific style of music is made" to those that are more inclusive, e.g. "the study of all organised sound". For our purpose, however, we are only considering the aspects of music theory that concern themselves with the language of western music, which enables notation of music in such a way that it can be shared through other mediums than sound and be recreated without significant loss of accuracy from the original. We also utilise various established conventions of WMT, which revolve around how this notation is structured within various musical styles and genres such that it sounds "aesthetically pleasing" to a listener with certain expectations. Although these expectations are as varied as there are people, there are certain elements of melody, harmony and rhythm that have been (and still are) recurrent in most of the popular western music documented since the renaissance era.

In the first section of the chapter, the most significant composite elements of musical notation are described, including their meaning and some visual representation for future reference. While this musical notation is not directly applicable to digital systems, it is important to be aware of the fact that much of the data from which we can draw musical knowledge is stored in the form of sheet music. A significant part of the implementation of music generation software that seeks to utilise this data is to be able to replicate its behaviour and purpose, if not its exact format. This especially applies if the software is to be a tool for actual composers to use in their work, as this is the language of the craft itself.

Next, there is an introduction to harmony theory; the theory of combining distinct notes to create movements of music in a traditional western style. This section does not utilise much of the musical notation defined previously as it can be explained directly in relation to the note classes themselves, and how they sound together. Thus, extensive knowledge of the notation is not required to understand how it works, but it is helpful if one wishes to visualise it.

The "rules" (or guidelines) of harmony theory are therefore highly conventional, and are a direct culmination of both the musical and cultural evolution in the western world for the last 600 years. In simple terms, harmony theory tells us what combinations of notes that have historically been popular, and which ones have not. Political and cultural shifts throughout history has also affected the prominence of new or "controversial" additions to the ever expanding book of harmonic theory, as a large part of all documented western music has either been heavily influenced and regulated by religious institutions or the cultural elite of their respective times. For this reason, the domain of folk music is often viewed as disjoint from this theory, and can vary greatly in nature dependent on both geographical location and time.

In the final section, the various concepts presented from harmony theory will be used to define the "rule-set" of the *chorale*, a rather strict (traditional) musical style which was made especially popular during the baroque period of the 18th century by being frequently featured in the works of some of the era's greatest composers, most notably *Johann Sebastian Bach*. Due to rigid and well-defined requirements, it is often used as the style of choice for teaching students in music how to apply traditional harmony theory in practice [7]. This thesis argues that it is also well-suited as the target style of music to generate by computer, as the validity of a chorale depends upon its adherence to these rules, which are quantifiable.

5.2 Rudiments

5.2.1 Tones and frequencies

In music theory, a tone describes a sound of a certain pitch and its duration. The pitch of a tone is derived from the frequency of the (clean and stable) sound wave it describes, and allows us to arrange various frequencies into *scales* which classify specific frequencies and their relations to each other. Some of the frequencies and corresponding tone names that occur in the *diatonic scale* of the *tonal system* are shown in table 1 below.

As an aside, an attentive reader might notice that while one of the note classes follows a pattern of magnitude, namely that the frequency of the tone labelled as A4 is exactly twice the number of A3, the frequencies for other pairs of tones labelled the same are not (but some are closer than others). This is because the pitches used in western music today are of *equal temperament*, meaning that the perceived distance between two neighbouring tones (a *semi-tone*) is the same, regardless of point of origin. This is done at the cost of some accuracy with regards to the *harmonic series* and physical resonance, but also provides valuable versatility for musical composition and performance for reasons that are outside the scope of this thesis. If interested, the reader is advised to self-educate on the differences between equal temperament and other tuning systems, such as *pythagorean* tuning or *just intonation*.

As the table shows, we use a circular *twelve tone system* to describe an *octave* (the distance

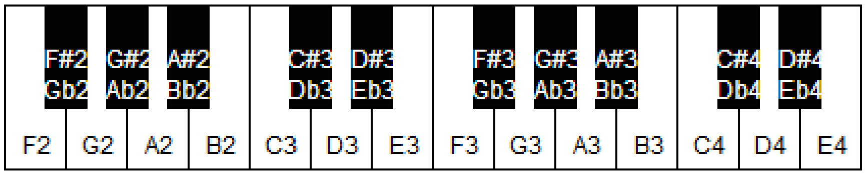
Name	Frequency	Name	Frequency
C3	261.63	C4	523.25
C#3	277.18	C#4	544.37
D3	293.66	D4	587.33
D#3	311.13	D#4	622.25
E3	329.63	E4	659.25
F3	349.23	F4	698.46
F#3	369.99	F#4	739.99
G3	392.00	G4	783.99
G#3	415.30	G#4	830.61
A3	440.00	A4	880.00
A#3	466.13	A#4	923.33
B3	493.88	B4	987.77

Table 1: Note and frequency chart for two octaves

between two neighbouring tones of similar alphabetical class). For all intents and purposes, we consider any octave, e.g. C0-C1, F2-F3 or A3-A4, to be similarly resonant regardless of the fact that the chart show that only A's are exactly so. For our ears, the difference is small enough to not significantly affect this perception. The labels themselves can be confusing at first due their semi-alphabetical nature with the symbols # (pronounced sharp) or b (pronounced flat) appearing now and again. However, this convention is helpful when considering musical scales which we will delve further into later.

To help visualise the distribution of these tones, Figure 5 depicts a section of the keys of a piano along with their respective tone names. Note that all black keys can be referred to either by their sharp or flat name (effectively meaning "one up from" or "one down from", respectively).

Figure 5: Tone distribution on a section of the piano



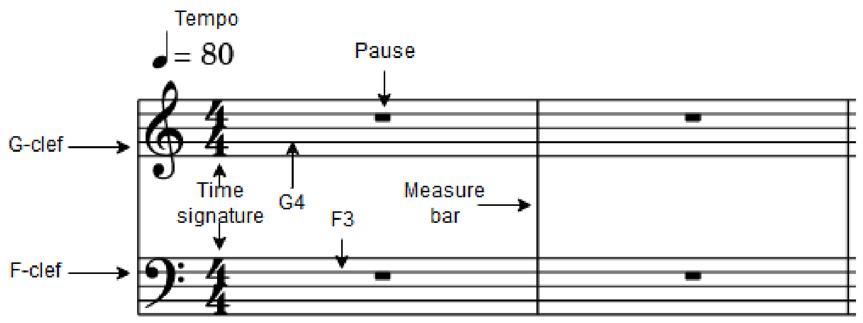
5.2.2 Notes, staves, clefs and measures

Notes are, unironically, notated tones. The pitch of a note is defined by its placement in in a *staff*, a vertical grid. A staff usually consists of five horizontal lines, but can be expanded to include as many as needed. However, if a broader range of pitches are used, staves are stacked vertically instead to make the *score* (the complete notation of a musical piece) simpler to read. Each line of the staff, including the space between two lines, represents the domain of a specific alphabetical note class. However, it does not include all the sharps and flats here, only the raw alphabetical values, i.e. C, D, E, F, G, A, B. Sharps and flats are shown by adding its respective symbol to the line in question. This means that a line for E is actually shared by all of E, E# and Eb, dependent upon a symbol being used or not.

At the start of the staff is the *clef*, which explicitly states what tones the lines on this staff represents. There are many different clefs, and their usage varies depending on the tone distribution for a given musical piece and the instrument that is performing it. Two commonly used clefs are the G and F-clefs, which give the positions of G4 and F3, explicitly, and all others by relation. While persons who are unfamiliar with the system will need to count their way to other tones in the staff, the tone distribution eventually becomes instinctively apparent to those that practice using the notation.

After the clef is the *time signature*, which defines the rhythm, or "beat structure", of the piece. Once in a while, a staff has a vertical line going through it, which depicts the end of the current *measure*. The length of the measure is tied to the time signature, e.g. a 4/4 time signature contains 4 beats per measure. Finally, there is usually some indication of the tempo of the piece, telling how many beats there are per minute (**BPM**). This is often explicitly stated in newer musical works, but not in old, classical work. Instead, composers would use latin keywords to indicate the tempo of the piece. One such example is *Largo*, which means "broadly" and is roughly translated to 40-60 BPM. Figure 6 shows a typical score layout for a piano piece in 4/4, with no sharps or flats. The score is read in-time, left-to-right, and a seasoned musician is able to continually decipher and play the notes correctly as they occur, unless it is of significant technical difficulty.

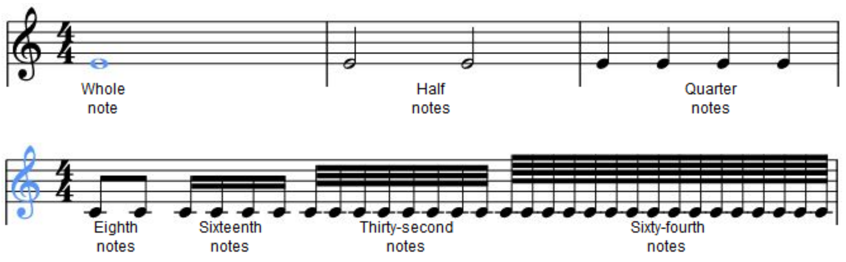
Figure 6: An empty score layout with labels.



As there is no music with only an empty staff, we can now populate the measures with actual notes. But as each musical note in the piece needs to have both a duration and a pitch, there are different shapes we can give them to describe how long they last. The previous figure, for instance, shows a *whole note pause* in each of the bars of measure, simply saying that there is nothing to play in them for the entire duration of the measure. Likewise, there are different shapes for pauses that only last for half, a quarter, or even further subdivisions of the measure. This also applies to the notes themselves, which have different shapes for *whole notes*, *half notes*, *quarter notes*, *eighth-notes*, *sixteenth notes*, *thirty-second notes* and *sixty-fourth notes*. There is in theory no limit to how far you can subdivide the notes, but it is uncommon to use anything below thirty-two, or even sixteenth notes. Also, as we can see in Figure 6, the tempo is actually described by a note itself (quarter note = 80), which means that one quarter note has a duration of 1/80 minutes, i.e. 0.75 seconds. The time signature, 4/4 is also tied to the quarter note, as it tells us there is 4 * quarter (1/4) notes per measure. By relation, this makes a whole note last for 4*0.75=3 seconds, a half note last for 2*0.75=1.5 seconds, and an eighth (1/8) note last for 0.75/2=0.375 seconds.

Whichever way we chose to combine these note-durations, their sum all needs to be equal to the total length of the bar, which is 3 seconds. Intuitively, this can equally be achieved by one whole note, 4 quarter notes, 2 half notes, 8 eighth notes or a mixture between them and the other available choices. Figure 7 shows how the various note-types add up in relation to a beat. Note that the pitch of all the notes of the first row of measures is E4, as they are all two steps down from G4 which is defined by the G-clef. Similarly, we can deduce that the note in question for the second row is C4. In the first row, the position of pitch E4 is highlighted, and in the second row the G-clef is highlighted for the position of G4 (second line from the bottom, in the middle of the clef).

Figure 7: Different notes and their respective lengths in relation to the given time-signature



There are also many additional symbols that can be used to mutate a given note, for instance *punctuation*, a dot after the note indicating that its duration is extended by 1/2 of its original length, or various accentuations that describe how the note is to be played. There are also symbols that mutate sets of notes together, e.g. *triols* which enables the writer to subdivide a rhythmical phrase into thirds instead of fourths. In short, these mutators are highly relevant for advanced musical notation, but too numerous to all be thoroughly examined by this thesis. Fortunately, this information can easily be attained elsewhere if needed.

5.2.3 Harmony and Melody

Having briefly explored the realm of distinct musical notes, we now define some terminology that describes the relations between them, namely *intervals*. A musical interval in an equal temperament system is the distance between two notes. Intervals can be used to both describe melodic (i.e. sequential notes) and harmonic (occurring at the same time) relationships, as the difference in pitch is the same regardless of temporal placement.

Intervals

Perhaps the most important motivation for thinking in intervals instead of absolute notes is the fact that it is completely translatable to any *key*, or chosen scale with a corresponding *root* note. As we have already seen, there are twelve notes to chose from, and each of these notes can have a completely different musical function depending on how they are combined. To visualise this, Figure 5 depicting the piano is again referenced. If we imagine playing only the white keys (7 within an octave) sequentially, we can create more than seven scales with different musical properties depending on which note we choose as the root. If we play C3 up to C4, we are playing the C-major scale, but if we play A3->A4, we are playing the A-minor scale, which creates completely different associations for the listener. Similarly, all of the other five notes can act as the roots of unique *modal* scales, creating seven scales in total using only white keys. Additionally, there is no rule that states that a scale must have exactly seven notes; there exist five-note scales which are hugely popular in modern rock, pop and blues (e.g. the *pentatonic* scale), and some which feature more than seven by using relative sharps or flats. Naturally, it quickly becomes apparent that a system which disjoints the absolute pitch of a note from its function is required to describe all of them in simple terms.

As intervals are always relative, we begin by defining the basic components; the distance from any chosen root note to any other note in the spectrum. For this to be possible, one of the notes must be chosen as the point of reference, which by convention usually is whichever of the notes that is lower in pitch. Negative intervals are therefore seldom used, but are nonetheless fully capable of describing the same things if we decide to use them. The circular nature of the twelve-tone system also means that any positive interval has a mirrored negative if one were to use the higher-pitched note as point of origin instead. The naming scheme for musical intervals in WMT uses two properties; the quality and relative position. Using C3 as root, the simplest interval to understand is the Perfect Unison (**P1**), which is simply the distance from the root to itself. Next we have the Minor Second (**m2**), which is the distance between the root and the next semi-tone (e.g. C3-C#3). This is followed by the Major Second (**M2**, C3-D3), Minor Third (**m3**, C3-D#3) and Major Third (**M3**, C3-E3). Make special note of difference in capitalisation for minor/Major intervals, as they will be referred to in short from now on. Unfortunately (yet descriptively) this pattern does not continue ad infinitum, as there are other qualities to be considered as well. Instead of writing them out here, table 2 is referenced, which contains all intervals within an octave of the root.

A large part of harmony theory is based upon how we perceive *dissonance* and *consonance*. While these terms describe physical phenomena that can objectively be studied within the field of acoustics, its usage in WMT is more dynamic and dependent upon conventional factors. Here, dissonance can be used to describe how pleasant something sounds, how well it corresponds to established historical conventions or cultural expectations for the music created in a certain period. In the context of intervals, less dissonant intervals are viewed as "safer", or easier to combine in a manner that sounds pleasant. This does not mean that dissonance is to be avoided at all costs, but rather that it should be used with a specific purpose in mind. P1, P5 and P8 are considered perfectly consonant, while P4 can be either viewed as consonant or dissonant depending on historical context

Name	Short	Distance in semi-tones	Relative to C3
Perfect Unison (root)	**P1 (r)**	0	C3
Minor Second	**m2**	1	C#3/Db3
Major Second	**M2**	2	D3
Minor Third	**m3**	3	D#3/Eb3
Major Third	**M3**	4	E3
Perfect Fourth	**P4**	5	F3
Tritone	**dim5/aug4**	6	F#3/Gb3
Perfect Fifth	**P5**	7	G3
Minor Sixth	**m6**	8	G#3/Ab3
Major Sixth	**M6**	9	A
Minor Seventh	**m7**	10	A#3/Bb3
Major Seventh	**M7**	11	B3
Perfect Octave	**P8**	12	C4

Table 2: All intervals within an octave to root

and its usage. Next are thirds and sixths, which are imperfect consonances in WMT. Seconds and sevenths are dissonant, along with the interval we choose to call the tritone. When describing a scale, the tritone will in practice either be called an augmented fourth or diminished fifth as it will act as a substitute for either P4 or P5. Intervals that extend beyond the octave range also have unique names, such as the ninth, tenth, eleventh, etc., but these will have the same broad musical function as the second, third, fourth, ..., respectively, only separated by one or more octaves.

The Major and Minor scales

With the knowledge of the basic intervals at hand we can now construct universal scales without the usage of specific notes. Table 3 shows the intervals that construct the major and minor scale, along with a eight-note mutation of the latter which is its frequent substitute both historically and contemporarily.

Scale step	Major scale	Minor scale	Harmonic Minor
1	P1 (root)	P1 (root)	P1 (root)
2	M2	M2	M2
3	M3	m3	m3
4	P4	P4	P4
5	P5	P5	P5
6	M6	m6	m6
7	M7	m7	m7/M7
1	P8	P8	P8

Table 3: Three common scales: Major, Minor and Harmonic Minor

As we can see, the major scale is aptly named after the fact that it only utilises the major quality whenever applicable, while the minor scale is full of minor intervals with the exception of the M2.

The reason for this, other than it is how the layout of the keys present themselves, is that the m2 interval is extremely dissonant due to the proximity of the two notes. This is especially noticeable when played simultaneously such as portrayed in this clip [55]. However, it is then somewhat ironic that the harmonic minor sometimes actually substitutes the m7 with a M7, effectively producing a m2 between M7 and P8. This is because the M7 and the P1 of a scale almost never is played simultaneously, but rather sequentially. In such situations the M7 is referred to as the *leading note*. Therefore, both in classical music and modern music, harmonic minor is regularly used to enhance specific chords in such a way that the note at M7 is *resolved* (becomes) P8 to complete some climax of the piece. Keep in mind that his only applies to a very select few chords, and m7 is the interval used for this step in the scale for all other situations. The next section describes how to identify such cases.

To reiterate, we can now revisit the piano keyboard and confirm that both the major and minor scale are playable using only the white keys (no sharps or flats). By cross-checking with the examples presented in the interval chart, we see that playing all white keys from C to C produces the major scale. Additionally, playing white keys from A to A produces the minor scale, even though the same set of notes is used. This is the main reasoning behind the naming scheme used, as the sharps or flats are only required if playing these scales in different *keys*, or with different root notes. And even then, one would only need to identify the new selection of 7 notes through their relative intervals depending on the key/scale chosen. The end result is a mixture of white and black keys which behave similarly to the all-white C major or A-minor scales, only with a different tonal centre.

This also reveals the purpose behind having doubly-named black keys; to make sure that each line and space in the staff of the musical notation only can contain notes of a certain alphabetical letter, it varies which white piano-key neighbour we need to substitute for a black key. F major, for instance, only needs one black key in addition to the white keys, which is the one labelled Bb or A#. By counting the intervals we notice that we already have an A in the scale, which is the M3, but no B which would be the tritone. We therefore chose Bb as the name for our black key, such that the staff filled with notes from the scale would be evenly distributed instead of one line containing both A and A# with nothing in the line reserved for B. Similarly, the G-major scale contains one black key, F#/Gb, along with all the whites (except F) and it therefore makes much more sense to label it F# than having both a G and a Gb cluttering the system with an empty space for F. This convention is so rigid that in some strange situations, one would even be inclined to refer to F as E#, E as Fb or even D as C##. The notes sound the same, but their functions are completely different within the scale and the notation.

By utilising the full range of the twelve tone system, the music created can be more varied, as well as be tailored to instruments with a more limited range such as human voices. It is also interesting to note that similar *transposition* can be done to the other five modal scales that sound more exotic and strange, as well as every other possible combination of notes imaginable.

Chords

Having identified some common scales, the next step of basic harmony theory is to identify the *chords* that populate the *harmonic field* of a scale. In a general sense, a chord is a combination of three or more notes played at the same time. The simplest example of a chord would be the combination of any three notes, a *triad*, which are played to oscillate simultaneously. However, anyone who has ever tried to press random keys on a piano will have experienced that there are many such combinations that are not aesthetically pleasing to listen to. On the contrary, much consideration must be placed into the selection of notes as different chords produce entirely different associations to the listener, both in relation to the selected notes themselves, and the notes that come before or after. *Harmonic progression* is the term for a sequence of chords, and the possible permutations for such a progression is numerous.

The harmonic field for a scale consists only of combinations of notes that exist within it. The most typical chord for both common scales is the 1-3-5 chord, where each number indicates the step chosen in relation to a root note. Explicitly, it consists of any root note that can be selected from a scale, a relative third (minor or major, depending on the scale) and a fifth. In most cases, this is the P5 and the quality of the third establishes the *tonality* of the chord. For most people, a m3 sounds "sad" or "melancholic", while a M3 sounds "happy" or "joyful". For brevity, we call these triads the minor and major chords with respect to the quality present. These chords are notated by the name of the root note alone if major (C = C major chord), or with an "m" added if minor (Cm = C minor chord). There is a significant pattern existent in these chords, which is that all three notes are related by ascending thirds: a major chord has a M3 from 1-3 and a m3 from 3-5, while the minor chord is the opposite. As mentioned earlier, thirds are considered less dissonant than many other intervals and many typical chords that contain more than three notes follow this pattern (1-3-5-7 is called a 7-chord).

If we stick with our chord template pattern, 1-3-5, and transpose it to all other positions in our scale we quickly achieve a sizeable domain of chords to work with; 2-4-6, 3-5-7, 4-6-8(1), 5-7-9(2), etc. These chords constitute a large part of the scale's harmonic field, and these seven combinations alone can interestingly be used to reproduce recognisable (if somewhat simplified) versions of a large portion of most popular music created since the birth of blues. If we consider 7-chords in our reproductions as well, there is probably a significant chance of plagiarism. So fundamental is this relatively small collection of chords from the harmonic field to western popular music, that commercially successful music is still being made today using nothing more.

Keeping in line with the universal application of intervals, we can refer to these basic triads by their root note position relative to the tonal centre of the scale in roman numerals. Table 4 shows the basic triads belonging to the harmonic fields of the major and minor scales, along with examples of the specific notes that comprise the basic triads of C-major and A-minor. Although a full harmonic field for each scale would contain many additional chords that are not listed here, most of them would functionally only be small mutations of one that is.

Immediately it is made obvious that the harmonic fields of both scales consist of the same set

Step/Numeral	Relative steps	Notes: C major	Chord name	Notes: A minor	Chord name
1/I	1-3-5	C-E-G	C	A-C-E	Am
2/II	2-4-6	D-F-A	Dm	B-D-F	Bdim
3/III	3-5-7	E-G-B	Em	C-E-G	C
4/IV	4-6-1	F-A-C	F	D-F-A	Dm
5/V	5-7-2	G-B-D	G	E-G-B	Em
6/VI	6-1-3	A-C-E	Am	F-A-C	F
7/VII	7-2-4	B-D-F	Bdim	G-B-D	G

Table 4: Every triad in the harmonic field of major and minor scale, with corresponding numeral, note positions and examples in C and A minor

of chords, which is natural since all their notes are common. Yet, their actual usage sounds very different, due to the selection of tonal centre, C or A. Note that there is one step (VII/II) which contains the diminished chord, a minor chord where the relative P5 is substituted by the tritone as mentioned earlier. Among all the chords bar I, the V chord is perhaps the most important as it is considered a suspenseful chord. This is due to it (potentially) containing the leading note (M7) as its third and is the case where the harmonic minor scale would substitute its m7 with a M7, effectively creating a major chord to allow for leading note resolution. Additionally, the three triads that reside on I, IV and V are all of the same tonality as the scale and together represent the full spectrum of available scale notes. For this reason, the harmonic progression of I-IV-V-I is perhaps the most common and representative motif within western music, regardless of scale tonality.

The most popular music is often that which has recognisable elements, due to being in line with the cultural expectations of a listener, and music kept within the boundaries of the harmonic field presented here is likely to be so. However, it is equally important to understand this only describes the most basic and typical motifs of harmony in western music. Different musical styles and genres will exhibit various degrees of adherence to such boundaries, and the most extreme and experimental of them might place significant effort in avoiding them completely.

Melodies, and their roles in harmonic space

At this point we have only considered the potential contents of any singular temporal point in a piece of music, and pinpointed ways to identify which chords that are suitable. We have also introduced the concept of harmonic progression, which is integral to selecting an order of such chords to use. Yet, it is equally important to consider the various melodic aspects of the piece, both in regards to how it evolves over time and how complex or difficult it would be to perform. Even though only three notes are required to form a chord, most compositions have a wider selection of notes to distribute among its instruments at any point in time. Typically, but dependent on the *orchestration* of the piece, i.e. which instruments are performing it, it is usually important to give equal attention to the *melodic progression* of every individual note as to their harmony. While a piano is capable of playing as many notes at a time as there are fingers available, many instruments are limited to a few, or only one, at once. Many such instruments, and especially those that rely on breath, are also limited in the range of notes they are able to produce, including human voices. In short, an erratic melodic progression will both affect the difficulty of performing a piece as well as a listeners ability to process it, due to lack of continuity and flow. Melodies are often the first things a listener will notice and familiarise themselves with, and the harmony acts as *accompaniment* that seeks to provide emphasis to it. This does not however mean that there is necessarily only one melody in a musical piece, as all notes that are used simultaneously can be perceived as part of some melodic sequence. Different styles of music place varying emphasis on the independence and prominence of simultaneous melodies and a musical piece can be said to be either *homophonic* if being focused on a single melody or *polyphonic* if focused on two or more at a time.

In great classical music, the various melodies that evolve and together form harmonies over time are distributed in such a way that they emphasise and create room for each other instead of crowding anothers' segment of the harmonic space. This is also a concept that is frequently used in modern music as well. As an example, a large portion of songs that feature on the radio today contains a vocalist that presents the *lead melody*, which is the central part of the piece and easy to notice. Lead melodies that are "catchy" will have some structural form that makes it easy to memorise and recognise for the listener. To achieve this, they often have limited motion over time, or at least be systematic in the way it foreshadows, prepares and/or repeats significant changes in pitch. A leading melody is also oftentimes provided with ample "space" around it, meaning that there are few other instruments that occupy the same *band of frequency* in the audio spectrum. Where most instruments that are part of the accompaniment generally occupy the lower frequencies, lead melodies like vocals or guitar solos are placed both higher in pitch and frequency to be more prominent to the listener. This also applies to traditional western music, where instruments that produce such sounds have more melodic emphasis, and are more thus more free to explore the harmonic space available. On the other end, the *bass* which occupies the lowest frequencies of the spectrum is generally viewed as the second easiest melodic sequence to identify, as it has no "competition" below it. In WMT, composers are therefore generally encouraged to place extra consideration on the structure and inter-play of the lead and bass melodies in their music. Since all notes together are needed to construct the discrete harmonies at every temporal step of the song, it is usually left to the middle frequencies and notes to "fill in the gaps" to achieve the desired chord or harmony. Great composers have the ability to make the melodic sequences of every instrument in the orchestration display a pleasant flow and structure, while simultaneously fulfilling their respective roles in the harmonic space.

This concludes our brief glance into general concepts of WMT, which, although limited in scope, is hopefully sufficient to understand both what the accurate representation of musical information requires, and the effective meaning behind every musical rule and guideline of the chorale style.

5.3 Four-part harmony: the Bach chorale

5.3.1 Form and structure

As the goal of the thesis is to describe methods capable of generating music by computer, it is necessary to establish some criteria from which the quality of resulting compositions are judged. To this end we select the chorale as it was used by Johann Sebastian Bach. More specifically, this is the homophone four-part harmony style for mixed choirs [7] as it is both limited in structural complexity and needs to adhere to many quantitatively verifiable rules to be considered *valid*. Being popularised during the baroque era, these chorales are relatively short, tonal compositions which in their simplest form consist of only four human voices. The highest pitched voice is the Soprano (**S**), which is responsible for presenting the melody of the piece. Below it is the Alto (**A**), which is the highest pitched of the three accompanying voices that together seek to support the melody with harmony. Third is the Tenor (**T**), whose role is the same as the alto, while the lowest voice is the Bass (**B**). While the bass is equally responsible for constructing an harmonic framework, it is also viewed as secondary only to the soprano with regards to melodic importance. Similarly to how the leading melody of the soprano is easy to recognise, an erratic bass melody will stick out like a sore thumb and detract from the homophonicity of a piece. The names of the voices imply the range in pitch at which they operate, and a more advanced chorale composition may utilise additional or alternative voices than the ones listed here. For our purpose, we will only be considering those that utilise only the exact four voices **S**, **A**, **T**, **B**.

Bach's chorales are, fundamentally, considered as religious music, with their single melodies often being based upon *Lutheran hymns* already existent at the time and with complementary religious lyrics. Bach would compose the remaining three voices in such a way that they would produce harmonies to emphasise both the melodic progressions and the lyrical expositions eloquently within the boundaries of tonal expectation. Since Bach was very systematical in constructing these solutions, it is possible to define a rule-set which accurately describes both the approach he used to achieve them and the structure of any completed work. This method and its corresponding rule-set is often still taught today as part of the education of students of music, due to its exemplary and consistent usage of the tonal system.

There are intrinsically a large number of rules that in varying degree must be upheld to ensure the validity of a piece. While some of them are related to each individual voice alone, others are related to their harmonic or rhythmic combination. In the following sections, only a subset of the most central rules are highlighted and explained as they are too numerous to all be included and analysed in the text. An attempt has been made to compile short but understandable descriptions of the rules that are mentioned in the first four chapters of Bekkevold's book, and are fully listed as implemented in a later section (8.4.3) of the thesis. A functional prototype of the software will seek to achieve as many of them as time allows for, with a priority based upon level of strictness.

5.3.2 General voicing rules

There are many rules in the style which revolve around the individual voices themselves. By the nature of the human voice, different people can have different ranges of pitches which they manage to sing. Since Bach's chorales are intended to be performed by a group of singers that do not necessarily share the same vocal range and there can be many singers per voice-part, they are specifically assigned a vocal range which is not too demanding for the performer. In a mixed choir (with both men and women), the bass and tenor are considered as voice-parts for males, while the alto and soprano are usually performed by women. However, it is entirely possible for a male with a higher pitched vocal range to be singing the alto, or for a female to sing as a tenor. What matters is the ability to reach the required notes for a piece. In Bekkevold, the vocal range that is to be expected from each voice-part is, as follows:

- Soprano: C3-G4
- Alto: F2-C4
- Tenor: C2-G3
- Bass: F1(E1)-C3

Note that the parenthesised E1 may only be used if it is introduced *step-wise* down (as some 2 interval) from F1 or F#1. The total range across all voices is then F1(E1) through G4, which is roughly three octaves.

During performance, each voice-part will at all times be responsible for representing one out of the necessary notes to form a chord, or a note which aids the transition between two chords. This is usually a triad chord or a four-note chord. In the case of a triad, one of the required notes will need to be doubled by two voices, while a four-note chord will contain an unique note per voice. Be aware that in such situations, a "note" means a tonal class such as C, D, E, ..., and not necessarily the same pitch.

The melody (soprano) sets the pace of the rhythmic progression, but other voices may at times display independent (but complementary) rhythm as long as it does not compromise the homophonic feel of the piece. It is also imperative that each voice-part is composed in such a way that it is not overly difficult to sing, meaning that a composer should generally strive to minimise the vertical motion (change in pitch) between each subsequent note. Large and/or dissonant interval jumps (called *leaps*) are significantly more difficult to sing and are therefore to be avoided when possible. To be specific, the book suggests the following boundaries for leaps in voices:

- The middle voices (A, T) can not leap beyond a P5.
- The outer voices (S, B) may occasionaly leap further than a P5, but then preferably only a 6 or a P8.

The vertical structure of the voices is also immutable, meaning that each voice must stay within the hierarchy of $S \geq A \geq T \geq B$ in pitch. Crossing a voice over another is therefore strictly forbidden. The same applies to the event where all the voices move simultaneously in the same direction. This is called an *avalanche* and is not allowed. In the rest of the thesis, all events that break such forbidden rules are referred to as *illegal* events. With regards to rules that are not as strictly phrased (guidelines), catalyst events are called *undesirable*.

5.3.3 Harmonic rules

To fully understand the harmonic rules of Bach's chorales, we need to revisit table 4 from section 5.2.3, which presents the harmonic field of the major and minor scale. The harmony of a chorale revolves around the use of *functions*, which are classifications of the various chords found at each step of the harmonic field. The most basic form of a function is the triad listed at each corresponding step. Every chord function is to be used for a specific purpose, and the system is designed in such a way that as long as these are used correctly, some aesthetic value is more or less guaranteed. That is to say, chords that are used to fulfil their functional purpose are guaranteed to not sound dissonant in the immediate setting.

Functions

The terminology of these functions can more or less be directly related to the roman numerals of the table referenced above. According to the theory, a scale contains three main functions. The function of step I is called the *Tonic*, or *T*. A chorale must both begin and end with T, as this is the tonal centre of the piece and resolves all tonal suspense. Next is the *Dominant*, *D*, that lies at step V of the scale. In the section on chords, we have already presented some of the reasoning behind its importance, as it is considered the main source of suspense in the tonal system. Due to this, it is also convention that the penultimate function of a chorale must be a D, such that it ends with the functions D->T. Additionally, except for very specific situations, a D must always be resolved into a T whenever it occurs in the piece. The last of the main functions is the *Subdominant*, *S*, which begins at step IV and whose chief purpose is to bridge between T and D. S is often used in combination with both T and D, as it demonstrates the tonality of the scale without causing excessive suspense by itself. The only basic rules on the usage of S is that the combination D->S is illegal unless very specific rules apply and that it should not be used as the ending of a phrase. Each of these, and "lesser" functions as well, may have several different *inversions* (i.e. permutations based upon which note is in the bass voice) that affect usage, or may contain inclusion/mutation of additional/existing notes.

To keep things relatively short, table 5 shows a summary of the different functions and their various forms that are presented in the first four chapters of Bekkevold. This includes the most common inversions of each function. Even though there are some additional functions to consider when composing an advanced piece that are not listed here, the thesis limits itself to these as they are more than sufficient for the implementation of a functional prototype. The note composition is relative to the scale root, and does not imply the order at which they must appear except for the bass note.

This produces a sizeable set of chords to work with when creating a chorale, and is sufficient to demonstrate most of the basic rules it requires. Note that some of the functions have an s or a m following their main function classifier. These are *submediants* and *mediants*, and represent all other chords in the previously examined harmonic field that are not of the main functions (i.e. II, III, VI and VII). The s signifies that it is placed two steps down from the main function and an m signifies its placement two steps above the main function. While this table contracts some mediants that share the same location (Ts/Sm, Tm/Ds) they are functionally different when used

Function name	Note composition	Doubled note	Bass note	Example: C-major (B-T-A-S)
T	1, 3, 5	1	1	C-E-G-C
S	4, 6, 1	4	4	F-A-C-F
D	5, 7, 2	5	5	G-B-D-G
T-5	1, 3	1 (Tripled)	1	C-C-E-C
D-5	5, 7	5 (Tripled)	5	G-G-B-G
T/3	1, 3, 5	1 or 5	3	E-C-G-C
S/3	4, 6, 1	1 or 4	6	A-F-C-F
D/3	5, 7, 2	2 or 5	7	B-D-G-D
T/5	1, 3, 5	1 or 5	5	G-C-E-C
S/5	4, 6, 1	1 or 4	1	C-F-A-F
D/5	5, 7, 2	2 or 5	2	D-B-G-D
T64	1, 4, 6	1	1	C-F-A-C
T54	1, 4, 5	1	1	C-F-G-C
D64	5, 1, 3	5	5	G-C-E-G
D54	5, 1, 2	5	5	G-C-D-G
D7	5, 7, 2, 4	None	5	G-B-D-F
D7/3	5, 7, 2, 4	None	7	B-G-D-F
D7/5	5, 7, 2, 4	None	2	D-G-B-F
D7/7	5, 7, 2, 4	None	4	F-G-D-B
D7-1	7, 2, 4	2 (rarely 4)	2	D-B-D-F
D7-5	5, 7, 4	5	5	G-G-B-F
S65	4, 6, 1, 2	None	4	F-A-C-D
S65/6	4, 6, 1, 2	None	4	D-A-C-F
S6	4, 6, 2	4	4	F-A-D-F
Ts/Sm	6, 1, 3	1 or 6	6	A-C-E-A
Tm/Ds	3, 5, 7	3 or 5	3	E-G-B-E
Ss (if major scale)	2, 4, 6	2 or 4	2	D-F-A-D
Dm (if minor scale)	7, 2, 4	7 or 2	7	(ex: C-minor) Bb-D-F-Bb

Table 5: All functions and inversions that are to be considered by this thesis

to notate a composition. The last two functions, Ss and Dm are almost exclusively only used in major and minor respectively, at least for basic compositions, as this is the step in which the tritone would otherwise make its appearance and is more difficult to utilise properly. While a true Bach chorale may frequently make use of these chords, their implementation is left for future work. The inversions are denoted by /x, where x signifies the bass note. This has significant effect on our perception of the chord which make them functionally different from its primary and other secondary inversions.

Nearly every function that is listed has more than one individual rule related to it, and it is left to the appendix to describe them. To summarise, most have to do with how the function is prepared and resolved (which functions can come before and after) and how the individual voices move through the function. Other examples are those related to placement in the rhythmical beat of the piece, rules that state how the notes can or can not be internally distributed between voices and those regarding how frequently a specific function can be used in a phrase.

Finally, it will be mentioned that Bach utilised the harmonic minor when composing chorales in a minor key, such that D->T always consists of a M7 leading note resolution. All in all, there are hundreds of distinct rules or guidelines that must be considered when creating a chorale piece, which in theory should prove useful in constraining a computer program in a sufficient enough manner to ensure a valid product. To round off the introduction to four-part harmony, the final section presents and explains the notion of *cadences*.

Cadences

It has already been mentioned that the complete piece is required to begin with T and end with D->T, but there is potentially a lot of room in between these two points to fill with functions. In most music, a melodic sequence does not last continuously from the beginning all the way until the end. This means that the usage of pauses, or intermediate endings, is a natural inclusion to any composition. In the chorale, a *phrase* usually consists of a sequence which is bounded by the length of a given textual verse. When the sentence is complete, the phrase must necessarily end before a new one can be started. To signify such an intermediate or final ending, we utilise *cadences* which are specific function combinations that adhere to the tonal expectations of suspense and resolution. We have already identified one such cadence, the final D->T, which is called an *authentic cadence*. This can be used for all endings, both intermediate and final. If we were to move through all three main functions, T->S->D->T, we are conducting the complete *tonal cadence*, as these three functions together contain all notes in the scale. This can also be used as the final cadence of a piece.

Some cadences are only legal for intermediate endings, as they are not considered to resolve the tonal suspense sufficiently enough to be the ultimate ending of the chorale. The mirrored authentic cadence, T->D is often used in such a way, or S->D as the suspense supplied by the Dominant lingers across the pause and is eventually resolved later on. S->T, when used as a cadence, is called the *plagal cadence* due to the lack of suspense and is not used as endings in chorales. It is worth noting that this cadence was frequently used in many other musical styles during history. Other cadences include the *phrygian cadence*, S/3->D, chains of *suspended* functions such as D64->D54->D->T and the *disappointing cadence* D->Ts or its mirrored Ts->D. Some cadences may also substitute some of their functions with others that sound similar, as for example D7 can substitute for D as long as it is not the final function.

To summarise, knowing the various cadences and their impact is important when composing a chorale, and is necessary to ensure the validity of the generated music. Each individual piece must make sure to begin with T and end with the authentic cadence, and utilise the various cadential movements intermittently to create a flow of suspense and resolution throughout the music.

6 Optimisation Strategy

6.1 Optimisation objective

For this thesis, the main objective of the optimiser is to ensure the validity of the generated output with regards to the rules and constraints set by the musical style of Bach's chorales. This implies that the optimiser must be capable of not only judging the musical structure, but also adjusting it whenever necessary. A choice must be made early on regarding how these adjustments should be handled, as there is a difference between finding or approximating an *optimal* solution to the problem, or simply identifying one or more valid solutions. Some potential strategies only seek to achieve the latter, for example by formulating it as a *constraint satisfaction problem* (**CSP**), which is a *decision problem*. Solving a decision problem proves or disproves the existence of a solution which adheres to some set of constraints, but there is no innate metric which implies the optimality of the solution. Meanwhile, *optimisation problems* operate with the assumption that there exists at least one such quantifiable metric in the data which, if maximised or minimised, undeniably determines the quality of any given solution. For instance, if we reformulate the CSP as a *constraint satisfaction optimisation problem* [56] (**CSOP**) instead, such judgement is made possible. While there is an inherent difference in these two problem formulations, this thesis may refer to the implemented software module as "the optimiser", regardless of the strategy utilised finding an optimal solution or just any solution.

The intent of the thesis is nonetheless to utilise true optimisation to solve the problem at hand, and there are many potential methods and techniques to consider that can achieve this. Later in this chapter, some of those that are deemed the most promising are presented with a short description and some thoughts on their suitability for our specific purpose.

6.1.1 Initial problem complexity

Before moving on to the various optimisation techniques, we will take a closer look at the ball-park initial complexity of our problem as a motivation for why such a specialised approach is required. As it was shown in chapter 5, table 5 contains the total domain of possible functions that may at any point in time be correctly utilised, amounting to 27 functions per scale. If we separate the musical piece into *t* time-steps, where each time-step needs to contain one valid function from the domain of *n* possible chords, the act of looking at all possible combinations of functions alone for a piece of length *t* is n^t, which is exponential with regards to *t*. In the current training data set used by the program, the average length of each chorale is roughly 60 seconds (at a glance). Upon listening, it also becomes apparent that each function rarely lasts for more than one second, with some changes being much more frequent than that. As such, it is not unrealistic to expect that a generated chorale should be able to contain at least 60 discrete functions. A brute-force exhaustive search would therefore need to look through 27^{60} permutations for functions alone, and this is ignoring all the different voice permutations that are possible per function. Continuing with the napkin math, if we assume that each function has an (optimistic) average amount of 10 possible voicing permutations, the total number of possible permutations is at least 270^{60} which is completely unrealistic to process in such a manner.

It is obvious that the program will need to significantly reduce its search-space, and while much work can be done in explicitly restraining the potential contents of some of the time-steps (since we know the piece must begin with T and end with D->T), or splitting the piece into shorter phrases that are optimised separately, the complexity is much too large to efficiently process for any more than a couple of time-steps at a time. Fortunately, a sound optimisation technique will significantly reduce the effective complexity if utilised correctly, and the next section takes a closer look at some of the more promising ones.

6.2 Optimisation techniques

6.2.1 Integer Programming

IP is a method for defining optimisation problems with discrete or integer variables [57], where variables model indivisible phenomena in reality or represent binary choices such as on/off, buy/sell or include/exclude. Although IPs may be non-linear, the abbreviation usually refers to *linear integer programs* (**ILP**). Many ILPs are *NP-Hard*, meaning that there are instances where the existence of a solution can be verified quickly but no good way of finding a solution exists, but there are also specific situations when ILPs can be solved in polynomial time. When the problem is properly formulated, an IP-solver may either identify or approximate a solution if the problem is solvable in the first place.

IP has seen much improvement over the last decades due to, among other things, improved available processing power, better modelling strategies and new heuristic methods such that it is possible to find solutions that are within 0.1% of the optimum. This accuracy depends on the nature of the problem itself, how it is formulated and which solving methods are applicable.

The formulation of an IP problem consists of some optimisation objective, i.e. maximise or minimise, subject to some conditions or constraints. A simple formulation example may look something like this:

$$Maximise \sum_{j=1}^{n} b_j x_j - c_j x_j$$

subject to:

$$\sum_{j=1}^{n} x_j \leq d$$

$$d \geq 0 \; and \; integer$$

$$x_j \geq 0 \; and \; integer$$

where c_j represents the integer costs of production of some product *j* and b_j its potential income, respectively, and d is the maximum total products that can be made. When executed by an IP-solver, returns the vector x which for which the profit is maximised. This formulation requires that all variables in the formulation are integer.

In cases where this can not be done, e.g. $c_j \; or \; b_j$ are float values, the problem becomes a *mixed integer problem* (**MIP**), and is treated differently. The problems that consist of entirely binary variables are called *binary integer program* **BIP**. Many examples of each type of problem can be found in Wolsey, including some applications to more famous problems like the *Knapsack Problem*, whose decision problem is NP-complete and optimisation problem is NP-Hard. To combat this, IP-solvers may employ a variety of strategies that can approximate a solution within reasonable time. In the case of *relaxation*, which is bounding the IP search space by the solutions of less constrained versions of a problem, an IP can *prune* much of the unnecessary search space and avoid spending time on exhaustively exploring them. Such relaxations are often done on the unconstrained LP, and produces some lower bounds which the IP can not possibly beat. Two common classes of algorithms that are usable in IP-solvers are *cutting plane* and *branch and bound* (**BB**) algorithms. The cutting plane method incrementally adds linear restrictions upon the LP-relaxed problem until it approaches integer solutions, while branch and bound uses the lower bounds from the LP-relaxation in combination with heuristics to determine whether or not to proceed down a given portion of the search tree. BB has a significant advantage over cutting plane as it allows for early-termination and may return multiple feasible (or even optimal) solutions, but relies heavily on the accuracy of the heuristics to be efficient and can otherwise be painfully slow. A combination of the two methods, called *branch and cut* [58], is quite popular to use in IP solvers today as it combines the reliability of BB with the efficiency of cutting planes. As an example, it has been used to great success to both solve and prove the optimality of relatively large instances of *the travelling salesman problem*.

Suitability

It was originally the intent of the thesis to research specifically the optimisation and generation of music by means of Integer Programming, as it was theorised that the discrete nature of musical notes and the large amount of absolute constraints to apply. Especially with regards to Bach's chorales, one could view the optimisation objective to be to minimise the distance that each voice moves at every time-step, while still fulfilling the requirements of function combinations. Unfortunately, after much trial and error, it was discovered that the complexity of the formulation significantly increases whenever conditional or exclusive constraints are needed in the model. Often, two or more discrete functions are equivalent, or equally fitting, for a given situation, and no good method of sufficiently formulating an integer program with probabilistic constraints (*PCIP*) for the task was found. With potentially hundreds of equivalent choices at every time-step, the corresponding integer program was simply too difficult to formulate in due time. The solver would also need to support non-linear models, as the structure of musical elements are hard to express linearly. For instance, one is to express the circular nature of the diatonic scale as an IP formulation, and another is to find meaningful ways for portraying multi-note chord structures as linear integer variables without loss of generality.

In the end, the attempts lead to little success. As a final note on the subject, this thesis does not conclude that it is entirely impossible to formulate the problem as an IP, but after much deliberation it was decided to move on and attempt other avenues of approach to reduce the risk of having no results to show for at the end of the allotted time of the project.

6.2.2 Simulated Annealing

Although the notion of using IP as method for solving the problem was abandoned, research on the matter gave rise to some inspiration for other strategies to attempt. One of them is *Simulated Annealing* (**SA**), which in IPs is used for heuristic estimation. Originally proposed by Kirkpatrick, Gelett and Vecchi in 1983 [59], and inspired by the simulation of the annealing of solids in metallurgy, it is used to approximate the global optimum of a given function by mutating between valid solutions and probabilistically accepting or rejecting "inferior" solutions depending on the current "temperature" of the system. The analogy stems from the act of cooling metals in a controlled matter to promote the formation of larger crystals and discourage structural weakness.

SA requires knowledge of the full state space $\boldsymbol{S}$ and some *neighbour-generator* which maps from any given state s to a neighbouring state $s+$. A neighbour is defined to be some state that lies sufficiently close to s, meaning it should only display minor mutations from the origin. In the case of the travelling salesman problem, which SA has proven useful for, a neighbour can for instance be any state that is identical to the original except for the reversal of one of the directional dependencies.

Having identified a neighbouring state, the algorithm either decides to accept the new solution or stay in its current state. Initially, the global temperature T of the system is high, and will therefore much more easily accept the worse of the two options, since a greedy algorithm which always selects the best option is prone to being trapped in local optimum. Across iterations, the goal is to locate the state with the least internal energy *E(s)* by progressively lowering the global temperature. The *acceptance function P(E(s), E(s+), T)* decides whether to stay or move, with decreasing probability for choosing the worse state as T approaches 0. Close to termination, the algorithm becomes greedy and only picks the immediately best solution at every iteration. The global temperature is decided by the *annealing schedule*, which can for instance be directly proportional to the remaining available computation time. Example pseudocode for the process is found below (more or less identical to wikipedia [60]):

```
Let s = s_0
For k = 0 through k_max (exclusive):
    T = temperature(k_max / (k+1))
    Pick a random neighbour, s_new = neighbour(s)
    if P(E(s), E(s_new), T) >= random(0, 1):
        s = s_new
Output: the final state s
```

In short, SA can be applied successfully to a multitude of problems, and is especially effective when an approximation of the global optimum is sufficient. It does require that the state space is fully mapped and that the neighbour-generator is capable of generating neighbours with similar local internal energy across the entire space. This means that good solutions have relatively good neighbours; that the landscape of the objective function across the state space is not exceedingly jagged and erratic. If there is little to no correlation between the gravity of the mutation and the change in internal energy, it is difficult for the algorithm to move beyond identifying only local optimum. Given the wrong conditions, both problem- and algorithm related, SA runs the risk of not locating an approximation of the global optimum at all.

Suitability

Immediately after IP was discarded as an option, SA was the most promising approach to consider next. First of all, it alleviates many of the difficulties of mathematically modelling the problem by requiring little to no such modelling at all. Many of the pieces required are already in place; the *optimisation goal* remains the minimisation of voice movement across all time-steps, the *annealing schedule* is deduced organically from designated computing time, and *acceptance function* remains as is described in the theory. However, it is not immediately apparent how to define a *neighbour-generator* for this specific problem. For us, a relatively minor mutation, such as the substitution of a single function, can have an immense effect on the quality of the entire solution as there are serial dependencies between it and whatever lies before or after. The mutation function must either be able to generate the entire search space of $num\_voicings^t$, or find some means of only generating all valid candidate solutions in the space. The latter is decisively more difficult and not in any way obvious, while the former would undoubtedly result in a very unsmooth and jagged objective function landscape. Whether or not the algorithm would be able to overcome this significant drawback, it has been shown that it performs best on problems where all candidate solutions are probabilistically equally valid. For instance, in the case of the travelling salesman problem, every candidate solution is trivial to generate and there are few non-linear aspects to the rate of cost change between permutations.

Another important factor of the optimiser is that its intended use is to impose structure upon the output of a neural network. This means that the optimisation criteria must both be based upon the global objective (the minimisation of voice movement) along with some cost of distance from the original. If we theoretically were to identify a complete valid state space and acceptable neighbour-generator, it would still be necessary to judge the produced sequence both on its individual quality and its congruence with the input sequence. This would probably contribute significant noise to the landscape of the internal energy function across states due to an additional metric being imposed which has no immediate correlation to the intrinsic validity of the sequence. Ultimately, an objectively less optimal solution might be "better", simply because it resembles the input sufficiently much more. It is integral to the whole generative system that different input produce different output, as it is not the intent to spend much time and effort in implementing a system which locates the same, optimal yet identical sequence of functions and voicings every time.

There is without doubt many possible applications of using SA for music generation, but it also presents a new set of challenges for our specific problem. While no solutions to these issues have presented themselves yet, their existence are in no way conclusively denied by this thesis.

6.2.3 Branch-and-bound search

The last of the optimisation strategies considered is also inspired by our foray into IP, namely Branch-and-bound as mentioned in section 6.2.1. BB was first proposed by Land and Doig in 1960 [61] as "an automatic method of solving discrete programming problems", for which it is still used today as an integral part of many IP-solvers in some variation. However, its applications have a broader range than only those that are formulated as integer problems as its approach is relatively generic. BB operates on the assumption that the state space of the problem can be expressed as a rooted tree, where all outer leafs constitute the complete set of *candidate solutions*. The depth of the tree is equal to the number of optimisation variables, and the root is said to contain the full set. Beginning its iteration at the root, every node that it can branch to is compared to some *lower* and *upper* bounds of the best solution yet found. For a minimisation problem, the lower bound is an optimistic estimation on what the best case solution may be if it continues down a given path, while the upper bound contains the cost of the best solution yet found. The algorithm then determines if the sum of the cost at the current node and the optimistically estimated remaining cost is greater than the upper bound, which would mean that there is no point in continuing down this path as it can not produce a better solution than one already found. The node, and all its children are pruned from the search space, potentially reducing the size of the search space by a large factor depending on the depth of the discarded node. If the child has potential for improvement upon the best found solution so far, it is added to the *active set*, which holds all the nodes yet to explore. For a maximisation problem, the roles of the lower and upper bounds are reversed, i.e. the lower bound is the best solution found and the upper bound estimates how much a given path at most could add to the current value.

The algorithm performs this routine in a top-down iterative search, starting with only the root node in the active set. As it is the only node in the set, all its children will be processed and added to the active set if they can feasibly produce better results than the initial upper/lower bound for the minimisation/maximisation problem. As one would generally want to start the pruning as high as possible in the tree, there is much to be gained from initialising the algorithm with *tight* bounds, meaning that the cut-off for both bounds is as close to the real optimum as possible. To achieve this, it is not unusual to either construct a feasible solution at random and use its bound as initial value, or have the active-set be a *LIFO*-queue so that a depth-first search quickly identifies the bounds of a valid solution. Otherwise, the initial cut-off bound is generally set as 0 for maximisation problems and sufficiently high to not make the problem unsolvable (e.g. infinity) for minimisation problems.

Below is pseudocode for a Branch-and-bound algorithm for a minimisation problem with no initial upper bound:

```
activeset = {root_node}
upperbound = infinity
currentbest = NULL
while activeset is not empty:
    select a branching node k from activeset
    remove k from activeset
    locate all children of node k, c_i (i=0,...,n_k)
    for each child c_i:
        calculate optimistic bound ob_i
        if ob_i > upperbound:
            kill c_i
        else if c_i is complete solution:
            upperbound = ob_i
            currentbest = c_i
        else:
            add c_i to activeset
return currentbest
```

It is apparent that branch and bound requires sensible upper and lower bounds, as without it would function similarly to an exhaustive search. With tight bounds, the size of the search space, and by result the run-time, may be reduced by a large amount. Although its worst case remains the same as brute-force if no nodes are pruned, it will generally perform much better than this and can be terminated early to at least provide *some* feasible solution.

Suitability

As BB is quite generic in nature and is mostly dependant on the quality of the bounds, it remains to see if good estimator heuristics can be developed. At any rate, the search itself seems quite simple to implement with regards to the chorale optimisation problem. By constructing a rooted tree with depth of size t_{tot} and branching on all function choices, a partial candidate solution equals the string of functions which leads from root to depth $t_i,\ 1 \leq t_i \leq t_{tot}$. If $t_i = t_{tot}$ we have found a full candidate solution. During iteration, every new child can be validated and penalised for illegal or undesirable local events, and any invalid partial solutions are pruned without the algorithm spending any more time on them. In addition, a cost can be assigned for distance moved between parent and child and for the distance from the original input. If the tree is made to search in a depth-first manner, the program would at least produce candidate solutions that are legal, if not optimal, relatively quickly. Still, it is clear that identifying a good estimator is paramount to finding optimal solutions, and it is uncertain whether or not this is possible. At any rate, BB is by far and large the seemingly most suitable option out of the optimisation strategies considered at this point, and is also the most probable to produce results of some quality within the thesis deadline.

6.2.4 SMT-Solvers

In spite of being mainly applicable to decision problems, solving for feasible solutions alone may well be a valid strategy to fall back on if the optimal strategies fail to deliver. Similar to the regular *boolean satisfiability problems* (**SAT**), *satisfiability modulo theories* (**SMT**) problems are presented as a set of *predicates* for which a solution must satisfy in order to be feasible, only also including the possibility of forming predicates over non-binary variables such as those used in regular arithmetic. An SMT problem needs to be supplied with corresponding theory that defines the nature of variable relations and the variable space. Since SMTs are decision problems, a solver will verify and pinpoint the existence of a valid solution or otherwise terminate with no solution found. As exemplified by Moura and Bjørner [62], SMT and corresponding solvers are applicable to a variety of problems, and a formulation of the encoding of the classic *job scheduling problem* is shown in the article. It also lists some of the most state-of-the-art solvers for SMT problems; *Z3*, *Barcelogic*, *CVC*, *MathSAT* and *Yices*, which are continuously being developed and together support many theories which are required to make sense of different formulas. For instance, Z3 [63] lists *empty theory*, *linear arithmetic*, *nonlinear arithmetic*, *bitvectors*, *arrays*, *datatypes*, *quantifiers* and *strings* as theories with built-in support.

Suitability

Due to the versatility of SMT-solvers such as Z3 and the fact that many are open-source and free to use, it is probable that the problem can be expressed as an SMT with corresponding theory. By using MIDI note numberings, it can be said that every note with the same modulo 12 value function similarly in a chord, and each voice can be constrained to select a permutation of notes which fulfils the requirements of the function within their respective ranges. Additionally, time-step dependant constraints are added that maximise the allowed movement from time-step to time-step, such that the solution (if one is located) is at least of some quality. To be sure a relatively good solution is located, the solver can be asked to locate solutions that are especially constrained on the optimisation objective itself. Since the usage we are intending it for is a minimisation problem, it is possible to run SMT-solvers on the problem with an initial constraint of 0 movement and relax this constraint across multiple iterations. In this way, the decision problem becomes an optimisation problem by proving the existence of a solution at one minimal value of the objective function, and disproving the existence of any solutions that are better as no solution was found at that level of constraint. Such a method may indeed yield as good results as a true optimisation algorithm, but also makes it difficult to state anything about its run-time as potentially could continue forever if the relaxation is not bounded and no solution exists.

Finally, a major benefit of using the SMT strategy is that it allows for gradual implementation of rules, and does not require a full representation of the problem to be functional and testable. Regardless, its suitability hinges on the difficulty of expressing all rules as predicates, but this should be entirely possible when allowed the full flexibility of e.g. bitvectors, arrays and linear/non-linear arithmetic.

7 Data Representation

7.1 Requirements for the data representation scheme

When converting musical information to data, as with most transformations, it is first of all important that there is as little loss of information as possible. The neural network will not differentiate between correct and erroneous training data, and any inaccuracies will be propagated into its generated results. Likewise, although we have more control over the optimiser and may use it to correct some obvious mistakes, it must be able to use and manipulate data that portray the various aspects of music properly. Even though we have only glanced at the representative power of musical notation in this thesis itself, anyone familiar with it understands its elegant description of the interplay between pitches, harmonies, velocities, accentuation, rhythm and tempo in a pedagogic and intuitive way. While musical notation has been perfected over hundreds of years and is designed to be used by adaptive and conscious humans, the digital representation must strive to define abstract concepts such as the *feel* and *groove* of music by numbers and bits. While it can not be claimed that this implementation will surely achieve what great scientists have been attempting since the birth of computing, it is nonetheless imperative that the representation minimises loss of information wherever possible. In addition, it is intended that the process both begins and ends with an audio format, which means that the transformation needs to be bi-directional. This increases both the risk and severity of any loss if the scheme is not robust.

Secondly, the mutability of the data is of utmost importance. The optimiser will likely need to make in-place adjustments, disassemble/rebuild segments and apply a variety of arithmetical operations to the data in real time. Needing to constantly reconstruct immutable data structures from the bottom in an environment where millions or even billions of elements must be processed would both hamper any notion of efficiency and cause much unnecessary complexity for both the design and implementation of the software.

Third, the representation should be as simple and multi-purpose as possible, as there is already enough variables in music to keep track of without adding to this complexity. For instance, versatile and computationally efficient data structures such as numerically typed multidimensional matrices or bitvectors are to be preferred over lists of un-typed data or strings. Due to possible memory requirements of representing such a large state space, special consideration must also be given to the memory usage of a variable and its type. Representing low, always positive integer values as 64bit signed floats is meaningless and a waste of resources.

Using one, generalised data structure for a multitude of operations is preferable to having fragmented parts that need to be synchronised and reassembled. Although not directly a requirement, it is also desirable that the data is both printable and readable to make both implementation and testing easier. The software produced by the thesis will be open-source, and both the code and approach should strive to be comprehensible to anyone who wishes to explore, replicate or continue the work.

7.2 Input and output file formats

As mentioned in the previous section, it is the intent to both initialise and terminate the software with actual music that can be listened to. The reasoning behind this is two-fold. Music is an art-form that is best absorbed by the ears, and although software whose application is only to written music may well be useful for many people with specific interests, audible music caters to a much broader audience of people. In addition, the availability and usability of data to use as input is significantly skewed towards that which is recorded and stored digitally. That is not to say that there is not an immense amount of sheet music attainable both online and in paper form, but the task of compiling, translating and converting this visual information into usable digital data for our purpose is daunting in itself.

Luckily, it has been shown that it is possible to use both recorded music in lossless/lossy audio format (.WAV, .FLAC, .mp3, etc.) and digital interfaces (MIDI [64]) as input to neural networks. Google's very own *WaveNet* [65] uses such raw audio to train and generate both spoken language and music by means of machine learning, with impressive results. Similarly, one of its biggest competitors in the AI industry, OpenAI, have developed *MuseNet* [66], a "deep neural network that can generate 4-minute musical compositions with 10 different instruments, and can combine styles from country to Mozart to the Beatles". The major difference between the two is that *MuseNet* uses MIDI files as input, which contain meta-data on the composite musical elements in addition to being playable by software synthesisers that are native to many operating systems.

While both approaches have merit, a MIDI file is much simpler to convert into digital data that seeks to replicate standard musical notation. The file itself does not contain any direct audio representation, but rather acts as a blueprint that software synthesisers use to produce sound. The internal structure of a file is some global header information which initialises the correct state of the synthesiser, and a sequence of *messages* that are transmitted serially at a *sampling rate* of 31.25 kbit/s and contain all the necessary information about how and what it should play at a given point in time. An example message, or *event*, may at one point in time inform the synthesiser to switch on note x, with velocity y for instrument/channel z, while some message later on will tell it to switch it off. By regularly sampling the messages and reading the note-on/note-off instructions, the synthesiser recreates the piece of music within the accuracy of the sampling rate.

7.3 Piano roll

As the MIDI-file contains all the time-dependent information regarding note composition, instruments and pitch, it can be converted into a *piano roll* format which not only states the disjoint note-on or -off events, but also signifies when a note is active. To produce this, the MIDI-file is sampled at a rate which identifies all such note-related messages and populates a nxt matrix with information on all active notes n that are active at time-step t. This provides a visualisation of which notes (ranging from 0 to 127 (C(-1)-G9)) are pressed on a piano keyboard at any given point in time. A visual example of the implementation of piano roll functionality can be found in the documentation for *LMMS* [67], a freeware software for music production.

Whereas raw sound-wave manipulation depends on the processing of continuous data, MIDI and piano-roll provide the discrete representation of a continuous phenomena. By using piano roll as data structure for portraying note-related events, the music generation software can both maintain an overview of the entire musical structure, as well as directly access and manipulate the contents at a specific point in time. As a piano roll only needs to contain a matrix filled with integer values, it can be stored digitally as a simple *.csv* file, which are easy to import by most programming languages. Similarly, while raw audio files such as .WAV are rich in information and require relatively huge amounts of disk space, both MIDI- and .csv-files of standard-length music are negligible in their disk space requirements.

7.4 Internal data structures

During run-time, the data representation pipe-line will maintain the piano roll format starting from the conversion of the MIDI-input before the neural network module and until the reverse conversion after the optimiser is done. Both modules are designed to both allow input and generate output of the same format, which will be stored as .csv-files in within the hierarchical directory of the software on disk between modules. Using disk storage is necessary simply due to the fact that both the neural network training and optimisation search are potentially lengthy operations that may not be completable in "one sitting" for the user, and the software should cater to independent usage of both modules if so desired. Suitable libraries for converting between MIDI and piano roll representation is readily available online for most relevant programming languages. If additional meta-data is required to propagate between modules, it will either be added as a section of each .csv-file or propagated through separate text-files. At this point, it is expected that all potentially needed meta-data will be limited to relatively short length textual or numeral strings. As for the internal representation of each module, any significant departure from the piano roll format is specified in the next chapter regarding the implementation of the system.

8 Implementation: Bach Chorale Generator

8.1 Design and software architecture

8.1.1 From the research project

This section describes how the design ideas presented in the precursor research project are revisited and adapted into the final software architecture that has been used and implemented. The basic idea of the program flow is visualised in Figure 8, which featured in the final chapters of the research project.

The system is designed using several independent, yet mutually interfacing modules, such that the output of one matches the input of another. Initially, a preprocessor module manipulates and prepares the raw input data for use by the Deep Learning module. Having already stated that the raw input format is .MIDI and the neural network is to accept .csv (piano rolls), the preprocessing performs all required transformation between them.

Second, the Deep Learning module (**DLM**) contains a neural network which trains upon the supplied data and generates some output, also in .csv piano roll format. Although the generated data can be immediately transformed back into music (.MIDI), the intent is to pipe it into the postprocessor module.

This module is, in abstract terms, supposed to both correct and improve upon (validate) the generated music by enforcing the rules of the four-part harmony chorale and applying musical structure. In the research project, Integer Programming was proposed as a possible method of achieving this, but the further investigation conducted during this thesis has revealed that there are other options that seem more viable, most notably a Branch-and-bound search algorithm. Additionally, the module must perform the transformation from .csv to .MIDI, i.e. the reversal the preprocessing.

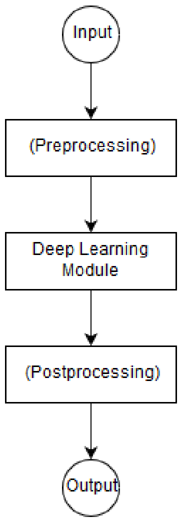


Figure 8: The discrete parts of the software.

At the very end of the project, a more exact architecture was proposed as depicted in Figure 9.

To briefly restate the idea, the prepared data enters the DLM which consists of an *Anticipation-RNN*, an advanced neural network architecture which claims to improve upon target prediction by the use of several separate units together. The output generated would then be validated by an IP which used a defined rules database, and then be re-checked for plagiarism (as this is a real problem within the industry). Finally, although the output could be transformed and played as .MIDI, it was also theorised that the system could possibly be circular in nature, i.e. using the IP validation as target function for the network itself. A novel idea, but one which imposes a whole new set of requirements to each module and the system as a whole, as it hinges on several factors such as both

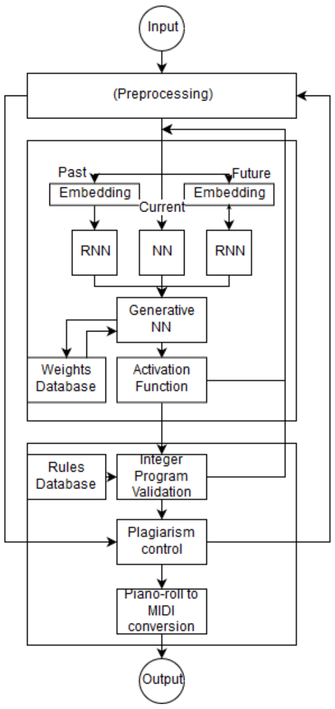


Figure 9: The concept model for a Deep Learning system validated by an IP.

quality and run-time of the IP validation.

Although the architecture of the software that has actually been implemented differs significantly from the one presented here, it still nonetheless follows a similar modular idea and process workflow.

8.1.2 Final design and process workflow

The final design of the implemented system is as depicted in Figure 10, which also depicts its workflow and the names of files that contain the main function for each module.

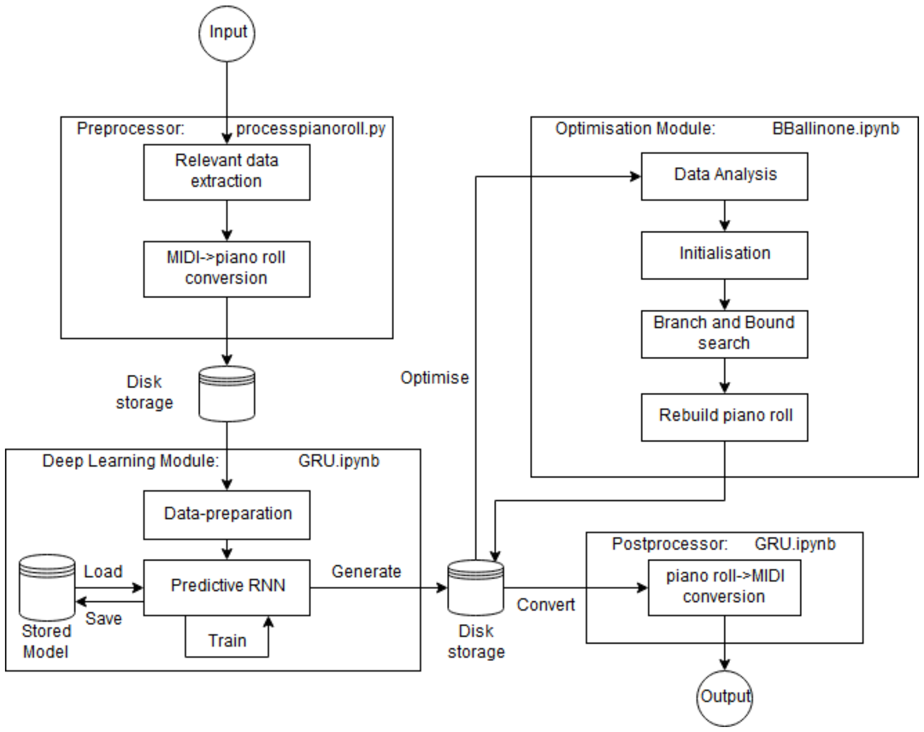


Figure 10: Design and workflow of the software

The exact contents of each module, and their reasoning, is described more detailed in the sections below. The program code is uploaded as a Github [68] repository, and is publicly accessible to view. All path references are relative to the root of this repository.

8.2 Preprocessor module

The main function for the preprocessor is located in "*neuralnet/helpers/processpianoroll.py*", and is a python script for extracting the *SATB* voices from the initial input; a collection of 46 of Bach's chorales in .MIDI format downloaded from *learnchoralmusic.co.uk* [69]. Although the website owner cites fair use, the raw input is not uploaded to the repository to avoid any possible copyright infringement. The complete list of works that have been utilised during implementation, testing and result generation is as follows (and is available for free on the website):

- Ascension Oratorio:
  - AC-06 Nun lieget alles unter dir
  - AC-11 Wann soll es doch geschehen
- Cantata 137
  - C137-05 Lobe den Herren, was in mir ist

- Cantata 140
    - C140-07 Gloria sei dir gesungen
- Christmas Oriatorio
    - CO-05 Wie soll ich dich empfangen
    - CO-09 Ach mein herzliebes Jesulein!
    - CO-12 Brich an, o schönes Morgenlicht
    - CO-17 Schaut hin! dort liegt im finstern Stall
    - CO-23 Wir singen dir in deinem Heer
    - CO-28 Dies hat er alles uns getan
    - CO-33 Ich will dich mit Fleiss bewahren
    - CO-35 Seid froh, dieweil
    - CO-46 Dein Glanz all' Finsternis verzehrt
    - CO-53 Zwar ist solche Herzensstube
    - CO-59 Ich steh an deiner Krippen hier
- Motet 3
    - M3-01 Jesu, meine Freude
    - M3-07 Weg mit allen Schatzen
    - M3-11 Weicht, ihr Trauergeister
- Mass in B-minor
    - MBM-03 Kyrie eleison
    - MBM-06 Gratias agimus
    - MBM-08 Qui tollis
    - MBM-13 Patrem omnipotem
    - MBM-16 Crucifixus
    - MBM-24 Dona nobis pacem
- St. John Passion
    - SJP-07 O grosse Lieb, o Lieb ohn alle Masse
    - SJP-09 Dein Will gescheh, Herr Gott, zugleich
    - SJP-15 Wer hat dich so geschlagen
    - SJP-20 Petrus, der nicht denkt zurück
    - SJP-21 Christus, der uns selig macht
    - SJP-27 Ach grosser König, gross zu allen Zeiten
    - SJP-40 Durch dein Gefängnis, Gottes Sohn muss uns die Freiheit kommen
    - SJP-52 In meines Herzens Grunde
    - SJP-56 Er nahm alles wohl in acht
    - SJP-65 O hilf, Christe, Gottes Sohn

  - SJP-68 Ach Herr, lass dein lieb Engelein

- St. Matthew Passion
  - SMP-03 Herzleibster Jesu, was has du verbrochen
  - SMP-16 Ich bins, ich sollte büssen
  - SMP-21 Erkenne mich, mein Hüter
  - SMP-23 Ich will hier bei dir stehen
  - SMP-38 Mir hat die Welt trüglich gericht
  - SMP-44 Wer hat dich so geschlagen
  - SMP-48 Bin ich gleich von dire gewichen
  - SMP-53 Befiehl du deine Wege
  - SMP-55 Wie wunderbarlich ist doch diese Strafe
  - SMP-63 O Haupt voll Blut und Wunden
  - SMP-72 Wenn ich einmal soll scheiden

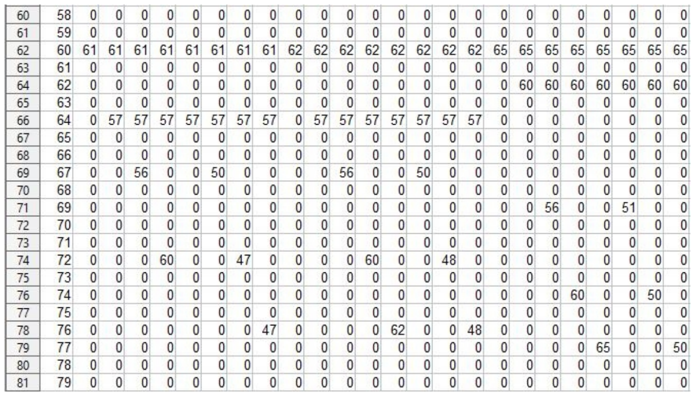
As a note, the software expect these tracks to be put in the "*neuralnet/datasets/training/*" folder.

Most, or all, of these files contain the full arrangement of instruments in addition to the SATB voices. Since the piano roll format is incapable of separating one from the other, the preprocessor begins with examining the *tracks* of each .MIDI file by utilising the *pypianoroll* [70] library. As each separate instrument is given a track, and these files consistently using tracks 1-4 for SATB, it is merely required to delete all others and write a new midifile with the same metadata. Afterwards, the preprocessor uses pypianoroll to export the contents into a .csv piano roll file, sampled at 1 timestep-per-second. Although a finer sampling would produce a more accurate representation of most pieces, chorals are generally slow and having long repeating section of the same notes could have a negative effect on the generated output unless a more comprehensive data preparation scheme was implemented. As for a prototype, a constant sampling rate serves to demonstrate the capabilities of the system.

As an example, Figure 11 shows the beginning of the piano roll representation for Bach's Prelude in C-major, sampled at 5 timesteps-per-second. As MIDI note numbers are 0-indexed, the first column describes what MIDI note number resides within each row. The pitch is decided by position, while the number of each cell is the velocity (volume) at which it is played. The voice-only versions of each file is stored in "*neuralnet/datasets/training/voicesonly/*", and the piano roll files in "*neuralnet/datasets/training/voicesonly/piano_roll_fs_1/*".

Since the removal of additional instruments can leave big pauses in between each choral verse, all empty sections in the piano roll are stripped down to length 1. Otherwise, since the neural network will train on a timestep by timestep basis, the generated output runs the risk of being completely void of notes.

Lastly, 12 *transposed* duplicates of each piano roll is created, ranging from -6 semi-tones to +6 semi-tones. This effectively creates 13 versions of each chorale, at least one for each key in the diatonic system with one key occurring as octave transposed. The reasoning for this is that



60	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
61	59	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
62	60	61	61	61	61	61	61	61	61	62	62	62	62	62	62	62	62	65	65	65	65	65	65	65	65
63	61	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60	60	60	60	60	60	60
65	63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
66	64	0	57	57	57	57	57	57	57	0	57	57	57	57	57	57	57	0	0	0	0	0	0	0	0
67	65	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
68	66	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
69	67	0	0	56	0	0	50	0	0	0	0	56	0	0	50	0	0	0	0	0	0	0	0	0	0
70	68	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
71	69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	56	0	0	51	0	0
72	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
73	71	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
74	72	0	0	0	60	0	0	47	0	0	0	0	60	0	0	48	0	0	0	0	0	0	0	0	0
75	73	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76	74	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60	0	0	50	0
77	75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
78	76	0	0	0	0	0	0	0	47	0	0	0	0	62	0	0	48	0	0	0	0	0	0	0	0
79	77	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	65	0	0	50
80	78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
81	79	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 11: The opening segment of BWV846, fs=5.

neural networks rely on having sizeable data for *generalised learning*. A training data of only size 46 scattered among 12 possible keys runs the risk of poor quality output and/or overfitting, as each note combination might only appear once or twice. Granted, it is highly likely that some of the transposed versions might teach the network voicings that extend beyond the allowed range for any given voice, but the optimiser will correct any such transgressions in the generated output.

As we now have a training set that consists of 598 files with an approx. length of over 60 timesteps, we are ready to feed these into the Deep Learning module.

8.3 Deep Learning module

8.3.1 Intermission: some thoughts and ramblings on the purpose of the DLM

As an aside, it seems relevant to specify the intended purpose of the neural network in the context of the thesis. Although the subject was touched upon in the thesis introduction, it is reiterated here to shed more light on the choices that have been made with regards to the implementation of the DLM.

Originally, the research project was focused entirely on the prospects of generating "good" music chiefly by the use of machine learning. However, the discoveries made during this time implied that this is quite a daunting task. The design and implementation of neural networks that produce desired output can either be ridiculously easy for some problems, or excessively difficult for others. Developing a model which accepts some set of notes and tries to predict/replicate/reformulate them is of the former, as all that is needed is a solid foundation, enough training data and enough computing power/time. The beauty of a neural network is that it nearly always produces *something* of interest, even if that thing is not exactly what was desired in the first place. We can juxtapose this to many traditional, specifically tailored algorithms that either do what they are supposed to do, or "fail" in their endeavour. To avoid stepping on any toes, it is not claimed here that the seemingly meaningless jumble of chaotic output from a poorly designed or trained neural network is more valuable than a non-optimal solution being provided from an optimisation algorithm. On the contrary, the non-optimal solution might be good enough, or nearly acceptable while few would want to listen to music that sounds like it was created by random number generation. But, what separates them is that the seemingly poor results of an NN may quickly improve over time or with the correct parameter changes, while other algorithms appear less dynamic in such regards. For an optimisation algorithm to work one must already have figured out how the problem can be solved, or the path leading to its solution, while machine learning leaves this problem to the computer in its entirety and merely seeks to accommodate its, and our, journey into the unknown.

As such, there is an intrinsic *mystery* to Deep Learning, the notion of unexplored territory, uncapped potential and unknown applications that are yet to reveal themselves. While there undoubtedly are many such uses for other methodologies as well, it is the current opinion of yours truly that they do not inspire quite as much interest and encouragement that they are preferable to use as foundation for a year of work. The black-box nature of a neural network lends itself to those that have a penchant for "tweaking", such as changing a parameter on one side, and seeing what comes out at the other, or theorising on what the output is useful for. Yet, it was readily apparent that improving upon the existing solutions and cutting-edge models of resourceful, prominent scientists within the field that are currently immersed in the same realm of exploration would be extremely unlikely or even outright impossible within the span of 6 months. Major entities in the industry such as Google and OpenAI have dedicated entire teams of experienced people and huge amount of resources towards generating music by Deep Learning, and have achieved notable success as was mentioned before. They are, in frank terms, impossible to compete with within the scope available to this thesis.

Therefore, with the intent of maintaining some pioneering spirit and combat the creeping sense of futility, the focus of this thesis has shifted more towards what the possibilities are when *interfacing* with a neural network with regards to music generation. The apparent challenges of ensuring structure, aesthetics and compliance with musical conventions can be approached from other angles than within the self-contained space of the DL model. Although nothing would please more than actually developing and exploring the possibilities within a more advanced and nuanced RNN model, its purpose in the software developed for this thesis is merely to create some data which can be processed later on. Of course, it would be great if its quality was self-sufficient and without need of further intervention, but such an outcome was deemed unlikely. The desire for having something to show for at the end which somewhat achieves the goal of generating aesthetically pleasing music has skewed a disproportionate amount of the effort and available work-hours over to the optimiser module. As such, the criteria for the implemented neural network is to produce something that does not appear random, and hopefully bears any resemblance to some work of art. Anything beyond this is a bonus.

8.3.2 Model and specification

The implemented neural network uses a fairly simple model, especially when compared to the *Anticipation-RNN* explored in the research project. Due to time constraints and difficulties of identifying a viable optimisation strategy, we are left with a bare-bones but functional RNN with a customisable amount of hidden layers and hidden nodes per layer. To kick-start the implementation, the contents of a public git repository [71] was used as skeleton code. It was supplied initially as part of an assignment for the NTNU course, TDT76, and already contains many useful helper functions (found in datapreparation.py and dataset.py) for music generation by Deep Learning. As such, much of the internal file structure is kept as is, with only minor modification outside the actual implementation of the neural network model itself.

The code for the RNN is found in "*neuralnet/helpers/GRU.py*", and is implemented using pytorch for python 3.7. It uses 2 layers of *Gated Recurrent Units* (**GRU**) to learn to predict the next time-step of a sequence as accurately as possible by looking at the time-steps preceding it. The network has an input layer size of 128, where each node is the binary representation of the activity of a specific MIDI note at a given timestep, 0 for off and 1 for on. The velocity of each note is not considered, only if it is on or not. Likewise, the output layer size is also 128. A forward pass in the model consists of the following internal layers:

- A *dropout* layer to encourage generalised learning,
- A *linear* layer for encoding the input,
- A *GRU* layer,
- A *ReLU* activation layer
- A *linear* layer for decoding the output
- An optional *sigmoid* layer (necessary for other loss functions that were experimented with but not used by current state of the system)

The model uses the *logits* version of *Binary-Cross-Entropy* (**BCE**) as loss function and *Adaptive Moment Estimation* (**ADAM**) as optimiser. If the sigmoid layer is used instead of the ReLU, a regular BCE must be used instead. The RNN is CUDA compatible, and will preferably use it if possible, but otherwise it trains using CPU which is significantly slower. For generation there is no noticeable difference between the two. There is save and load functionality for the model and optimiser parameters, such that training can be continued or generation be done immediately with pre-trained weights. There is also logging-functionality which shows the training history (number of epochs, loss) of any loaded state.

Many of the existing helper functions also provide a means to easily transform the data between piano roll and internal *tensor* representation, and to plot/play/embed the results by means of several libraries. The neuralnet directory contains a file (requirements.txt) which states all the various packages that are needed for usage. The training and generation processes are presented in their own respective sections.

8.3.3 Neural network initialisation

In this thesis's program code, the location of the main function for the DLM is found in "*neuralnet/helpers/GRU.ipynb*", a *jupyter notebook* [53] file. Jupyter notebook was chosen due to the convenience of being able to have saved states of the program to work and test with, especially when dealing with long run-time processes such as training a neural network.

The initialisation begins with importing necessary packages and instancing the model and neural optimiser with the desired parameters. For the training and generation of the results that are presented in the next chapter, the model was initialised with an input size of 128, 2 hidden layers and 256 hidden nodes per layer. The ADAM optimiser is initialised with default parameters and a learning rate of 0.0005. The input data is fetched from disk and the contents of each .csv file is represented as tensors, which allow for fast access and matrix calculations. At this point, the network is ready to be used either for training or generative purposes, with or without the loading of pre-existing parameters.

8.3.4 Training

The training process is straight forward. A total number of epochs to train is decided, where one epoch is one iteration through the entire data-set. By default, this amount is 2000, as the testing and generation was limited to the computational prowess of one *NVIDIA Geforce 980ti* using CUDA. If it is a fresh start, no additional initialisations are necessary, otherwise it imports the necessary information first.

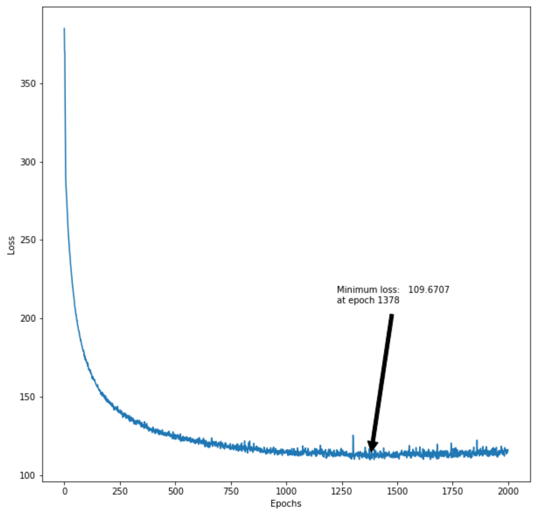
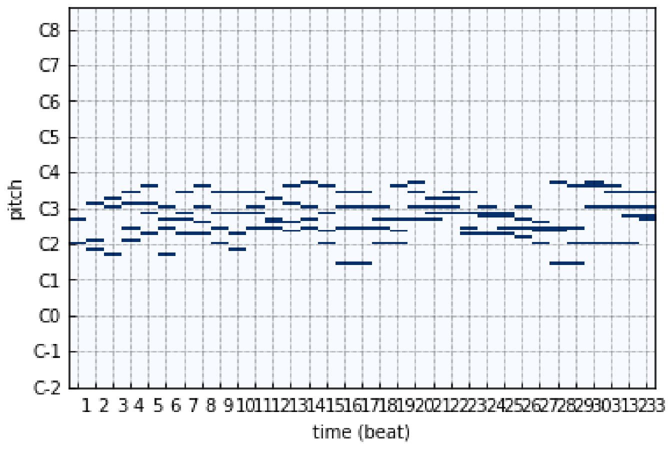
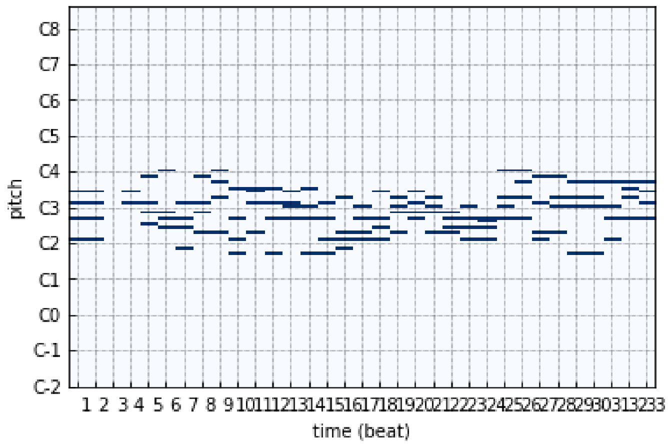


Figure 12: The loss graph of the training used to generate results.

During an epoch, each of the songs are processed separately, with the full length of each file processed as one batch. The order of processing is randomised, as this is best practice as it can mitigate overfitting. After predictions have been made for every time-step and the loss calculated, the network does its backward pass and adjusts its weights. The current best total loss result, and the corresponding parameters, is kept across epochs as it reverts to the "best" state found at the end of training. For instance, for our results the best state was found in epoch 1378/2000, and is therefore the one returned at termination. A complete graph of the loss in the training used for generating our results is shown in Figure 12. Notice how the curve quickly flattens out which could imply that adjustments need to be made to the parameters if learning is to be improved across longer sessions. It is however difficult to extrapolate any conclusions to this end as the loss could begin to steadily improve again at some later point. For our purpose, this is sufficient for the prototype as it is desirable that the output of the neural network is "imperfect" enough for the effects of the optimiser to be noticeable.



neuralnet0.mid



neuralnet13.mid

Figure 13: Two 33 second pieces generated from 5 input time-steps

8.3.5 Generation

After training has concluded, or parameters are loaded, the module is primed for music generation. As it is based upon a predictive model, an initial sequence of time-steps from the data-set is supplied as the initiator of the generated sequence. The model will predict the next time-step to follow, and then continuously predict upon its own output from there on out. The process uses a "moving window" of analysed time-steps, meaning that if it is supplied with the first five time-steps of "Wie soll ich dich empfangen", the next prediction will be on the four last of these with its own generated output as the fifth. By the end of the fifth generated time-step, no more of the original is considered and the generation has to be self sufficient from this point on.

After a generated sequence of the desired length is built, it is converted back into piano roll format and can be visualised by using the embedding functionality found in IPython for jupyter. Figure 13 shows the visualisation of two sequences generated by using the first five (sampled) seconds of "Nun lieget alles unter dir", and "Wann soll es doch geschehen", respectively. These results are stored in "*results/neuralnet*" for the .MIDI files and .csv versions in "*results/neuralnet/pianoroll*" for later use with the optimiser. In the repository, a full range of 33 second snippets generated by the first 5 seconds of all original work is already pre-made, and will be more thoroughly discussed in the next chapter. For now, the DLM has fulfilled its purpose and its outputs are processable by the optimiser.

8.4 Optimisation module

8.4.1 The purpose of the optimiser

All code that comprise the optimiser can be found in "*optimiser/fourpart/BBallinone.ipynb*". Although containing a large amount of cells that need to be executed, the main function can be found near the bottom which automatises the entire process from importing all available .csv's in the NN output directory to analysing, optimising and storing its own results in "*results/optimiser*".

The optimiser is written in Julia and is based upon a Branch-and-bound search which uses *dynamic programming* (*memoisation*) for the calculation of lower estimated bounds for candidate solutions. However, a significant portion of the optimiser code is also used for *input analysis*, as several assumptions need to be made before any rules can be enforced. Specifically, the means of defining the target *key*, *scale* and the *approximation scheme* has no obvious or trivial solution. It can therefore not be claimed that the methods employed for achieving these things are entirely objectively optimal or exact as there has been little explicit prior research found to draw any conclusions from in these specific contexts. Also, the challenge of choosing accurate and workable data structures for each musical element becomes readily apparent throughout some of this analysis. The thesis attempts to support original inventions with the gravity of existing work, but this is unfortunately not universally achieved.

The main objective of the optimiser is to adjust the output from the neural network "as little as possible" while achieving the requirements or adhering to the guidelines of the theory behind Bach's chorales. While only some of these rules were written out in plain-text in section 5.3, all requirements that have been implemented are clearly stated throughout the subsections below and are generally derived from Bekkevold [7]. The secondary objective is to adhere to the chief guideline of the musical style; to minimise the movement of each voice. There is an inherent issue with this goal however, as this guideline is intended for human composers who seek to harmonise an existing melody or improve upon other work. Objectively, the minimisation of movement in a computational sense can often be contrary to the goal of music composition as a whole; creating aesthetically pleasing music. Without some boundaries, the program might simply find a low-cost pattern and repeat it *ad nauseum*, or avoid movement altogether and generate piece consisting of little excitement and development.

Consequently, a third optimisation objective is derived, not from any specific rule-book, but from common musical sense; accommodate and encourage variation over strict repetition. As such, there will at times be mentioned rules or imposed costs that have no basis other than that they seem to improve upon the (subjectively) asserted quality of the generated music.

8.4.2 Analysis and initialisation

Tonal map

The optimisation process begins with importing a NN-generated 33 second snippet of piano roll, which is subsequently mapped into a *tonal map*. A tonal map is a 33x12 matrix, where the modulo 12 of active notes at a given time-step is inserted at the resulting index, and thus provides an overview of the number of each note class that is active at this time. As all MIDI notes whose modulo 12 equals 0 are C's, a *tonal vector* is "C-indexed". As an example, the tonal vector

$$[1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0]$$

describes a time-step which contains one C and two G#'s across all 128 pitches.

Identifying key and scale

The tonal map is used for two things; identifying the key and scale of the snippet and influencing the approximation scheme for calculating *tonal distance* which is used as the chief metric for "distance from the original" for the first optimisation objective. The key and scale identification is achieved through scoring every time-step of the tonal map against the specific pattern that depicts the major scale (and the minor, by relativity). After scoring for the first position (C-major/A-minor), it circularly shifts one step to the right and scores for the next pair(C#-major/A#-minor), etc. After shifting twelve times we are left with the initial scores for each pair of scales, and the one with the highest score is chosen as the candidate pair. In case of ties, all possible candidates proceed to the next step of scoring. Below is the fourth scoring (Eb-major/C-minor) for the tonal vector above. Tonal vector:

$$[1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0]$$

(Eb-major/C-minor) pattern:

$$[1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0]$$

which calculates to a full score of 3. If this pattern contains the most hits across all time-steps, the piece will be optimised with regards to one of these two keys.

The second step of the process is deciding which of the two candidate scales to choose, for which Rohrmeier and Cross's [72] study on the statistical properties of tonal harmony in Bach's chorales was interpreted and applied. In a figure near the bottom of the paper, a ranking of the frequency of each chord in a scale occurs, and how they statistically occur in conjunction are shown, for both the major and minor scale. The highest ranked functions are, in order:

- Major: T, D, S, Ts, Ss, D7
- Minor: T, Tm, D, Dm, S, Ts

Looking at the steps that comprise these functions in the total ranking, we can differentiate which alignment of the pattern that is the best fit for the most common chords for our specific tonal map. Naturally, the roots are the most frequent, so high weights should be placed on the tonal centre. Almost equally frequent is the mirrored tonal centres, Ts and Tm for maj/min respectively. Then are dominant functions, which places emphasis on step 5 and 7+, and third are the subdominant variations. Although if we outright just count the occurance of every note in these six functions, one would create a relatively homogeneous distribution (as is the intention with the popular chords as they comprise the entire tonal field), the thesis proposes to focus on the roots for the tonic, dominant, subdominant and the seventh step to decide tonality, as the root of the mirrored Ts/Tm cancel eachother out when trying to separate the scales. Hence, tonal patterns that heavily weight the root, fourth, fifth and major seventh are utilised to decide which scale suits best. The pattern for the major scale in its base position (unshifted for C) becomes

$$[1, 0, 0.25, 0, 0.25, 1, 0, 1, 0, 0.25, 0, 1]$$

and the mirror pattern for A-minor is

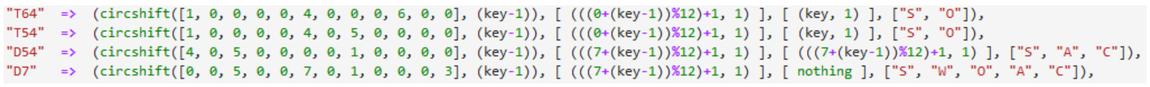
$$[0.25, 0, 1, 0, 1, 0.25, 0, 0.25, 0.75, 1, 0, 0.25]$$

These two weight-distributions emphasise quite differently and are proposed as sufficient to pass informed judgement on what is, after all, potentially chaotic output from the neuralnet. One could possibly comment how the minor pattern sacrifices some of the weight of its major seventh (harmonic) to the real minor, whereas it seems like the emphasis on maj7 is quite important to Bach in his chorales in minor. However, this choice was made to even out the playing field so that one would score better than the other for a given random distribution. The occurrence of the major seventh in minor is also a clear marking due to its uniqueness.

The end result of this scoring round is an approximation of the most accurate key and scale to use for the entire input snippet, and all subsequent optimisation is done in relation to it.

Initialising all legal functions

After the key and scale has been found, a dictionary holding the meta-data for every possible legal function is built. The meta-data for a given function consists of its tonal map (which is circular shifted with regards to key (key-3 for minor) index), the bass note class, note classes that can be doubled or tripled for the given function, and finally a definition of which rhythmic beat subdivisions the function can occur on. As this is better visualised than explained, the code for the initialisation of all functions belonging to a given major scale at key i is shown in Figure 14.



```
"T64"   =>  (circshift([1, 0, 0, 0, 0, 4, 0, 0, 0, 6, 0, 0], (key-1)), [ (((0+(key-1))%12)+1, 1) ], [ (key, 1) ], ["S", "O"]),
"T54"   =>  (circshift([1, 0, 0, 0, 0, 4, 0, 5, 0, 0, 0, 0], (key-1)), [ (((0+(key-1))%12)+1, 1) ], [ (key, 1) ], ["S", "O"]),
"D54"   =>  (circshift([4, 0, 5, 0, 0, 0, 0, 1, 0, 0, 0, 0], (key-1)), [ (((7+(key-1))%12)+1, 1) ], [ (((7+(key-1))%12)+1, 1) ], ["S", "A", "C"]),
"D7"    =>  (circshift([0, 0, 5, 0, 0, 7, 0, 1, 0, 0, 0, 3], (key-1)), [ (((7+(key-1))%12)+1, 1) ], [ nothing ], ["S", "W", "O", "A", "C"]),
```

Figure 14: Example code for initialising some functions

After execution, the dictionary can be referenced with corresponding function name as key whenever the meta-data is required.

Distance from original: Tonal method

Next is a highly subjective interpretation of the objective goal, distance from original. First of all, some reasoning and assumptions behind the chosen approximation scheme is in order. As there initially was no expectation regarding the distribution of notes for the input, the distance had to be calculable for each time-step regardless of number of notes that occur. If the NN has deemed that a current time-step is to be void of notes completely, or consist of nearly all 128 possible notes played simultaneously, some metric could still be applied to differentiate between what function could occupy the space.

A simple, and perhaps the most obvious, distance cost scheme would be to simply count the absolute distance that any proposed note combination has to the original. In a world where we are guaranteed an input of exactly four notes, this option is more palatable, however the optimiser was not implemented with this assumption in mind. It is neither assumed that the real MIDI note number found in the original lies within the limited range provided by the outer voices S and B. Given an original that has generated the extreme note 0 or 127, the relative distance-difference between note 60 and 61 is negligible. The chosen scheme proposes to emphasise the note class (tonality) over absolute distance. MIDI note 0 is a C, and to our ears it will function as a C. If all other notes in the time-step form the remaining notes in a C-major within acceptable range, the distance to the most suitable C should be less than the distance to the closest legal note within range (40, E). As such, distance from the input is based upon the tonal map instead of the original distribution across all 128 notes.

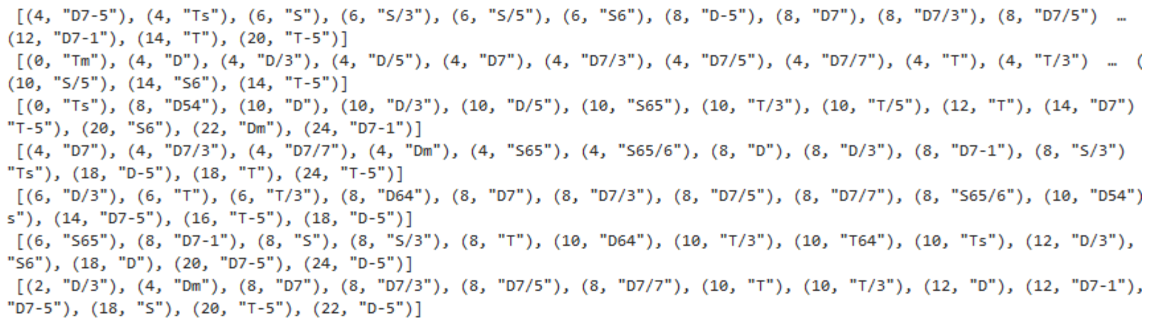
This has some significant effects on the output, both positive and negative. One such negative is that if we listen to the original, and then immediately listen to the optimised version which uses only the tonal map for distance calculation, it might sound very different at first, A tonal distance calculation may have lead to a selection of notes that are significantly higher or lower pitched than the original, changing the "soundscape" of a chord even though the note classes and their function might be the same. Contrarily, a purely absolute distance optimisation might change an original input which contains all of the necessary note classes to form a specific function bar one which is extremely distant to a functionally opposite chord. The extreme outlier disproportionately affects the combined distance cost to believe it is far away from any legal combination. Both of these extreme situations were weighed up and it was theorised that out of the two, a tonal distance would more consistently produce a metric that accurately portrays the human perception of the original harmony (or disharmony).

An additional drawback with using the tonal metric which should be mentioned, is that it increases the complexity of the required algorithm somewhat due to the circular modulo 12 system. The calculation needs to accept two tonal vectors, and then decide what the distance between them is. At first it was considered whether or not a simple *Hamming distance* would be sufficient, but this metric only measures amount of bit switches, and the bit switch from a C to a G (7 semi tones) would be equal to a switch from C to B which is only one semi-tone away. Not satisfied with this, an attempt was made to develop an algorithm which does a mixture of both original methods; the absolute distance within a modulo 12 number system.

To achieve this, a separate calculation is made for every chord in the valid function dictionary per time-step. It subtracts the input vector from the target vector (of the current function being analysed) which together forms a merged vector (*subvector*) filled with potentially both negative and positive values. If, by pure chance, the vectors match evenly up, the distance cost for the function at this time-step is 0 as the resulting subvector is filled with only 0's. But in most cases, there will be a distribution of some positive and some negative numbers. The gist of the algorithm is to select any entry and iteratively search for its closest neighbour of opposite sign in a left-right manner. If a neighbour is found one step to the left or right, the count is reduced (or increased) towards 0 in both positions and the algorithm continues on to the next occurrence of a non-zero number. After one full pass it can be assumed that all matches are within distance 1 of any position has been removed and counted. Starting over from the beginning, it does another complete pass with search-radius increased by 1, and looks for neighbours within two positions, etc. Naturally, one enters into the realm of possibility where it tries to search outside the bounds of the vector array, which should behave circularly. To accommodate this, the subvector is padded on both sides with the nearest 6 elements from the other direction, and the algorithm only iterates through the original 12 in the middle. Any matches that are found outside the original bounds are adjusted for both the outside positions and inside its own bounds for the same modulo position. The search continues until the contents of the subvector are all zero, or none of either positive or negative integers are left. The total amount of distance steps that had to be moved is returned as the function cost for the time-step.

It is to be noted that this algorithm was relatively complex to implement (for me), and although it has appeared consistent throughout rigorous testing, the thesis provides no guarantee or proof about objective consistency across all possible permutations.

By the end of the distance calculation scheme, a sorted *n_timesteps x n_functions* matrix has been initialised which contains the tonal distance to every possible chord at all time-steps (scaled up by a factor of 2 to compete with the voicing costs). One partial excerpt from such a "*function-potential*" matrix can be seen below 15.



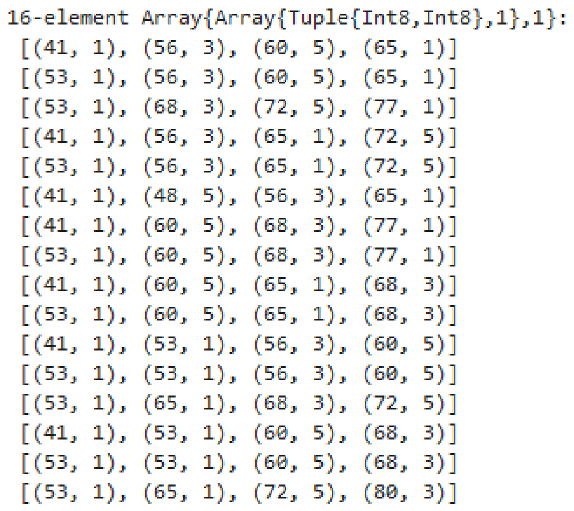
```
 [(4, "D7-5"), (4, "Ts"), (6, "S"), (6, "S/3"), (6, "S/5"), (6, "S6"), (8, "D-5"), (8, "D7"), (8, "D7/3"), (8, "D7/5")  …
(12, "D7-1"), (14, "T"), (20, "T-5")]
 [(0, "Tm"), (4, "D"), (4, "D/3"), (4, "D/5"), (4, "D7"), (4, "D7/3"), (4, "D7/5"), (4, "D7/7"), (4, "T"), (4, "T/3")  …  (
(10, "S/5"), (14, "S6"), (14, "T-5")]
 [(0, "Ts"), (8, "D54"), (10, "D"), (10, "D/3"), (10, "D/5"), (10, "S65"), (10, "T/3"), (10, "T/5"), (12, "T"), (14, "D7")
"T-5"), (20, "S6"), (22, "Dm"), (24, "D7-1")]
 [(4, "D7"), (4, "D7/3"), (4, "D7/7"), (4, "Dm"), (4, "S65"), (4, "S65/6"), (8, "D"), (8, "D/3"), (8, "D7-1"), (8, "S/3")
"Ts"), (18, "D-5"), (18, "T"), (24, "T-5")]
 [(6, "D/3"), (6, "T"), (6, "T/3"), (8, "D64"), (8, "D7"), (8, "D7/3"), (8, "D7/5"), (8, "D7/7"), (8, "S65/6"), (10, "D54")
s"), (14, "D7-5"), (16, "T-5"), (18, "D-5")]
 [(6, "S65"), (8, "D7-1"), (8, "S"), (8, "S/3"), (8, "T"), (10, "D64"), (10, "T/3"), (10, "T64"), (10, "Ts"), (12, "D/3"),
"S6"), (18, "D"), (20, "D7-5"), (24, "D-5")]
 [(2, "D/3"), (4, "Dm"), (8, "D7"), (8, "D7/3"), (8, "D7/5"), (8, "D7/7"), (10, "T"), (10, "T/3"), (12, "D"), (12, "D7-1"),
"D7-5"), (18, "S"), (20, "T-5"), (22, "D-5")]
```

Figure 15: A partial function-potential matrix

Generating all voicing permutations

When the module is optimising, it should be free to propose any permutation of tonal notes within legal range per voice that together form a valid function. For most functions, this means that the only note class that is strictly set is the bass note, while the rest are free to mix and match in whatever way suits best. As the Branch-and-bound search also needs to consider these voicings, a data structure that portrays them is necessary.

To find all possible permutations that can be assigned to a function, the program builds an array for each voice which contains all the MIDI numerical notes that match the requirements of the chord and subsequently assembles them in all permutations that achieve its bass note and doubling requirements. All completed legal voicing permutations are represented using a list of four tuples *(note_number, note_step)*, where position i is voice B, T, A, S in order. The note number is the MIDI note pitch, and the note step is the interval distance from the root of the chord, as this information is necessary to properly implement the various rules. Figure 16 displays all potentially legal permutations of T in F-minor (F-minor itself). The number of valid combinations for a given function depends upon the chosen key and scale. The total number of voicing permutations that are generated across all functions for the F-minor scale is 409, unevenly distributed between the 27 possible functions.



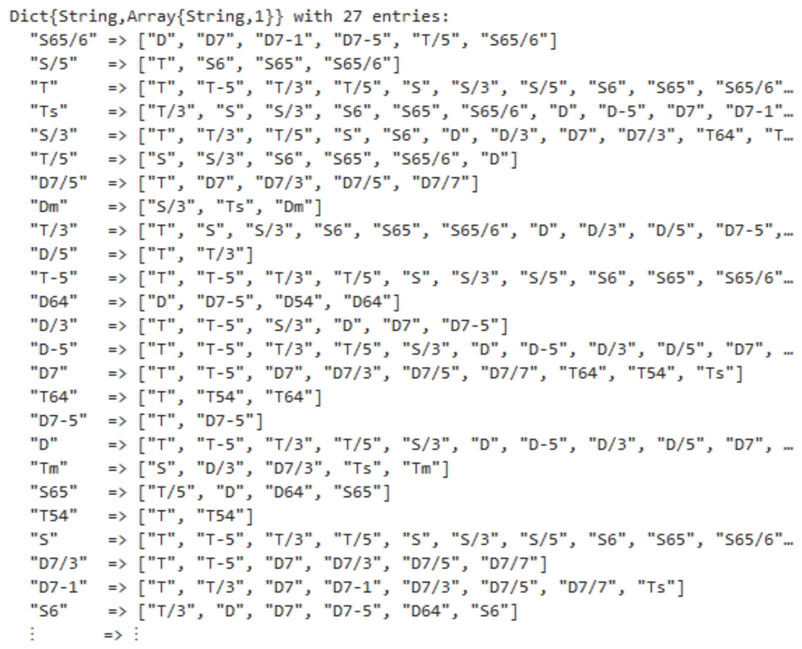
```
16-element Array{Array{Tuple{Int8,Int8},1},1}:
 [(41, 1), (56, 3), (60, 5), (65, 1)]
 [(53, 1), (56, 3), (60, 5), (65, 1)]
 [(53, 1), (68, 3), (72, 5), (77, 1)]
 [(41, 1), (56, 3), (65, 1), (72, 5)]
 [(53, 1), (56, 3), (65, 1), (72, 5)]
 [(41, 1), (48, 5), (56, 3), (65, 1)]
 [(41, 1), (60, 5), (68, 3), (77, 1)]
 [(53, 1), (60, 5), (68, 3), (77, 1)]
 [(41, 1), (60, 5), (65, 1), (68, 3)]
 [(53, 1), (60, 5), (65, 1), (68, 3)]
 [(41, 1), (53, 1), (56, 3), (60, 5)]
 [(53, 1), (53, 1), (56, 3), (60, 5)]
 [(53, 1), (65, 1), (68, 3), (72, 5)]
 [(41, 1), (53, 1), (60, 5), (68, 3)]
 [(53, 1), (53, 1), (60, 5), (68, 3)]
 [(53, 1), (65, 1), (72, 5), (80, 3)]
```

Figure 16: All valid voicing permutations for T in F-minor

Initialise function relations hashmap

To reduce the problem search space, a hash-map 17 holding all the legal successors to any function is supplied. Once again, as with the function meta-data dictionary, its contents differ on the chosen scale tonality. While 26 out of 27 functions are the same, the 27th is either an Ss for major scales or a Dm for minor scales. During cost calculation on a candidate solution, any function in time-step t which does not occur in the dict for the function at t-1 is pruned immediately as it is an invalid combination, regardless of voicing permutation.

While this table contains an absolute ruling, it must be stated that there is no such contracted overview in Bekkevold, but it will mention specific situations that are legal or illegal throughout the text. The book has been checked twice and thrice over for any such statements, and the findings compiled into this table. It is possible that some statements have been missed or wrongfully interpreted, but this should not severely reduce the quality of the output regardless. At worst, it slightly hinders flexibility or variation.



```
Dict{String,Array{String,1}} with 27 entries:
  "S65/6" => ["D", "D7", "D7-1", "D7-5", "T/5", "S65/6"]
  "S/5"   => ["T", "S6", "S65", "S65/6"]
  "T"     => ["T", "T-5", "T/3", "T/5", "S", "S/3", "S/5", "S6", "S65", "S65/6"…
  "Ts"    => ["T/3", "S", "S/3", "S6", "S65", "S65/6", "D", "D-5", "D7", "D7-1"…
  "S/3"   => ["T", "T/3", "T/5", "S", "S6", "D", "D/3", "D7", "D7/3", "T64", "T…
  "T/5"   => ["S", "S/3", "S6", "S65", "S65/6", "D"]
  "D7/5"  => ["T", "D7", "D7/3", "D7/5", "D7/7"]
  "Dm"    => ["S/3", "Ts", "Dm"]
  "T/3"   => ["T", "S", "S/3", "S6", "S65", "S65/6", "D", "D/3", "D/5", "D7-5",…
  "D/5"   => ["T", "T/3"]
  "T-5"   => ["T", "T-5", "T/3", "T/5", "S", "S/3", "S/5", "S6", "S65", "S65/6"…
  "D64"   => ["D", "D7-5", "D54", "D64"]
  "D/3"   => ["T", "T-5", "S/3", "D", "D7", "D7-5"]
  "D-5"   => ["T", "T-5", "T/3", "T/5", "S/3", "D", "D-5", "D/3", "D/5", "D7", …
  "D7"    => ["T", "T-5", "D7", "D7/3", "D7/5", "D7/7", "T64", "T54", "Ts"]
  "T64"   => ["T", "T54", "T64"]
  "D7-5"  => ["T", "D7-5"]
  "D"     => ["T", "T-5", "T/3", "T/5", "S/3", "D", "D-5", "D/3", "D/5", "D7", …
  "Tm"    => ["S", "D/3", "D7/3", "Ts", "Tm"]
  "S65"   => ["T/5", "D", "D64", "S65"]
  "T54"   => ["T", "T54"]
  "S"     => ["T", "T-5", "T/3", "T/5", "S", "S/3", "S/5", "S6", "S65", "S65/6"…
  "D7/3"  => ["T", "T-5", "D7", "D7/3", "D7/5", "D7/7"]
  "D7-1"  => ["T", "T/3", "D7", "D7-1", "D7/3", "D7/5", "D7/7", "Ts"]
  "S6"    => ["T/3", "D", "D7", "D7-5", "D64", "S6"]
  ⋮       => ⋮
```

Figure 17: The valid function table for minor scale

Rhythm map

Last before the rule-specific sections of the code is the *rhythmify* function. Originally intended to be a dynamic system which somehow differentiates between the various time signatures and subdivisions, it is implemented in a very basic form in the current prototype. Rhythm proved to be one of the most difficult aspects of music to capture, as by the nature of our midi sampling and neural network training, there exists only one tempo at the moment; 60 bpm, or 1 beat per second. The whole system is built around the assumption that there needs to be some change in notes at worst every second time-step, whereas a finer sampling of the original could produce sequences of varying lengths which are stagnant. For instance, if the original is sampled five times per second, one rhythmical beat might suddenly have a very erratic number of time-step representations. On the other side, a flat 1 per second sampling makes it so that finer grained movement is lost to the system. However, the neural network itself will also have this shortcoming, else an equally fine sampling rate is used from the very original musical piece. Again, this would propagate into the learning mechanism, where there would be a much higher frequency of long stretches of repeating notes in the output.

At any rate, for a proper implementation of rhythmic elements, it would undoubtedly be required to make some significant changes all the way back to the very first pre-processing action. Some brainstorming on how this could be done is left for the discussion regarding future work at the end of the thesis, but as of now the *rhythm map* of the piece is hard coded to follow a 4/4 time signature with no or one subdivision per beat. Across 16 time-steps, a raw rhythm map with subdivision would look like this:

$$["S","O","W","O","S","O","W","O","S","O","W","O","S","O","W","O"]$$

and contain two measures worth of rhythm. The strings represent Strong-, Off- and Weak-beats respectively. To visualise a rhythm map with no subdivision, one only needs to remove every occurrence of "O" and populate with alternating "S" and "W" producing four measures worth of information.

To introduce some structure to the music, some of the indexes of the array are replaced to signify Authentic cadences ("A") and intermediate Cadences ("C") which have specialised rules related to them, and can only feature a specific subset of all available functions.

Distance cost

There has been implemented a function that utilises a compromise of absolute distance from the original input to a suggested voicing. Instead of just counting the distance from ever note in input to a corresponding one in the voicing, it seeks to describe the *width* of the input. A voicing can have four relatively low notes and reside in the low spectrum, mid or high spectrum. If there is little distance between bass and soprano, we say that the voicing is *tight*. Alternatively, if the internal distance between each note is maximal according to the rules, the voicing is *spread* and overall represents a larger spectrum of the total pitch range.

This distance function calculates *how many octaves away* the lowest note in the input is to the lowest note in the suggested voicing, and likewise the octave distance between the two top notes. Naturally, a voicing that resides within the same width of the input will score better, but it does not overly differentiate between those that are roughly in the same area. This distance cost is specifically used to select the starting T for the first chord, as this makes it more probable that the input and output will at least sound similar in the beginning. Although this is indeed a nice metric to use, there are some major issues related to using it as a term in the cost calculation for the actual Branch-and-bound search which is presented next. For now, the reader should be aware of its existence as its implications are not negligible for the objective goal even though it is not actively used for the optimising search.

8.4.3 Rule implementation

The first cost-function which is implemented is the *musical function cost-function* (excuse the term), which stops any attempts at forming candidate solutions that contain illegal chord combinations in their tracks. In general across all rules, a flat penalty of 1000 is added to signify strictly illegal moves, and any total costs that exceed 999 in the search algorithm is pruned away.

Below is a list of the implemented rules that are solely dependent on function choice alone, along with corresponding penalty if broken. The parenthesis signify if it is directly derived/interpreted from the book by Bekkevold, or any additional notes).

Function rules:

A The current function is a valid successor to the previous function as shown in Figure 17, 1000 (book) (numerous distinct rules)

B Intermediate cadences:

1. must end with T, D or Ts, 1000 (book, all cadences)
2. must rest on its final function, 1000 (implied by book, punctuated notes for phrase end)
3. must begin with T, S, S/3 or any D, 1000 (book, all cadences)
4. (any) D -> T, Ts or D, 1000 (book, tonal/authentic/disappointing cadence)
5. can not contain only D, 1000 (book)
6. T -> T, S, or D, 1000 (book, tonal cadence)
7. can not contain only T, 1000 (book)
8. S -> S or D, 1000 (book, tonal cadence)
9. can not contain only S, 1000 (book)
10. S/3 -> S/3 or D, 1000 (book, phrygian cadence)
11. can not contain only S/3, 1000 (book)
12. Ts -> Ts or D, 1000 (book)
13. can not contain only Ts, 1000 (book)

C Authentic cadences:

1. must rest on final function T, 1000 (book)
2. can only contain T, S or any D, 1000 (book)
3. must end on D->T, 1000 (book)

D Chord repetition can only happen from stronger to weaker beats, 1000 (book)

E T/5, S/5, D/5 require the same function before and after, 1000 (book)

F Ts -> S/3 or Ss -> D7-1 must decrease in rhythmic accentuation, 1000 (book)

Next are the voice-specific rules which require the additional information about which voicing permutation is being judged. It is assumed that all partial candidate solutions that reach the voicing cost calculation stage has passed the previous functional requirements from the table above at the current time-step. It is worth noting that some rules have dependencies that extend beyond the current and previous time-step, and in such situations we need to look at time-step t-2, which is referred to as the *ancient* time-step.

Similarly to the list of function rules, the voicing rules that are implemented are listed below in the same format. One major difference is that in addition to considering strict legality we also need to encourage the various guidelines that exist within the rule-set. Due to this, rules that are formulated with a "should", or "encourage" are only assigned a small cost. The objectively best values for each of these costs are not certain, as only small adjustments can have significant effect on the generated output. In any case, an attempt has been made to create a balanced cost distribution which can be backed by some rationale, but the thesis does not claim to have found the optimal balance across all rules. Also, make specific note of the different usage of interval names and plain numbers. An interval signifies absolute distance between two time-steps, whereas the possible numbers 1, 3, 4, 5, 6, 7 signify active steps in the current chord with relation to the root of the function, not the tonal centre of the scale.

Voicing rules:

A General voicing rules:

1. Any voice can hold the same note over a maximum of 4 time-steps, 1000 (not book, but to combat stagnation)
2. Leaps in bass should be countered by a move in the other direction, 2 (book)
3. Bass should be encouraged to "walk", i.e. move in the same direction, 2 (book)
4. Dead (stationary) bass should be avoided unless a part of some cadence or augmented chord, 2 (implied by book)
5. Soprano should be encouraged to not switch direction, 2 (not book, but promotes melody)
6. Voicing avalanches are illegal, 1000, (book)
7. Voices should strive to minimise movement (within reason), *situational* (book)
8. Discourage voicing repetition, 1 (book)
9. Discourage voicing repetition in authentic cadence before end, 4 (not book)
10. Voices are encouraged to not double the same MIDI note, 4 (implied by book by careful use of tight voicings)
11. Encourage opposite contrary or oblique motion between S and B, 2 (book)
12. B should be encouraged to move step-wise, 2 (book)
13. Mid voices (A, T) can only leap 7 (P5) or less, 1000 (book)
14. Outer voices (S, B) should only leap 7 (P5) or less, 2 (book)
15. Outer voices may move < 9 (m6/M6) or 12 (P8), 1000 (book)
16. Parallel P5, P8 or d5->P5 are illegal, 1000 (book)
17. Hidden parallel P5 and P8 are illegal, 1000 (book)

B Function specific rules:

1. Dominants

   a Unless in a cadence, S->D requires B moving step-wise up, and rest shortest path down, 1000 (book)

   b In D->T, leading note (3) must resolve into 1, or down into 5 if cadence, 1000 (book)

   c D7, D7/3, D7/5, D7/7, D7-5: leading note must resolve into 1 (or 5 if cadence), 1000 (book)

   d D7, D7/3, D7/5, D7/7, D7-5: 7 must resolve into 3 of (suitable) T, 1000 (book)

   e D7, D7/3, D7/5, D7/7, D7-5: 1 must go to 1 or 5 in (suitable) T, 1000 (book)

   f D7/5 -> Any: 1 must remain stationary, 1000 (book) (2 rules)

   g D7/5 -> Any: Bass (5) must move step-wise in and out of the chord, 1000 (book)

(2 rules)

h D7-1 -> T: Leading note must resolve into 1 (or 5 if cadence), 1000 (book)
i D7-1 -> T: 7 must resolve step-wise into 3 or 5, 1000 (book)
j D7-1 -> T: Doubled 7's must resolve in opposite directions, 1000 (book)

2. T/5, S/5, D/5: Bass (5) and 1 must move max step-wise in and out, 1000 (book) (2 rules)
3. Augmented Chords

   a T64, D64: 4 must be introduced stationary, 1000 (book)
   b T64, D64: 6 should be introduced step-wise, 1 (book)
   c T64, D64: 4 must be resolved step-wise down (unless going to T54, D54), 1000 (book)
   d T64, D64: 6 must be resolved step-wise down, 1000 (book)
   e T64, D64: Bass (1) must remain stationary after resolution, 1000 (book)
   f T54, D54: 4 must be introduced stationary, 1000 (book)
   g T54, D54: Bass (1) must remain stationary after resolution, 1000 (book)
   h T54, D54: Can not have same function before and after, 1000 (book)
   i T54, D54: 4 must be resolved step-wise down, 1000 (book)
   j T54, D54: All 1 and 5 notes must remain stationary after resolution, 1000 (book)

4. T/3, D/3, S/3:

   a T/3->D/3, S/3->T/3: bass needs to leap downwards, 1000 (book)
   b T/3->S/3: bass needs to leap upwards, 1000 (book)
   c T/3->D/3, S/3->T/3, T/3->S/3: The leap in bass must be countered by step-wise movement in the other direction, 1000 (book) (2 rules)
   d Only use two X/3 chords in conjunction if soprano moves in contrary or oblique motion to the bass.

5. Mediants:

   a Any D->Ts: leading note becomes 3 of Ts, 1000 (book)
   b Mediants should be used sparsely and not dominate the progression (limited to max 4 occurences across 4 measures), 1000 (stated in book, but no decisive number supplied)

6. S65, S6:

   a S65: 5 and 6 must be introduced and resolved in oblique motion, 1000 (book) (3 rules)
   b S6: 6 should be introduced step-wise, 2 (book)
   c S6: Upper voices must resolve downwards or stationary, 1000 (book)

7. Final T in authentic cadence should be spread, 1000 (not explicitly stated in book, but

appears to be convention in supplied examples)

Most rules are fairly straight forward, but some are quite vague. For instance, the optimisation objective **A6** is implemented with some subjective interpretation. A rigid interpretation would be to assign a linear penalty equal to the distance travelled across time-steps, but an extremely high risk of creating monotonous music. Another possibility is to allow all semi-tone or full-tone increments (distance < 3) as if it was zero, which is currently in use as step-wise movement is generally encouraged or allowed between any two legal chords. But how should the non-step-wise movement be penalised? One option is to heavily differentiate between 0-2 and 3+, imposing the real distance cost for whatever leap has been performed, resulting in a major bias towards step-wise or stationary movement at any point in time. Unfortunately, this has some undesirable results as the functions will tend to get stuck going back and forth between two low cost voicings.

Alternatively, one could impose the linear cost increase from this point on, i.e.

$$cost = abs(x) > 2 \ ? \ abs(x) - 2 : 0$$

which is a more forgiving approach but still realises that a fifth is more extreme than a third. Its drawback is the fact that a scale has varying absolute distance intervals within it. A m3 (d = 3) would be cheaper than an M3 (d=4) although they are functionally the same and only dependant on position in the scale. To circumvent this, the thesis proposes a flat movement cost scheme

$$cost = abs(x) > 2 \ ? \ c : 0, \ c = constant$$

where any leap within the legal bounds of each respective voice is considered equally "bad" in comparison to step-wise motion or no movement at all. While this somewhat contradicts the guideline of absolutely minimising movement, it is important to realise that some legal function and voicing combinations are by nature more "expensive" than others while still being desirable for richly varied musical results. As there already are other fail-safes in place which would prevent directly illegal situations.

With this in mind, the three lower voices are set with flat penalty c=3 for any leaps, heavily encouraging step-wise or stationary motion. Mid voices are generally more restricted than S and B, but we wish to encourage a "walking-bass" which means that it also must be heavily penalised. The soprano is allowed more freedom and only penalised with c=2 for leaps. This scheme and selected constants are not conclusive and is open for further discussion.

8.4.4 Branch-and-bound search

At last, the Branch-and-bound method is almost ready to be executed to locate what is, hopefully, optimal candidate solutions with respect to the three objective goals. There are certain necessities before the main loop can begin to iterate, which is the final manipulation of many of the analysis data structures that have been created previously.

Preparation

First, the search has been split into two phases, phase 1 which searches through the partial candidate solutions from time-step 1 through 16, and phase 2 which begins at the previous time-step

16 and searches for the completion of the sequence. The main reasoning behind this is simply that, although significant pruning can be done, no implementation of the algorithm has been found that reduces the complexity to non-exponential levels. Secondly, the search uses memoisation to mitigate the search time, which requires a very large data structure to be initialised. More specifically, its size is

$$(n\_timesteps - 1) * n\_voicings^3$$

due to the existence of serial dependencies up to two time-steps behind. To emphasise this magnitude, for an input piece that is analysed to be in F-minor, there exist 409 voicing permutations that can be used. The required matrix for storing every visited combination of three at every time-step out of 16 contains

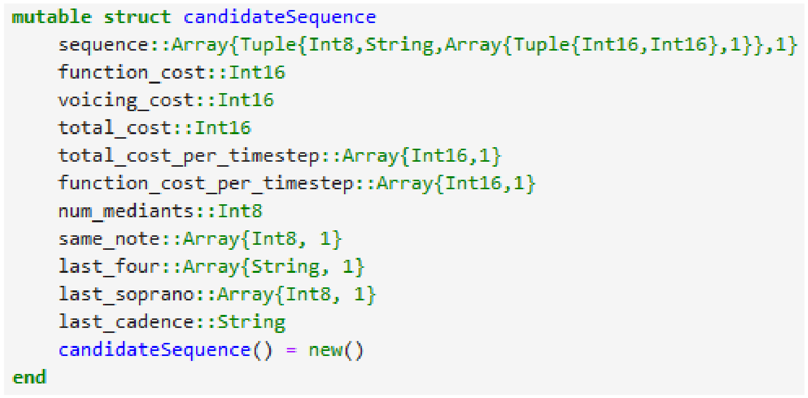
$$15 * 409 * 409 * 409 = 1,026,268,935$$

cells, and this is with removing the necessity of storing the last time-step. In reality, there is little need for the first time-step either and the number could be reduced to 957,851,006, which is still very size-able. For this reason, it was decided that although separating the piece into at least two parts would infringe upon the concept of finding the optimum low cost across the entire sequence, it serves to both mitigate the memory usage and run-time. It can also be argued that a chorale verse, or phrase, rarely lasts much longer than a couple of measures, and the rules state little about the requirements for the transit from one verse to the next.

During the preparation stage, the function potential list (which is sorted on tonal distance cost) is further manipulated. For phase 1, all functions except T are pruned at time-step 1 as it is the only valid choice to begin with. Next, it marks the rhythm map with the location of the cadences, which at the moment is set to be leading up to the middle of phase 1 (t=6, 7, 8) and at the end (t=13, 14, 15, 16). Phase 2 is more free and lets the algorithm chose whatever during the middle until the authentic cadence at the last four time-steps (t=13, 14, 15, 16) and locks 15 and 16 to T. Next, it iterates through every time-step of the function potential list and prunes away all functions that are not allowed at the current beat by the information supplied by the rhythm map and function meta-data. The last operation is to select the starting voicing of T which suits best with regards to the distance_cost from time-step 1 in the input. In case of a tie, all best scoring options are added to the active set.

The candidateSequence struct

Every candidate solution uses a custom *struct* type for representation. A candidateSequence struct and its fields can be seen in Figure 18.



```
mutable struct candidateSequence
    sequence::Array{Tuple{Int8,String,Array{Tuple{Int16,Int16},1}},1}
    function_cost::Int16
    voicing_cost::Int16
    total_cost::Int16
    total_cost_per_timestep::Array{Int16,1}
    function_cost_per_timestep::Array{Int16,1}
    num_mediants::Int8
    same_note::Array{Int8, 1}
    last_four::Array{String, 1}
    last_soprano::Array{Int8, 1}
    last_cadence::String
    candidateSequence() = new()
end
```

Figure 18: The candidateSolution struct

The first field, sequence, contains an array of tuples which signify the chosen function at all time-steps up to this point. A time-step tuple has the format

$$(function\_cost,\ "function\_name",\ [(bass\_note,\ chord\_index),...])$$

and the final solution consists of 32 of these. While the function_cost in a sequence tuple only depicts the tonal distance cost of that specific function at the time-step on which it was chosen, the function_cost field in the struct keeps a running total of the entire sequence after it has passed through the chord-cost function defined earlier. The same applies to the voicing_cost field, the total_cost sum between them and separate containers total_cost_per_timestep and function_cost_per_timestep that are used at various points of the search and memoisation strategy. The rest are used at various points for tracking information that extends beyond the immediate vicinity.

The search

The active set, initalised with one or more candidateSequence(s) with cost 0 and sequence length 1, is passed into the main search loop. The general procedure is similar to the pseudo-code presented in chapter 6 but with some alterations. First of all, there are (at least) two discrete factors to estimate and differentiate between, function cost and voicing cost. This means that we first need to select a function, then build new candidateSequences for every available voicing permutations. The function is selected in a best-first manner, choosing the one from function potential at time-step 1 with the lowest distance cost. It then does one pass through the chord-cost function before exploring each potential child and their respective voicing, such that an invalid chord combination does not need to be processed multiple times. If the function combination survives pruning, it iterates through all the available voicing permutations, checks its function cost + voicing cost + estimate from the lookup table and compares it to the current best value (upper bound). As its estimator lookup-table has not yet been updated, it uses a flat constant*remaining time-steps as heuristic, which is a customisable number. To guarantee optimality, it needs to be set to be lower than the remaining cost of the optimal solution as this is the definition of the optimistic lower estimate. However, if it is assumed that no perfect input would be supplied by the neural network, this estimate does not need to be zero. If, for instance, we assume that the typical average cost for the best solutions found for the first 16 steps of neural net output lie somewhere around 200, this translates into an average cost-per-time-step of 200 / 15 (as the first step is free) = 13.33. If the real cost is uniformly distributed across all time-steps, an "unoptimistic" lower bound set at 13.33 would then be likely to achieve the same quality of solution, but not any better.

If the estimate exceeds the initial upper bound (which only happens if it breaks any rules) it is pruned away and the memoisation cell in $[time{-}step, ancient\_voicing, prev\_voicing, current\_voicing]$ is updated in the lookup-table to be 1000, which shows that this is an impossible combination to proceed from. The indeces of each voicing is retrieved from a dictionary which returns its position in the order they were generated, ensuring consistency throughout the lookup-table. When the active node locates a child which is both legal and is of lower cost than the upper bound + an optimistic estimate it is added to the activeset as a new potential node to explore at depth+1. As one active node will generate and test all children before dying, there are potentially a number equal to the total amount of voicing permutations of new nodes being added to the active set at every timestep towards the bottom. However, they are processed in such a way that the *best* matched function by tonal distance is added *last*. As the active node dies, this best child is at the top of the LIFO stack and is the first to be explored further. This ensures that a seemingly greedy candidate solution is built relatively quickly which both ensures that a solution can be returned if the search needs to be prematurely terminated and that the upper bound is quickly reduced to a tighter value enabling better pruning.

If the child is at the solution space depth (16) and is better than the previous best, it is accepted as a solution and the upper bounds are adjusted. In addition, as the search is depth first, we can guarantee that the best out of all the children generated by this specific function + voicing sequence can at the end of this active node's lifetime be used to update the lookup-table from final depth to final depth - 1 for this specific combination. All other sequences that lead up to the same penultimate decision with the same two precursor voicings can do no better than this one did after an exhaustive search. As such, the lookup table for $[15, V_{13}, V_{14}, V_{15}]$ is set to be equal to the best found cost from 15 to 16.

After all children have been searched and/or pruned, the active node is dropped and a new one is popped from the top of the stack. Since it is DFS, this active node will be the most recently added child with a sequence length no shorter than the longest one in the stack. In such a way, the search initially identifies the greediest solution possible, and then conducts the real search from end to start, all the while adjusting its internal lower bounds and lookup-table heuristic to match the current best found for any partial solution from the current depth back to the end. This however, would prove more difficult than anticipated.

The most important thing to keep in mind when designing the algorithm for tracking the estimated lower bound is that you can never make any conclusive decision about a combination before all its successors have been explored. The updating of the lookup-table will therefore have to wait until the active node is dead and we can be certain all its children have been properly explored. By the nature of DFS, we can be sure that whenever the depth of a new active node is *less* than the previous active node, the best solution found previously is optimal for the partial sequences of all children of the last node to be processed from *this* depth until end. If there existed a better solution, it would have been found before the depth was reduced as all the children were exhausted. Using this knowledge, the algorithm is intended continuously propagate the remaining cost of all previously found solutions in the lookup-table from end up to the new, lesser depth. The rationale is that there is nothing stopping the algorithm from search new combinations starting from the lesser depth, but whenever it finds itself reverting back into the path of a previous solution it will be given the remaining cost up front. This, however, turned out to be quite complex to implement properly, and the current prototype contains bugs that cause it to not function as intended. This has the unfortunate effect of not mitigating the exponential complexity sufficiently to make the search terminate within reasonable time. In fact, it has been necessary to utilise drastic "shortcuts" to produce results which are not optimal, but valid. As for now, the reader is made aware that the intended implementation with proper usage of estimated heuristics does not terminate in reasonable time, and all results are generated using a greedier method which locates a number of valid solutions beyond the most greedy and then terminates. It is still the intent for the prototype to be implemented using proper heuristics, so these shortcuts are deemed irrelevant for the original design and are rather presented in the discussion chapter.

Re-conversion into piano roll

As part of the singular main function, the optimiser rebuilds the chosen candidate solution with the smallest cost back into piano-roll format as is, except for the final touch of adding a *Picardy third* to the last T if the piece is in the minor key (transforming the minor T into a major T). This was usually the case for most chorales that Bach wrote, and also hugely popular in other contemporary styles. The finished piece is stored back on disk in the "*results/optimiser/pianoroll/*" directory.

8.4.5 Output finalisation

The last step of generation is simply to transform the optimiser output into .MIDI using the same python tools as was used for the reverse transition. As the optimiser was written in Julia, we return back to *GRU.ipynb*, where the bottom sections are reserved for both transforming the output of the neural network and the optimised version. The generated and validated Bach chorale versions of the neural network outputs are located in "*results/optimiser/*". To round of the chapter before actually listening and commenting on the results, Figures 19 and 20 show the visualised piano rolls of the neural interpretation of "Wann soll es doch geschehen", along with the optimised version.

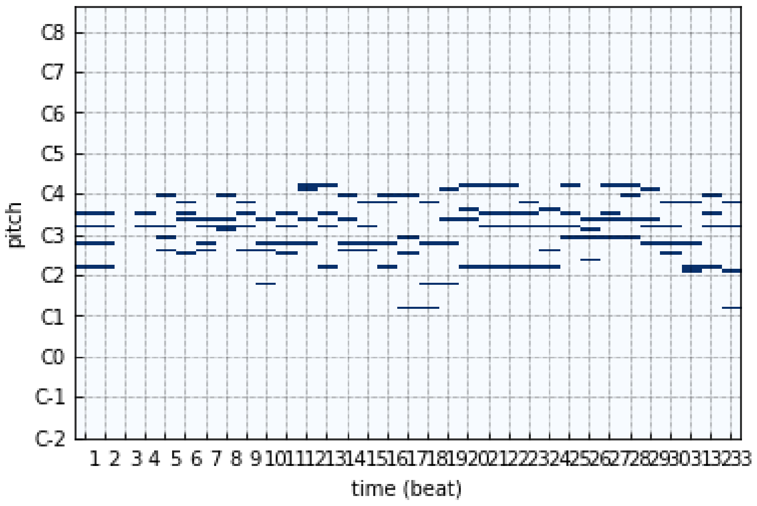


Figure 19: The neural interpretation of BWV11

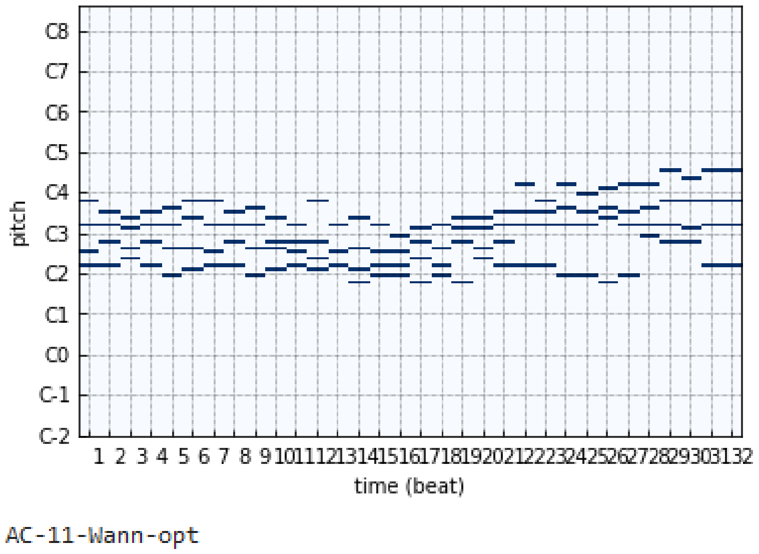


Figure 20: Its optimised counterpart

9 Results

9.1 Overview and comparison of generated results

Original file	original key	analysed key	cost	search time	solutions found	rating
AC-06	D-Major	A-Major	443	14.95s	56	4
AC-11	D-Major	D-Major	444	22.6s	30	5
C137-05	C-Major	C-Major	339	10.0s	34	3
C140-07	Eb-Major	Bb-Minor	419	33.5s	60	2
CO-05	A-Minor	A-Minor	456	36.9s	38	5
CO-09	D-Major	D-Major	484	9.2s	15	2
CO-12	G-Major	B-Minor	494	8.5s	23	3
CO-17	C-Major	F-Major	464	8.8s	69	3
CO-23	G-Major	G-Major	450	30.1s	54	3
CO-28	D-Major	E-Minor	512	13.5s	36	3
CO-33	G-Major	C-Major	391	12.8s	60	4
CO-35	F#-Minor	E-Major	440	18.5s	28	3
CO-46	A-Major	A-Major	443	37.6s	50	3
CO-53	A-Major	A-Major	412	9.2s	38	3
CO-59	G-Major	E-Minor	446	21.4s	65	4
M3-01	E-Minor	Bb-Minor	435	15.3s	41	4
M3-07	E-Minor	Eb-Major	434	14.0s	58	2
M3-11	E-Minor	Bb-Minor	435	15.2s	41	3
MBM-03	F#-Minor	F-Major	452	20.7s	44	3
MBM-06	A-Major	A-Major	411	10.9s	44	5
MBM-08	B-Minor	D-Major	472	4.6s	37	3
MBM-13	D-Major	A-Major	431	21.4s	62	4
MBM-16	E-Minor	E-Minor	425	12.5s	53	3
MBM-24	D-Major	A-Major	411	11.8s	44	4
SJP-07	G-Minor	G-Minor	459	20.2s	42	4
SJP-09	D-Minor	F-Major	480	86.6s	80	3
SJP-15	A-Major	A-Major	445	32.5s	63	2
SJP-20	A-Major	C#-Minor	404	16.3s	71	3
SJP-21	E-Phrygian*	C-Major	507	6.8s	25	4
SJP-27	A-Minor	E-Minor	495	6.3s	62	2
SJP-40	E-Major	F#-Minor	413	16.45s	62	3
SJP-52	Eb-Minor	Eb-Minor	417	19.6s	22	3
SJP-56	A-Major	A-Major	470	37.4	58	3
SJP-65	B-Mixolydian*	Bb-Minor	441	12.3s	61	3
SJP-68	Eb-Major	Eb-Major	464	11.5s	44	2
SMP-03	B-Minor	F#-Minor	489	4.9s	47	2

SMP-16	Ab-Major	Ab-Major	493	8.0s	57	3
SMP-21	E-Major	E-Major	437	19.8s	60	4
SMP-23	Eb-Major	Eb-Major	431	21.4s	20	3
SMP-38	Bb-Major	Bb-Major	438	9.7s	44	3
SMP-44	F-Major	F-Major	439	14.0s	35	5
SMP-48	A-Major*	A-Major	405	16.7s	43	3
SMP-53	D-Major	D-Major	405	13.5s	51	2
SMP-55	B-Minor	F#-Minor	523	92.3s	49	3
SMP-63	F-Major*	F-Major	483	7.6s	39	3
SMP-72	A-Minor	C-Major	464	12.7s	62	4

Table 6: Overview of the generated results

The results listed above are located in the repository at *results/optimizer/* in MIDI format.

9.1.1 Key error rate

Some notes:

- The phrygian scale is the modal scale starting at step 3 in a major scale, meaning that its notes are shared with its Ts. While it is by all rights an independent scale, it was not considered for the program and SJP-21 correctly translates it from E-phrygian to C-Major.
- The SJP-65 mixolydian is also a modal scale which begins at step 5 of a major scale, sharing the same notes with its subdominant. In this case, B-mixolydian -> Bb-minor is not the correct translation and is therefore considered wrong, although many notes are shared between the scales.
- SMP-48 is listed online as beginning in F#-minor and ending in A-major (relative scale).
- SMP-63 is listed online as beginning in D-minor and ending in F-major (relative scale).

The key analysis is completely correct for only 22 out of 46 snippets, but there are several factors to consider that might prove mitigating to this number. Interestingly, 7 times it returns the major scale which begins on the dominant position (e.g D-Major -> A-Major) and is a modulation that Bach very commonly used. 2 times it does the same for the subdominant, which is also a regularly occurring motif, and finally 4 times to the relative scale such as in the notes above. While these pieces have not been more thoroughly analysed for their exact modulations, it is rather probable that at least some of the original works do exactly those things.

To summarise:

- Correct : 22 (including SMP-48 and SMP-63)
- Major dominant modulation: 7
- Relative scale modulation: 4 (excluding those that are known, SMP-48 and SMP-63)
- Subdominant modulation: 2
- Completely wrong: 11

If we discount the modulations, the analysis produces a 47.8% success rate. If we allow the common modulations, i.e. relative scales and the dominant major as no such considerations were implemented in the algorithm, it is 71.74% correct. Finally, if we choose to give it all the benefit of the doubt, this number increases to 76.07%. While conclusively deciding which one of these is ultimately the most correct can only be done by analysing the average tonality of the first 32 seconds in all of these pieces, it is nonetheless important to remember that the analysis was conducted on the output of the neural network, not the original work. Even through this level of abstraction, which is only as good as the neural network itself, it manages to correctly maintain the tonality well enough for it to be correctly judged nearly half of the time, and very close in some others. One could assume that some of the more grievous misses, e.g. those that are transposed a semi-note up or down and share little to no notes in common with the original, are due to the neural network not being able to differentiate between the duplicated and transposed input which would follow the same pattern, only in neighbouring steps.

As the neural network output is available in the repository, the exact accuracy can be judged by analysing them for tonality. However, this is left for future work (or the very enthusiastic reader).

9.1.2 Cost rate

These results are not optimal, so there is limited knowledge to be drawn from their value. It does provide an idea on the cost magnitudes that are necessary to transform the input into something that follows the chorale rules. A typical total function cost for a piece lies around 3-4x that of the voicing cost by design, as it is intended that as much of the original input "feel" as possible is maintained.

9.1.3 Solutions found

This is the number of solutions that were located and then improved upon during the final phase (last 16 time-steps). For some it converged so quick that one could only assume that we had merely skimmed the surface, while others had relatively long search times with few good results found. Otherwise, there is not much knowledge to be gained from this number other than knowing that it is possible to build a number of solutions to choose from by ear-test.

9.1.4 Elapsed time

Although it may appear like the algorithm is completing the searches in record time, this is not the case. The results have been generated using major shortcuts that will be explained in the next chapter. As for now, these values bear no scientific significance.

9.1.5 Rating (1-5)

The rating is exactly that, an ear-test with a corresponding score of how good/bad it sounds to someone who is fairly familiar with the style at this point. However, there is definitely the possibility of significant bias, and the rating should not be considered canon. The reader is encouraged to have a listen and make up their own judgement. As these opinions are better left for the discussion section, that is where the thesis goes next.

10 Discussion

This chapter is separated into two sections, one regarding music and the other technology. The first discusses the results that were presented in the previous chapter, and some thoughts on the level of success they achieved with regards to the intended goal. The second is focused on the implementation of the software itself. It provides a discussion about the various challenges that were encountered and what mitigating measures or compromises, if any, were used to handle them, and at what cost. Finally, some brainstorming about possible further development is presented.

10.1 Results

10.1.1 Initial thoughts

After having listened to the entire generated set, I am quite pleased with the results from a musical sense. To the extent at which quality could be expected beforehand, the optimiser seems to be performing quite well even though the results can not be claimed as numerically optimal. It was already theorised that any piece fulfilling the major rules of tonal music would by default sound "nice", and this assumption has been strengthened by these findings. It must be stated that there is not much variance between most pieces as only a subset of the total possibilities of the chorale style is implemented. If the reader is interested in listening to some of the best of the older "generations" of generated music that have been produced during the implementation of the software and compare them to the latest versions, a collection of some of the "best" throughout time can be found on soundcloud [73]. While the output is originally MIDI, these versions have been converted to .mp3 using a *soundfont* to play the music with better quality virtual instruments than the general MIDI native to windows can produce.

Before moving on to the technical aspects of the implementation, some discussion is warranted about the quality of generated output from each module.

10.1.2 DLM output

The music generated from the neural network is actually not as bad as one might have feared, as it is quite obvious that it has learnt something about tonality, progression and rhythm. However, as it is a predictive system, it is easy to recognise the thing it is trying to do, to repeat what it has been given. To make a corny analogy, it is almost as if a small child has been listening in to some piano lessons and seeks to replicate it by himself. The *movement* of the music is correct, but the wrong keys are frequently being pressed. As such, it is not something that I would listen to in my spare time. However, as input to the optimiser it functions very well. The (at times) clear tonality is of use to the key analyser, and makes function selection more differential. The worst case is if either no keys are pressed or too many are pressed in the same time, which the network has seemed to learn. In fact, it tends to only play four notes at a time even if this was not explicitly coded in the generation or learning sequence. Had it been differently, the tonal distances might have been too even to make any significant pruning operations and the search would magnify in run-time. There isn't much else to discuss, except some immediate improvements that could be made.

The limited rhythmic and tempo capabilities is a major flaw to the system. Although the network could easily have processed piano rolls sampled at a higher frequency, it would perhaps have been wiser to make more structured input from the get go. In an optimal situation, the piano rolls would be completely disjoint from the actual tempo or sampling rate, merely being automatically subdivided into the correct beats and measures. As the input would be on a format which represents the entire piece, it would be able to generate something equally accurate in a format that the optimiser can use. For each file, the tempo and time-signature could then be propagated as meta-data all the way from the input preparation stage to the optimiser where it is sorely needed. It is not unlikely that this would improve upon the generated results since of right now it is merely trying to learn from an erratic selection of the notes from the real piece.

10.1.3 Optimiser output

After a full listen on the current batch, it is clear that significant improvement is made to the neural network output with regards to the rules of the Bach-style chorale. While the former contains obvious errors and inaccuracies across all musical aspects, the optimiser output has clear structure, exclusively scale-compliant note selection and functional harmonic progression. Being at worst the greedy solution to a constraint satisfaction optimisation problem, if the system is capable of generating invalid solutions it is due to bugs and not design, as it is clearly capable of creating some that are correct.

In general, as the rating column in the result presentation table shows, I am generally happy with the quality of the music that is generated. The pieces are clearly structured, albeit overly similarly to each other. There is a diverse selection of functions and voicings being used across songs, and it is clear that different input has a significant effect on the chosen function and voicing progression. While these things are directly a consequence of the implemented rules extracted from the book, there are many other and primarily subjective guidelines that have been implemented.

In the early versions, the harmonics were relatively functional immediately after implementation. Due to the nature of the cost scheme, it is simply impossible to do something out of accordance with the rules when it comes to functional requirements in voicings. There were cases where the rules were (or are still) wrongfully implemented, but in such cases the most common side-effect is merely that those functions or voicings will not feature in a generated work. In other words, preventing illegalities was relatively trivial to achieve.

What is not trivial is the balancing of the costs related to all the official and subjective guidelines that have been implemented. Small penalties in cost here and there add up, and their internal distribution may quickly skew the output excessively towards one another. For instance, while almost every generated piece in this current batch seems to contain a clear melodic motif, this was not always the case. An un-steered soprano would leap back and forth, lie completely still for long stretches of time at the least opportune moments or wiggle between the same two notes across several measures. While such behaviour is relatively hard to notice for the middle voices, there is an entirely different expectation for the melody. After all, it is usually the case that chorales are created with a specific melody in mind, whereas this software will just choose one that fits the functional mold. As such, additional, sometimes contrasting, guidelines were implemented. There are rules which demand constant movement, but also those that penalise excessive movement within cadences. There are guidelines that apply a cost to changes in direction, but others that penalise *not* changing direction if a leap is performed. Step-wise motion is encouraged, but not to the extent to which there are no leaps at all. The melody has an unique responsibility for producing variation, excitement and to develop the structural contents. While a perfect balance of these things has not been found, it is clear that most of the pieces contain at least some notion of predictable and supportive melody. Ultimately however, the dominating factor for achieving success in this regard is what functions that are selected, and that is primarily based on the input. If the input is "flat" and contains no potential for harmonic development, the resulting harmonics will also be uneventful.

Analysis: SMP-44-chorl-opt

While this thesis is mostly technical in nature, it would be negligent to not conduct some analysis on the quality of the generated output in a musical sense. After all, that is the purpose of the software. To this end, the piece SMP-44-opt has been selected for further inspection, and is among those uploaded to soundcloud in a better audio quality than those found in the repository. Being one of my personal favourites from the batch, SMP-44 is the twice-generated adaptation of "Wer hat dich so geschlagen" from St. Matthew Passion. Looking at the result chart, it is also one of the pieces for which the key analysis algorithm managed to identify the correct scale; F-major. It does not sport an especially low cost, and is fairly average in this regard. It was fully generated within 14.0s, during which time 35 other, but cost-wise worse, candidate solutions were found.

Upon listening, while its melody opens with a slight rhythmic shift, it quickly recovers into a more structured motif within the first two measures. The shift itself gives associations to counterpoint styles, but not to the extent of breaking homophonicity. The rhythm is simple throughout, as it is bounded by the capabilities of the system. The structure is quite clear, with the first four measures being self-contained and constituting one singular phrase which ends with an intermediate cadence at the half-way point. From there on out, a predictable, yet beautiful, melodic pattern is portrayed, with varied harmonic progression which utilises many of the available tools in the book; variations upon the main major triads, mediants in minor and step-wise bass movement counterpointed by the melody.

While chord progression *sounds* valid, it must be analysed further with regards to the rules themselves. As such, table 7 shows the function progression of the piece, with one row representing two measures of 4 strong/weak beats each (4/4):

	1 S	2 W	3 S	4 W	1 S	2 W	3 S	4 W
1	T	D	D7/7	T/3	S	D7/3	T	T
3	S	D-5	S/3	T/5	S	D7	Ts	Ts
5	S	D7/7	T/3	S	T/3	D7/5	T	T
7	S65	D	Ts	S/3	D64	D	T	T

Table 7: The chord progression of SMP4-44-chorl-opt

At first glance, the structure becomes more apparent. With resting T's or Ts's after every second measure, the phrases always manage to work themselves back home. The first interesting and demanding function that is used is D7/7. D7/7 is a four-note chord which must, according to the rules, be resolved into a T/3. This happens to be the case. Also, the third of the D7 (leading note) must resolve into 1 of T, which happens in alto (MIDI: 64->65). Next, the T/3 contains a doubled 5, and allows for the bass to pivot into S. D7/3 is used in a similar fashion as the preceding D7/7, but must lead to T as is the case. The double resting T's here function as respite *within* a longer phrase, as the earlier shift in rhythmic motif somewhat distorts the sense of beat. As such, the phrase continues on into the next measure, presenting a simple S->D-5->S/3 motif with ascending bass. Here we have the one situation where D can go to an S, which is the inverted phrygian D->S/3. T/5 requires the same function class before and after, which is here S. Next is another D7 (beat 2, measure 4) whose progression into Ts is an allowed special case of the disappointing motif D->Ts. Here, the leading note must resolve into the 3 of Ts, which happens in the tenor. The phrase ends and rests on Ts across two beats in descending accentuation.

There is a noticeable shift at the fifth measure, which is natural due to the fact that the two pieces are generated separately, but bridged. It is still a requirement that the beginning of this measure has legal connection to the one that is preceding. An S becomes a D7/7 allowing the bass to remain stationary for one beat, before pivoting around for another descent through S->T/3->D7/5. This motion is somewhat special as it is completely mirrored by the soprano, which is usually encouraged to move contrarily or obliquely. However, as there is a distance relative to a parallel third between the voices during descent, it avoids any illegal parallels. The last inversion of D7, D7/5 presents itself and is considered to be the most difficult inversion to use. Its rules are the following:

- D7/5 must be placed on weak or off-beat.
- The bass (5) must move step-wise in and out of the function.
- The 1 must remain stationary throughout the function motion.
- It can therefore only be used in two ways: T->D7/5->T, or T/3->D7/7->T.

In the case of this chorale, D7/5 is indeed placed on a weak beat, the bass descends step-wise down and the root remains stationary in the tenor. It is noted that the software is able to use one of the most difficult chords perfectly.

While the repeating T's at the end of the measure may appear to mark the end of a shorter phrase, it maintains momentum by using two different voicings and consecutively prepares the introduction of one of the other more advanced functions available in the toolkit; S65. S65 has both positional, functional and voicing restrictions placed upon it, in similar fashion to D7/5. A S65 on a strong beat is usually resolved into D, while on a weak-beat must go to D64 or T/5. Additionally, the 5 and 6 need to be both introduced and resolved quite specifically. Upon introduction, the 5 needs to be prepared and the sixth must move in step-wise, oblique motion. At resolution, either of the two voices can move obliquely step-wise while the other rests. In our case, the 5 is introduced stationary from the 1 in T (alto) and resolved step-wise down into the 3 of D. The 6 arrives step-wise down from 3 in T, and stays put as the 5 in D (tenor). Once again, full marks for execution.

D becomes Ts, marking a brief shift in tonality and the climax of the piece, before passing through S/3 to form an augmented D64 which triggers a classical authentic cadence D64->D->T.

It turns out that every function-dependent rule is achieved, at least by this specific piece, and included are many of those with the most amount of rules related to them in the code. There should neither be any illegal parallels, hidden or otherwise, but this is more difficult and time-consuming to confirm. We can have a short look at the piano roll visualisation in Figure 21. There appears to be no avalanches across the voices, and bass and soprano portray general contrary motion. The spread is even, except for at the middle cadence, and there is significant overlap on notes across time-steps which indicates that shortest paths are often chosen.

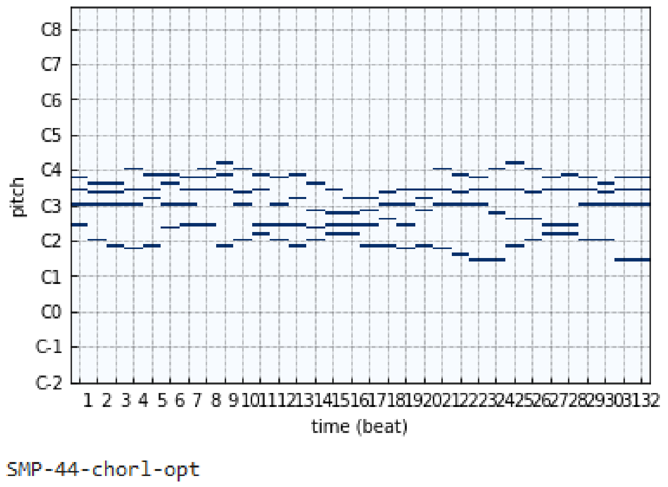


Figure 21: Piano roll representation of SMP-44-chorl-opt

All in all, it is my subjective opinion that some of the works, especially SMP-44-opt, are deemed to be of sufficient quality in both an aesthetic and craftsmanship sense, at least to the lengths of the developers limited expertise. Although it will not be a part of the submitted thesis, it is the intent to probe the minds of both my musical co-supervisor, Prof. Brandtsegg, and any others experts that may be contacted on their opinions on the matter as well. For a conclusively scientific approach, a peer-review must be conducted with a larger amount of participants.

10.2 Software

10.2.1 Initial thoughts

While having a functional prototype at hand which actually fulfils many of the aspirations that sparked the entire thesis provides a great sense of accomplishment and joy, it leaves much to be desired. There are many of the composite parts which need to be revisited, adjusted or completely re-worked if better results are to be achieved. This includes both theoretical aspects about the problem domain and how it could be approached differently, as well as technical implementations that have proven themselves limiting, of poor quality or outright faulty. As a whole, the system includes many moving parts that need to be juggled properly, and it is clear that both modules of the software have suffered from lack of full attention. Regardless whether it was due to lack of time, expertise or simply incompetency on my part, neither of the Deep Learning or optimisation modules provide any major new discoveries to either field of computer science. On the other hand, it must also be stated that if I had known beforehand the quality of results that was to be attained, I would gladly do it again. Music has been a lifelong passion of mine, and this thesis has provided the opportunity to explore both my professional and personal interests at the same time.

At any rate, it has been an incredibly educational journey, and much of the newly acquired knowledge about implementing larger systems, designing software from the bottom up from just an idea and the process of transforming some abstract notion of how a problem with no obvious answer could be solved into actual code that performs to a certain level. A major part, if not the majority, of the effort was spent simply theorising about algorithmic design and data representation, which undoubtedly has led to more insight into how one should approach such problems.

About half-way through the process, after trying and failing to squeeze the problem into an Integer Programming-shaped hole, there was a realisation that simply wanting something to work does not make it so, and presented a dilemma where some cold-hard decisions were required. One of them was to continue in the current path, attempt once again to rephrase or reformulate the problem, search for more tools that could help or read more books on the subject in the hopes of attaining a higher level of understanding. The other was to admit defeat, admit that enough was enough and look for some other option that could possibly attain some success within the relatively short, remaining time until deadline. As somewhat of a perfectionist by heart, this was a tough decision to make, but I do strongly believe that I made the right one. I suppose that is part of the job, really, to realise when you are not making enough progress in the path you are travelling, and make a change in direction even if it means abandoning a large amount of work and effort. It did certainly not induce any sort of pride at the time, but it was a necessary pill to swallow and has proven relatively fruitful in the end.

With the mixed bag that is success and failure, we move on to discussing the two module implementations.

10.2.2 The not-so-Deep Learning Module

It is disappointing that the DLM ended up being such a small focus of the thesis, but the limited time available forced more attention onto the optimiser to make it functional. Within the boundaries of this thesis and the results, the output from the DLM acts as only little more than some glorified and time-consuming random number generator. As such, there is not much to discuss about it other than what the few "tricks-of-the-trade" that were implemented could have effected.

Some *test data manipulation* was used. With only a small training set available, it was duplicated and shifted in such a way that each original object was represented by 13 distinct versions providing a better foundation for *generalised learning* and combats *overfitting*. Likewise, overtraining was avoided when it became apparent that the loss was no longer decreasing, and the first, best weight combination was used to produce the end result. Additionally, a *dropout*-layer was added to induce further variance and a *ReLU* was selected over a simple *sigmoid* to help prevent *vanishing gradients*. Without these, the output would have been at significant risk of overfitting, which it does not appear to overly do.

Teacher forcing was the default method of training, and used constantly during training. For possible improvements, one of the first things to explore would be to make the teacher forcing dynamic, such that the network gets to "practice" on its own output. While this could improve the variance of generation, it is uncertain whether or not this is a good thing when the goal is to imitate a specific style. For generalised music however, this could have been a cheap way to introduce some "creative freedom".

The model did not use any *validation* set, as the training set was deemed too small to split and not be used entirely for training. One possibility could have been to choose only one of the duplicates for each piece, but this was not attempted. Regardless, it is unlikely that a validation set would have helped much when the loss graph shown in the previous chapter was as it was. As the curve descends quickly to its minimum and then flattens with no better result appearing for 1/3 of the total number of epochs, there was little fear of overfitting as long as the first and lowest weight set was used for generation.

Beyond this, there is little to mention about the DLM in its current state. While more time could have been spent tweaking and adjusting some of the optimiser or loss function parameters, it is likely that it would prove more fruitful to immediately pursue a more advanced model, such as the *Anticipation-RNN* that was analysed in the research project.

10.2.3 The Approximiser

At first the prospects of using *Integer Programming* had to be dropped, then no ideas of how to implement the system using *Simulated Annealing* became apparent either. After much dismay, focus shifted towards a safe and seemingly simple option; Branch-and-bound search. Unfortunately, it turns out that this was not as straight forward as one would have hoped. There are some big gaps in the implementation that need to be filled, somehow, as it does not work fully as intended at the moment. There are also several shortcomings of the design itself that could require a full rework to improve.

There are some positives however, as the initial analysis seems to be of acceptable quality when considering its input. The various data representations created during initialisation and used during execution are working as intended, and seem effective for the purpose. Having functions and voicings as independent entities, yet expressively combining them to form candidate sequences allows for all the information to be processed separately, yet transferred in between the various methods together. The rules appear to be working as no illegalities have been discovered yet, and the costs are balanced in such a way that the output has both a nice harmonic structure and inter-voice flow. The use of hash structures (dicts) to contain function and voicing meta-data allows for easy access on demand. The tonal map proved to have many applications, and it does appear like the custom distance algorithms implemented work to some extent, although they can not be said to be conclusively correct. There is also some hope for the rhythm map as a concept, as any such implementations as were mentioned earlier about having a universal piano roll representation independent of sampling rate would be work well in conjunction with such a generalised container for rhythmic elements.

In short, while not entirely a success, these things together managed to secure at least valid output, which was a secondary objective after all. As for not finding optimal solutions, there are several reasons.

The heuristic problem

A big factor to the effectiveness of Branch-and-bound is how tight the optimistic bounds can be calculated. This implies needing some way to identify a good heuristic on how good a solution could possibly be in the future. Since the problem is multi-faceted, with several dependent-yet-independent cost-factors in the functions, voicings and distance costs from original, good estimators for each must be located for them to be usable as part of the total cost calculation.

As for the function estimator, some headway was made by implementing a shortest-path-to-end look-up table which was built ahead of the main execution. The algorithm consisted of constructing a

$$num\_t * (num\_functions^2)$$

3-D matrix where each cell would contain two-time-step dependent costs until end. It can be pictured as a graph that looks like Figure 22, where each node represents the chosen previous->current function combination. To the right is the local cost of each function at the time-step, and every edge signifies the calculated shortest path. Beginning from the bottom, the total-cost-until-end is calculated fully per time-step before continuing on to the next. Instead of having to calculate all the way down at every branch, the function cost method which is already implemented would be used to identify the cost increase from one time-step to the next. As such, the costs from each locally calculated combination would be summed with the already calculated shortest path from the lower cell. Since only nodes which have identical last and first function (for upper and lower depths, respectively) need to be calculated, any unreachable (illegal) nodes would be marked with cost > 1000. If the problem had only consisted of the function space costs, this would have been a complete and fast solution to the optimisation problem. Unfortunately, this is not enough alone when additional cost terms are added to the equation, although it does help somewhat with the pruning process.

As an addendum, the reader should be aware that there are some numerical errors in the actual simulation of the graph, as it was originally created for own use during implementation. Since the algorithm no longer is a part of the implementation, it was not deemed necessary to make it any more presentable than it already is.

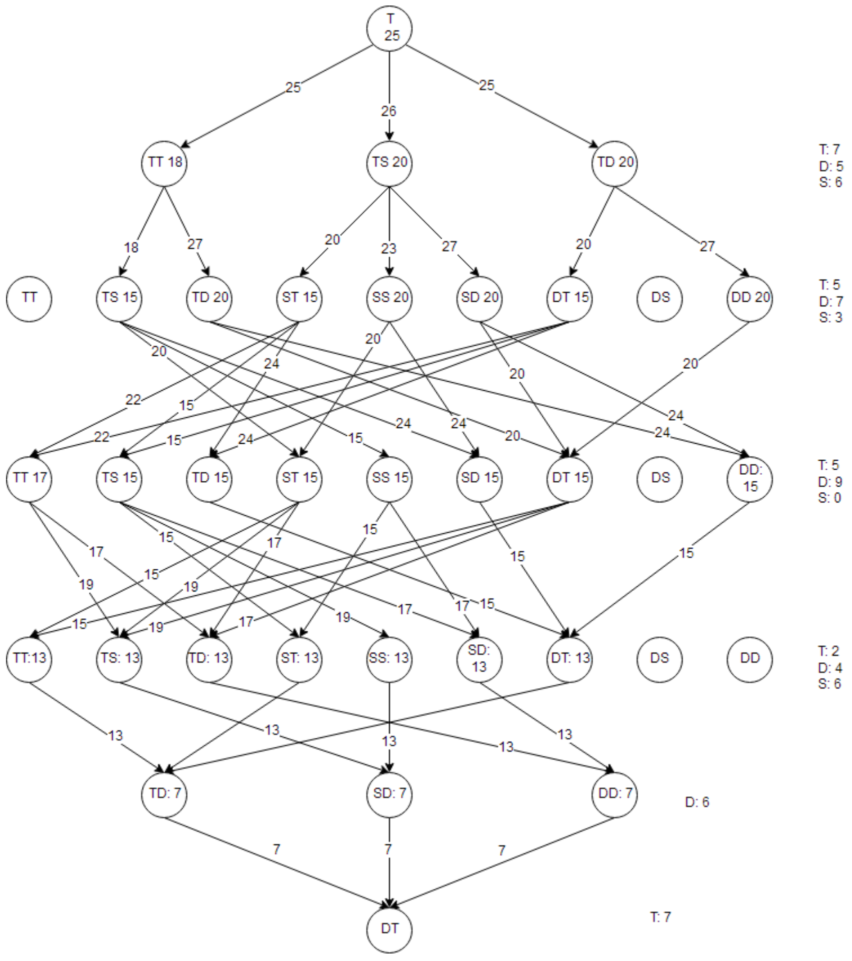


Figure 22: A graph representing construction of a function heuristic for lowest-cost-paths until end.

As an alert reader would perhaps already have pieced together, this table was later converted into the full memoisation scheme which is intended to hold the total cost at any point instead of just the function cost. While it would have been equally suitable to identify independent voicing and absolute-distance heuristics, they are much trickier to figure out. To consider the first, voicings are beholden to a large set of abstract rules that are not easy to represent in a numerical sense. While a note of a specific value might be perfectly viable in one combination, it is completely the opposite in another. It is easy to calculate in a singular point in time when both sides of the gap is presented, but costs vary from time-step to time-step and chord to chord to such a degree that it has been incredibly difficult for this thesis to make any headway on the issue. Likewise, voicings are equally dependent on the functions they belong to, as they would not exist otherwise. There might exist some angle to utilise this fact in some way, but none has come to mind yet. If anything, it has been considered to split up the search into two parts, where one just builds a list of the cheapest possible legal function combinations such as in the heuristic above, and then does a relatively much more contained search on the voicings alone. When the functions are already set in place, the search space is reduced from $\tilde{\ }400^t$ to $\tilde{\ }15^t$, depending on the functions chosen. Although it would have to be iteratively repeated, it would still fare better in pure complexity if no improvements can be made to the main implementation.

The last term that would need to be estimated for a complete implementation is a distance metric from the original to the suggested voicing at every time-step. One could choose to estimate based upon the current position in the note-space, provided some statistical analysis which tells us typically how far away it is possible to get within some amount of time-steps in a chorale. However, in music, stagnation is generally a bad thing, and there is significant effort being made by human composers to ensure contrary movement between soprano and and bass, producing a wide array of variant distances. Simply stating that anything that moves outside a given range is bad would portray poor musical sense. There is of-course a limit to how far a set of voices can go, but given that the opportunistic estimate needs to be exactly that; *opportunistic,* there is no obvious limit on how *little* it can go. If cost-term is lower bounded by 0, it does not make any mathematical difference to include it.

In the end, as music is a dynamic thing, dynamic estimates are required. While some attempts have been made at identifying possible solutions, little success has been achieved on this front. This effectively means that the Branch-and-bound search is left with poor worst-case complexity and can not be expected to optimise within reasonable time, even if all other parts of the module are working.

Changes to the Branch-and-bound search

There are some ways to "cheat" around this, of course, if one removes the requirement for an optimal solution. As an "approximiser", it can function better with some greedier bounds. For instance, the results generated and presented here are not the optimal results of fully completed Branch-and-bound searches which was the intent, since that might very well take decades with the current implementation. The method by which these specific results are made, includes a static and *pessimistic* lower bound which is as tight to the average best solution that can be produced within reasonable time. The side-effects of this are non-optimal solutions as the search would prune possible improvements, but the alternative is significantly worse.

Given a non-terminating search, everything from the current depth at the point of timed cancellation back to the beginning will have its greediest form as it was never revisited after first being built. Every partial solution in the active set would comprise of the same, identical permutation up to this point, excluding its last step. This results in a "Dr. Jekyll and Mr. Hyde" situation, where half of the piece is the most naive greedy solution and the rest is fairly optimised. In a regular use of the algorithm with no mitigation measure, it would be told to terminate after 500,000 iterations (roughly 20 minutes) and provide a solution which never was processed beyond the middle of the total depth at time-step 8-10 out of 16. At such a depth, working its way back to the end is a 400^{6to8} operation before any pruning, which regardless is a too large number to make any progress in. Examples of such a situation can be heard between the first and second generation examples in those uploaded to soundcloud, as both beginnings are identical due to never having been revisited. Additionally, those specifically were also generated with some inaccuracies in the implementation such that a proper run through the algorithm would have shifted this point of divergence further up the time-steps.

It was then decided that it was more desirable to achieve something consistently average, than two, extremely disjoint sections that have little apparent reason to be connected. While the greedier approach still suffers somewhat from this problem as the end is more mapped out than the start, and there is significant risk of poor combinations in the beginning, it is easier to accept an initial strange manoeuvre than a longer sequence of it. If one chooses to be positive, at least it can be considered as "starting with a bang".

Though a static and high lower bound does speed up the process, it logically has a very different effect on different snippets. Some snippets are solvable with lower average costs than others, much dependent on how well the neural network did in formulating proper chords in the first place. For those that can do much better than the average, they are still terminated at non-optimal solutions as soon as the limit is struck, while those that do significantly worse proceed as if there was no lower bound at all. To combat this, an extremely greedy strategy was used to force quick, valid results within only seconds. The problem formulation was basically re-phrased from "finding the optimal combination from beginning to end", to "finding the optimal combination from the end to the start of the greedy solution". After the total depth is hit and the greedy solution is constructed, an array keeps track of the "best-total-cost-per-time-step" in reverse. Whenever the algorithm works its way through all the children of a given depth, the table is updated to hold the best cost located from new depth to end. In effect, the new candidate solutions are lower bounded by the tail-part of the best solution found, which risks being much higher than the real lower bound of the optimal solution.

In a scientific sense, this removes all premise of optimality. As the algorithm now just operates on the chance that there exists some improvement within the children that have already been generated up to the greedy solution, nearly not even a fraction of the total search space is explored. It becomes the greediest version of itself that it could possibly be, sans just accepting the first solution immediately. It must be clearly stated that it is not the intent of the thesis to present this as a viable strategy, merely a means to an end to produce solutions that have *some* improvement over the greediest solution. Having already explained about the current alternative, this is considered a lesser evil. One perhaps mitigating factor, if such a thing can be claimed, is that it performs better within seconds than the original did within 10-20 minutes and returns a lower cost solution almost immediately than the original was able to locate in this time. It is assumed that this is both due to the fact that quite a lot of the "best" partial candidate solutions are initialised on the way down which will all be explored before the algorithm terminates. It does not do anything for the voicing or distance costs, so it is safe to assume that there exist lower total cost solutions than those seen here.

In short, until progress can be made on developing proper memoisation, heuristic scheme or some other strategies present themselves, this acts as a placeholder method to produce output that is representative of the chorale rules.

Other shortcomings

Among these major issues, there are some smaller ones that are present but are plausibly fixable. While it is believed that the data representations themselves are sound for most tonal matters, the implementation of rhythmic elements leaves much to be wanted. As was briefly mentioned in the DLM section of this chapter, the input probably needs to contain more information than it does at the time. Alternatively, the program must be altered to contain a comprehensive analysis which would allow more dynamic structural elements to be added to the generation scheme. At this point, only a time signature of 4/4 (or 4/4 with subdivision) is supported and all generated pieces follow the same format with similar placements of cadences and endings. While there is some variation in the generated output, it has limited expressive range. As such, the rhythm function needs to be extended or re-worked.

The custom made tonal analysis would probably also allow for more advanced methodology. At the moment, it naively ignores modulations, absolute distance and overly biases in favour of the preset rhythmical map. Combining this with the fact that there are still many more rules yet to be implemented, such as those regarding change of tonal centre, additional functions and non-chord tones (to bridge functions), it is safe to say that almost every part of the module needs revisiting. Lastly, whenever major parts are to be re-implemented, it should be done with a comprehensive testing scheme in mind. Since this was not prioritised enough, the current system is cumbersome to validate and may contain many bugs.

11 Conclusion

As the last chapter of the thesis, we will summarise what has been discovered by the project as a whole and attempt to provide some answers to the main research questions that were proposed in the introduction.

11.1 Research questions and goals

What are the explicit rules of a Bach chorale?

Although too numerous to be listed here, the rules of the Bach chorale style were in full harvested from Bekkevold [7] and all those that made it into the implemented system are listed in section 8.4.3. For the most part, this is an exhaustive list of all the rules from the first four out of five chapters of this book, only excepting the section on *nonharmonic notes* in chapter 2. It must be considered that those featured here are subjective interpretations of said rules, as not all of them are conclusively phrased or supplied in absolute terms. Certainly, another source material might possibly present a different formulation to some of them.

How can the various elements of musical information best be represented as digital data?

Having explored various data structures and representations, it was decided to make use of a piano roll matrix representation throughout most of the pipeline. A piano roll by itself is able to capture the melodic and harmonic aspects of music quite well, and if structured correctly can also embody rhythmical elements also. While this was not as well achieved in this thesis as one could have hoped, it has certainly presented a foundation on which further improvements can be made. In general, as with most problems, the best suited representation is closely related to the operations and calculations that is to be performed on them. For numerical calculation, matrices and vectors perform well. Hashed structures such as dictionaries proved very useful to easily represent relational information, as for instance the possible successors to a given function, the meta-data of a function itself or retaining information on different voicing permutations. To make a blanket statement, the thesis has found no efficient way of representing the many elements collectively, and most have required their own, specifically tailored, data structure.

There is some merit to the use of such an "archaic" system as tonal functions, as it gives a simple way to classify scale dependencies disjoint from actual tones and notes. This is also the main reasoning behind using intervals or scale steps as representation instead of absolute chord names. As a means to relay musical harmonic and melodic information digitally, such systems are second to none.

Although rhythmic elements were poorly propagated, it is likely that if the input format is sufficiently structured, rhythm as a whole can be treated as horizontal, discrete vectors, e.g. across columns in a matrix or payloads in a stream. All (western) music relies on the division and subdivision of a tempo indicator, which means that all that is required is the correct subdivision structure which can be scaled to the desired tempo. Musical structure and progression is oftentimes embedded in the rhythmical framework, such that the former simplifies the implementation of the latter.

Little headway or effort was made to discover means of representing dynamics, texture or the remaining elements of music, as they were considered secondary to harmony, melody, rhythm and form.

What optimisation strategies are suitable for this problem? Is it solvable within reasonable time?

Due to the gargantuan magnitude of the search space complexity, $num\_voicings^t$, which grows exponentially with respect to the number of time-steps, a good optimisation strategy and algorithm is required for it to be of any hope to be solvable within reasonable time. The thesis explored three possible candidates, Integer Programming, Simulated Annealing and Branch-and-bound search, where the latter was chosen for implementation. A small summary is in order for each, along with some thoughts on suitability.

Integer Programming

Albeit perhaps the least successful aspect of the thesis, several optimisation strategies have been considered or attempted. It seems natural to formulate music generation as a constraint satisfaction or optimisation problem, as music is an art of rules which only need to be clearly identified for replication or inspiration. Strictly mathematical optimisation strategies like Integer Programming which depend upon a set of algebraic equations are extremely efficient at what they do best, but the major challenge with regards to the music generation problem is the expression of said rules in algebraic terms. While elements of music are generally discrete and in theory lend themselves well to such methods, they are also riddled with non-linear factors that are difficult to express generally. Additionally, it is difficult to implement probabilistic models over constraints in an IP, as music optimisation at times consists of choosing between numerous but equally valid choices. As such, no significant progress was made for developing a system utilising IP optimisation, but if a proper formulation could be made it should perform no worse than what is typical for a given solver.

Simulated Annealing

Simulated Annealing resides on the other side of the spectrum as a relatively "free" and unconstrained search. By using statistical probability analysis on mutations of already confirmed candidate solutions, SA can locate global minima or maxima by avoiding getting stuck locally. While there is no guarantee of optimality when using SA, it should be able to find relatively good solutions if the optimisation objective function has a relatively smooth curve across the search space domain. It is neither required to extensively formulate as comprehensive of a problem definition as it is for mathematical optimisation. One necessity to using it, however, is some way to define a mutating function which generates neighbouring candidate solutions from a given candidate solution. It is also required that this mutating function has the ability to reach the entire search domain within a finite amount of mutations.

For our problem of locating optimal solutions to a Bach chorale "sudoku" problem, all these things are indeed possible to implement. If one considers the entire search space of every possible permutation (even if invalid) as the global search space, it is easy to define a simple mutating function which only modifies one function+voicing combination at a time for one time-step. However, it is highly likely that such an implementation would produce a much too uneven landscape for an efficient hill-climber search to be used, or at least without significant chance of getting stuck in some local optimum regardless of temperature adjustment. As for the case of generating search space which only consists of actually valid permutations, where one *could* assume that good partial solutions have other good partial solutions within a close range of mutation, identifying all such candidates are half of the problem.

Finally, the multi-layered objective function with dependencies both within the search space (all permutations) and outside of it (to the original input) adds additional noise to any such landscape which might further diminish the effectiveness of the strategy. As a whole, it is still worth to further explore an implementation and see how it works in practice, but no such attempts were made during this thesis. Until such an endeavour has taken place, it is difficult to be conclusive about the search time complexity.

Branch-and-bound

Lastly, the thesis has attempted to implement a Branch-and-bound search with heuristic estimators to solve the problem. Branch-and-bound is known for being applicable to almost any problem with a defined search space, but only performs as well as its ability to prune and avoid significant portions of the tree. At worst, it performs no better than a brute force algorithm, but at best it can outperform many other specialised algorithms. For our problem, the efficiency of Branch-and-bound relies solely on the ability to estimate an opportunistic lower bound from which the algorithm is able to decide whether or not a partial candidate is worth exploring. Although such an estimator was created for one of the cost terms (functions) no such methods were discovered for voicing costs or distance costs, which means that no tight bounds could be produced for the total cost. As any function can contain a number of voicings, each possibly distributed in a completely different manner across the available range of notes, the cost from $F_1V_1- > F_iV_j$ might be worlds apart from the cost of $F_1V_2- > F_iV_j$. As such, without sufficiently tight estimates, the search performs poorly and is unable to terminate within reasonable time.

In a bid to mitigate this issue, the implementation of a memoisation strategy in conjunction with the search was started. By using a data structure capable of storing the four-way dependant costs of $time - step, V_{t-2}, V_{t-1}, V_t$ for explored nodes, it was in the hopes of the thesis that this would provide a sufficient reduction in complexity as a whole. However, the memoisation has not been properly implemented by the time of deadline and must be further improved to be functional.

As a means to produce some results to present, an approach was used where only a small subset of the total search space consisting of only the immediately apparent best function combinations was explored for additional improvements upon the initial greedy solution. This means that the results that have been generated here do not represent what globally optimal solutions would eventually look like, but serves to portray their adherence to the various rules of the Bach chorale style.

The run-time of the proper implementation of Branch-and-bound is unknown, and unreasonable, but it has at least been shown that valid solutions of some quality can be generated within the span of seconds.

How is a neural network model designed, implemented, trained and used?

The Deep Learning module is written as a jupyter notebook [53] file, using 3.7 [49] as programming language and pyTorch [48] as framework. The RNN is designed and implemented using GRUs and the piano roll .csv files as input and output format. The contents of each piano roll matrix is of binary nature, so a BCE-with-logits loss-function is used in conjuction with ReLU activation to reduce the frequency of vanishing gradients. An ADAM optimiser is used with a learning rate of $5e - 04$. The implementation includes various best practice strategies for avoiding the most common problems of Deep Learning systems, such as the retaining of best loss weights, randomised processing, data set manipulation and magnification and teacher forcing. To mitigate the amount of time required for training it has been implemented with CUDA support. The training itself is conducted over 2000 epochs on a training set of size 598, and allows for the saving and loading of model and optimiser parameters for continued training or immediate generation.

Generation consists of feeding the network with the first n time-steps from one of the training data piano rolls and building upon the networks own predictions from that point on. A shifting "window of attention" is used to supply *some* of the previous sequence for continued prediction instead of the entire sequence, as the goal of the generation is not to replicate, but innovate.

How should the chosen optimisation strategy be implemented to achieve desired results?

Although partially answered by the previous research questions, the implementation itself is done in Julia [52] for performance. It has been written procedurally in a jupyter notebook file, and contains three stages of operation. First, the input data is imported and analysed by a key and scale analysis scheme. Next, initialisation of all necessary data structures is performed, the first being the calculation of tonal distances from each input time-step to pre-defined valid functions decided by the identified key and scale. Following that, every conceivable voicing permutation for each function is generated and stored. A memoisation look-up table is initialised, A simple data structure is created for keeping track of the rhythmic contents of the time signature and all Bach chorale style rules are defined in the form of a chord-cost function and a voicing cost function.

The search itself is prepared by populating the first time-step with one or more voicing permutations of the tonic function which achieves the lowest calculated distance with regards to the first input step. The rhythmic map is populated with cadences in suitable locations. A candidate sequence *struct* is specifically designed to hold and propagate any vital information across time-steps. The search is two phased, the first finding a better-than-greedy solution to the first 16 time-steps, while the second phase continues the sequence from the 16th to the 32nd. The main search function contains two nested loops. The first iterates over all possible functions for the given time-step. The second inner loop generates all voicing permutation children of the chosen function if it survives the first pruning step, and again prunes for the total function + voicing cost. At the end of each outer loop iteration, the surviving fully generated children for a given function are sorted and pushed to the node LIFO set in such a way that the lowest total cost child will be popped first. The search is Depth First, Best First and quickly generates a greedy solution from which the upper bounds are recalculated. From here on out, the algorithm depends on good heuristics to provide efficiency.

After both sequences have been generated, they are fused together and stored as piano roll .csv's, ready for .MIDI transformation.

How well does the generated output, if any, adhere to the requirements of the Bach chorale?

It is the opinion of this thesis that it performs very well. A thorough musical and structural analysis was conducted on a selected example from the generated set, and found that most, if not all, rules were duly followed. Both functions and voicings appear to be in not only legal, but also structurally sound order. While purely subjective, it would not be profound to claim that some of the generated pieces sound at least on par with at least amateur level work within the style.

Goals

As all of the goals were met during the lifespan of the thesis, no further comments are required that have not already been stated elsewhere.

11.2 Future work

The software designed and implemented by this thesis is not comprehensive enough to be considered final, and there is severe need for improvements across all modules and parts. It will however serve as a functional prototype of the possibilities for utilising external optimisation in conjunction with neural networks for music generation by computer. As with all prototypes, there must be defined some clear and concise avenues to explore for future improvements. While many such opportunities have already been mentioned throughout the text, this section will summarise and mention those that the thesis deems the most important and/or promising.

Deep Learning module

The number of improvements that can be made to the neural network itself is so numerous that it is hardly any point in mentioning them all. In its very basic state, there is little that can not be done to further its capabilities. A more advanced model needs to be developed such that better raw material can be produced. This might for one include the addition of several neural units working in conjunction, such as portrayed by Hadjeres et al. [4]. Alternatively, significant work must be made to improve upon the training gains as the current loss quickly plateaus with no further gain over time. Whether this is due to learning rate, training set limitations or poor choice of optimisation or loss-function is unknown, but all should probably be revisited and reconsidered.

Preparation of raw input

Also, the input preparation needs adjustment to more accurately portray the music it seeks to generate from. As of now, a flat sampling rate of 1 time-step per second neither captures more than a fraction of most raw data and does nothing to aid accurate learning or properly describes the rhythmical elements of music. Even though the neural network itself is capable of making sense of piano rolls sampled at a higher rate, this does not translate well into generation or provide any such boundaries for the optimiser to work with.

The thesis proposes that a less greedy approach is used for gathering the data from the original files, as these do contain meta-data on the tempo and possibly time signature of the piece. If some analysis can be made to interpret these, a specifically tailored piano roll format could be generated which is sampled at a dynamic rate which coincides with the rhythmical structure of the song. This way, every piano roll could be handled similarly by the neural network, yet there is more information being passed along down to the optimiser to work with. Such an implementation would significantly reduce the loss of information, drastically improve the quality of the neural net output and by extension the optimiser and ensure a much closer resemblance to the original work in general.

There is also always the possibility of working directly with scores and the actual musical notation, albeit this introduces a whole lot of new factors to consider and many work hours used to attain and transcribe them into digital format.

Distance metric for the optimiser module

When the distance cost metric was being designed, little to no source material on the matter was located at the time. However, towards the end of the lifetime of the thesis, some options were discovered.

A small portion of the book *Encyclopedia of Distances* by Deza et al. [74] is available online and mentions the *Estrada distance*, or intervalic distance between chord c and c' as

$$\sum_{i=1}^{6} |c_i - c_i'| \; and \; max \; |c|, |c'| - |V(c) \cap V(c')| - 1$$

where the interval vector $V(c) = (c_1, ..., c_6)$ depicts the number of times the i'th interval within an octave occurs in c. In simpler terms, it proposes using the internal intervalic structure of a chord to decide how different it is from another. While this is only briefly mentioned in passing, further research into the Estrada distance could prove fruitful for a more accurate representation of note distance. It is also worth noting that this metric implies the use of purely tonal distances, which is already supported and mapped out by the module during run-time.

In a similar vein, Cambouropoulos [75] proposes a representation for chord transitions which utilises a 12-dimensional vector to portray the intervalic distance between all pairs of notes between two successive chords. Each chord is represented by its intervalic structure, ranging from 0 (root) to 6 (tritone) along with a directional sign for all intervals except the extreme points. The distance is equal to the calculated interval distance from each note in the first chord to all of the second. It is called a Directional Interval Class (DIC) vector, which he claims is specifically useful for computer modeling tasks due to being, quoted; "...easy to compute, independent of root finding, independent of key finding, incorporates voice leading qualities, preserves chord transition asymmetry, transposition invariant, independent of chord type, idiom-independent, chords can be uniquely derived from vector (except for symmetric chords such as the diminished seventh). If all of these claims hold, this would be a natural place to begin searching for any improvements to the harmonic representation."

There are probably additional avenues to explore as well, such as possible applications of *earth mover's distance*, but each must be judged with regards to the specific purpose in mind.

Regardless of chosen metric, it is a requirement for using the Branch-and-bound method that there are ways of estimating distances into future time-steps. This premise does not change regardless of chosen distance metric.

Optimisation module

First of all, as there are still more rules and functions yet to be implemented, the first course of action would be to formulate and implement the remaining chapter of the source material book. *Modulations* and *nonharmonic notes* are integral parts of the chorale style and need to be added for it to be called comprehensive. However, in the current implementation, nonharmonic notes are especially hard to define as the whole system revolves around pre-set and pre-defined chord permutations. For the fastest implementation without needing to re-work large parts of the pipeline, accidentals and nonharmonic notes could be added *post-search* if a sufficiently accurate analysis method was made. They would then be used to add embellishments or known-to-be legal transitions between two chords that might appear in the best candidate solution. Modulation would also need to be tied in with the harmonic analysis, perhaps splitting it up into smaller sections and defining a key and scale for segments of the snippet instead of in its entirety. These would need to be properly aligned with the rhythmical structure of a piece, since modulations need to be prepared and introduced over some time (often as part of cadences) to not detract from the quality of the piece.

The memoisation method needs to be thoroughly examined and attempted before any rash decisions are made about dropping Branch-and-bound entirely, but it would be worth the time to explore other opportunities. The SMT-solvers that were mentioned could be valid alternatives, as they allow for both defining a suitable theory with regards to the diatonic system along with dynamic constraint enforcement to approximate or even locate optimal solutions to constraint satisfaction problems. It might even be worth to return to Simulated Annealing and give it a go to see if it works, but there are probably heaps of possible strategies and algorithms unknown to this thesis which could be feasible to use given the right definitions and formulations.

Further output improvements

After working with (and therefore listening to) MIDI format music for a long stretch of time, it is mentioned that the results would do well to be converted into some other format with a richer and more immersive sound. For the average listener one could assume that the quality of the audio might be just as important for reception as the actual contents as long as it is not obviously bad. If the score for the generated SMP-44 had been transcribed and performed by a real choir, it is quite possible that it would pass in many walks of society as music as composed by a human (but perhaps not quite at Bach level). There exists software and softsynths that allow for a transformation from regular midi into digital and real audio simulated by *virtual instruments*, some of which are frequently used by real musicians, artists and composers.

Potential use

While researching the possibilities for music generation using computers is interesting by itself, a couple of ideas have emerged along the way about what such a system could be of use for. One prominent option is to continue to develop a system which is able to generate more styles of music in a similar fashion. As chorales are relatively obscure and caters to a very specific audience, there are many industries that rely on a steady stream of new original music, especially in the entertainment industry as a whole. Film, television, advertisement and large event organisers oftentimes have use for music which is not necessarily the most advanced compositions known to man. Even a generative software which could produce tens, hundreds or thousands of hours of "elevator music" by the click of a button could have some use to someone.

Another possible goal for continued development is to either create or merge the system with a user-friendly and interactive tool for music composition. Budding musicians, hobbyists or established artists alike often experience less productive stretches of time, and sources of inspiration and motivation may be far and in between. Imagine a musical notation software with playback capabilities, which in addition to being a work-space for musical creativity provides a means to automatically create, extend, harmonise or re-arrange existing ideas into something new, or to a specific style. With access to well trained neural network weights and an efficient optimised optimisation module, it is not unlikely that such transformations could be done on-demand and fast. It is rarely a requirement for such use that long stretches of music are processed at a time, and even the current algorithm will be able to handle the completion of a measure or the immediate harmonisation of a melody in reasonable time.

The research project theorised an idea about using the optimiser, if efficient enough, in a circular fashion in conjunction with the neural network. In many situations, a large enough representative training set could be difficult to come by, whereas the optimiser could theoretically run on random input alone and still produce valid chorales. If no initial training set is available, one could be generated and fed into the neural network which again would produce new output to optimise. As the neural network initially makes mistakes, variance is created which the optimiser again validates. This effectively creates a scenario where the neural network uses the optimiser as its target function, and the loss is calculated based upon distance from the optimised piece. There is perhaps a possibility that, given enough time, the neural network will manage to learn the rules that seem to be afflicting its output and generate fully valid chorales by itself.

Whether or not these things are realistically achievable or just naive pipe dreams is uncertain.

What is certain is that no progress is made without experimenting on new, and perhaps somewhat strange, ideas.

Peer review

Finally, for the generated music to be conclusively deemed as valid Bach-style chorales, a peer review must take place such that experts in the field can conduct the necessary analysis required. Although, as mentioned previously, it is the intent to get some post-delivery opinions from some professors in music who specialise within the field, it will not be conducted in a scientific manner. There is some grounds for mentioning a full survey with "regular" people, but it is to be said that any findings would not be conclusive when considering the specific target style. While many might agree or disagree whether or not it "sounds like Bach", it can not be expected that they would be aware of the structural, melodic and harmonic requirements.

Bibliography

[1] Kaspersen, S. K. 2018. Music composition and generation using deep learning methodology.

[2] Youtube: Lejaren hiller - illiac suite for string quartet. URL: https://www.youtube.com/watch?v=n0njBFLQSk8&list=PL0pv-xCXkg1k0666mFfBcE1pFGgmlRdtW&index=1.

[3] Briot, J.-P. & Pachet, F. 2017. Music generation by deep learning - challenges and directions.

[4] Gaetan Hadjeres, F. P. & Nielsen, F. 2017. Deepbach: a steerable model for bach chorales generation.

[5] Youtube: Deepbach: harmonization in the style of bach generated using deep learning. URL: https://www.youtube.com/watch?v=QiBM7-5hA6o.

[6] Vimeo: Deepbach: a steerable model for bach chorales generation. URL: https://vimeo.com/240608371.

[7] Bekkevold, L. 2005. *Harmonilære og Harmonisk Analyse*. Aschehoug.

[8] Doornbusch, P. 2017. Early computer experiments in australia, england and the usa. *MUSA 2017 Conference*.

[9] The university of melbourne: Reconstruction of the music played by csirac. URL: https://web.archive.org/web/20190107145457/https://cis.unimelb.edu.au/about/csirac/music/reconstruction.html.

[10] Wikipedia: Music-n. URL: https://web.archive.org/web/20190107145632/https://en.wikipedia.org/wiki/MUSIC-N.

[11] Csound.com. URL: https://csound.com/.

[12] Wikipedia: Computer music. URL: https://web.archive.org/web/20190107145837/https://en.wikipedia.org/wiki/Computer_music.

[13] Barbrick, G. 2012. Book review: Keyboard presents synth gods.

[14] Holmes, T. 1983. *Electronic and Experimental Music*. Routledge.

[15] Buxton, W. A. S. 1977. A composer's introduction to computer music.

[16] Örjan Sandred, Mikael Laurson, M. K. Revisiting the illiac suite - a rule based approach to stochastic processes. URL: https://web.archive.org/web/20190107150013/http://www.sandred.com/texts/Revisiting_the_Illiac_Suite.pdf.

[17] Hoffman, P. 2001. *Xenakis, Iannis*.

[18] Keller, D. & Ferneyhough, B. 2004. Analysis by modeling: Xenakis's st/10-1 080262. *Journal of New Music Research*, 33(2), 161–171.

[19] Youtube: Iannis xenakis - st/10=1,080262. URL: https://www.youtube.com/watch?v=9XZjCy18qrA.

[20] Koenig, G. M. 1983. Aesthetic integration of computer-composed scores. *Computer Music Journal*, 7(4), 27–32.

[21] Wikipedia: Serialism. URL: https://web.archive.org/web/20190107150045/https://en.wikipedia.org/wiki/Serialism.

[22] koenigproject.nl: Project 1. URL: http://www.koenigproject.nl/indexe.htm.

[23] koenigproject.nl. Programmed music. URL: https://web.archive.org/web/20190107150109/http://www.koenigproject.nl/Programmed_Music.pdf.

[24] Conklin, D. 2003. Music generation from statistical models .

[25] McCormack, J. 1996. Grammar based music composition. *Complex systems*, 96, 321–336.

[26] Chomsky, N. 1956. Three models for the description of language.

[27] David cope: Experiments in musical intelligence. URL: https://web.archive.org/web/20190107150753/http://artsites.ucsc.edu/faculty/cope/experiments.htm.

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

[29] Youtube: David cope emmy vivaldi. URL: https://www.youtube.com/watch?v=2kuY3BrmTfQ.

[30] Youtube: David cope bach style chorale. URL: https://www.youtube.com/watch?v=PczDLl92vlc.

[31] The guardian: Interview with david cope. URL: https://web.archive.org/web/20190107150826/https://www.theguardian.com/technology/2010/jul/11/david-cope-computer-composer.

[32] Youtube: Emily howell from darkness 2 beg, cope. URL: https://www.youtube.com/watch?v=mnBUxG-wSVg.

[33] Turing, A. 1950. Computing machinery and intelligence. *Mind*, 59, 433–460.

[34] Goldberg, D. E. & Holland, J. H. 1988. Genetic algorithms and machine learning. *Machine Learning 3*, 2, 95–99.

[35] Horner, A. & Goldberg, D. E. 1991. Genetic algorithms and computer-assisted music composition.

[36] Genetic algorithms tutorial. URL: https://web.archive.org/web/20190107150553/http://www2.econ.iastate.edu/tesfatsi/holland.gaintro.htm.

[37] Biles, J. A. 1994. Genjam: A genetic algorithm for generating jazz solos.

[38] Genjam website. URL: https://igm.rit.edu/~jabics/GenJam.html.

[39] Youtube: Genjam's journey: From tech to music: Al biles at tedxbinghamtonuniversity. URL: https://www.youtube.com/watch?v=rFBhwQUZGxg.

[40] Chen, C.-C. J. & Miikkulainen, R. 2001. Creating melodies with evolving recurrent neural networks.

[41] Huang, A. & Wu, R. 2016. Deep learning for music.

[42] Hochreiter, S. & Schmidhuber, J. 1997. Long short-term memory.

[43] Goodfellow, I., Bengio, Y., & Courville, A. 2016. *Deep Learning*. MIT Press, http://www.deeplearningbook.org.

[44] colah's blog: Understanding lstm networks. URL: https://web.archive.org/web/20190107150920/http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[45] Hale, J. Towards data science: Deep learning framework power scores 2018. URL: https://web.archive.org/web/20181227080902/https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a?gi=602a60911e0e.

[46] tensorflow.org. URL: https://www.tensorflow.org/.

[47] keras.io. URL: https://keras.io/.

[48] pytorch.org. URL: https://pytorch.org/.

[49] python language. URL: https://www.python.org/.

[50] Nasa modeling guru: Basic comparison of python, julia, matlab, idl and java 2018. URL: https://modelingguru.nasa.gov/docs/DOC-2676.

[51] Oracle: Java programming language. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html.

[52] The julia programming language. URL: https://julialang.org/.

[53] jupyter. URL: https://jupyter.org/.

[54] Øyvind Risa. 1995. *Musikkteori og arrangering: innføring for lærerstudenter*. TANO. URL: https://www.nb.no/items/f3d35a2526ddeae917b980add4a0066f?page=5&searchText=musikkteori.

[55] Music Rudiments Sounds. Youtube: Minor 2nd harmonic interval. URL: https://www.youtube.com/watch?v=sMRkaXaeB_0.

[56] Tsang, E. 1993. *Foundations of Constraint Satisfaction*. Academic Press.

[57] Wolsey, L. A. 1998. *Integer Programming*. Wiley-Interscience.

[58] Mitchell, J. E. 1999. Branch-and-cut algorithms for combinatorial optimization problems. URL: http://eaton.math.rpi.edu/faculty/Mitchell/papers/bc_hao.pdf.

[59] S. Kirkpatrick, C. D. Gelatt, M. P. V. 1983. Optimization by simulated annealing. *Science*, 220, 671–680.

[60] Wikipedia: Simulated annealing#pseudocode. URL: https://en.wikipedia.org/wiki/Simulated_annealing#Pseudocode.

[61] M. Jünger, T. Liebling, D. N. G. N. W. P. G. R. G. R. L. W. 2010. *50 Years of Integer Programming 1958-2008*. Springer.

[62] Leonardo De Moura, N. B. Satisfiability modulo theories: Introduction and applications. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.367.9961&rep=rep1&type=pdf.

[63] Github: Z3 repository. URL: https://github.com/Z3Prover/z3.

[64] Wikipedia: Midi. URL: https://en.wikipedia.org/wiki/MIDI.

[65] Wavenet: A generative model for raw audio. URL: https://deepmind.com/blog/wavenet-generative-model-raw-audio/.

[66] Openai: Musenet. URL: https://openai.com/blog/musenet/.

[67] Lmms documentation: Piano roll editor. URL: https://lmms.io/documentation/Piano_Roll_Editor.

[68] Github: Master thesis repository. URL: https://github.com/nsthtz/Master.

[69] learnchoralmusic.co.uk: John's midi file choral music site. URL: http://www.learnchoralmusic.co.uk/.

[70] github: pypianoroll. URL: https://salu133445.github.io/pypianoroll/getting_started.html.

[71] github: neural-composer-assignement. URL: https://github.com/zehsilva/neural-composer-assignement.

[72] Rohrmeier, M. A. & Cross, I. 2008. Statistical properties of tonal harmony in bach ' s chorales.

[73] Soundcloud: Computer generated bach-style chorales using deep learning and combinatorial optimisation. URL: https://soundcloud.com/nsthtz/sets/computer-generated-bach-style-chorales-using-deep-learning-and-combinatorial-optimisation.

[74] Michel Marie Deza, E. D. 2009. *Encyclopedia of Distances*. Springer, https://books.google.no/books?id=q_7FBAAAQBAJ&pg=PA408&lpg=PA408&dq=estrada+distance+music&source=bl&ots=QZ9TcCXRAw&sig=ACfU3U0OxWClA_19cmyIAjecgIegp1XWiw&hl=en&sa=X&ved=2ahUKEwiJkf6-kLrhAhUSw8QBHb9lAfQQ6AEwDHoECAgQAQ#v=onepage&q=estrada%20distance%20music&f=false.

[75] Cambouropoulos, E. 2012. A directional interval class representation of chord transitions.

Simen Kværneng Kaspersen

Chorales by Deep Learning and Combinatorial Optimisation