FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

David Samuel

Artificial Composition of Multi-Instrumental Polyphonic Music

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2018

First of all, I would like to thank Martin Pilát for being my supervisor. He kept me motivated throughout the three semesters with his endless positivity, and his responses to my questions were always quick. His suggestions helped me to choose the right direction of both the research and the writing process. Still any mistakes remaining are my own.

I would also like to thank my roommate Tomáš Nekvinda for patient corrections of many versions of this thesis. Since he is an experienced web developer, he helped me to run an online survey, an essential part of this work. His constant supply of fried toasts has been a great help, too.

Speaking of the survey, I really appreciate everyone who sacrificed the time and decided to take part. The number of respondents was a pleasant surprise.

I owe a lot to all my other friends for their support and interest, and also for healthy distractions from work. Finally, I would like to thank my family for great support and understanding.

Title: Artificial Composition of Multi-Instrumental Polyphonic Music

Author: David Samuel

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: We propose a generative model for artificial composition of both classical and popular music with the goal of producing music as well as humans do. The problem is that music is based on a highly sophisticated hierarchical structure and it is hard to measure its quality automatically. Contrary to other's work, we try to generate a symbolic representation of music with multiple different instruments playing simultaneously to cover a broader musical space. We train three modules based on LSTM networks to generate the music; a lot of effort is put into reducing high complexity of multi-instrumental music representation by a thorough musical analysis. Our work serves mainly as a proof-of-concept for music composition. We believe that the proposed preprocessing techniques and symbolic representation constitute a useful resource for future research in this field.

Keywords: music composition, music analysis, deep learning, LSTM, artificial creativity

Název práce: Generování polyfonní hudby o více nástrojích

Autor: David Samuel

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Představujeme generativní model pro skládání klasické i populární hudby, jehož cílem je vytvářet hudbu na lidské úrovni. Hlavní překážkou je její složitá hierarchická struktura a absence rozumného automatického vyhodnocení její kvality. Na rozdíl od ostatních prací na podobné téma se snažíme generovat symbolickou reprezentaci hudby o více nástrojích hrajících současně, abychom pokryli širší hudební spektrum. Pro samotné skládání využíváme tři moduly založené na LSTM neuronových sítích; velké úsilí je vynaloženo na zjednodušení vstupní hudební reprezentace důkladnou analýzou dostupných dat. Naše práce slouží především jako ukázka toho, že současné technologie umožňují skládání hudby. Věříme, že námi navržený hudební analyzátor a generátor poslouží jako základ pro další výzkum v této oblasti.

Klíčová slova: skládání hudby, analýza hudby, hluboké učení, LSTM

Contents

Chapter 1

Introduction

The field of artificial intelligence continually competes with humans in a variety of tasks. Computer programs have already beaten human in chess [Campbell et al., 2002], face recognition [Lu and Tang, 2014] or Go [Silver et al., 2016]. One field where humans still excel are tasks involving *creativity.* There are many attempts to synthesize paintings [Elgammal et al., 2017] or write poems [Liu et al., 2018], but it seems that computers would need knowledge about the whole world to create *art* unrecognizable from humans. One thoroughly studied subfield of *artificial creativity* is the automated music composition. Philosophers sometimes interpret music as a direct expression of pure human emotions and the most sensual of all the arts [Sontag, 2009].

> *We humans are a musical species no less than a linguistic one. This takes many different forms. All of us (with very few exceptions) can perceive music, perceive tones, timbre, pitch intervals, melodic contours, harmony, and (perhaps most elementally) rhythm. We integrate all of these and "construct" music in our minds using many different parts of the brain. And to this largely unconscious structural appreciation of music is added an often intense and profound emotional reaction to music. "The inexpressible depth of music," Schopenhauer wrote, "so easy to understand and yet so inexplicable, is due to the fact that it reproduces all the emotions of our innermost being, but entirely without reality and remote from its pain... Music expresses only the quintessence of life and of its events, never these themselves.*
>
> Oliver Sacks, Musicophilia: Tales of Music and the Brain (2007)

We believe that music composition presents one of the most significant challenges of artificial intelligence. Not only because of a seeming connection with human intelligence, but more due to its high structural complexity (which seems similar to natural languages, but is not as thoroughly studied) and the inability to automatically evaluate the quality of results.

Unlike the studies mentioned in Chapter 3, we propose a generative model that can create music simultaneously played by multiple instruments in a wide range of genres (not only classical music, but also popular music genres). This feature dramatically increases the complexity the model should comprehend. We, therefore, focus on reducing this complexity by simplifying the available music data without limiting their expressiveness too much.

Our model consists of three parts: a chord, note and volume predictor, where each of them is based on an LSTM neural network described in Chapter 2. The chord predictor is used to layout the harmonic structure, and then the note predictor generates melodies based on it. After the notes are generated, we use the volume predictor to give them a volume expression.

The primary goal of this thesis is to create a generative model able to compose multi-instrumental music on the same level as humans do. We are aware that this goal is quite ambitious, but we hope that our ideas will at least help to reach the ultimate goal faster.

1.1 Contributions

As far as we know, our work is the first that managed to deal with multi-instrumental composition of music using deep learning techniques. Our proposed music representation and neural network architecture for this problem can serve as a basis for further research in this field.

Chapter 5 presents novel approaches to meter detection, key detection and chord detection in a symbolic representation of music. We believe that our enhancements can improve the performance of general music analyzers. Moreover, preprocessing a musical dataset by our algorithms should easily improve efficiency of generative models proposed by others. Researchers in the field can therefore focus more on developing the models than on analyzing music as we did.

1.2 Outline

The thesis is laid out in the following way: First, we describe some definitions and methods used throughout the thesis in Chapter 2. We explain basics of the music theory, of the MIDI format and of machine learning – particularly of (recurrent) neural networks and LSTMs. After establishing the background, we discuss approaches to music composition by other researches in Chapter 3. Then we briefly discuss the data used to train our model in Chapter 4. This is naturally followed by Chapter 5 devoted to the process of analyzing and simplifying the music pieces in the dataset. Although this process is essential for our work, this chapter can be skipped if the reader is interested only in the generative model itself. Then we describe the architecture of the generative model and its representation of the input and output in Chapter 6. An online questionnaire about the quality of the output is outlined in the following Chapter 7. Finally, we discuss the results in Chapter 8.

Chapter 2

Background

2.1 Music Theory

Since music is the central theme of this thesis, we provide a brief introduction to music theory so that the implementation and motivation of the further algorithms can be better understood. Of course, we cannot provide a comprehensive explanation of music, as the music theory is very complicated and there are many different views on the hidden structure of music; refer to Kostka and Payne [2008] for a complete description. Please note that we are explaining the theory from a view of a computer scientist so some terms may be defined ambiguously or slightly different than in the traditional music theory.

Music can be most basically defined as a sequence of *sounds* organized in time. These *sounds* can be described by their **timbre** (color), **pitch** (frequency), **dynamics** (loudness) or by their **rhythmic function**. Although these characteristics are not complete nor required (for example *sounds* of percussion and drums do not usually have any pitch), we will assume they are, and we will consider music as a sequence of **notes** defined by these four characteristics – their function will be further described in the following sections.

2.1.1 Timbre

Timbre (or tone color or tone quality) is the characteristic that distinguishes two different instruments from each other (in a way that they are perceived distinctively even when played with the same pitch and volume).

Defining full, measurable and finite description of timbre is still an open problem of psychoacoustics, so it is usually represented by a set of incomplete and nonorthogonal attributes such as noisiness, periodicity, spectral envelope, time envelope (attack, sustain, release strength) or by changes of these attributes during the time. A survey of more than 50 audio descriptors can be found in Peeters et al. [2011],

Since we cannot sufficiently describe timbre by mathematical means, it is still impossible for computers to synthesize a musical score in the same quality as experienced human performer would – for example, a violinist can intentionally affect the timbre by changing the bowing speed, tilt or pressure, by playing at different parts of a string and using different fingerings or by playing vibrato or glissando to transmit the intended mood of an interpreted piece. The complexity of the timbral space presents a considerable problem for the evaluation of our algorithm, because we create only a symbolic representation that has to be automatically synthesized into an audio signal, so that a non-expert could rate it – we cover this problem in the chapter about MIDI files and in the chapter about evaluation by an online survey.

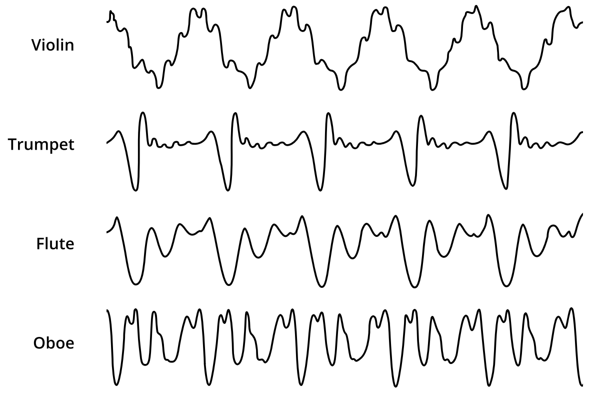

Figure 2.1 *Figure showing sound waves of four different instruments playing the same tone. Reproduced from Howard and Angus [2010]*

2.1.2 Pitch, Melody, and Harmony

Pitch is the perceived *height* of a note that is based mostly on its fundamental frequency. Although frequency is a continuous unit, the pitch is usually defined as a categorical variable.

The absolute value of a pitch frequency is not as relevant as its ratio to the surrounding notes' frequencies. Ratios of frequencies that can be simplified into small integers are perceived as more consonant. The most harmonious ratios are the **octave** $(2 : 1)$, the **perfect fifth** $(3 : 2)$ and the **perfect fourth** $(4 : 3)$.

Western music usually uses a *twelve-tone equal temperament* tuning system which divides frequencies into categorical pitches in a way that each octave contains 12 pitches, and the ratio of two nearest pitches (a **semitone**) is constant. The frequencies of pitches in equal temperament can be computed by the following formula, where P_a is a reference pitch (usually denoted as A_4 with the frequency of 440 Hz).[1]

$$P_n = \sqrt[12]{2}^{(n-a)} P_a$$

When notating a pitch P_n, we use the **reference pitch** P_a (notated as A_4) and assign a symbol from $(\ldots A_3$, $A\sharp_3$, B_3, C_4, $C\sharp_4$, D_4, $D\sharp_4$, E_4, F_4, $F\sharp_4$, G_4, $G\sharp_4$, A_4,

[1]Note that the most harmonious pitch ratios cannot be expressed exactly in the equal temperament, but their approximation is so close that the dissonance is almost inaudible. For example the perfect fifth is approximated by $\sqrt[12]{2^7} = 1.498307 \approx 3/2$ and the perfect fourth by $\sqrt[12]{2^5} = 1.334840 \approx 4/3$.

$A\sharp_4 \ldots$) according to its distance $(n-a)$ from P_a. A group of notes with the same latter and a different subscript is called **pitch-class** and since all pitches from the same pitch-class differ only by several octaves, they are treated as almost equivalent.

A sequence of pitches is called **melody**. The melodies usually contain pitches from sets of pitch-classes called **keys.** The ratios of key's pitch-classes to its central **tonic** pitch-class should convey a certain mood. For example the C major scale (C, D, E, F, G, A, B) is often described as sounding happy and strong, whereas pitch-classes from the C minor scale $(C, D, D\sharp, F, G, G\sharp, A\sharp)$ imply a more melancholic mood.

Chord is a combination of at least three (sometimes also two) simultaneously sounding pitches from different pitch-classes. Chords are divided into classes based on the ratios between their pitches, two most commonly used classes are the *major* class (consisting of a root pitch-class and pitch classes 4 and 7 semitones higher) and the *minor* class (root with pitch classes 3 and 7 semitones higher).[2] Other classes (voicings) usually exchange one pitch-class with another or add an additional pitch-class to create chords of various degree of dissonance.

In practice, a chord is not always indicated by all its pitches being played at once, but it may be indirectly implied by the melody line, the order of notes or the context of other chords.

The chord (harmonic) structure of a piece is an essential characteristic of Western music; chords may be viewed as unstable short-term keys that build tension and guide the melody by differing from the actual key. The tension is released when harmony returns to the tonic chord of the real key.

2.1.3 Volume

Volume is an important characteristic that adds expression to music by changing loudness of notes. Although it is often not notated in a musical score explicitly and left to the performer's interpretation, music without changes in volume sounds dull and boring.

2.1.4 Rhythm and Time

As far as timing is concerned, the most critical time characteristic of an individual note is its **onset** (start) time and then its **duration**.

Rhythm is the global structure of notes (and silences) in time. The rhythm should be regular and recurring so that a listener can develop anticipation of the next note's onset time; breaking the regularity usually leads to an unpleasant sound.

Meter tracks regular **beat pulses**, which can be informally described as the moments in time in which people would tap their foot when listening to a song.

[2]Note that the major/minor chords use the first, third and fifth pitch-class (called triad) from a key with the same name. The fifth pitch-class corresponds to the ratio of the perfect fifth $(3:2)$ and the third pitch class to the major/minor third $(5:4/6:5)$, which is why their sound is very consonant.

The meter is usually emphasized by playing percussion and accented notes, and by changing chords precisely at the beat pulses. Duration of a note is often notated as a fraction of beats it lasts (for example an eight note lasts one half of a beat, and a whole note lasts four beats).

Tempo is the frequency at which the beat pulses occur, which is usually measured in *beats per minute* (bpm). Slight changes of the tempo are often used by classical music performers to add expression to certain parts of a musical piece; on the other hand, the majority of popular music does not contain any tempo changes at all.

A widespread rhythmic phenomenon is **syncopation**, which *"is a disturbance or interruption of the regular flow of rhythm. It's the placement of rhythmic stresses or accents where they wouldn't normally occur."*[Hoffman, 1997]

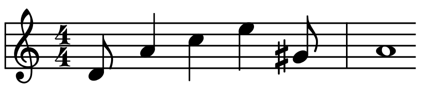

Figure 2.2 *Off-beat syncopation in 4/4 time. [Hyacinth, 2013]*

2.1.5 Dynamics

Dynamic changes of the four characteristics above are fundamental to the music composition. At the lowest level, the weak and strong beats are continually changing their accents, the melody lines are ascending or descending and their pitches flow from consonant to more dissonant. On a higher level, the average volume is changed, the chords build up or release different amounts of tension and different melodic lines alternate between each other. On the highest level, we can see cyclic changes of verses and choruses in popular music or even of whole movements in classical suites.

A successful artificial music composition model should understand this complex hierarchical structure, where events spanning several minutes depend on events occurring for a fraction of a second and vice versa. It should also balance between sounding too dull by following all the rules of music theory and sounding too chaotic by breaking them.

2.2 MIDI

Musical Instrument Digital Interface (MIDI) is a communication protocol for musical instruments, computers, and other audio hardware. The protocol also defines a file format (Standard MIDI File) that we use as both the input and output of our algorithms. Unlike audio file formats (MP3, WAV or FLAC), MIDI file does not store explicit sound waves, but a more abstract description of the music. MIDI is mostly a stream of *event messages* that characterize notes to be played, their timbral attributes and also metadata about the musical piece. We will describe the basics of this protocol according to The Complete MIDI 1.0 Detailed Specification [1996].

Each note starts with a *note-on* message and ends with a *note-off* message. Each of these events has to be specified with a time (in internal units), pitch (128 tones in the range from C_{-1} to G_9 in the standard twelve-tone equal temperament tuning), channel and volume. Notes on the 10th channel are mapped to percussive instruments (so their "pitches" specify different percussive sounds and not an actual pitch). All other 15 channels are assigned with a musical instrument from a set of 128 instruments. The sound of a note can be further modified by messages specifying attributes such as attack strength, sustain, panning or pitch bending (which enables us to produce a sound of arbitrary frequency and change the frequency over time).

Metadata messages can, for example, specify tempo, key signature or text information about the composer and interpreter.

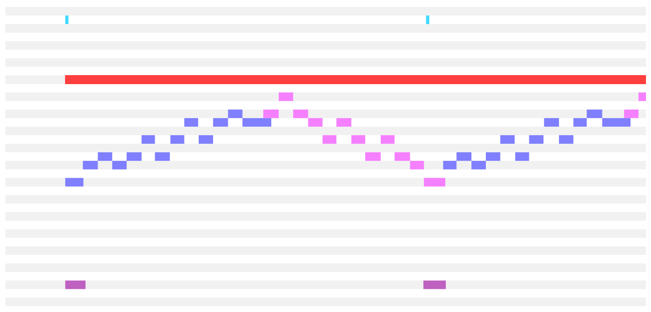

Figure 2.3 *MIDI files can be visualized by a so-called piano roll; this figure shows the beginning of the main theme of the Moldau by Bedřich Smetana, each colour represents a different musical instrument.*

As this representation is much more compact than a direct encoding of sound waves, MIDI files used to be very popular for music storage, and the protocol is still widespread in electronic music production.

A considerable disadvantage of this approach is that the sound of a MIDI file is ambiguous and depends mainly on a synthesizer that renders the file; the representation by event messages is rather elementary and cannot capture all aspects of tone colour. The insufficiency is especially valid in the case of the human voice, because the specification does not define any instrument capable of singing lyrics or even sounding close to the human voice.

The previous paragraph implies that every MIDI file sounds inevitably *synthetic* in a way. As the main question of this thesis is whether we can automatically generate music indistinguishable from human-composed music, it is important to remember that we have restricted the full "music space" to just the synthesized MIDI files. On the other hand, MIDI files can be interpreted as a kind of musical scores, so the question should be instead interpreted as whether we can automatically *compose* music indistinguishable from human-composed music.

2.3 Machine Learning

The traditional approach to solving problems in computing is to formulate a step-by-step algorithm and translate it into a computer program. Although this approach is highly effective and transparent (we know exactly what the computer is doing and why), there is a significant class of problems where finding a step-by-step solution is impossible. These problems involve tasks such as image recognition, natural language understanding, spam detection, medical diagnosis or customer segmentation – often tasks people are good at but are unsolvable by classical computing methods.

One possible way to overcome such problems is to obtain a set of examples containing inputs and desired outputs (i.e., a set of images and descriptions of their content for an image recognition task). Machine learning methods then try to understand the underlying structure of that examples and estimate the desired behavior. A traditional formal definition of machine learning by Mitchell [1997] is as follows:

> *A computer program is said to* learn *from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

This definition contains all three necessary parts of a machine learning program. Task T is usually either **classification** (assign a correct label out of a finite set of labels to a given input) or **regression** (assign a real number to a given input). Experience E can be either **supervised** (we provide annotated examples with desired outcomes) or **unsupervised** (we only have a dataset without any annotation; it is mostly used for clustering). A performance metric P of a solution for a task T measures how close is the solution to the desired behavior (for example how many images are classified correctly).

Traditional machine learning models need a set of **features** (feature vector) that explicitly describe input characteristics. The ideal features should be correlated with the output and independent on each other. The feature vector should be comprehensive on one side, and short on the other side to avoid the so-called *curse of dimensionality* (the *complexity* of a model grows exponentially with the number of features). Thus designing the right set of features usually requires expert knowledge and is one of the most significant obstacles to the traditional machine learning methods described in this chapter.

One can easily design a highly complex model that fits all the examples provided for training perfectly. However, when we show this model examples it has never seen, it will likely fail. We say that the model is **overfitting** on the training distribution and fails to generalize. To detect overfitting, we randomly divide the provided examples into a **training set** and a **test set**. We use the training set to develop the model, and only after the training is finished, we evaluate the model on the test set. A model able to generalize well should produce similar results on both of these sets. A general approach to avoid overfitting is to reduce the *complexity* of the trained model or use a larger and more diverse dataset.

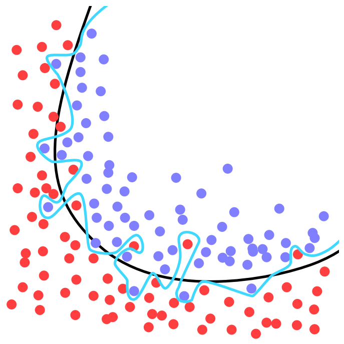

Figure 2.4 *Classification of red and blue dots in a two-dimensional feature space; the green line is overfitted on the training data, whereas the black line probably estimates the separation well even for unseen data. [Chabacano, 2008]*

In the following sections, we will briefly introduce several common machine learning models that we use in next chapters. Since we utilize them mostly as black boxes, their descriptions lack any implementation details. Another family of machine learning models – neural networks – are more thoroughly discussed in the next chapter.

2.3.1 Decision Tree Learning

A **decision tree** is a rooted tree, where internal nodes represent decisions and leaf nodes represent outcomes. The decision tree learning algorithm automatically builds such a tree based on a training dataset. When the leaf nodes are labeled by discrete values, we get a classification decision tree; when labeled by continuous values, we get a regression decision tree. An example of a simple decision tree is depicted in Figure 2.5.

2.3.2 K-Nearest Neighbor Algorithm

The k-nearest neighbor (k-NN) algorithm is an instance-based learning method, which is a family of learning methods that do not explicitly estimate the target distribution, but instead "lazily" compares an input instance with instances from the training set. The k-NN classifies an instance by searching for the k closest examples in the feature set. Then, the instance is classified by the majority vote of the nearest neighbors. The algorithm can also be used for regression – then the instance is assigned a value by averaging values of the nearest neighbors. When $k = 1$, the process reduces to an assignment of just the nearest neighbor's value.

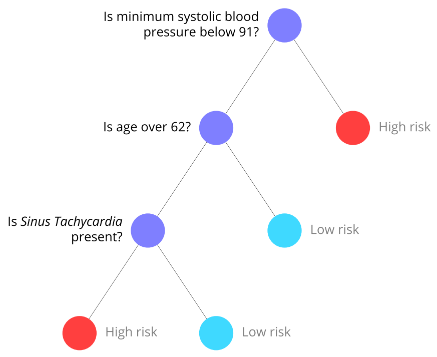
Is minimum systolic blood pressure below 91?
High risk
Is age over 62?
Low risk
Is *Sinus Tachycardia* present?
High risk
Low risk

Figure 2.5 *Diagram of a decision tree for risk of a heart attack reproduced from [Breiman et al., 1984]*

Apart from k, another essential hyperparameter of this algorithm is the distance metric. The Euclidean and Manhattan distances or the cosine similarity are used most often.

$$\begin{aligned} euclidean(x, y) &= \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \\ manhattan(x, y) &= \sum_{i=1}^{n}|x_i - y_i| \\ cosine(x, y) &= \frac{x \cdot y}{\|x\| \, \|y\|} \end{aligned}$$

2.3.3 Support Vector Machines

Support Vector Machines (SVMs) are supervised learning classifiers. The main idea of the SVM is to find a hyperplane that linearly separates the classes in the feature space. When it is impossible to find such hyperplane, the features are expanded into a higher number of dimensions and some small error is allowed. The target hyperplane not only has to separate the feature space linearly but also has to have the largest margin between the target classes – so it is most likely to generalize outside the training set.

The original linear SVM allowed only a linear separation. Boser et al. [1992] suggested a generalized algorithm that can perform an effective nonlinear classification by using a *kernel function*.

After learning parameters $(\alpha_1, ..., \alpha_n, \beta)$, the label y of an input instance x is calculated as $y = sgn(\sum_{i=1}^{n} \alpha_i y_i k(x_i, x) + \beta)$, where $k(\_, \_)$ is the kernel function and (x_i, y_i) are training examples. The most commonly used kernel functions are:

$$\begin{aligned} k_{linear}(a, b) &= a \cdot b \\ k_{polynomial}(a, b) &= (\gamma a \cdot b + \delta)^{\epsilon} \\ k_{radial}(a, b) &= e^{(-\gamma\|a-b\|^2)} \end{aligned}$$

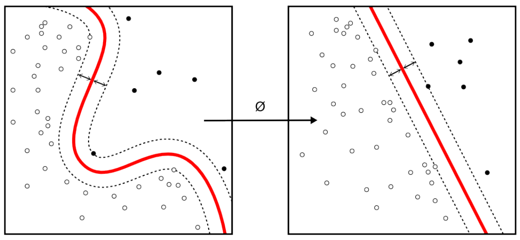

Figure 2.6 *Illustration of a transformation by a kernel function. [Alisneaky, 2011]*

2.3.4 Ensemble Learning

Instead of training just one model, we could train many of them and then combine their hypotheses. An **ensemble** of learners is a finite set of different machine learning models whose output is usually a (weighted) majority vote (for classification) or an average (for regression). When the models (learners) in such ensemble make different errors, the combination of their predictions is much more accurate and general.

The performance of an ensemble is inversely proportional to the correlation of its learners. To find diverse learners, we usually train decision trees and use their seeming disadvantage – their high sensitivity to any change of a training set. There are two main methods of training models on different datasets: bagging and boosting.

Bagging (bootstrap aggregating) methods train each learner independently on a slightly different subset of the training set T. The subset is created by repeatedly drawing a random sample from the training set $|T|$ times (with repetition), which is called **bootstrapping**. About 63.2 % of examples in T are selected in each bootstrapped subset.[3] **Random forests** by Breiman [2001] use such technique to learn an ensemble of decision trees. To make the decision trees even more various, they choose a random subset of features before constructing each tree.

Boosting methods do not train each learner independently, but they give a weight to each training sample so that each learner focuses more on the samples that the previous learners predicted incorrectly. **AdaBoost** (adaptive boosting) by Freund and Schapire [1997] is a typical method that uses boosting.

2.4 Neural Networks

Artificial neural networks are the primary mechanism used in our algorithms for the music composition. This chapter follows up on the previous chapter since neural networks constitute a subfield of machine learning. First, we will briefly describe the basics of artificial neural networks, and then we will explain the specific neural architectures used in our algorithms. The descriptions will be mostly based on Deep Learning by Goodfellow et al. [2016].

[3] We draw a specific sample with probability of $1/n$. After n draws, the probability that we do not pick that sample is $(1 - 1/n)^n$, which tends to $e^{-1} \doteq 0.368$.

2.4.1 Motivation

The main limitation of the standard machine learning methods is their inability to extract relevant features from raw data automatically. This limitation can be circumvented by a careful design of specific features by experts in the particular area of interest. But only up to a point, since designing such features is usually very difficult. Neural networks try to learn a hierarchy of features, where each level in the hierarchy depends on lower and simpler levels of features. When provided with a sufficient amount of data, the artificial neural networks can usually outperform expert systems based on handcrafted features.

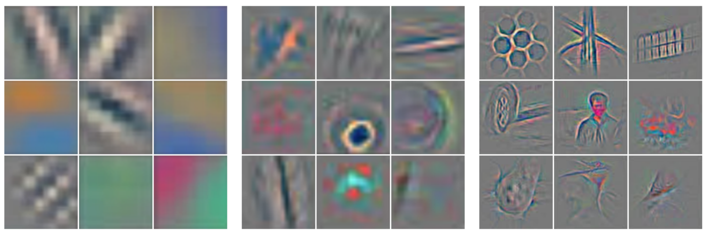

Figure 2.7 *Figure illustrating features detected at three different layers of a convolutional neural network for image classification. The first layer detects simple edges and gradients from raw pixel data. Next layers detect more and more abstract representations until the network can correctly guess what object is in the picture. Reproduced from Zeiler and Fergus [2013].*

2.4.2 Feedforward Neural Networks

Similarly to other machine learning models, the goal of a neural network is to approximate some function f^* by a function f_w defined by learned parameters w.

Feedforward neural networks represent f_w by chaining together simpler functions $f^{(i)}$ so that the ith function depends only on the output of the $(i-1)$th function. The approximation can be thus expressed as $f_w(x) = f^{(n)}(\dots(f^{(1)}(x))\dots)$. Function $f^{(1)}$ is called the **input layer**, the last function $f^{(n)}$ is the **output layer**; the other layers are called **hidden layers**. Number of layers n gives the **depth** of the network.

Each layer in the network is a vector function, where each output is computed independently by a **neuron**. We call neural networks *"neural"* because they are loosely inspired by how actual biological neurons work. Each neuron computes a linear combination of its inputs and puts the result into an **activation function** ϕ – similarly to real neurons that fire a signal according to the strength of incoming signals from other neurons. Formally, the output of a neuron is given by $f_j^{(i)}(x) = \phi(w^T x + w_0)$, where w are **weights** that need to be learned. The following figure gives an illustration of how a feedforward neural network works.

The most often used activation functions ϕ are the rectified linear unit (ReLu) $\phi_{ReLu}(x) = max(0, x)$, the hyperbolic tangent $\phi_{tanh}(x) = tanh(x)$ and the sigmoid function $\phi_\sigma(x) = \frac{e^x}{e^x+1}$.

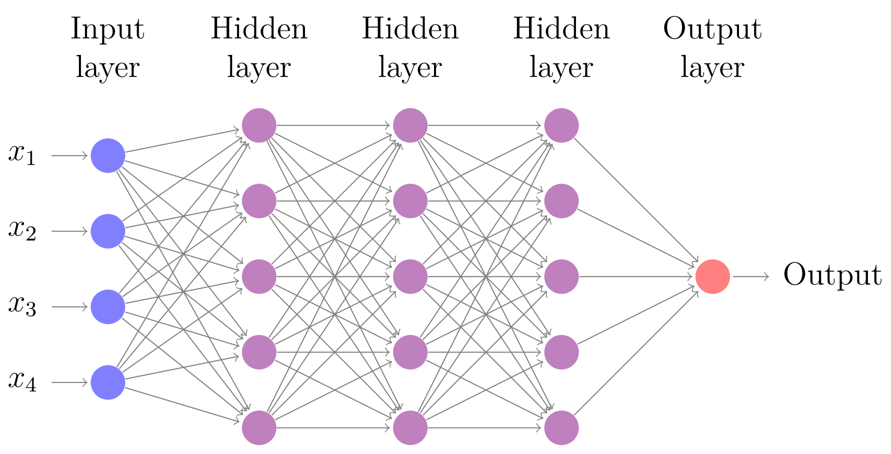

Figure 2.8 *Illustration of a simple feedforward neural network with three hidden layers of 5 neurons each. Every arrow symbolizes different weight value w_i.*

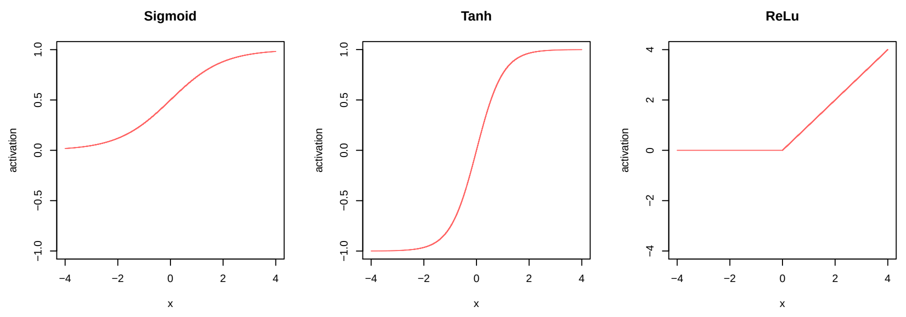

Figure 2.9 *Plots of three different activation functions. The left plot shows the sigmoid function, the middle plot shows the hyperbolic tangent and the last plot shows the rectified linear unit.*

To fully describe the neural network architecture, a **loss function** L – metric stating how close does f approximate the ideal function f^* – has to be defined. Since f^* is unknown, we have to approximate the loss with samples from training data. A commonly used loss function is the mean squared error for regression and the cross-entropy loss for (binary) classification:

$$\begin{aligned} L_{MSE}(y^*, y) &= \frac{1}{n}\sum_{i=1}^{n}(y_i^* - y_i)^2 \\ L_{CE}(y^*, y) &= -y^* log(y) - (1 - y^*) log(1 - y) \end{aligned}$$

The goal of obtaining the most accurate approximation of f^* can be then interpreted as an optimization problem: we would like to find such weights w that the loss $\hat{L}(w) = L(y^*, f_w(x))$ is minimized on a training set x. Since the function $\hat{L}(w)$ can be differentiated by w, we can use a **gradient descent** algorithm to find the optimum. The algorithm uses the observation that since f_w is defined and differentiable, its value at a point a decreases *fastest* when going in the direction of the negative gradient $-\nabla_w \hat{L}(a)$. Hence the suboptimal solution is found by iteratively making small steps in the direction of the steepest descent. Computing a gradient based on all of the available data is too slow in practice – therefore

stochastic gradient descent, which estimates the gradient with a **batch** of B examples, is used instead. Some modifications are needed to make the stochastic gradient descent more effective and more robust against local minima; please refer to Goodfellow et al. [2016] for more details.

2.4.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks capable of processing sequential data (unlike feedforward neural networks which can process only fixed-sized input). In order to process sequential data of variable length, the neurons should have a memory for information to persist, which is done by allowing *loops* in the network.

However, a general cyclic network cannot be optimized by a gradient descent optimizer. To avoid this issue, we allow only cycles of unit length. The network is **unfolded** during training – it is cloned for each time step so that the loops transfer information between two successive time steps. Since the unfolded network is acyclic, it can be trained with the same algorithms as feedforward neural networks.

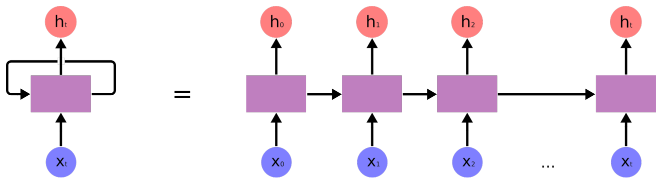

Figure 2.10 *Diagram showing a recurrent unit with a looped hidden state on the left; the right-hand side shows the same unit unfolded in time. Reproduced from Olah [2015]*

In order to capture long-term dependencies, unfolded RNNs need to be very deep (usually more than 100 time-steps), which causes problems with exploding and vanishing gradients that prevent the network from converging.[4] **Long Short-term memory** (**LSTM**) architecture by Hochreiter and Schmidhuber [1997] is specially designed to solve this problem.

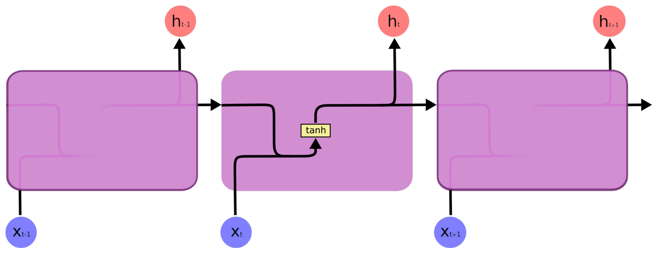

Figure 2.11 *Illustration off a simple RNN cell consisting of only one ϕ_{tanh} activation. [Olah, 2015]*

[4]The derivatives in the gradient descent algorithm are computed gradually from the output layer to the input layer by the chain rule. As the derivatives are multiplied by the same weight vector for each time step, their values can easily either vanish to zero or explode into unreasonably large values. Please refer to Goodfellow et al. [2016] for more information.

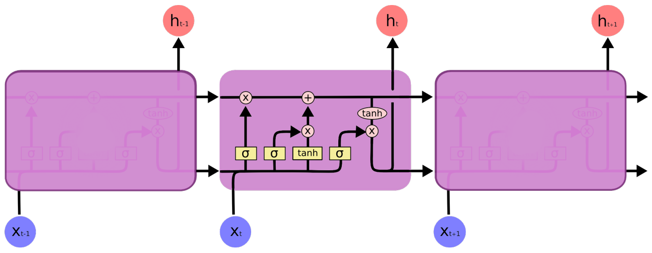

Figure 2.12 *Illustration of a more complex RNN cell called LSTM (described below). Note that the cell state (upper horizontal line) proceeds through the cell without any nonlinearities. [Olah, 2015]*

A cell state c can flow through an LSTM layer without any change, which solves the issue with exploding and vanishing gradients. When a layer wants to modify some information in the cell state c_t, it can do so by using **gates** that linearly interact with the cell state and are computed according to the input vector x_t and output vector h_{t-1}. Each gate consists of a sigmoid layer and a bit-wise multiplication operation influencing how much information should be let through.

The first gate is the **forget gate** f, which is multiplied directly by the cell state. The forget gate removes information from the cell state. The second gate is the **input gate** i which is multiplied with a *tanh* layer of the input and the result is then added to the cell state. The input gate can insert new information into the cell state. The last gate is the **output gate** o, which selects information from the updated cell state c_t that should be let through to the output h_t.

More formally, one forward pass of a single LSTM layer can be computed by the following equations, where t represents a time step, $*$ is a bit-wise multiplication operator and W, U and b are weight matrices and bias vectors that need to be trained:

$$\begin{aligned}
f_t &= \phi_\sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \phi_\sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \phi_\sigma(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t * c_{t-1} + i_t * \phi_{\text{tanh}}(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t * \phi_\sigma(c_t)
\end{aligned}$$

The initial values of c_0 and h_0 are usually set to zero. Zimmermann et al. [2012] suggest initializing the initial values with some noise during the training for better generalization. Hence we initialize them by a uniform noise in our implementation.

Chapter 3

Related Work

The first research in artificial music composition dates back to 1950s. Illiac Suite (String Quartet No. 4) is the most famous example of the early efforts. It was automatically composed by a computer program by Hiller and Isaacson in 1957 (see Sandred et al. [2009] for more details). The early approaches used generative grammars, rule systems, and Markov chains to compose music. See Nierhaus [2009] or Fernández and Vico [2013] for comprehensive surveys of the different techniques used. Note that these methods often rely on hardcoded rules by musical experts and create music in a very limited space.

With the recent advent of deep neural networks, people started using them for artificial composition. There exist two different approaches – one can synthesize either a raw audio signal or create a more abstract symbolic representation of music (such as sheet music). Van Den Oord et al. [2016] tried the first approach as a possible application of their WaveNet architecture, but it seems that the raw audio signal contains too much information to be understood by current neural networks. Another example of such approach with similar result can be found in work by Nayebi and Vitelli [2015].

The second symbolic approach typically utilizes a recurrent neural network (RNN) capable of processing a sequential input. Such a network is then used to predict a note N at time $t+1$ given the previously played notes as the input. It essentially approximates a conditional probability distribution $P(N_{t+1}|N_t, N_{t-1}, \ldots)$ by learning it from a large corpus of music. When the network sufficiently estimates the distribution, we can stochastically sample from it to generate new music.

Eck and Schmidhuber [2002] proposed using an LSTM network instead of a simple RNN and created a generative model for short pieces of 12-bar blues music. The LSTM architecture helped them to overcome the limitations of RNNs – Mozer [1994] attempted to compose music with them and reported their inability to understand the phrase structure and rhythmic organization. For this reason, most of the further work almost exclusively uses some variant of more robust RNNs.

Architectures of neural networks for music composition are usually compared on rather small datasets, most commonly on JSB Chorales [Radicioni and Esposito, 2014] (60 pieces) and Nottingham Music Database [James Allwright and Shlien, 2003] (1037 pieces). To overcome difficulties caused by the insufficient amount of data, highly specialized architectures are being proposed. Boulanger-

Lewandowski et al. [2012] combined recurrent neural networks with *restricted Boltzmann machines* to estimate the joint probability distribution of notes. Vohra et al. [2015] improved this technique by combining LSTM with a *deep belief network* to create the state-of-the-art model for the datasets mentioned above. Another interesting architecture was proposed by Johnson [2017], who combines LSTM with convolutional neural networks to enable his model not only to be time-invariant but also transposition-invariant. Magenta [Simon and Oore, 2017] managed to create a very expressive music model by encoding its input and output into various *events*. The model can use 128 *note-on*, 128 *note-off*, 100 *time* and *32 volume* events to play notes and manipulate the tempo and volume. We use a similar (though simpler) approach in our model. Another work worth mentioning is a model by Žídek [2017] who experiments with LSTM network enhanced by residual connections [Wang and Tian, 2016], and a model by Johnson et al. [2017] for generating jazz melodies using two different music generators and combining them using a *product of experts* [Hinton, 2002]. An extensive survey of deep learning techniques for artificial music composition can be found in Briot et al. [2017].

Chapter 4

Dataset

Contrary to other experiments on music composition (to our best knowledge), we train the generative model on a much larger dataset (only hundreds of pieces are usually used). We have obtained a dump of MIDI files from various sites on the Internet which contains 129,506 files in total. Our MIDI parser can validly process 126,784 of these; 72.53 % is multi-instrumental and 27.47 % of them contains only one instrument. Since 32.13 % of them is labeled according to their genre, we can use the genre distribution to estimate the contents of the whole dataset.[1] The dataset is not cleaned in any way, which means that it contains some duplicates (or different representations of the same piece).

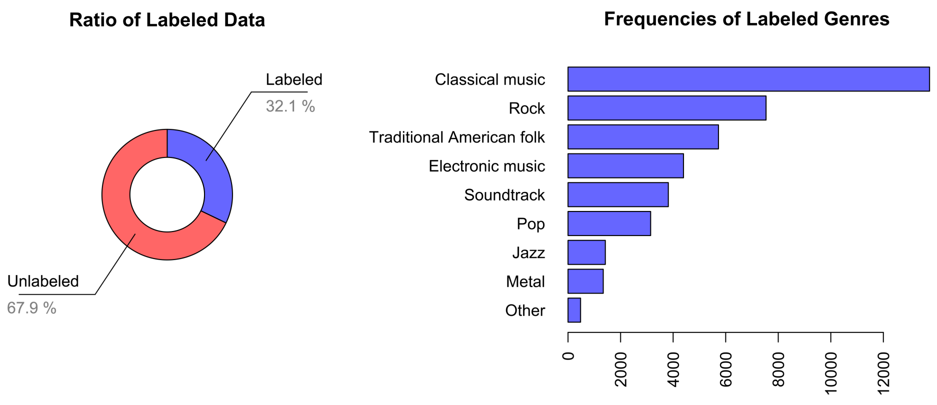

Figure 4.1 Left*: Ratio between pieces classified by their genre and pieces without a label.* Right*: Numbers of occurrences of each genre in the labeled part of the dataset.*

The whole dataset is split into 1,000 files for validation set and the rest for the training set. Traditionally, the final generative model should be evaluated on a separate test set, but since it is not possible to perform a meaningful automatic evaluation, we use an online survey for judging the quality of generated composition instead.

[1]At least to an extent, for example the traditional American folk music can be probably found almost exclusively in the labeled portion, because it was (presumably) obtained solely from one site specialized only on this genre of music. On the other hand, it seems that the frequency of pop music in the whole dataset is underestimated.

Due to the large size of the dataset and copyright issues, we cannot attach the data publicly, but we can provide them on a personal request.[2]

[2]The dataset should be also available online (but with fewer labeled pieces) on `https://www.reddit.com/r/datasets/comments/3akhxy/the_largest_midi_collection_on_the_internet/`

Chapter 5

Preprocessing

This chapter describes preprocessing steps applied to input MIDI data to reduce their complexity without significant compromises on the possible musical expression. Training a neural network without this procedure would be impossible for current state-of-the-art neural architectures because the space of raw MIDI files is too large.

First, we describe a meter detection algorithm, which is a necessary step in the preprocessing procedure, because it makes the time discrete. Then a key detection algorithm is introduced and is followed by a chord detection algorithm, which combines the approaches of the previous two algorithms. Finally, we describe other less complicated (albeit critical) procedures, that also simplify and discretize the input.

5.1 Meter Detection

The rhythmic structure is one of the most critical parts of both traditional and modern western music. Nevertheless, capturing it correctly is difficult, as the rhythm is dependent on time which is inherently continuous – but our model needs the time to be discrete. For this reason, we need to discretize the time. To do this with as low loss of information as possible, we have to find the underlying metrical structure and normalize the tempo according to it.[1]

When detecting the metrical structure, occurrences of beat pulses have to be determined. A certain meter is encoded in all of the MIDI files, but it is not always the true one. For that reason, we define a metric to measure the fitness of the meter, and when the meter is likely to be wrong, we recalculate it based on an algorithm from the book Cognition of Basic Musical Structures by Temperley [2004].

In this chapter, we introduce the procedures used to normalize and discretize the tempo. First, the Meter Fitness Metric is defined to approximate how well does an estimated metrical structure fit the analyzed composition. Then, the al-

[1]As discussed in the introduction to music theory (section 2.1.4), dynamic changes of tempo are important characteristics of a musical piece performed by professional musicians. When the tempo is normalized, we lose that expression. On the other hand, MIDI files are usually transcribed from a note sheet where such information is unavailable, which is why this should not be an issue.

gorithm for meter estimation by Temperley is presented. Next, we propose some changes to the original algorithm and show that they indeed enhance the performance of our dataset. After that, we present some illustrative pseudocodes that should better explain our algorithm. Finally, the normalization and discretization procedures are described.

5.1.1 Meter Fitness Metric

In order to decide how well does the detected meter represent the rhythmic structure of the composition, we have to define a metric. When comparing two meter estimations, we assume that the correct one has more notes occurring at its beat pulses (or at the pulses of lower metrical levels). Following this intuitive assumption, we then propose the metric of a meter to be the proportion of note onsets explained by the pulse of the meter. In practice, the notes are not played at the exact times (as expected by a note transcription for example), but they are played with some variation in time. For this reason, we assume that for an ideal meter the note onsets should be normally distributed around its pulses with *some* variation.

The **Meter Fitness Metric** (**MFM**) is defined according to these two assumptions as the average of **note fitnesses**. The fitness of a note is calculated with the following algorithm:

- Calculate the relative time `t` as the time delay of the note start and the latest beat pulse and normalize it by the beat length – to stay in the range of [0,1].
- Return the likelihood[2] of the relative time `t` when assuming that a note should occur close to the main beat pulses (or less probably close to pulses of two lower levels), which is defined as follows:

$$fitness = \begin{cases} 1 \cdot \exp(-(16(t-0))^2) & t < \frac{1}{8} \\ 0.25 \cdot \exp(-(16(t-\frac{1}{4}))^2) & t < \frac{3}{8} \\ 0.5 \cdot \exp(-(16(t-\frac{1}{2}))^2) & t < \frac{5}{8} \\ 0.25 \cdot \exp(-(16(t-\frac{3}{4}))^2) & t < \frac{7}{8} \\ 1 \cdot \exp(-(16(t-1))^2) & otherwise \end{cases} \tag{5.1}$$

We use MFM to measure the quality of the detected meter and decide whether does the default meter (that is already implicitly encoded in the MIDI file) represent the tempo well. If we assume that the default meter is either *matching* or *mismatching*, we can interpret the histogram (Figure 5.2) of all MFMs from the dataset as two normal distributions: one *matching* with the expected value around 0.65 and one *mismatching* with the expected value around 0.125. Histogram of MFMs of a random meter[3] is shown as red dots in the same figure (scaled down to have the most frequent value at the same height), to give stronger support for

[2] In fact, the probability distribution is scaled up so that the "ideal" meter has MFM 1.

[3] For each piece, a beat length between 400 and 1200 ms is chosen at random and then all beats in the piece are created with this length.

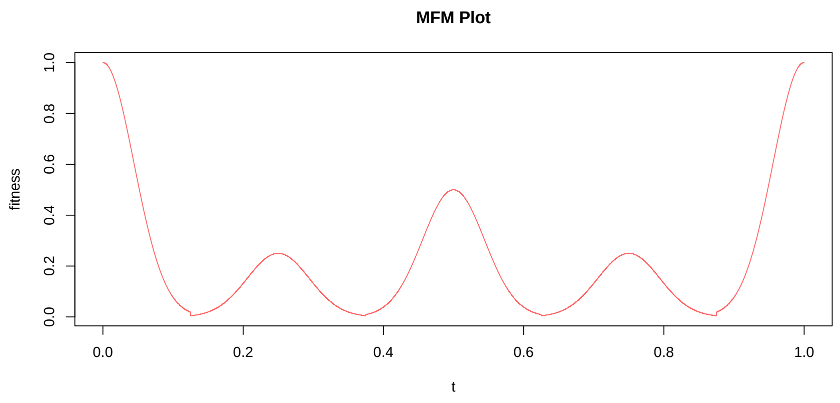

Figure 5.1 *Plot of the MFM equation 5.1.*

this claim. Based on the distribution from the histogram, we create a threshold of 0.3 to classify the default meter as either *matching* or *mismatching*.

If the default meter is classified as *mismatching*, we try to estimate it better by using the following dynamic program based on the David Temperley's book.

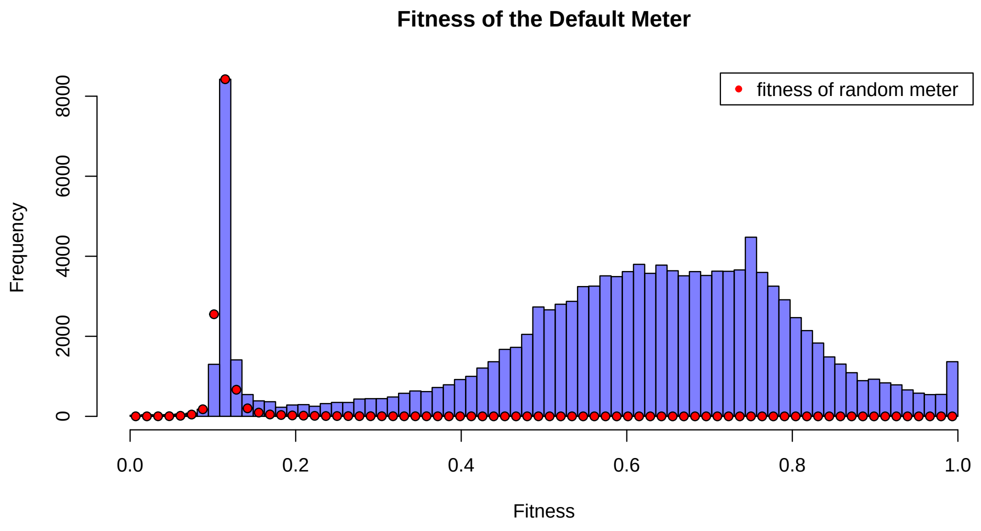

Figure 5.2 Blue: *Histogram of Meter Fitness Metric values of default meters in the whole dataset, higher values mean better fitness.* Red: *Scaled histogram of MFM values of random meter measured on the whole dataset. The peak around value 0.125 suggests that many default meters do not match to the actual meter of the piece.*

5.1.2 Algorithm for Meter Estimation

The algorithm tries to find an optimal metrical structure based on a set of hard and soft constraints. First, we describe a general system of these constraints that can be used for detection of various musical phenomena. Next, a specific set of constraints for meter estimation is defined, and a program for an effective search of the optimal solution is introduced. Then, we propose some additional constraints that improve the performance on our dataset. Finally, we measure the quality of the estimation by using the MFM.

The Preference Rule System

The preference rule system was first proposed by Lerdahl and Jackendoff [1985] in A Generative Theory of Tonal Music. This system consists of three kinds of rules which are based on observations of actual human perception of music.

The **preference rules** measure how *well* does our estimation correspond to an expected human cognition of the estimated musical structure. **Well-formedness rules** can be seen as hard rules or limits on which estimations are allowed and which are not. They also defined a third kind of rules – **transformation rules**, but these are not used in our algorithm.

These rules provide an intuitive high-level abstraction. We can then separate the description of the constraints from the low-level implementation that searches for the optimal solution. One disadvantage of this approach is that the preference rules contain a lot of artificial weights that have to be set.

Rules for Estimating the Metrical Structure

This subsection describes the preference and well-formedness rules for estimating the metrical structure as proposed by Temperley [2004]. We only need to detect the beat level of the metrical structure for our use, but Temperley's rules are more general, which is why we have omitted all the rules that deal with the additional complexity (which however does not influence the accuracy of detecting the basic beat level).

Metrical Well-formedness Rule 1 (Beat Length Rule): The time distance between two successive beat pulses should be between 400 and 1200 ms and the shortest and longest distance in the whole analyzed piece should not vary more than 1.8 times. This rule is not implicitly stated in Temperley, but it can be inferred from his implementation.

Metrical Preference Rule 1 (Event Rule): Prefer a structure that aligns beat pulses with event onsets.

Metrical Preference Rule 2 (Length Rule): Prefer a structure that aligns beat pulses with onsets of *longer* events. This rule can be seen as an extension of the Event Rule. A certain amount of points is given to the segments for each of their notes, where the points are weighted by the note's **effective length**.

The effective length is calculated as the maximum of the actual length and the **registral interonset interval** (**RIOI**) of the note, which Temperley defined as the time difference between the start of the note and the start of the nearest successive note within some range of pitch – we use eight semitones. As effective length could easily gain unrealistically large values, it is limited to be 1000 ms at most. The intuition behind this is that the perception of a note ends when it is interrupted by another note in the same melodic line, not necessarily when the note stops playing (playing *staccato* should not change how we perceive the underlying meter).

Metrical Preference Rule 3 (Regularity Rule): Prefer beat pulses to be maximally evenly spaced. Ideally, the meter should stay the same throughout the whole analyzed composition. However – as it is common for compositions to feature some deliberate tempo changes and as it is common for human performers to slightly vary the tempo as well – the Regularity Rule is not stated as a well-formedness rule, but as a preference rule instead (which is the difference from the original metrical model proposed by Lerdahl and Jackendoff).

When implementing this rule, the difference of the duration between the current beat and the previous beat is calculated and – if the difference is larger than the quantization time – the square root of this difference is subtracted from the score of the segment.

Dynamic Program for Optimization of Preference Rules

The main idea of the estimation algorithm is that it uses the preference rules to choose the best estimation out of a set of all the ones that satisfy the well-formedness rules. A dynamic programming approach is used to search through the space of all possible structural descriptions effectively.

First of all, the time of the analyzed piece is quantized into **segments**. Each of these segments is linked to the notes that start at the same quantized time. The metrical structure of the piece is then defined as a subset of those segments. Generally, the quantization time should be small enough to be precise, but at the same time large enough, because groups of notes meant to be played at the same time are usually varied for a few milliseconds from each other, and we would like them to stay in the same segment. We used quantization time of 35 ms, which was also used by Temperley.

The **base scores** of each segment are computed first. These conform to the Event Rule and the Length Rule. Because of this, the base score of a segment is not dependent on any other segment, and its computation is straightforward. On the other hand, when computing the **regularity score** according to the Regularity Rule, previous segments and their scores have to be considered. The following algorithm computes these backward dependencies.

After calculating the base score for all the segments, we iterate through all the segments from the first one to the last one to compute their **total scores**. For each segment, we assume that it is the start of a new beat and all its possible lengths (the time interval between the segment and the next beat pulse) are considered. We calculate the total score for each of these lengths.

To calculate the total score for a length `l` of a segment `s`, we must try all possible lengths of the previous beat `prev_l`, calculate the possible scores `score` for each of them and then select the best one as the total score. The start of the previous beat corresponds to some previous segment (if it does not exist, because we are at the beginning, we set `score` to a default value). All total scores of the previous segment are already calculated, and we can thus get its total score for `prev_l`. The Regularity Rule gives us a negative regularity score based on the difference between `l` and `prev_l`. Then `score` for `prev_l` can be finally determined. When we have the scores for all possible previous lengths, we select the largest one and set it as the total score. The previous length belonging to that score is also saved for later backpropagation. To better explain the algorithm, we present illustrative pseudocodes in Section 5.1.2.

When all total scores are computed, segments with beat pulses can be selected. We start from the last beat, which is detected as the segment with the largest total score. A certain previous interval corresponds to this total score. We can thus move to the previous beat and select the *total score* which belongs to that interval. By iterating this process, we get all the beats.

When done like this, the algorithm prefers *shorter* beat length, because more base scores can be summed up, when there are shorter beats. Temperley suggests weighing the base score by the square root of the beat length when calculating the total score to avoid this problem.

Enhancements

The original algorithm by Temperley was created to detect meter in polyphonic classical compositions with just one instrument, but we would like it to work well for any western musical genre (especially for popular music) with any number of instruments. In addition to this, the resolution of the beat timing is based on the initial time quantization, but we would like to represent even smaller changes in tempo. We try to solve both of these problems by introducing some changes to the original algorithm and also by using gridsearch to find better parameters of the preference rules.

First of all, we try to solve the problem with small resolution. We propose two ways to deal with the small resolution issue, *smoothing* and *averaging*.

The **smoothing** is based on the following observation of the original algorithm: when the quantization is set to t ms and the correct length of successive beats is l ms, then the algorithm will likely output alternating multiples of t which are closest to l and their average tends to l (for $t = 20$ and $l = 610$, possible lengths of consecutive beats in the output are $(600, 620, 600, 620, \dots)$), which is an expected result given the preference rules.

The smoothing procedure then takes all **sequences of alternating beats** (inclusion-wise maximal sequences of successive beats whose lengths differ for at most t) and adjust each of them so that all its lengths are set to the average length of the sequence. As there may be a slight tempo change in the original composition (less than t), a Gaussian blur has to be used to detect this change instead of a simple global average. Although this procedure usually works as intended, it turned out in practice to have two flaws: it still does not estimate the small tempo changes well and most importantly – it cannot differentiate between small intended changes of the detected meter (for example in human performance) and alternating beats caused by small resolution.

For this reason, we introduce **averaging** as a simpler solution to the small resolution problem as it turned out to be more accurate in practice (at least on our dataset). To *average* a beat, we estimate its time as the average of its notes' onset times (and leave it to the default value, when the beat has no notes assigned). This procedure seems to give better results on the majority of compositions, so we decided to use it instead of the smoothing mentioned before. However, as it still does not work on some inputs, we believe there is room for future work on this problem.

In addition to the postproduction by averaging, the Regularity Rule should be adjusted, so that a metric structure is not underestimated, when it alternates the beat lengths by `t` to approximate the hidden tempo. On the other hand, some penalty has to be set when the algorithm misuses the Regularity Rule by slightly changing the lengths to get a totally different tempo. For example when `t = 20`, the original Regularity Rule would not punish beat lengths `(600, 620, 640, 660, 680)`, which would ultimately allow it to change the tempo from 600 ms to 680 ms without any penalty. For this reason, we define **alternation** for each beat pulse. It can be either `None`, `Declining` or `Raising`. The alternation of a beat is implied by the alternation of the previous beat, quantization time `t` and the difference of beat lengths between the current and the previous beat called `d`. The finite automaton at Figure 5.3 defines the transition function. The alternation of the first beat is set to `None`. The red edges in the figure are those transitions that should result in a penalty by the Regularity Rule.

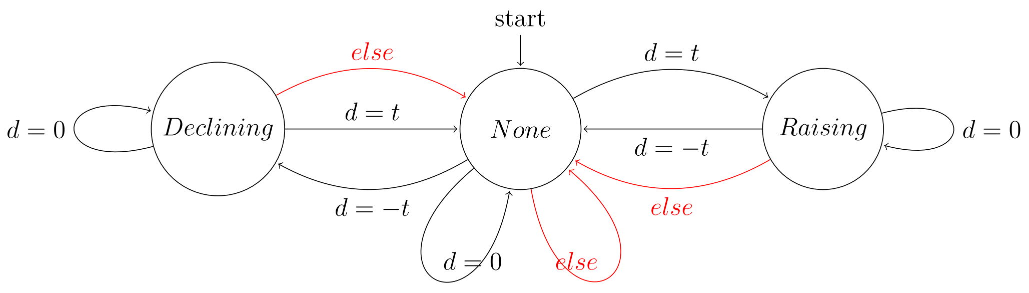

Figure 5.3 *Automata for alternation in the updated Regularity Rule*

Popular music often uses syncopation (explained Section 2.1.4), which leads to the following rule:

Metrical Preference Rule 4 (Off Beat Rule): Prefer beats that have notes with onset at the half of the beat, third or quarter of the beat. For each of these notes, a portion of points is added to the total score. The portion can be calculated when computing the negative scores for Regularity Rule.

As our dataset has volumes assigned to each note, the fifth preference rule is straightforward:

Metrical Preference Rule 5 (Volume Rule): Prefer a structure, that aligns beats with onsets of *loud* events. This rule is an extension of the first and second preference rule, the points from each note are not weighted only by the length of the note, but also by its volume.

Not only has each note a volume, but it is also assigned to some musical instrument. We use this information to better estimate the melodic lines when computing RIOI. In addition to this – as percussion is usually used as the main means of conveying the tempo in popular music – we use the role of percussion to create the sixth preference rule:

Metrical Preference Rule 6 (Percussion Rule): Prefer a structure that aligns beats with onsets of *percussion* events. This rule is an extension of the first, second and fifth preference rule, it gives the percussion events a bonus when computing base scores.

Pseudocodes

The description of the algorithm has been so far very high-level and misses some important implementation details. For this reason, we present illustrative pseudocodes for calculating *total scores* of segments in this section.

```
REG_MUL ← 10
SYNC_MUL ← 0.25
NOTE_MUL ← 1
NOTE_BASE ← 0.2


function TOTAL__SCORE(segment s, length l)
    best ← (score:  -1, prev_length:  -1)
    for each possible length of the previous beat prev_l do
        prev_s ← segment occurring prev_l earlier than s
        if prev_s is null then
            score ← DEFAULT__SCORE(l, prev_l)
        else
            score_prev ← prev_s.total_score[prev_l]
            reg_score ← REG_MUL * REGULARITY__SCORE(s,prev_s)
            sync_score ← SYNC_MUL * SYNCOPATION__SCORE(s, prev_s)
            base_score ← NOTE_MUL * s.base_score
            score ← score_prev + base_score + sync_score - reg_score
        end if

        if score > best.score then
            best ← (score, prev_l)
        end if
    end for

    return best
end function


function BASE__SCORE(segment s)
    notes ← notes in s
    percs ← notes where note.instrument is percussion
    rioi ← average RIOI of notes in seconds
    volume ← average volume of notes

    return $\sqrt{\texttt{notes.count + percs.count}}$ * rioi * volume + NOTE_BASE
end function


function REGULARITY__SCORE(segments s, prev_s, lengths l, prev_l)
    d ← |l - prev_l| in seconds
    s.alternation[l] ← ALTERNATION(prev_s.alternation[prev_l], d)
```

```
        if not s.alternation[l].penalty then
            return 0.0
        else
            return $\sqrt{d}$
        end if
    end function
```

Results

In the following figures, we compare the original algorithm with our enhanced version. We illustrate the difference by histograms of MFMs on our whole dataset. We also show the histogram of the final algorithm, which uses the default meter when MFM is greater than 0.3 and our algorithm for meter estimation otherwise.

Two important notes have to be made about the results from the Temperley's algorithm:

1. We used our own implementation based on his description so we may have overlooked some details.
2. His suggested penalty in the Regularity Rule is set fairly low, which means that his algorithm focuses more on aligning beats with clusters of notes than on keeping the meter regular. The MFM does not measure the regularity of the meter, but it solely focuses on the alignment of the beat pulses with notes. This may result in a slight advantage of his original algorithm, as we set the penalty about three times higher in our algorithm.

According to Figure 5.4, it seems that our enhanced algorithm suits our dataset better. When comparing the MFMs, the original algorithm by Temperley has the sample mean fitness of 0.452 with the standard deviation of 0.132 and our algorithm has the mean equal to 0.602 and the standard deviation to 0.196. Moreover, when compared with the default meter on the *matching* compositions (which should be the correct meter), our algorithm produces similar results. The difference between fitnesses of our enhanced algorithm and the default meter has the mean of 0.029 with the standard deviation of 0.130. We thus believe that we have improved the algorithm and that it approximates the hidden metrical structure well.

5.1.3 Meter Normalization and Discretization

The main reason for estimating the metrical structure is to normalize the tempo and discretize the time.

The tempo normalization should help the generative model not to consider different tempos and their changes. We argue that the tempo does not have such important role in music.

The tempo is normalized to 100 beats per minute (BPM) for all sampled compositions. To achieve this, we change the time duration between each two consecutive beat pulses to 600 ms and scale onset of every event accordingly (so that their relative position in a beat remains the same).

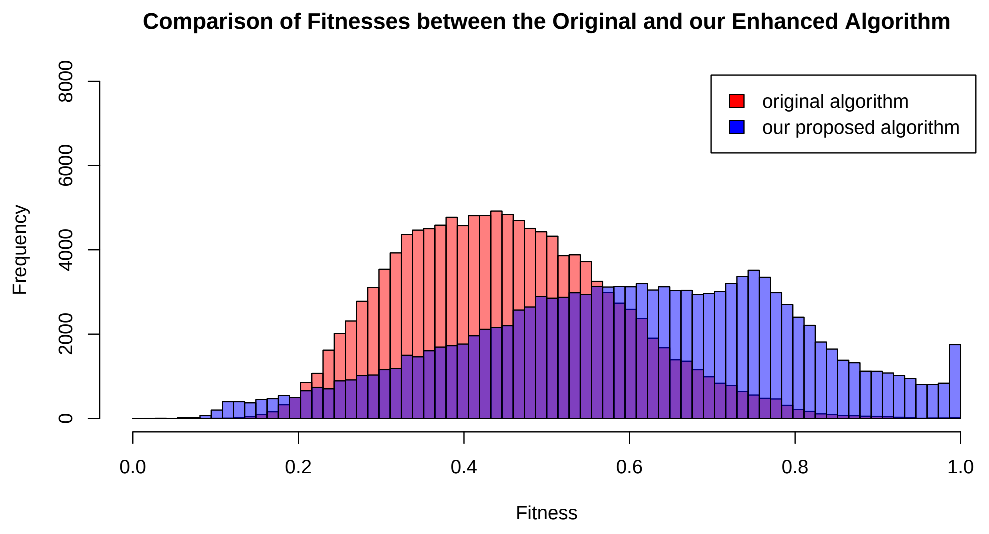

Figure 5.4 *Histograms of Meter Fitness Metric of the original meter detection algorithm by Temperley (in red) and our improved version of that algorithm (in blue); higher values mean better fitness. The distribution shows that our algorithm detects the meter better on the majority of our dataset.*

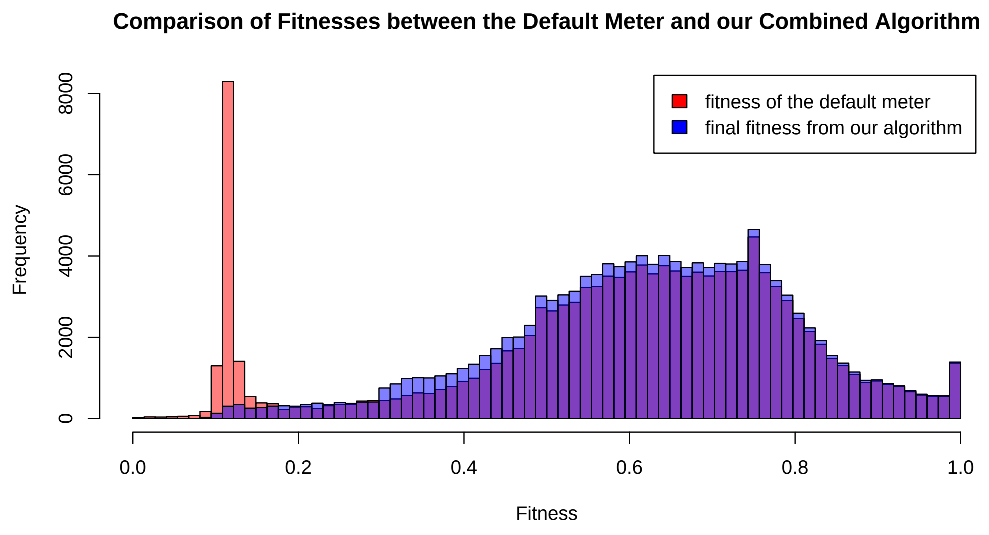

Figure 5.5 *Histograms of Meter Fitness Metric of the default meter of the midi files (in red) with a high peak in the left side and our final algorithm which uses the improved meter estimation only if the default meter fails (in blue). The distribution shows that our algorithm (in blue) got rid of the incorrectly detected meters (the red peak).*

Then the time of every event is rounded to the nearest multiple of 50 ms. As each beat lasts 600 ms, it can be divided into 12 units of 50 ms. In that way, we can represent eighth, sixteenth and triple quarter notes without any time distortion and it is also the smallest division that allows that. This means that if the metrical structure is estimated correctly, then most of the time information is preserved without any loss.

5.2 Key Detection

When perceiving a note in a music composition, its individual role is much less important than its role in the nearby context. This phenomenon is especially valid in the case of a tone pitch. There are many different kinds of context that influence the perception of a pitch – such as its part in a harmonic structure or the relation to the surrounding notes in a melodic line. In this chapter, we will deal with the context of a key. When considering the most usual major and minor modes, there are 24 different keys in total. A single pitch has a different role in each of these keys, which adds a lot of complexity that a prediction model should learn. On the other hand, when a piece of music is transposed to another key, it is perceived almost the same.[4] Hence, we would like to transpose the key of each piece in our dataset to a unified key.[5]

In order to transpose a key, we have to classify it first. The problem of key detection has been studied for a long time, but a robust solution still does not exist. The algorithms can be divided into two groups, the first uses a symbolic representation of a piece and the second uses raw acoustic information (for example Campbell [2010]). Since we are dealing with MIDI files that contain only symbolic data, we will use the symbolic approach.

The classical symbolic approach to key detection is the one proposed in Cognitive Foundations of Musical Pitch by Krumhansl [1990]. We will briefly describe his algorithm and then introduce our classifier based on machine learning methods. Finally, we will compare these two approaches.

[4]Real instruments sound slightly different when played in different keys. But we deal only with synthetized midi files whose timbre is not affected by transpositions.

[5]We would like to preserve the original mode, so we in fact transpose to two different keys. Major keys are transposed to C major and minor keys are transposed to its relative key — A minor.

5.2.1 Krumhansl's Method

The core definition of the Krumhansl algorithm is the **key profile**. A key profile kp_K of a key K is a vector of 12 values; the indices represent 12 different pitch classes, and the values represent how much is each pitch class harmonious with the key K. The values are scaled – in this way it can be interpreted as the probability of a pitch class being played given the context of a particular key.[6] so that the sum of each key profile is 1. The actual values in the key profiles are based on a series of experiments conducted by Krumhansl and Kessler [1982]. Based on their experiments, they created two different key profiles – one for the major mode and one for the minor mode.

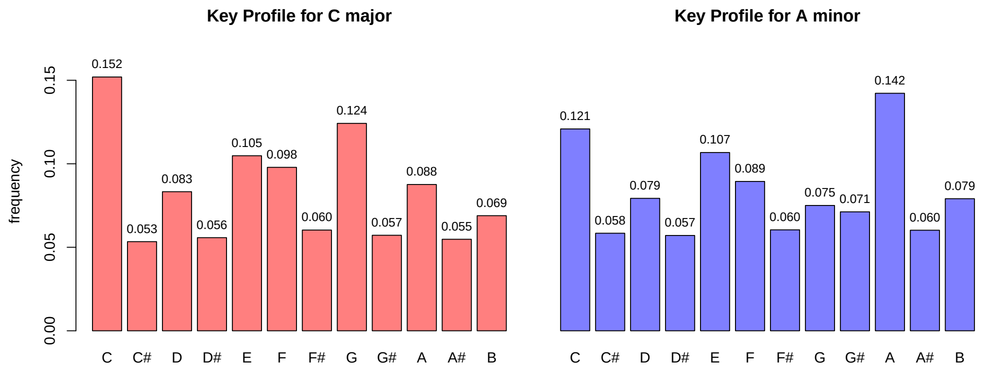

Figure 5.6 *Key profiles of C major and A minor obtained from experiments by Krumhansl and Kessler [1982].*

Apart from an ideal distribution of pitch classes represented by a key profile, an actual distribution of a piece I has to be calculated. This distribution can be described by a vector of 12 values which we will denote as the **pitch profile** pp_I. One way of computing the pitch profile is to sum up the lengths of notes, grouped by their pitch classes, and to normalize these sums to obtain a valid distribution (which is what Krumhansl [1990] originally did). However, as Temperley [2004] has shown, it is more robust to first divide the analyzed piece I into beats (which is already done in our case) and then to create an indicator function $\mathbb{1}_B(p)$ returning 1 if a pitch class p is present in a beat B and 0 otherwise. Then we sum up these indicators for each pitch across all beats and normalize it to gather the pitch profile pp_I.

$$(pp_I)_i = \frac{\sum_{B \in Beats(I)} \mathbb{1}_B(i)}{\sum_{p=1}^{12} \sum_{B \in Beats(I)} \mathbb{1}_B(i)}$$

After obtaining the pitch profile pp_I we can measure its correlation to each key profile – then the most probable key is the key K, which key profile kp_K has the largest correlation with pp_I.

[6]The key profiles were not normalized in the original definition, we decided to rescale them so that the following statistical operations are more intuitive.

$$
\begin{aligned}
corr(x,y) &= \frac{\sum_{i=1}^{n}(x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\overline{x})^2\sum_{i=1}^{n}(y_i-\overline{y})^2}} \\
key(I) &= \underset{K\in Keys}{\operatorname{argmax}}\, corr(pp_I, kp_K)
\end{aligned}
$$

5.2.2 Machine Learning Models

Correlating the pitch profile with key profiles is a straightforward and effective way to detect the key. However, it is not precise enough for our purposes (see the results in Section 5.2.3). Since we have a lot of annotated musical pieces in our dataset, we decided to detect the key by machine learning methods. The problem of key detection can be formulated as a supervised learning classification task, where the classes correspond to the 24 different keys.

It is possible to define the key signatures of a musical score by an optional meta-events in the MIDI format. The key signature should be almost identical to the correct key.[7] The key signatures are specified in 57 % of our dataset, which makes 72,228 possible inputs to a classification model. However, 9.20 % of the annotated files have more than one unique key; therefore they are excluded because we deal only with pieces in a single key (this issue is discussed in the next section). On top of that, we have found that many pieces annotated as C major are in fact in a different key.[8] For this reason, we decided to remove all inputs annotated as C major. After this reduction, we ended up with 25,438 annotated files and divided them randomly into 2,500 files for final evaluation and 22,938 files for training.

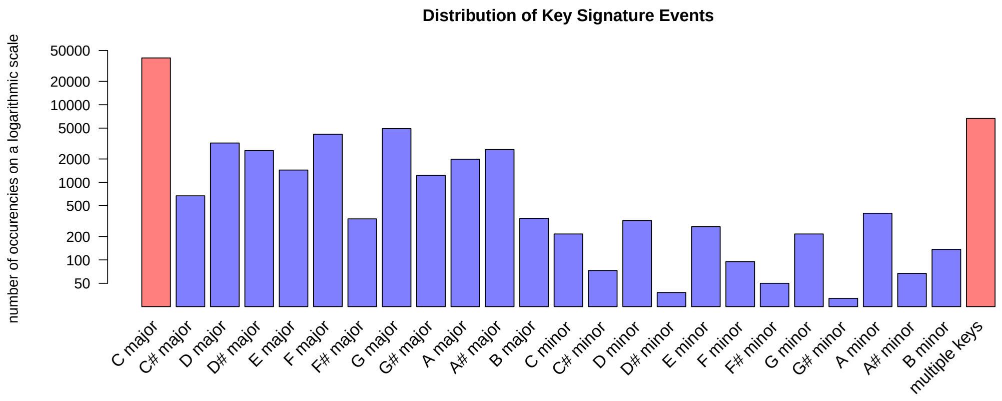

Figure 5.7 *The distribution of key signature events in our dataset; "multiple keys" label represents files with more than one unique key signature; key signatures highlighted with red color were excluded from the training set.*

[7]Although the mode (major or minor) has to be specified, it is not used consistently, so the distinction between relative keys is sometimes unclear. Also note that a modulation to another key may not always be signalized by changing the key signature.

[8]A possible explanation is that C major is the default key signature in some midi editors and is not changed to the correct key signature when creating a midi file.

In order to make the distribution of keys more uniform and to have more training data, we augmented the dataset by transposing each annotated piece to all other 11 different keys in the same mode. In the end, we have 252,318 annotated pieces in the training (development) set.

As for a feature vector, we only use the pitch profile from the Krumhansl algorithm. Madsen et al. [2007] proposed a more comprehensive feature extraction based on an observation that, in addition to their distribution, the order of notes is also very important for humans to detect the underlying key. Hence they defined *interval profiles*, which are represented by a stochastic matrix describing transitions between each pair of pitch classes. Although such features should make the model more accurate, a generalization of the interval profiles for polyphonic music is non-trivial, since a segmentation to different melodic lines has to be done first. Therefore we will use only the pitch profiles.

Multiple Keys

Relatively common musical phenomena (especially in classical music) are key changes (modulations). However, we have dealt only with pieces that are in a single key up to this point. We could modify our algorithm to use a sliding window technique (as proposed by Krumhansl [1990]) or create a preference rule system (as proposed by Temperley [2004]). Nevertheless, transposing each of the temporal keys to a single key would not make any musical sense, as we would like to preserve the relative differences between the modulated keys. For that reason, we try to detect only the most essential central key and transpose other keys relative to it. We assume, that the central key influences pitch profiles the most and should be therefore selected with a high probability.

5.2.3 Results

We have trained a random forest ensemble model (200 trees with no upper bound on maximal number of splits), an AdaBoost ensemble model (200 trees with max-split = 20), SVM with a Gaussian kernel and k-nearest neighbors model ($k = 10$ with the Euclidean distance metric and weighted majority vote). All the hyperparameters were tuned using 10-fold cross-validation on the training data.

We have also evaluated the accuracy of the classical Krumhansl algorithm for comparison. Temperley [2004] has suggested using different key profiles to obtain more precise results, and we have thus tested both the original key profiles and the modified key profiles.

Aside from predicting the correct key, different kinds of misclassification can occur. The least severe one is a prediction of the relative key (the key with the same pitch-classes and key signature), which does not matter much because we transpose to relative keys in the subsequent transposition. Another error is the prediction of a key that is the fifth of the correct key (e.g., C major and G major); it is also a rather small error, as the two keys are very similar. A more serious error is the prediction of a parallel key, which is the key with the same tonic but a different mode (e.g., C major and C minor). The rest of errors is denoted as the “other errors”.

The most accurate classifier is the random forest. As it is also less computationally

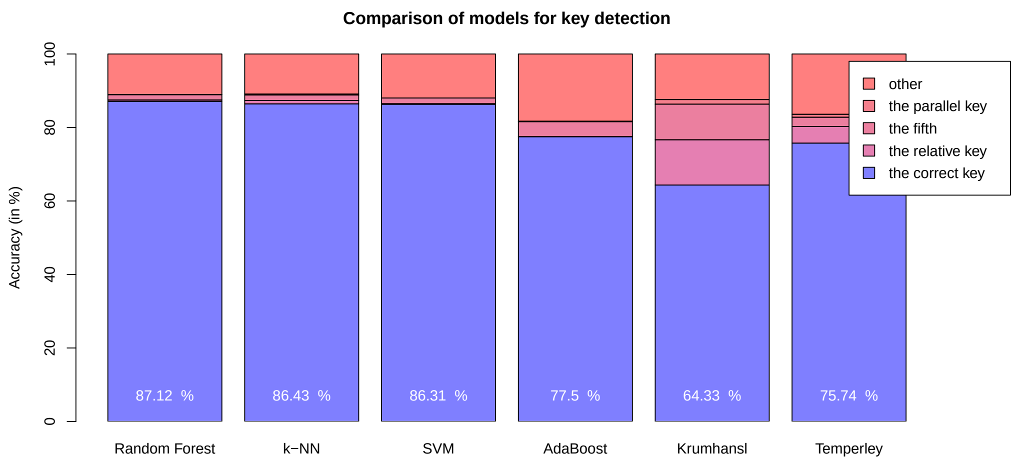

Figure 5.8 *Comparison of different models proposed for key detection.*

expensive than the SVM, we use it as the key detector in the preprocessing of the dataset.

Future Work

Since the error rate is still quite high, there is still some space for improvement. One possible weakness of our model is the feature extraction – we think it is quite unlikely that a vector of just 12 values would be comprehensive enough. One could design a more thorough feature set [Madsen et al., 2007] or use a neural network that could design more suitable features itself. Korzeniowski and Widmer [2017] proposed a convolutional neural network with a spectrogram image as the input, which achieved state-of-the-art performance on raw audio data. We think it would be interesting to use a recurrent neural network, as it is a natural model for sequenced data and it can be utilized for both the symbolic and acoustic representation of music.

5.3 Chord Detection

The memory of current RNN models is not large enough to remember long dependencies in a music piece – the models can remember only about 6 seconds of notes.[9] For that reason, we would like to obtain a more abstract description, that could represent longer dependencies in a more condensed way.

We use the harmonic structure as the representation of a higher abstract level. However, trying to thoroughly describe the harmonic structure would be too complicated. We thus decided to simplify the representation to progressions of basic chords (in major and minor modes without any voicings). In this way, a standard LSTM can remember more than a minute of music represented by chords.[10]

To represent the chord progressions in a piece, we assign a chord to each of its beats. This may be limiting because chords may not change on a beat pulse and there may be multiple chords occurring in a single beat. We argue that this is not a real issue, because the lower-level note representation can better model these phenomena. Besides, compromising for such issues would make our harmony representation much more complex and thus more memory-demanding. We believe that the most prominent limitation of our approach is dealing only with major and minor modes and not considering inversions, extensions or diminished chords. These different voicings are essential for the description of the harmony and the implied mood, but their automatic distinction is too complicated; we believe more complex harmony analysis would greatly benefit the composition model and should be considered in future work.

Unlike other approaches to chord detection [Zenz and Rauber, 2007, Stark, 2011, Hausner and Sauer, 2014], we do not try to derive the harmony from a pure audio signal, because we already have the symbolic (MIDI) representation. The latter two theses try to estimate pitch profiles and then use a simplified preference rule system to predict the chords (though they do not use it explicitly, it can be interpreted as such). Temperley [2004] has suggested a much more comprehensive preference rule system for chord detection, and we will base our algorithm on his concept. However, Temperley solves only a simplified version of our problem, because he tries to predict the roots of chords. We will thus try to generalize his solution to distinguish between major and minor voicings, too. Another difficulty is that we do not have any *chord-annotated* data, so the evaluation of the algorithm is done solely by listening to pieces accompanied by the predicted chords.

First, we will describe the preference rule system, and then we will show some examples of chord estimations and compare them against the correct chords.

[9] Estimation for our proposed representation of music; please refer to Chapter 6 and specifically to Section 6.2 to better understand the following reasoning. Note Predictor is trained on sequences of 120 events, which can encode music samples of various length. If we assume that all events are *Short Shifts*, we can encode $120 \cdot 50$ `ms` $= 6000$ `ms` of music. In reality, some of the events would be *Large Shifts* (that extend the length) and some would be note events (that shorten the length) – hence we think this crude estimation is sound enough to illustrate the problem.

[10] According to Section 6.2, Chord Predictor is trained on sequences of 100 events, where each event encodes 600 `ms`.

5.3.1 Preference Rules

Similarly to the meter detection algorithm, we will describe the procedures by a preference rule system. The preference rules are often based on ideas from the key detection algorithm, so we think the previous chapter should be read prior to this.

Chord Well-formedness Rule 1 (Beat Rule): A single chord is assigned to each beat. The chord is described by its root note and mode (either major or minor). A sequence of beats with identical chords is called a **chord segment**.

Chord Preference Rule 1 (Strong Beat Rule): Prefer chord segments beginning on *strong* beat pulses. Different beat levels can be easily identified with a simple preference rule system extending the beat detector.

We denote the beats detected by Meter detector as beats of **level 0**. They are usually combined to duplets or triplets, where the first beat pulse of each group has **level 1**. One, two or three duplets (or triplets) are then combined into another larger group of beats which is called a *measure*. The first beat pulse of a measure is denoted as the beat of **level 2**. Although more complicated divisions exist, they are not as common, which is why we limit our scope to these divisions only.

Chord Preference Rule 2 (Compatibility Rule): Prefer chords that are suitable with notes in the considered beat. Similarly to key detection algorithm, we calculate pitch profiles for each beat and key profiles for each chord. Then we prefer those chords, whose key profiles have the largest correlation.

As the chord also depends on the surrounding notes, we create three pitch profiles of a different time span – pitch profiles between two consecutive beats at level 2, between beats at level 1 and between beats at level 0.

The pitch profiles are calculated differently than in the key detection algorithm. Each note in a beat is weighted by its length in the beat (the longer the better), by how close to the beginning of the beat does it start (a bonus is given if it starts precisely on the beat pulse) and by the height of the note's pitch (bass notes more strongly convey the harmonic structure, so that is why we give more significance to them). After the weights are determined, we group the notes according to their pitch classes, and then the pitch profile is created by summing every such group.

Chord Preference Rule 3 (Harmonic Variance Rule): Prefer chord segments that are close to the surrounding chord segments on the circle of fifths (Figure 5.9). Chords close to each other in the circle of fifths are perceived as harmonious and are thus more likely to be used. We count the number of steps it takes from one chord to another, and compute a penalty that is proportional to the number of steps.

Chord Preference Rule 4 (Key Rule): Prefer chords close to a key of the whole piece on the circle of fifths. Similarly to the Harmonic Variance Rule, we prefer chords that sounds well given the context of the key. We have to determine the key beforehand to apply this rule.

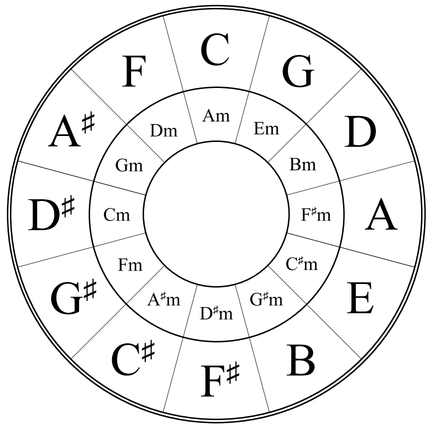

Figure 5.9 *Illustration of the circle of fifths.*

5.3.2 Implementation

The actual process of finding the optimal harmonic structure according to the preference rules is quite similar to the dynamic algorithm for the meter detection which was described in detail in Section 5.1. It is more straightforward so we will explain it only briefly.

After the meter and key detection are done, we divide the analyzed piece into beat segments. We would like to assign a chord to each segment so that the preference rules produce the highest score. The Compatibility Rule and the Key Rule are independent of other beat segments, so their contribution can be calculated ahead. As we do not know which chord will be selected, we count with all of the 24 chords and calculate the *base scores* for each of them.

The other two rules depend on the surrounding chords, so their optimal solution is not as trivial. We compute the *total scores* for each of the 24 chords C (as we do not know what chord will be the best for the next segment) of a segment S_n by the following pseudocode:

```
S$_0$.total_score[C] $\leftarrow$ S$_0$.base_score[C]
S$_n$.total_score[C] $\leftarrow$ $max_{H \in Chords}$(S$_{n-1}$.total_score[H] +
                              S$_n$.base_score[C] - VARIENCE_PENALTY(H, C))
```

Where the variance_penalty is calculated according to the Harmonic Variance Rule. If H and C differ, it means that a new chord segment starts at C – hence the penalty given by the Strong Beat Rule is computed inside this function, too.

After iterating through all beat segments from the first to the last and computing the *total scores*, we establish the final analysis by backpropagating through all chord choices that led to the largest total score of the last segment.

5.3.3 Results and Discussion

As we have said before, we do not have a sufficient amount of data annotated by chords to evaluate our algorithm. However, as we have optimized its performance by ear, we have at least a rough idea of how does the method behave on different music pieces. We present estimated chord progressions of two songs which should show its strengths and weaknesses.

We visualize the estimation by a text representation where one line shows mistaken chords and the other line shows all predicted chords. We use a popular chord notation where the chords are symbolized by capital letters according to their root notes. *Minor* chords are distinguished by a lowercase "m". The symbol "|" is used to divide measures (similarly to bar lines in the standard music notation).

```
The Beatles - Let It Be, first four measures:
mistakes:   |           |     Am7    |           |     Dm     |
prediction: | C  C  G  G | Am Am F  F | C  C  G  G | F  C  C  C |
```

This song uses a popular I - V - vi - IV chord progression that uses only the smallest steps on the circle of fifths. The harmonic structure is also clearly established by a piano accompaniment. Because of that, the detection of the chords is not difficult, and our algorithm managed it without a significant problem.

All the mistakes are caused by the restricted harmonic structure that we use. The A minor seventh chord cannot be detected, because we assume only simple triads, and The D minor chord at the end cannot be found, as we do not consider chords starting in the middle of a beat segment. We argue that both of these errors are minor because those two chords do not convey any substantive harmonic information; their purpose is rather melodic.

```
Radiohead - Creep, first eight measures:
mistakes:   |     |     |  B  |  B  |     |     |  Cm |  Cm |
prediction: |  G  |  G  |  Bm |  Bm |  C  |  C  |  F  |  F  |
```

Radiohead often make use of unusual and unstable chord progressions to create feelings of disorientation and alienation. These chords are dissonant and far away from each other on the circle of fifths. Our model fails to detect such progressions because it assumes that the harmonies should sound plausible in the first place.

We believe that the algorithm can detect the correct chords when they are implied the rhythmic section (percussion, bass and chords). When the harmonic structure is unclear or made purposely unstable, then the algorithm should select a close approximation which sounds more consonant. We argue that the most relevant information about the actual harmonic structure is still preserved and that the composition model is able balance the mistakes from the melodic structure (respectively from the note events).

5.4 Other Reductions

5.4.1 Instrument Clustering

Since we would like to produce multi-instrumental music, we have to deal with a number of different instruments – the MIDI standard defines 128 different ones. Not only is it too many for our model to comprehend, but also the majority of instruments is too rarely used to obtain a reasonable estimate of their distribution.

For these reasons, we cluster the instruments into 11 different classes. There are many possibilities how to do the clustering. The first possible solution is to use the 16 different classes that are already defined in the MIDI standard. However, we came to the conclusion that these classes are not reasonably chosen for our purposes – they often group different instruments together (such as timpani with a string ensemble) or put similar instruments to different clusters (such as a fiddle and a violin). Another way to cluster the instruments is to render their sound and use some unsupervised learning technique to cluster them by their timbral characteristics. While this may work, it would be dependent on the sound renderer (since MIDI does not adequately define the audio qualities). We also believe, that apart from their timbre, instruments should also be clustered according to their use: for example, instruments used as a bass accompaniment should be clustered together despite their sound.

Hence we decided to cluster the instruments by hand, as grouping 128 items is still practicable and is actually less complicated than creating a metric reflecting the requirements above.

The final equivalence classes are defined in Appendix A together with the number of occurrences of each instrument to illustrate the nonuniformness of their distribution. Finally, we present a figure depicting the final distribution of clusters, which is more uniform than the original distribution of single instruments.

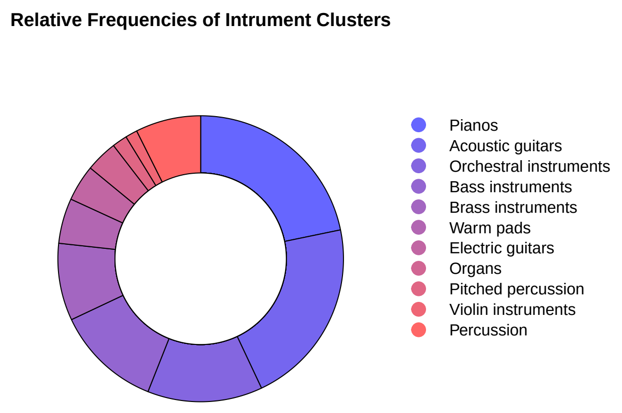

Figure 5.10 *Plot with proportions between different instrument clusters.*

5.4.2 Note Range Reduction

We can express 128 different tone pitches in the MIDI format, which is an unnecessarily large interval – for example, the standard piano has a range of 88 tones and a four-string bass guitar of just 47 tones. Moreover, bass instruments, for example, do not play higher notes very often, which means it would be difficult for the composition model to learn them. Because of that reasons, we reduce the range of each instrument cluster so that only their most frequent pitches are preserved.[11] When a note is outside of the range, it is shifted to the nearest correct pitch in the same pitch class.

The reduced ranges of all instrument clusters are shown in the table below. in Figure 5.11, we present a histogram of pitch occurrences counted across all files in our dataset for instruments in the piano cluster.

Instrument Cluster	**Lowest Pitch**	**Highest Pitch**
Pianos	C_2	C_6
Acoustic guitars	G_2	E_5
Orchestral instruments	G_2	C_6
Bass instruments	C_1	D_3
Brass instruments	G_2	C_6
Warm pads	G_3	A_5
Electric guitars	C_2	E_5
Organs	C_3	C_6
Pitched percussion	G_2	C_6
Violin instruments	G_2	C_6
Percussion[12]	35	82

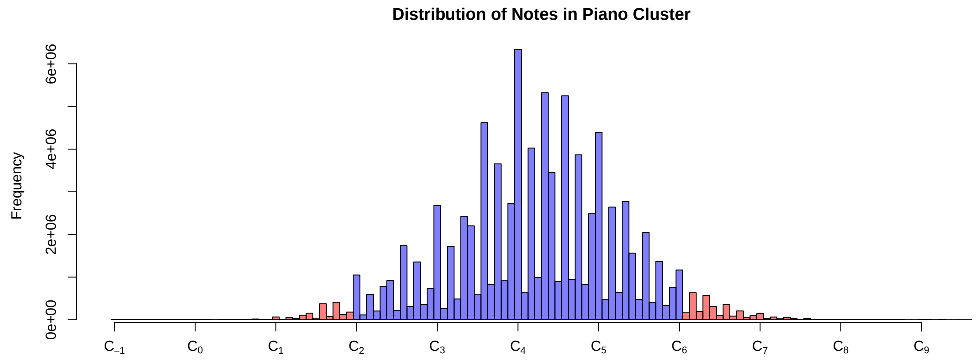

Figure 5.11 *Histogram showing pitch occurrences of notes played by instruments in the piano cluster (after being transposed to a unified key). Red color highlights note pitches that are outside the accepted range for pianos; permitted pitches are shown in blue.*

[11] We choose such range for each instrument cluster, that about 95 % of notes lie inside it.

[12] Percussion notes map their “pitches” to different drum sounds, hence the values cannot be interpreted in the same way as the other instruments.

5.4.3 Dynamics

Apart from the volumes specified in each note message, the MIDI format can also specify volumes in other ways: by relative volumes of each channel, by relative volume of the whole piece or by changing *expression* (messages imitating the expression pedal of pianos). After calculating the contribution of every such influence, we get the real volume of each note in the range $[0, 1]$. Since the absolute values of the volumes are not as important as their relation to the surrounding volumes, we normalize the volumes so that their mean is zero for each music piece.

5.4.4 Pitch Bending

The midi standard contains special note-modifying event messages that can express tones of arbitrary frequencies. Such messages are usually used to imitate *glissando* (i.e., a continuous slide from one pitch to another). We use only discrete pitches in the twelve-tone equal temperament tuning (Section 2.1.2), so we approximate each glissando by dividing it into parts, which are then rounded to the nearest note in that tuning.

Chapter 6

Generative Model

The previous chapter introduced some simplifications done to the raw input data. We will follow this by describing the final format that serves as the input and output of our generative model. But before that, we give a brief overview of the three parts our model consists of. Then in the last section, we explain and discuss the architecture of recurrent neural networks used as the generative model.

6.1 Model Overview

The generative model consists of three modules: **Chord Predictor**, **Note Predictor** and **Volume Predictor**. Chord Predictor is used to generate the underlying chords, one chord per beat. It is trained on data obtained from the chord detector. The most important module is Note Predictor, which generates **events** – either **note-ons**, **note-offs** or **shifts** in time; the meaning of the events is explained in the following section. The Note Predictor accepts a chord outputed from the Chord Predictor and the last event to generate a new event. If the event constitutes a start of a new note, then the Volume Predictor is used to assign a volume to that note.
Each module processes information in a sequential manner – it predicts a new state based on all of the previous states. Hence we use an LSTM network as the base of each module. This decision is more thoroughly discussed in Section 6.3.

6.2 Input and Output Format

6.2.1 Chord Predictor

The chord detector from Section 5.3 annotates each beat with one of the 24 possible chords – 12 major and 12 minor. We also create a special "chord" representing the end of a song. We can thus represent a chord as an integer from [0,24] on the input and as a vector of length 25, where each chord is encoded in *one-hot* vector representation,[1] on the output.

[1]Encoding of a state, where every bit is set to 0 except exactly one bit set to 1. Encoding n different states therefore needs a vector of length n.

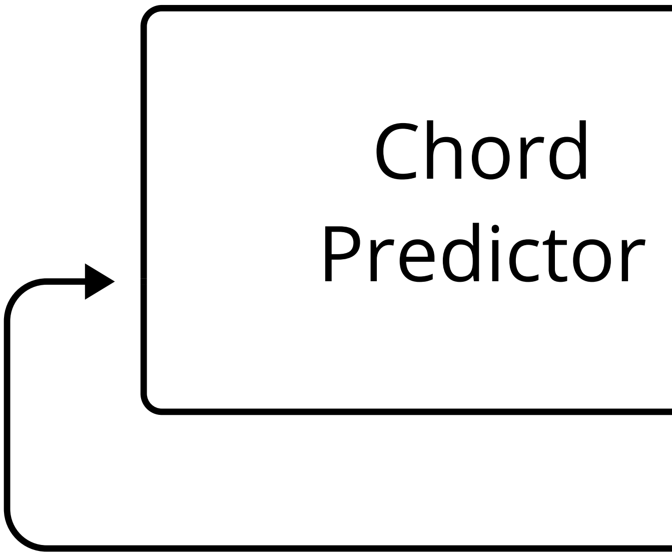

Figure 6.1 *Three connected modules of the generative model*

6.2.2 Note Predictor

Note Predictor generates all the notes that should be played. Each note consists of an onset-time, an end-time, a pitch and of an instrument that plays the note. Our proposed representation of notes, suitable for recurrent networks, was loosely inspired by the MIDI format, which encodes the musical score as a sequence of event messages (see Section 2.2 about MIDI format). Hence the input/output of Note Predictor is a stream of **events**.

The **timer** of the stream is set to 0 in the beginning. Specific **time-shift** can change it: **Short Shift** shifts the timer 50 ms ahead (corresponds to $\frac{1}{12}$ of a beat) and **Large Shift** shifts the timer 300 ms ahead (corresponds to $\frac{1}{2}$ of a beat).[2] The end of a piece is represented by **End of Song** control event, which also resets the time back to 0. Although the Large Shift event is superfluous, it effectively decreases the number of events needed to encode a piece, which lets Note Predictor remember longer time intervals.[3]

[2]The Large Shift event is allowed to happen only at the beginning and at one half of a beat; otherwise the generative model did not seem to understand the rhythmic structure that well.

[3]The idea of having two different time-shift events comes from work by the Google Magenta team [Simon and Oore, 2017] that use 100 of them (ranging from 10 ms up to 1 000 ms). Their main goal is to represent deliberate tempo changes by piano performers with high precision.

Note-on and **note-off** events encode each note in a way that every instrument and every pitch has its distinct pair of note-on/off events. Their order in the stream – with respect to time-shift events – directly encodes their onset time. Consecutive events happening at the same time are ordered so that the encoding is unambiguous and less complicated for the neural network: all note-off events come before note-on events, then the respective note events are ordered by their instruments and then by their pitches.

This encoding allows the recurrent neural network to generate music with an arbitrary number of simultaneous voices while performing only one-class classification. One disadvantage is that it needs two events per each note and many time-shift events to move forward in time, which make the encoding unnecessarily lengthy. The following table shows all 865 events used in the encoding.

Note Predictor accepts two inputs: an id of the last event (as described above) and also an id of the chord that will be played in the next beat. We offset the chords in this way, so that Note Predictor can react to note changes in advance and create a natural transition. The output of the module is a one-hot encoding of an event (a vector of length 865).

From	**To**	**Note-on Events**	**From**	**To**	**Note-off Events**
0	48	Pianos	431	479	Pianos
49	82	Acoustic guitars	480	513	Acoustic guitars
83	124	Orchestral instruments	514	555	Orchestral instruments
125	151	Bass instruments	556	582	Bass instruments
152	193	Brass instruments	583	624	Brass instruments
194	220	Warm pads	625	651	Warm pads
221	261	Electric guitars	652	692	Electric guitars
262	298	Organs	693	729	Organs
299	340	Pitched percussion	730	771	Pitched percussion
341	388	Percussion	772	819	Percussion
389	430	Violin instruments	820	861	Violin instruments

Number	**Control Events**
862	Short Shift (50 ms)
863	Large Shift (300 ms)
864	End of Song

Discussion of Other Methods

We have also experimented with more compact means of event representation; one such representation can, for example, encode the on/off state in a one-hot encoding of 2 bits, the note pitch in 61 bits and the instrument cluster in another

However, the experiments on our dataset suggests that a neural network cannot easily understand the arithmetic relationships between those events and therefore fails to keep a steady rhythm.

11 bits. Although this substantially decreases the dimensionality of the input and output, the neural network does not seem to understand it that well. Moreover, since the network uses an embedding layer, it can learn a preferable representation with a lower dimension by itself.

The lengthiness of the encoding can be easily avoided when considering only monophonic music. For example, Mozer [1994] encoded the musical pieces as a sequence of *notes* (each note has a pitch value and length) – similar to how one writes a musical score. Therefore he encoded each note in only one event and did not need any time-shift events.

When considering polyphonic music, one possible solution is to drop from the one-hot encoding and instead sum up all simultaneous notes into one message in the stream.[4] Then the timer is automatically incremented with each message. A similar representation was used for example by Žídek [2017] in his work. Although this approach considerably reduces the length of an input, it has two significant issues: For one, the neural network model has to perform a multi-label classification (classify m classes out of n) instead of a simpler one-class classification (classify 1 class out of n) – since there are 2^n possible outputs instead of n, the neural network cannot sufficiently estimate the hidden distributions and converge that well. The second issue comes with sampling from a trained model. Such model (usually) outputs the probability of each of n notes, which makes it unclear how to decide how many and what notes should be played. For example, it is impossible to differentiate between two notes that should be played together and two notes that represent two incompatible extensions of a melodic line with similar probability.

6.2.3 Volume Predictor

The input of Volume Predictor are the same events as in the case of Note Predictor. Volume Predictor should assign a volume (or more specifically a normalized real volume computed in the preprocessing chapter) to each note-on event. All other events have an *undefined* volume attached. Although no loss is counted for the events with *undefined* volumes, they are still fed into the network so that it can develop a better understanding of the context.

6.3 Neural Network Architecture

6.3.1 General Description

This section describes the neural network architecture of Note Predictor module. Nevertheless, the other two modules use a similar architecture, only slightly different inputs and outputs (as explained in the section above). Since the Volume Predictor is trained on a regression task (instead of classification), we will cover its differences in a separate section.

[4]For example instead of three messages encoding three notes played in one time step (001000, 010000, 000001), we could transfer the same information in only one event (011001).

The architecture consists of an embedding layer, k recurrent LSTM layers with recurrent batch normalization and a final fully-connected layer, which is connected to a softmax activation function.

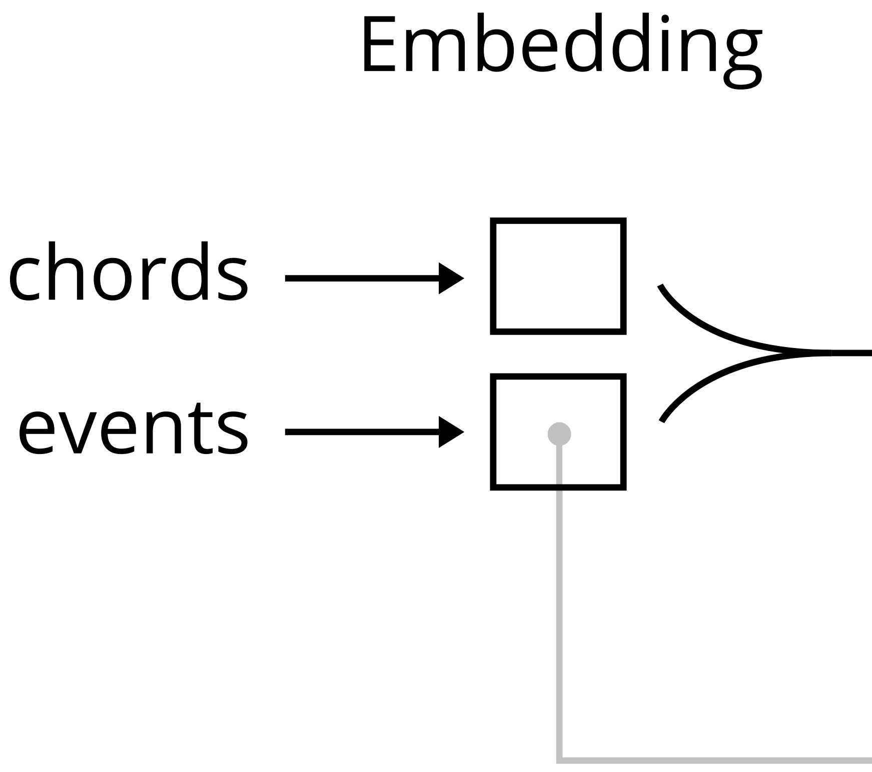

Figure 6.2 *Illustration of different neural layers that form Note Predictor.*

The **embedding layer** learns a compact vector representation of the input events. It maps each of the n events to a d-dimensional vector of real values, which is internally represented by a $n \times d$ matrix. We use the embedding layer instead of a simple one-hot encoding because embedding representation is denser and trained specifically to provide informational in an optimal way for the next layers. Since Note Predictor accepts two independent inputs, it uses two different embedding layers of sizes d_{event} and d_{chord} and then concatenates the embedded outputs into one vector of dimension $d = d_{event} + d_{chord}$.

We apply the so-called **dropout** technique [Srivastava et al., 2014] to the embedded outputs to regularize them for a better generalization. The dropout masks out a portion of an output vector to force the embedding layer to develop a more general representation (that works even if some of its elements are *dropped out*).

The d-dimensional vector from the embedding layers is then fed into k **LSTM layers**. We keep the number of units per each layer the same and denote it as h. We do not use the traditional LSTM layers (described in Chapter 2), but LSTM layers with recurrent batch normalization proposed by Cooijmans et al. [2016].

Batch normalization is a regularization technique that speeds up the training and helps to generalize better [Ioffe and Szegedy, 2015]. The normalization addresses one major problem of training a multi-layer network – while a layer is changing its output distribution during training (by updating its weights), the following layer cannot make any progress in learning. The batch normalization fixes the mean and the variance of each layer's output and thus makes the output distribution more stable. Its use is widespread for convolutional and feed-forward neural networks, but its implementation for recurrent networks is not straightforward due to their sequential nature; Cooijmans et al. [2016] applied the normalization to the hidden-to-hidden transition at each time step and reached better convergence rates than with the traditional LSTMs.

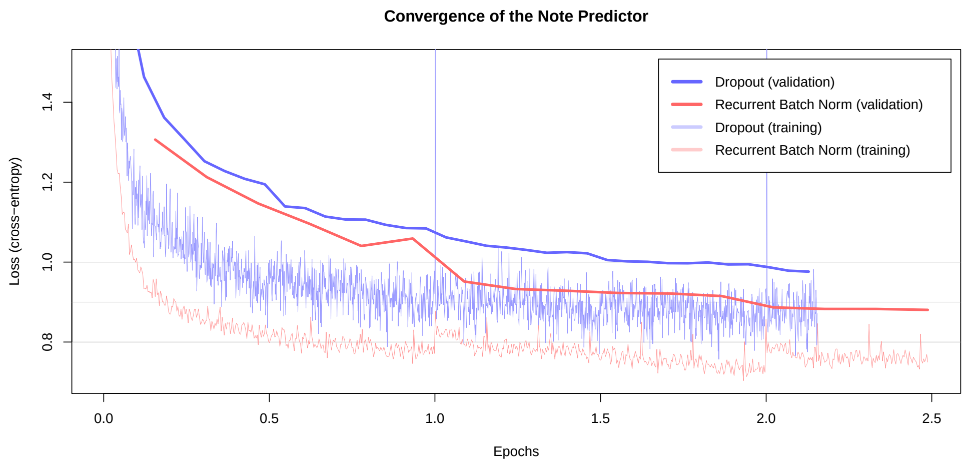

Figure 6.3 Red: *Convergence of Note Predictor with hyperparameters provided in Section 6.3.6. The thick lines show the cross-entropy loss on the validation set and the thin one show the loss on the training set.* Blue: *Convergence of Note Predictor when not using batch normalization but 10% dropout for regularization of LSTM layers. Although LSTM with batch normalization takes considerably longer to run, it converges in fewer steps and leads to more accurate results.*

The last LSTM layer outputs a h-dimensional vector that is converted into a n-dimensional vector (where n is the number of events to be classified) by a **fully-connected feed-forward layer** without any activation. Press and Wolf [2016] noticed, that when $d = h$, the output layer can be interpreted as the inverse of the embedding layer. Therefore they suggest using the same weight matrix for both of these layers, which should lead to fewer parameters and better generalization of the network. We use this technique in Chord and Note predictor and hence set $d = h$.

The output from the last fully-connected layer is converted to a probability distribution of events by a **softmax** activation function. Softmax is a function $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ given by $\sigma(x)_i = \frac{exp(x_i/T)}{\sum exp(x_j/T)}$, where T is a parameter called **temperature**. Softmax rescales an input vector so that it represents a categorical distribution; the temperature T influences the *softness* of the scaling – high temperature makes the distribution more uniform (the output is completely uniform when $T \to \infty$), and low temperature makes the maximal value more probable.

6.3.2 Training

We use the multi-class cross entropy loss as the loss function, which is given by $L_{CE}(c^*, p) = -log(p_{c^*})$, where c^* is the an index of a correct event and p_{c^*} is the probability of that event as provided from the softmax activation function.

The weight parameters are updated by the Adam optimizer – a variant of the stochastic gradient descent algorithm introduced in Chapter 2, which usually converges faster and is more robust against falling into local minima. Its main idea is to use *momentum* to regulate the direction of the current gradient by previous gradients and to change the learning rate (step size) adaptively. Please refer to the original paper by Kingma and Ba [2014] for more information.

6.3.3 Sampling

After the model is trained, we would like to generate a new composition by an iterative random sampling of new events, which are then used as an input in the next iteration. Since we are sampling directly from the categorical distribution provided by softmax, temperature T is an important parameter influencing the randomness of the generated composition.

One issue with sampling is that we have to initialize the inner states of the recurrent networks somehow. There are three possible ways how to do it: The first one is to set them to some constant value, which however creates very similar pieces and it as also unclear how to determine the right value. The second way is to initialize the inner states randomly to create diverse pieces. The problem is that the space of the inner states is (probably) very segmented and discontinuous, which is why the model often generates some random chaos as it does not know what the inner states represent.

The third way is to initialize the inner states with some other song – we call this process **priming** and the original song **primer**. Feeding the primer into the modules' networks sets their states to an internal representation of the primer. The advantage/disadvantage is that the model would then improvise in the style of the primer. The most important feature of this approach is that the inner states are *valid*, so the samples are not chaotic, but listenable.

Nevertheless, since the sampling is stochastic, the randomness may introduce a series of errors that lead to an invalid inner state. The invalid inner state then produces a chaotic composition that only intensifies the problem. This phenomena usually occurs after several minutes of sampling (single-instrumental music is almost immune to this, but the probability of falling into an invalid inner state raises as more instruments are played simultaneously), and we think it is inevitable when we sample stochastically, because the recurrent networks are not trained to learn the appropriate continuous inner state representation. We believe that solving this issue (maybe by randomizing or restricting the inner states during training) is an interesting and beneficial future work applicable outside artificial music composition.

6.3.4 Differences of the Volume Predictor

The Volume Predictor uses the same architecture as described above, but its final fully-connected layer has only one output, that is then directly used as the output volume (without any activation function). Since predicting a real volume is a regression task, we use the mean squared error loss instead of cross entropy. The predictor is also trained by the Adam optimizer.

6.3.5 Discussion of Other Architectures

As explained in the Introduction and Background, we opted to use LSTM instead of simple RNN, because it converges to better results. However, many variants of RNN with similar properties as LSTM exist. The best-known alternative is Gated Recurrent Unit (GRU) [Cho et al., 2014], which uses fewer parameters than LSTM and can thus generalize better on some tasks. Nevertheless, Nayebi and Vitelli [2015] has shown that GRU is not as suitable for music composition as LSTM. Zoph and Le [2016] used reinforcement learning techniques to artificially design an optimal RNN unit for natural language generation. Although they claim their *NASCells* are better than LSTM cells, we have not experimented with them due to time constrains. We believe that using NASCells or even designing a cell optimized directly for music generation is a future work that can result in a better performance of the whole architecture.

Other models for musical composition often use highly specialized architectures, for example, the state-of-the-art model combines a deep belief network (DBN) with LSTM [Vohra et al., 2015]. However, its performance is measured only on a small dataset (hundreds of examples), and we believe that the combination with DBN is used mostly as a regularization technique that would not be beneficial on our large dataset due to its higher computational demands. Johnson [2017] suggests integrating LSTM and a convolutional neural network to create a transition-invariant model – although such architecture seems to be suitable for one-instrument music, the transition-invariance is of no use for us, because we have already transposed the training data to a unified key. Moreover, it is not apparent how to generalize the architecture for multi-instrumental music.

Bai et al. [2018] suggest using *dilated convolution networks* (practically used in WaveNet [Van Den Oord et al., 2016] architecture for example) instead of RNNs for sequential modeling tasks because the convolution networks can be better parallelized,[5] which makes their training faster on modern highly parallel GPUs. However, our experiments imply that such networks are not as suitable for music composition as recurrent networks, because they are not able to capture long dependencies that well. Another architecture for more efficient dealing with sequence modeling has been proposed by Vaswani et al. [2017]. They use an *attention mechanism* to create the state-of-the-art model for natural language translation that is also faster to train than previous attempts based on LSTM units. We believe that exchanging RNN for a network using attention has great potential as future work.

6.3.6 Implementation and Hyperparameters

We used a Python deep learning library PyTorch for implementing the modules. Although PyTorch is still in a beta state, we decided to use it because it is open-source, has GPU support, its design is intuitive and is faster than the rival libraries [Stefbraun, 2018].[6]

We tried to select the optimal hyperparameters for each module, but the search had to be very limited due to significant computational demands of the recurrent neural networks (most notably, training Note Predictor for one epoch took around one and a half day on NVIDIA Tesla K80 with 8.73 teraFLOPS of single-precision computational power).

The hyperparameters used in the final **Note Predictor** are: `batch_size = 256`, `dropout = 10 %`, `gradient_clipping = 0.25`,[7] `d`$_{chord}$ `= 12`, `hidden_size = h = d`$_{event}$ `= 800`, `k = lstm_layers = 3`, `initial_learning_rate = 0.003` (divided by 4 if the validation error got worse after an epoch), `sequence_length = 120`[8] and `temperature = 0.95`.

The hyperparameters used in the final **Chord Predictor** are: `batch_size = 128`, `dropout = 10 %`, `gradient_clipping = 0.25`, `d`$_{chord}$ `= 16`, `hidden_size = h = 32`, `k = lstm_layers = 2`, `initial_learning_rate = 0.006` (divided by 2 if the validation error got worse after an epoch), `sequence_length = 100` and `temperature = 1.0`.

The hyperparameters used in the final **Volume Predictor** are: `batch_size = 128`, `dropout = 10 %`, `gradient_clipping = 0.25`, `hidden_size = d`$_{event}$ `= h = 256`, `k = lstm_layers = 2`, `initial_learning_rate = 0.003` (divided by 2 if the validation error got worse after an epoch) and `sequence_length = 80`.

[5]RNNs are unrolled in time during training, which creates a very deep, unparallizable network, that is demanding both computationally and memory-wise.

[6]Most notably faster than the currently-most-popular deep learning library Tensorflow. We also believe that Tensorflow's reliance on static computation graph definition is a bad design decision, because it makes debugging much harder without bringing any substantial advantage against PyTorch's Dynamic Computational Graph. A good comparison between PyTorch and Tensoflow can be found in Dubovikov [2017]

[7]We limit the maximum length of the gradient vector to this value to avoid problems with exploding gradients.

[8]This represents the depth of an unrolled network in time, i.e. how distant relationships are directly modeled during training.

Please refer to the attached source codes with documentation for more detailed information about the actual implementation,

Chapter 7

Survey

The quality of an artificially composed musical piece is challenging to measure automatically. Even the notion of *quality* is hard to properly define since it is subjective in its essence. The cross-entropy loss used to train the models is not a very meaningful measure of quality. Hence we have conducted an online survey to at least estimate it.

First, we state the goals we would like to answer and then the form of the survey based on them. The last section contains a discussion and an interpretation of the results.

7.1 Goals of the Survey

We would like to answer these questions:

A) Is the music composed by our algorithm distinguishable from real human-composed music?
B) Are there any differences in the *quality* of these two types of music?
C) Are the answers different when considering only simpler single-instrumental music?
D) Has the neural network model overfitted the training data?

7.2 Definition of the Survey

First, we describe how a representative set of pieces was created, then how we selected a subset that was shown to respondents and what we asked them to do. Next, we explain how it could answer the goals mentioned above.

We have randomly selected 30 multi-instrumental and 30 single-instrumental pieces and removed those which we thought are well-known. After this process, we ended up with 24 multi-instrumental and 28 single-instrumental pieces. Then we used these songs as primers and generated improvisations on them with the same hyperparameters as described in Section 6.3.6. The single-instrumental pieces were created by the same model as the multi-instrumental, but the output of the model was masked in a way that all instruments except one had probability 0.

In that way, we gathered 104 pieces in total. Finally, we cut a random 30-second sample from each piece. The sample should start at least 30 seconds after the priming point and should not end more than 2 minutes after this point.[1] One of the reasons to shorten the length is to make the survey briefer and thus more accessible. The other consequences are discussed in the last section.

Each respondent listened to 10 pieces out of 104 that were chosen randomly without a repetition. The randomness should minimize any bias caused by non-representative samples of songs or misleading order of songs. The respondents could repeat each song without any limitation and go back to an already rated song. They were asked two questions regarding each song:

1. Do you think this song was created by a human or by a computer?
2. How do you like this song on the scale from 1 (bad) to 5 (excellent)?

There was also an option to skip a song if it seemed to be familiar to them. In addition to these direct questions, we measured for how long have they been listening to each song before answering.

The first question is a "Turing test", that should provide an answer for the goal A. Similarly, the primary use of the second question is to answer the goal B. The goal C can be answered by comparing the results of single-instrumental and multi-instrument pieces. When the generative model overfits, it will likely produce well-known melodies, because of their abundance in the training data. We can thus use the option to skip a song to answer the goal D.

7.3 Results

We gathered answers from 293 respondents, which should serve as a representative sample for a meaningful evaluation.

A) Is the music composed by our algorithm distinguishable from real human-composed music?

The accuracy, at which people were able to distinguish between real and generated music, is 62.6 %. The precision[2] of the answers is 62.9 %, sensitivity is 63.5 % and specificity 61.6 % – confusion matrix of the answers is provided below.

The average play times of the rated pieces were very similar – real music was listened to on average for 23.1 seconds before answering, while the generated music for 24.9 seconds. The small difference suggests that the respondents were equally certain no matter what music they were listening to.

We can also calculate the accuracy per each sample; since every generated piece was primed from a real song and both of them were put into the survey, we can determine the correlation between those pairs. The results are provided in the figure below. The scatter plots show that there is indeed a correlation between the quality of the original and the improvised pieces.

[1]If the original song is too short to fulfill these requirement, then we allow it to start sooner. If the generated song is too short, we try to generate it again.

[2]Precision is the ratio between correctly guessed computer pieces and all pieces rated as "computer". Sensitivity is the ratio between correctly guessed computer pieces and all computer pieces. Specificity is the ratio between correctly guessed human pieces and all human pieces.

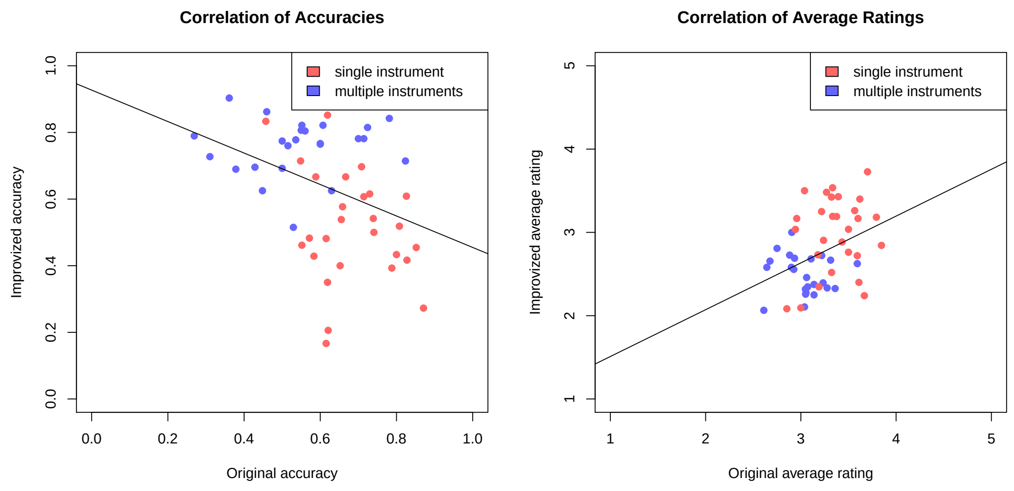

Figure 7.1 Left: *correlation between the accuracies of original pieces and their improvised counterparts. High accuracy of an original piece means that many respondents thought it was authentic; likewise in the case of low accuracy on improvised pieces.* Right: *correlation between average ratings of original and improvised pieces; the quality of original pieces positively correlates with the quality of improvised pieces. It is also evident, that the single-instrumental pieces were rated higher than the multi-instrumental ones.*

The 62.6% accuracy lies out of any reasonable confidence interval around 50 %, so the hypothesis that the generated music is indistinguishable is clearly disproved. However, we believe that 62.6 % is a reasonable result meaning that the generated music was not easy to distinguish but occasional mistakes sometimes revealed its origin.

It is important to note that the respondents were provided with only 30-second samples from the full-length songs. The results should be therefore interpreted with this fact on the mind. The generative model is not able to understand and use any musical phenomena spanning more than tens of seconds; thus the full-length samples would presumably be recognized with much higher accuracy. We conclude that the current state of neural networks is still not able to generate music with a clear high-level structure (this is a general problem of deep learning, it is the reason why even state-of-the-art machine translators cannot share context between sentences).

Another note has to be made about the similarity of a primer and its improvisation – if the improvisation were too similar, the results would not have any real validity. We have carefully listened to all the samples and conclude that although the pairs usually share a similar style, they always sounded as two distinct songs (as can be expected, given the short memory issue discussed above). We provide all the samples in the attachment, so the reader can evaluate the similarity and quality of the samples on its own.

	All		**Single**		**Multi**	
	“computer”	“human”	“computer”	“human”	“computer”	“human”
computer	929	533	395	361	534	172

	All		Single		Multi	
	"computer"	"human"	"computer"	"human"	"computer"	"human"
human	548	880	244	531	304	349

Table 7.1 *Confusion matrices of answers to the first question "Do you think this song was created by a human or by a computer?". Rows display the ground truth, and the columns show the actual answers. The left table presents confusion matrix of all pieces, the middle of single-instrument pieces and the right of multi-instrument pieces.*

B) Are there any differences in the *quality* of these two types of music?

The average rating of human-made songs is 3.216; the generated songs have the mean rating of 2.745. The drop in quality is noticeable, but not significant and corresponds with the aforementioned accuracy. Although the model is not perfect, we argue that it successfully managed to compose listenable music.

C) Are the answers different when considering only simpler single-instrumental music?

The accuracy at which people correctly guessed the origin is similar for both types of music: single-instrumental pieces have the accuracy of 60.5 % and multi-instrumental pieces have the accuracy of 65.0 %. More substantial difference can be seen in the sensitivity of the answers: one-instrument pieces have the sensitivity of 52.25 %, while multi-instrument of 75.64 %.

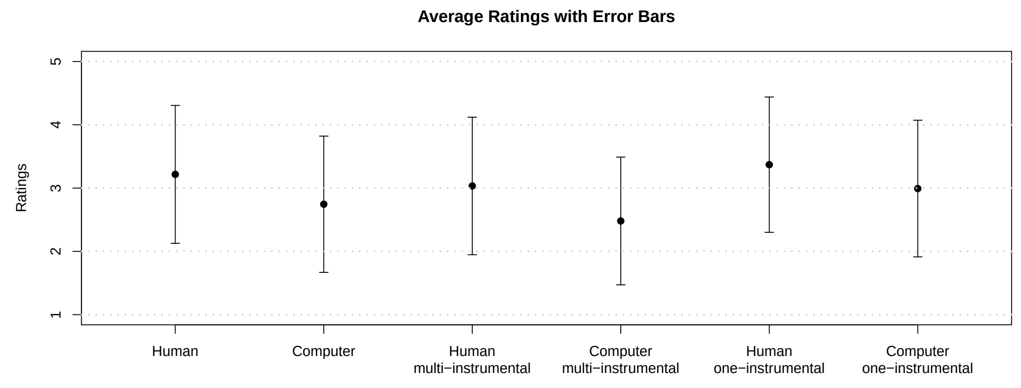

Figure 7.2 *Average ratings (dots) of various subset of the 104 evaluated pieces, error bars show the standard deviation.*

These statistics suggest that the multi-instrumental pieces sound artificial despite their origin, because people rated them as "computer" more often. The model is also more prone to mistakes when composing multi-instrumental music – this can be concluded from the 4.5% difference in accuracies, from the scatter plots in Figure 7.1 and also from the average ratings shown in Figure 7.2. The last plot shows that although there is a drop in quality between single-instrumental and multi-instrumental music, it is not significantly larger than the drop seen in the real music. More detailed view on differences between samples is provided in Appendix B.

D) Has the neural network model overfitted the training data?

From the total of 2,930 answers, only 40 of them selected the rated piece as familiar. Out of this, only 8 computer pieces were selected as such. Since our dataset contains many duplicates of well-known songs, the model should overfit to them and produce familiar melodies, if it were prone to overfitting. This (together with the loss function of the neural network in Figure 6.3) suggests that the model is not overfitted and generalizes well.

Chapter 8

Conclusion

The goal of this work is to develop a program capable of composing polyphonic music with multiple instruments playing simultaneously. We hope this thesis shows that it is indeed possible to let computers compose music. Although the output is not perfect and is limited to a symbolic representation, it can create listenable compositions that are hard to distinguish from human-made music. Since the neural network we use is still rather simple, we strongly believe that one day, with advancements in computational power and deep learning research, computers will be able to compose music on the same level as humans.

When we began experimenting with generating multi-instrument music, we quickly found out that it is impossible to train a plausible model with the current neural networks on raw, unprocessed data. For this reason, we started studying literature about automated musical analysis and then developed algorithms for restricting the musical space according to it. These algorithms are described in Chapter 5. We conclude that music is structurally very similar to natural language, but considerably less studied, unfortunately. Developing a musical analyzer was not only difficult for lack of data, but also for lack of exact music theories.

The neural network architecture that uses the preprocessed data to create new music is described in Chapter 6. The architecture is based on modern deep learning techniques and on approaches used by others to solve similar problems.

In order to evaluate the performance of the proposed model, we have conducted an online survey described in Chapter 7. The results show that there is still some space for enhancement, but overall the artificially generated music was rated almost as good as the real music and was rather hard to distinguish from it (with 62.6% accuracy).

For a more practical use, the generative model can improvise over a prepared composition, extend an incomplete composition and it should be easily modified to continually join two different compositions.

As a future work, more afford should be put into the architecture of neural networks, because we have focused more on the necessary musical analysis in the preprocessing steps. We have suggested some different neural layers, which are worth a try, at the end of section 6.3.5. In a future research, we would like to try developing an alternative to the three independent modules described in section 6.3.1 – sharing states and weights between the modules or even combining them into one larger module should be beneficial in our opinion. Another thing worth mentioning is the loss function used in Note Predictor – we used cross-entropy loss, a general loss function used in the majority of classification tasks, but a specific loss designed with expert knowledge should result in a noticeable improvement.

As far as musical analysis in preprocessing steps is concerned, the bottleneck seems to be the chord detection algorithm. We have discussed potential improvements mainly in section 5.2.3 for the key detection algorithm. These ideas can be as well used for more accurate detection of chord, which however suffers mainly from the lack of annotated data.

We should also mention, that the dataset described in Chapter 4 contains many pieces labeled by their genre, which is not utilized in our current generative model in any way. One possible use is to transfer genres between pieces. It also seems interesting to label the pieces according to their mood (which is a hard and open problem, however) and then generate music in a specific mood that could be interactively changed.

The most significant problem of all generative models based on deep learning is their short memory (as discussed throughout the whole thesis). One possible solution is to design a rule-based system that would utilize the deep learning models as subroutines for some limited tasks – for example composing a melody or creating an arrangement for given melody and chords. Although such system will be limited to the handcrafted rules, we believe it should be able to create believable and listenable music even with current deep learning techniques.

Bibliography

Alisneaky. Kernel machine.png, 2011. URL `https://commons.wikimedia.org/wiki/File:Kernel_Machine.png`. [Online; accessed May 9, 2018].

MIDI Manufacturers Association et al. The complete midi 1.0 detailed specification. *Los Angeles, CA, The MIDI Manufacturers Association*, 1996.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM. ISBN 0-89791-497-X. doi: 10.1145/130385.130401. URL `http://doi.acm.org/10.1145/130385.130401`.

Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*, 2012.

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL `https://doi.org/10.1023/A:1010933404324`.

L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984. ISBN 9780412048418.

Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation-a survey. *arXiv preprint arXiv:1709.01620*, 2017.

Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57 – 83, 2002. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(01)00129-1. URL `http://www.sciencedirect.com/science/article/pii/S0004370201001291`.

Spencer Campbell. *Automatic Key Detection of Musical Excerpts from Audio*. PhD thesis, McGill University Library, 2010.

Chabacano. Overfitting.svg, 2008. URL `https://commons.wikimedia.org/wiki/File:Overfitting.svg`. [Online; accessed May 9, 2018].

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL `http://arxiv.org/abs/1406.1078`.

Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron C. Courville. Recurrent batch normalization. *CoRR*, abs/1603.09025, 2016. URL `http://arxiv.org/abs/1603.09025`.

Kirill Dubovikov. Pytorch vs tensorflow – spotting the difference, 2017. URL `https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b`. [Online; accessed May 9, 2018].

Douglas Eck and Juergen Schmidhuber. A first look at music composition using lstm recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, 103, 2002.

Ahmed M. Elgammal, Bingchen Liu, Mohamed Elhoseiny, and Marian Mazzone. CAN: creative adversarial networks, generating "art" by learning about styles and deviating from style norms. *CoRR*, abs/1706.07068, 2017. URL `http://arxiv.org/abs/1706.07068`.

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

Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1504. URL `http://www.sciencedirect.com/science/article/pii/S002200009791504X`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Christoph Hausner and Tomas Sauer. Design and evaluation of a simple chord detection algorithm. 2014.

Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 9:1735–80, 12 1997.

Miles Hoffman. *The NPR Classical Music Companion: Terms and Concepts from A to Z*. Houghton Mifflin, 1997. ISBN 9780395707425.

David M. Howard and Jamie A.S. Angus. Copyright. In *Acoustics and Psychoacoustics (Fourth edition)*, pages ii –. Focal Press, Boston, fourth edition edition, 2010. ISBN 978-0-240-52175-6. doi: https://doi.org/10.1016/B978-0-240-52175-6.00008-9. URL `https://www.sciencedirect.com/science/article/pii/B9780240521756000089`.

Hyacinth. Syncopation example.mid, 2013. URL `https://commons.wikimedia.org/wiki/File:Syncopation_example.mid`. [Online; accessed May 9, 2018].

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Jay Glanville James Allwright, Eric Foxley and Seymour Shlien. Abc version of the nottingham music database, 2003. URL `http://abc.sourceforge.net/NMD/`. [Online; accessed May 9, 2018].

Daniel D Johnson. Generating polyphonic music using tied parallel networks. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, pages 128–143. Springer, 2017.

Daniel D Johnson, Robert M Keller, and Nicholas Weintraut. Learning to create jazz melodies using a product of experts. In *ighth International Conference on Computational Creativity, ICCC, Atlanta*, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

Filip Korzeniowski and Gerhard Widmer. End-to-end musical key estimation using a convolutional neural network. In *Signal Processing Conference (EUSIPCO), 2017 25th European*, pages 966–970. IEEE, 2017.

S. Kostka and D. Payne. *Tonal Harmony: With an Introduction to Twentieth Century Music*. McGraw-Hill Education, 2008. ISBN 9780073401355.

Carol L Krumhansl. Cognitive foundations of musical pitch. 1990.

Carol L Krumhansl and Edward J Kessler. Tracing the dynamic changes in perceived tonal organization in a spatial representation of musical keys. *Psychological review*, 89(4):334, 1982.

Fred Lerdahl and Ray S Jackendoff. *A generative theory of tonal music*. MIT press, 1985. ISBN 9780262621076.

B. Liu, J. Fu, M. P. Kato, and M. Yoshikawa. Beyond Narrative Description: Generating Poetry from Images by Multi-Adversarial Training. *ArXiv e-prints*, April 2018.

Chaochao Lu and Xiaoou Tang. Surpassing human-level face verification performance on LFW with gaussianface. *CoRR*, abs/1404.3840, 2014. URL `http://arxiv.org/abs/1404.3840`.

Søren Tjagvad Madsen, Gerhard Widmer, and Johannes Kepler. Key-finding with interval profiles. In *ICMC*, 2007.

Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

Michael C Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science*, 6(2-3):247–280, 1994.

Aran Nayebi and Matt Vitelli. Gruv: Algorithmic music generation using recurrent neural networks. *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*, 2015.

Gerhard Nierhaus. *Algorithmic composition: paradigms of automated music generation*. Springer Science & Business Media, 2009. ISBN 978-3-211-75539-6.

Christopher Olah. Understanding lstm networks, 2015. URL `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. [Online; accessed May 9, 2018].

Geoffroy Peeters, Bruno L. Giordano, Patrick Susini, Nicolas Misdariis, and Stephen McAdams. The timbre toolbox: Extracting audio descriptors from musical signals. *The Journal of the Acoustical Society of America*, 130(5): 2902–2916, 2011. doi: 10.1121/1.3642604. URL `https://doi.org/10.1121/1.3642604`.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *CoRR*, abs/1608.05859, 2016. URL `http://arxiv.org/abs/1608.05859`.

Daniele P. Radicioni and Roberto Esposito. Bach choral harmony data set, 2014. URL `https://archive.ics.uci.edu/ml/datasets/Bach+Choral+Harmony`. [Online; accessed May 9, 2018].

O Sandred, Mikael Laurson, and Mika Kuuskankare. Revisiting the illiac suite–a rule-based approach to stochastic processes. *Sonic Ideas/Ideas Sonicas*, 2: 42–46, 2009.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`.

Ian Simon and Sageev Oore. Performance rnn: Generating music with expressive timing and dynamics. `https://magenta.tensorflow.org/performance-rnn`, 2017.

Susan Sontag. *Reborn: Journals and Notebooks, 1947-1963*. Farrar, Straus and Giroux, 2009. ISBN 9781466812017.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Adam Stark. *Musicians and machines: Bridging the semantic gap in live performance.* PhD thesis, Queen Mary University of London, 2011.

Stefbraun. rnn_benchmarks, 2018. URL `https://github.com/stefbraun/rnn_benchmarks`. [Online; accessed May 9, 2018].

David Temperley. *The cognition of basic musical structures.* MIT press, 2004. ISBN 9780262201346.

Aaron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

R. Vohra, K. Goel, and J. K. Sahoo. Modeling temporal dependencies in data using a dbn-lstm. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–4, Oct 2015. doi: 10.1109/DSAA.2015.7344820.

Yiren Wang and Fei Tian. Recurrent residual learning for sequence classification. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 938–943, 2016.

Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013. URL `http://arxiv.org/abs/1311.2901`.

Veronika Zenz and Andreas Rauber. Automatic chord detection incorporating beat and key detection. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 1175–1178. IEEE, 2007.

Marek Židek. Generování polyfonní hudby pomocí neurovỳch sítí. 2017.

Hans-Georg Zimmermann, Christoph Tietz, and Ralph Grothmann. *Forecasting with Recurrent Neural Networks: 12 Tricks*, pages 687–707. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_37. URL `https://doi.org/10.1007/978-3-642-35289-8_37`.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016. URL `http://arxiv.org/abs/1611.01578`.

List of Figures

Appendix A

Frequencies of Instruments

This appendix presents additional information to occurrences of instruments in our dataset, as discussed in the section about Instrument Clustering.

Instrument	**Cluster**	**Frequency**	**Instrument**	**Cluster**	**Frequency**
Acoustic Grand Piano	1	23,873,793	Soprano Sax	5	359,246
Bright Acoustic Piano	1	3,854,812	Alto Sax	5	927,132
Electric Grand Piano	1	990,681	Tenor Sax	5	675,620
Honky-tonk Piano	1	633,306	Baritone Sax	5	282,161
Electric Piano 1	1	3,174,243	Oboe	5	734,043
Electric Piano 2	1	2,225,095	English Horn	5	139,451
Harpsichord	1	1,406,784	Bassoon	5	493,330
Clavinet	1	535,270	Clarinet	5	1,002,004
Celesta	9	208,520	Piccolo	5	291,300
Glockenspiel	9	132,791	Flute	5	1,358,280
Music Box	9	209,397	Recorder	5	262,492
Vibraphone	9	907,240	Pan Flute	5	391,009
Marimba	9	212,284	Blown Bottle	5	46,897
Xylophone	9	46,102	Shakuhachi	5	94,948
Tubular Bells	9	110,055	Whistle	5	120,340
Dulcimer	9	61,108	Ocarina	5	199,480
Drawbar Organ	8	943,598	Lead 1 (square)	5	291,953
Percussive Organ	8	1,185,942	Lead 2 (sawtooth)	5	604,899
Rock Organ	8	1,473,007	Lead 3 (calliope)	5	281,179
Church Organ	8	779,268	Lead 4 (chiff)	5	57,882
Reed Organ	8	242,730	Lead 5 (charang)	7	158,600
Accordion	8	439,123	Lead 6 (voice)	6	292,407
Harmonica	8	772,586	Lead 7 (fifths)	5	30,415
Tango Accordion	8	188,782	Lead 8 (bass + lead)	5	334,301
Acoustic Guitar (nylon)	2	5,801,167	Pad 1 (new age)	6	578,471
Acoustic Guitar (steel)	2	22,166,911	Pad 2 (warm)	6	1,839,644
Electric Guitar (jazz)	2	2,122,494	Pad 3 (polysynth)	6	492,069
Electric Guitar (clean)	2	3,306,754	Pad 4 (choir)	6	344,976
Electric Guitar (muted)	2	1,185,240	Pad 5 (bowed)	6	189,963
Overdriven Guitar	7	2,780,639	Pad 6 (metallic)	6	137,256
Distortion Guitar	7	3,901,458	Pad 7 (halo)	6	292,575
Guitar harmonics	7	69,189	Pad 8 (sweep)	6	327,202
Acoustic Bass	4	2,530,446	FX 1 (rain)	6	69,606
Electric Bass (finger)	4	12,919,009	FX 2 (soundtrack)	6	77,656
Electric Bass (pick)	4	910,327	FX 3 (crystal)	6	75,060
Fretless Bass	4	2,428,206	FX 4 (atmosphere)	6	283,853
Slap Bass 1	4	229,313	FX 5 (brightness)	6	236,728
Slap Bass 2	4	225,629	FX 6 (goblins)	6	72,534
Synth Bass 1	4	510,305	FX 7 (echoes)	6	126,060

Instrument	**Cluster**	**Frequency**	**Instrument**	**Cluster**	**Frequency**
Synth Bass 2	4	450,959	FX 8 (sci-fi)	6	126,582
Violin	10	882,987	Sitar	2	85,635
Viola	10	338,438	Banjo	2	356,601
Cello	10	608,288	Shamisen	2	39,396
Contrabass	10	298,718	Koto	2	101,156
Tremolo Strings	3	312,323	Kalimba	9	46,569
Pizzicato Strings	9	348,693	Bag pipe	8	51,858
Orchestral Harp	2	636,256	Fiddle	10	161,120
Timpani	9	191,248	Shanai	5	22,301
String Ensemble 1	3	8,630,252	Tinkle Bell	9	21,965
String Ensemble 2	3	5,013,596	Agogo	9	13,937
Synth Strings 1	3	3,359,927	Steel Drums	9	72,485
Synth Strings 2	3	901,647	Woodblock	9	35,539
Choir Aahs	6	2,267,499	Taiko Drum	9	61,746
Voice Oohs	6	789,188	Melodic Tom	9	38,698
Synth Voice	3	637,837	Synth Drum	9	47,925
Orchestra Hit	9	45,996	Reverse Cymbal	9	50,799
Trumpet	5	1,262,912	Guitar Fret Noise	none	19,994
Trombone	5	1,039,447	Breath Noise	none	15,321
Tuba	5	343,244	Seashore	none	107,119
Muted Trumpet	5	128,025	Bird Tweet	none	14,135
French Horn	5	1,156,509	Telephone Ring	none	12,650
Brass Section	5	962,624	Helicopter	none	15,637
Synth Brass 1	5	538,725	Applause	none	50,734
Synth Brass 2	5	295,044	Gunshot	none	26,836

Table A.1 *The first column shows names of instruments according to the MIDI standard, the second column contains IDs of instrument clusters of each instrument; the IDs are mapped to their actual names in the table below. The third column consists of frequencies of note – how many notes in our dataset is played with each instrument.*

Cluster	**Cluster Name**
1	Pianos
2	Acoustic guitars
3	Orchestral instruments
4	Bass instruments
5	Brass instruments
6	Warm pads
7	Electric guitars
8	Organs
9	Pitched percussion
10	Violin instruments
11	Percussion

Appendix B

Ratings of Songs in Survey

This attachment shows statistics of every song from the survey (Chapter 7). The table is horizontally divided into two parts so that each row correspond to two versions of a piece – an original one and an improvised one. The first column of each part corresponds to the song ID (that match to names of songs on the attached DVD), the second column to the number of ratings, the third column to the prediction accuracy, the fourth column to the average time of listening before answering and the fifth column corresponds to the average rating.

ID	#	ACC	TIME	RATING	ID	#	ACC	TIME	RATING
1	20	70.00%	18.19	3.05	53	32	78.13%	22.50	2.26
2	21	71.43%	24.51	2.90	54	32	78.13%	21.18	3.00
3	29	31.03%	17.51	3.11	55	22	72.73%	22.14	2.68
4	28	60.71%	19.88	2.93	56	28	82.14%	21.43	2.56
5	29	44.83%	24.42	3.14	57	40	62.50%	22.14	2.38
6	28	53.57%	18.19	3.07	58	27	77.78%	25.08	2.35
7	25	56.00%	24.36	3.36	59	46	80.43%	25.42	2.33
8	37	45.95%	21.65	3.05	60	29	86.21%	18.11	2.28
9	29	72.41%	21.79	3.28	61	27	81.48%	19.92	2.33
10	17	82.35%	26.88	3.31	62	21	71.43%	25.07	2.67
11	24	50.00%	21.52	2.75	63	26	69.23%	23.07	2.81
12	32	78.13%	26.40	3.22	64	19	84.21%	23.88	2.72
13	36	36.11%	25.55	2.61	65	31	90.32%	21.65	2.06
14	30	60.00%	26.80	3.23	66	34	76.47%	25.92	2.39
15	37	37.84%	19.75	2.68	67	29	68.97%	23.66	2.66
16	17	52.94%	21.98	2.88	68	33	51.52%	22.37	2.73
17	30	60.00%	26.58	2.93	69	30	76.67%	20.19	2.69
18	33	51.52%	21.90	3.06	70	25	76.00%	26.82	2.46
19	20	55.00%	20.68	2.90	71	31	80.65%	24.65	2.58
20	21	42.86%	23.32	3.05	72	23	69.57%	27.16	2.32
21	29	55.17%	22.24	3.14	73	28	82.14%	23.25	2.25
22	26	26.92%	25.28	3.04	74	38	78.95%	23.53	2.11
23	27	62.96%	21.16	3.59	75	24	62.50%	23.13	2.63
24	28	50.00%	19.70	2.64	76	31	77.42%	23.71	2.58
25	31	77.42%	19.04	3.67	77	26	92.31%	20.13	2.24
26	24	70.83%	23.31	3.00	78	33	69.70%	21.16	2.09
27	24	58.33%	19.63	3.38	79	21	42.86%	27.40	3.19
28	27	74.07%	22.37	3.59	80	28	50.00%	29.46	2.72
29	21	61.90%	17.65	3.19	81	27	85.19%	21.61	2.35
30	17	58.82%	22.53	3.50	82	21	66.67%	25.97	2.76
31	26	61.54%	26.09	3.04	83	24	16.67%	19.38	3.50
32	23	73.91%	19.17	3.22	84	24	54.17%	26.82	3.25
33	28	71.43%	25.84	3.39	85	28	60.71%	24.83	3.43

ID	#	ACC	TIME	RATING	ID	#	ACC	TIME	RATING
34	25	80.00%	18.13	3.60	86	30	43.33%	23.42	3.17
35	23	65.22%	20.57	2.96	87	30	40.00%	25.67	3.17
36	24	66.67%	22.48	3.24	88	21	66.67%	29.97	2.90
37	31	87.10%	24.09	3.70	89	22	27.27%	19.42	3.73
38	26	61.54%	24.62	3.27	90	27	48.15%	25.45	3.48
39	41	65.85%	20.74	3.61	91	26	57.69%	26.17	2.40
40	26	80.77%	22.96	3.50	92	27	51.85%	26.92	3.04
41	37	72.97%	25.50	3.43	93	26	61.54%	23.18	2.88
42	33	78.79%	21.30	3.33	94	28	39.29%	26.46	3.54
43	35	45.71%	22.33	2.85	95	24	83.33%	26.19	2.08
44	29	82.76%	18.88	3.79	96	24	41.67%	20.26	3.18
45	29	65.52%	24.04	3.18	97	26	53.85%	29.26	2.73
46	27	85.19%	24.72	3.85	98	33	45.45%	27.09	2.84
47	31	54.84%	23.34	3.32	99	28	71.43%	21.79	2.52
48	29	62.07%	21.72	3.32	100	34	20.59%	21.76	3.42
49	29	55.17%	22.61	3.33	101	26	46.15%	24.42	3.19
50	35	57.14%	20.71	2.94	102	29	48.28%	30.42	3.04
51	23	82.61%	23.42	3.57	103	23	60.87%	21.13	3.26
52	21	61.90%	20.44	3.62	104	40	35.00%	25.86	3.40