BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ALGORITHMIC MUSIC COMPOSITION
POČÍTAČEM KOMPONOVANÁ HUDBA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ADAM PANKUCH**
AUTOR PRÁCE

SUPERVISOR **MARTIN KOLÁŘ, M.Sc.**
VEDOUCÍ PRÁCE

BRNO 2020

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2019/2020

Zadání bakalářské práce

22952

Student:	**Pankuch Adam**
Program:	Informační technologie
Název:	**Počítačem komponovaná hudba**
	Algorithmic Music Composition
Kategorie:	Zpracování signálů

Zadání:

1. Seznamte se s hudební teorií, a s počítačem generovanou hudební tvorbou
2. Vytvořte kolekci metod automatického generování hudby, rozdělené podle metodologie
3. S použitím kolekce GuitarPro souborů vytvořte dataset hudby s časově zarovnaným popisem (sloka, refrén, sólo, ...)
4. Seznamte se s dopřednými neuronovými sítěmi
5. Natrénujte neuronovou síť generující skladbu
6. Hudební notaci konvertujte do audio signálů, které prezentujte jako sbírku skladeb

Literatura:

- McLean, A. and Dean, R.T. eds., 2018. *The Oxford handbook of algorithmic music*. Oxford University Press. (dostupná u vedoucího)
- https://en.wikipedia.org/wiki/Algorithmic_composition
- https://sites.google.com/site/sd16spring/home/project-toolbox/algorithmic-music-composition

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Kolář Martin, M.Sc.**
Vedoucí ústavu:	Černocký Jan, doc. Dr. Ing.
Datum zadání:	1. listopadu 2019
Datum odevzdání:	28. května 2020
Datum schválení:	4. prosince 2019

Abstract

The goal of this thesis is to create a system, which is able to generate guitar tracks. This problem consists of two main parts: acquisition of a training dataset and training of a suitable deep learning model. The first part of the problem was solved by series of scripts which filter and transform a set of songs with many instruments in Guitar Pro format to a set of guitar tracks in pianoroll format. The second part of the problem was solved by training a few convolutional and recurrent neural networks on the created dataset of guitar tracks. Guitar tracks generated by these networks were compared to each other and evaluated. Although, the generated tracks are not very harmonic and pleasing to the ear, they show that convolutional networks are more suitable for generation of polyphonic music than other types of neural networks.

Abstrakt

Cieľom tejto práce je tvorba systému schopného generovať gitarové stopy. Tento problém pozostáva z dvoch hlavných častí: získanie trénovacieho datasetu a trénovanie vhodného deep learning modelu. Prvá časť tohto problému bola vyriešená sériou skriptov, ktoré vyfiltrovali a transformovali sadu skladieb s viacerými hudobnými nástrojmi z Guitar Pro formátu na sadu gitarových stôp vo formáte pianoroll. Druhá časť problému bola vyriešená natrénovaním niekoľkých konvolučných a rekurentných neurónových sietí na vytvorenom datasete gitarových stôp. Gitarové stopy generované týmito sieťami boli navzájom porovnané a ohodnotené. Hoci vygenerované stopy nie sú veľmi harmonické a príjemné na vypočutie, ukázujú, že konvolučné siete sú vhodnejšie na generovanie polyfónnej hudby v porovnaní s inými typmi neurónových sietí.

Keywords

algorithmic composition, deep learning, neural networks, Guitar Pro, pianoroll

Kľúčové slová

algoritmická kompozícia hudby, deep learning, neurónové siete, Guitar Pro, pianoroll

Reference

PANKUCH, Adam. *Algorithmic Music Composition.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Martin Kolář, M.Sc.

Rozšírený abstrakt

Cieľom tejto práce je tvorba systému schopného generovať gitarové stopy. Tento problém pozostáva z dvoch hlavných častí: vytvorenie trénovacieho datasetu a následné natrénovanie vhodného deep learning modelu na generovanie hudby.

Prvá časť tohto problému bola vyriešená nasledovným spôsobom. V prvom rade som získal databázu, ktorá obsahovala okolo 55000 skladieb v Guitar Pro formáte. Každý z týchto Guitar Pro súborov obsahoval niekoľko rôznych stôp s odlišnými hudobnými nástrojmi. Guitar Pro súbory tiež obsahujú takzvané *značky* (markers), ktoré označujú rôzne sekcie skladby (napríklad: refrén, sloha, sólo, atď.). Mojím prvým cieľom bolo vytvoriť dataset, ktorý by pozostával z gitarových stôp, ktoré by mali vyznačené jednotlivé sekcie (napríklad: refrén, sólo, atď.). Po analýze databázy 55000 skladieb sa však ukázalo, že takýto dataset nebude možné vytvoriť. Namiesto toho som sa rozhodol vytvoriť 2 datasety, ktoré bolo možné vytvoriť z mojej databázy – dataset refrénov a dataset slôh. Tvorba týchto datasetov bola úspešná. Vytvorený dataset refrénov pozostáva zo 7840 gitarových stôp a dataset slôh pozostáva z 7030 gitarových stôp v pianoroll formáte. Vytvorené riešenie tiež poskytuje jednoduchý spôsob tvorby nových datasetov gitarových stôp z Guitar Pro skladieb.

Druhá časť problému bola vyriešená natrénovaním niekoľkých konvolučných a rekurentných neurónových sietí na vytvorenom datasete refrénov. Celkovo som natrénoval 5 rôznych neurónových sietí. Každá z týchto sietí bola trénovaná po dobu 20 epoch na celom datasete refrénov. Najlepšiu presnosť: 51.6% na trénovacom datasete dosiahla rekurentná LSTM sieť. Po natrénovaní každej siete som vygeneroval niekoľko testovacích gitarových stôp a navzájom ich porovnal. Napriek očakávaniam, vygenerované stopy neboli veľmi harmonické a príjemné na vypočutie. Prekvapivo sa však ukázalo že rekurentná LSTM, ktorá dosiahla najvyššiu presnosť generovala jedny z najhorších výsledkov. Po zhodnotení vygenerovaných stôp som usúdil, že vytvorené konvolučné siete sú vhodnejšie na generovanie polyfónnej hudby v porovnaní s rekurentnými sieťami.

Algorithmic Music Composition

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Martin Kolář, M.Sc.. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Adam Pankuch
May 28, 2020

Contents

Chapter 1

Introduction

Computational creativity is one of the fast growing areas of research in the field of artificial intelligence. This term can be loosely defined as an automated analysis and synthesis of works of art. Unlike many other practical applications of artificial intelligence, such as natural language and speech processing and face and object recognition, computational creativity is hampered by our inability to define it in objective terms. Questions like „How do we evaluate computational creativity?“ and „What counts as creativity in a computational system?“ are some of the still unanswered questions which motivate research in this area.

This thesis is focused mainly on a specific type of computational creativity: *algorithmic composition* – interpreted literally, the use of algorithms to compose music. The concept of algorithmic composition is nothing new. Musicians have been proposing methods that can be considered algorithmic for centuries but the true algorithmic composition developed only in recent years with advancement in computers and information technology as a whole. State-of-art music generation systems are capable of generating very interesting musical pieces. Songs generated by the state-of-art MuseGAN project [9] may serve as an example to the reader.

This work summarizes different approaches to algorithmic composition, mainly musical composition by neural networks. Neural networks are currently the fastest growing field of artificial intelligence and they seem to outperform other machine learning methods in many areas, including algorithmic composition. The next part of the work consists of the description of different musical formats which were used in this work and an assessment of current situation and plan of work.

In the following chapters I describe my own approach to the problem of algorithmic composition utilizing neural networks. This can be broken down to two different tasks: the acquisition of a dataset of guitar tracks with specific properties and training of some neural networks on the created dataset. In my work, I try to take advantage of information about different song sections (eg. intro, verse, bridge, chorus, solo, ...) and information about song tracks (eg. rhythm or lead guitar, bass, drums, ...) contained in songs in Guitar Pro format and use them for narrowing down a task of generating music for a neural network. The final chapter describes experiments with trained networks and evaluates musical pieces generated by the created neural networks.

Chapter 2

Algorithmic composition

Algorithmic composition could be described as: „*a sequence (set) of rules (instructions, operations) for solving (accomplishing) a [particular] problem (task) [in a finite number of steps] of combining musical parts (things, elements) into a whole (composition)*“ [7]. From this definition it is apparent that it is not crucial to use computers for algorithmic composition as we often infer. Mozart, for example, created a system called Musikalisches Würfelspiel (Musical Dice Game) which used dice to randomly generate music from precomposed options. [17] This game may be considered a type of algorithmic composition too.

The first computational model for algorithmic composition was created only in 1959, when Hiller and Isaacson used random number generators together with Markov chains to generate music. [17] Since then researchers have been addressing the problem of algorithmic composition from many different angles. The different methods for algorithmic composition can be categorised based on their most prominent feature, into these sections:

2.1 Grammars

2.2 Symbolic, Knowledge-Based Models

2.3 Markov Chains

2.4 Evolutionary Models

2.5 Self-Similarity and Cellular Automata

2.6 Artificial Neural Networks

These methods will be described in detail in next subsections. The categorization and descriptions of methods were loosely adopted from [17] and [10].

2.1 Grammars

In broad terms, a formal grammar may be defined as a set of production rules for strings in a formal language. Strings are generated by repeatedly applying rules, in a sequence of so-called derivation steps. Because of the repeated application of the same rules, grammars are well suited for representation of systems with hierarchical structure. This property is ideal for the problem of algorithmic composition as hierarchical structures are present in most styles of music.

The most important step for algorithmic composition by grammars is the definition of the set of rules. While early authors derived these rules by hand from principles grounded in music theory, other methods are possible. These methods usually distill a grammar by examining a corpus of pre-existing musical compositions (for example by evolutionary algorithms). Another important aspect of the automatic composition process is the selection of the grammatical rules which will be applied. Common solution is the use of activation probabilities for the rules (so called stochastic grammars).

The main disadvantages of formal grammars are:

- Difficulty to manually define a set of grammatical rules to produce a good composition.
- On the other hand, generation of the rules of grammar automatically (eg. by evolutionary algorithms) does not produce great results.
- Grammar can usually generate a large number of musical strings of questionable quality.

2.2 Symbolic, knowledge-based models

Knowledge-based system (KBS) is an umbrella term unifying various rule or constraint-based systems. These systems represent knowledge as more or less structured symbols. The use of KBS for algorithmic composition seems to be a natural choice because the knowledge about musical composition has traditionally been structured in the form of a set of rules for manipulating musical symbols. Their main advantage is their explicit reasoning (they can explain their choice of actions). Although, the rules implemented in KGB are usually static, part of the knowledge may also be dynamically changed or learned (eg. using evolutionary algorithms).

In spite of looking like the most suitable choice for algorithmic composition, KBS still exhibit some problems:

- Formation of rules and constraints for algorithmic composition is very difficult and time consuming and depends on the ability of an expert to clarify the base concepts and find a usable representation.
- KBS might become overly complicated when adding all the „exceptions to the rule“ which is necessary in this domain.

KBS are often combined with other methods for algorithmic composition to provide better results.

2.3 Markov chains

Markov chain is a stochastic model describing a sequence of possible events (states) in which the probability of each event depends only on the previous event (state). Markov chains can be represented as labeled directed graphs (nodes represent states, edges represent transitions and edge values represent their probability), or as probability matrices, which are more common. For the task of algorithmic composition, these probability matrices may be either induced by training from a corpus of pre-existing compositions, or derived manually from music theory. The former is a more common way to use them in research.

The states of Markov chain are usually mapped to a single note or a sequential group of notes played for a certain time.

Some of the main problems of the Markov chains for algorithmic composition are:

- Necessity to find the probabilities of notes by analysing many pieces when creating Markov chain by hand.
- Markov chains generated from corpus of pre-existing compositions often capture just local statistical similarities and they are not able to capture higher or more abstract levels of music.

Among others, Hidden Markov Models, generalizations of Markov chains, were used extensively for algorithmic composition too.

2.4 Evolutionary models

Evolutionary algorithms (EAs) use mechanisms inspired by biological evolution, where a changing set of candidate solutions (a population of individuals) undergoes a repeated cycle of evaluation, selection and reproduction with variations (eg. mutations or combination of individuals). These algorithms have proven to be very efficient for problems with large search spaces, such as the problem of algorithmic composition. This property together with their ability to provide multiple solutions makes them suitable for algorithmic composition. EAs may be divided into two groups based on the type of their fitness function.

Use of an objective fitness function. The candidate solutions are evaluated by some formally stated fitness function. This type of EAs depends heavily on the amount of knowledge that the system possesses (in form of a complicated objective fitness function).

Use of a human as fitness function (so-called Interactive Genetic Algorithms). In this case, a human replaces the fitness function and performs the evaluation of candidate solutions. This type of EAs suffers from two main problems:

- Subjectivity of users (evaluators).
- Efficiency – the user must hear all potential solutions to evaluate a population.

2.5 Self-similarity and cellular automata

According to research, the spectral density of an audio signal (for music of many different styles) usually follows a $1/f$ distribution, where f is the frequency of the signal. This type of signal is usually referred to as *pink* noise and it occurs naturally in many different data series, eg. meteorological data. It was discovered, that random compositions where the pitches of notes were determined by pink noise were perceived as more musical and pleasing by an audience. It is believed, that this is caused by *self-similarity* in the structure of pink noise.

A self-similar object is an object which is exactly or approximately similar to a part of itself. This is the defining property of fractals and also a common feature of many classical music compositions. That is why the fractals and chaotic systems (which have fractal characteristics) have been extensively used as a source of raw material for compositions. Commonly used techniques for generation of self-similar musical patterns are iterated function systems, non-linear maps, non-linear dynamical systems, L-systems or musical rendering of fractal images. These techniques also include cellular automata which are a discrete version of chaotic dynamical systems and can be used to generate fractal patterns.

A cellular automaton (CA) is a discrete model consisting of a regular grid of cells, each in one of the finite number of states. In each time step, each cell's state is updated according to some deterministic rules. These rules are usually defined by a composer but there were also attempts to learn the rules of CA by a neural network or evolutionary algorithms.

Overall, the methods that produce self-similar patterns and cellular automata are not suitable for generation of full-fledged algorithmic compositions and are more commonly used to generate raw material for composition or used together with other methods for algorithmic composition.

2.6 Artificial neural networks

Artificial neural networks or simply neural networks (NNs) are computational models vaguely inspired by biological neural networks that constitute animal brains. They are composed of interconnected sets of *artificial neurons* (imitation of a biological neuron) which transform some inputs to certain outputs. They are typically used in *supervised learning*, where a set of examples (inputs with corresponding outputs) is used to train the network.

As for the problem of algorithmic composition, this means that NNs need a pre-existing corpus (dataset) of music compositions (preferably in a similar style), to be able to „learn“ and imitate the style of these compositions. One of the most important aspects of algorithmic composition by NNs is the mapping between the music notation and inputs/outputs of the network. Mapping of music notation to pianoroll format (see 4.3) is among the most frequently used mappings. The type and the architecture of the NN is also very important as each architecture exhibits different behavior. Both feed-forward and recurrent networks were used to compose music in the past.

Neural networks are very often coupled with other approaches to algorithmic composition, for example:

- **creation by refinement** – gradient-descent algorithm (or similar) used to create a composition from a random pattern with NN used as critique (NN was trained to return a musicality score of a given composition)
- **NN + agents** – NN generates basic melody by segments and rule-based agents elaborate and extend the melody.
- **Markov Chain + NN** – Markov chain creates motifs with relative pitch and after that NN assign absolute pitch to each motif.
- **NN + chaotic system** – NN trained with pre-existing composition together with pseudo-random musical input from a chaotic system (goal was to generate more complex compositions).
- **NN + evolutionary algorithms** (most popular) – NN trained to act as a fitness function of an evolutionary algorithm for selection of best candidates.

Chapter 3

Neural networks

This chapter is dedicated to more detailed description of neural networks (NNs) as this thesis is focused on algorithmic composition by neural networks.

Neural networks are, as defined by Dr. Robert Hecht Nielsen: *„...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.“* [6]. In simple words, a neural network is a computational model loosely inspired by the way biological neural networks in animal brains process information.

A basic building block of neural networks is *artificial neuron*. It is a mathematical function with an arbitrary number of inputs and a single output, modeled after biological neurons [2]. Scheme of artificial neuron (later only neuron) is shown in Figure 3.1. Input of the neuron is a vector $\vec{x}$. For each of the inputs x_0, x_1, ..., x_n there is a corresponding weight w_0, w_1, ..., w_n in the vector of weights $\vec{w}$. The input x_0 is usually referred to as *bias* and is fixed to a value of 1. Upon receiving inputs, each of the inputs x_i is multiplied by its corresponding weight w_i. The products are then summed and an *activation function* φ is applied to the sum. Output of the activation function is the output of the whole neuron y. Mathematically speaking, the output of neuron y is a function:

$$y = \varphi\left(\sum_{i=0}^{N} w_i x_i\right)$$

where φ is an activation function, N is a number of inputs, x_i is i-th input and w_i is i-th weight. There exist a number of different activation functions. The most commonly used functions are *sigmoid* activation function and rectified linear activation function – also called *ReLU* (Rectified Linear Unit).

Neural network is formed by connecting artificial neurons: output of a neuron is connected to input of another neuron. Neurons are usually organised into layers of three basic types: *input layer*, *output layer* and *hidden layer*. Input layer is composed of „passive“ neurons (they do not modify the input data) and it brings data into the network for further processing by subsequent layers. Output layer is the last layer in NN that produces outputs for the program. Hidden layer is a layer in between input and output layers, which neurons take in a set of inputs and produce a set of outputs. NN with multiple hidden layers is called *deep neural network*. In Figure 3.2, there is an example of a deep neural network with an input layer, three fully connected hidden layers and a fully connected output layer.

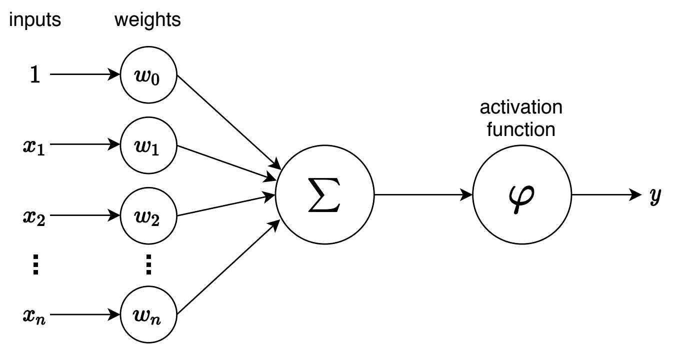

Figure 3.1: Artificial neuron with input vector $\vec{x}$, weights $\vec{w}$, activation function φ and output y.

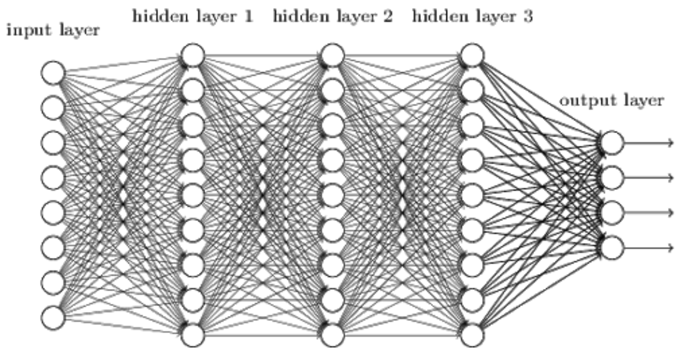

Figure 3.2: Deep neural network with input layer, output layer and 3 fully connected hidden layers. Source: [15].

3.1 Convolutional neural networks

Convolutional neural networks (CNNs) are a class of deep neural networks, most commonly applied to processing of data that has a known grid-like topology. They are primarily used for processing of visual imagery. By their definition convolutional networks are: *„... neural networks that use convolution in place of general matrix multiplication in at least one of their layers.“* [11]. Traditional layers (usually referred to as *dense* layers) multiply the input matrix by a matrix of parameters. This means that the value of every output unit of a layer depends on every input unit. On the other hand, convolutional layers have sparse connectivity, which means that each output unit depends solely on inputs selected by *convolutional kernel.*

Convolutional layer can be favourably used when it is expected that two inputs that are close to each other are more related than inputs that are further apart from each other. Convolutional layers are also shift invariant, which means that they are able to detect features/objects even if these features/objects do not exactly resemble the training examples. It is worth noting that a dense layer could perform the same transformation of input as a convolutional layer, but compared to dense layers, convolutional layers are more efficient – they require way less parameters and perform less floating point operations. Convolutional layer has following attributes:

- *shape of convolutional kernels* – defines their width and height, eg.: (3, 3), (5, 5), etc.

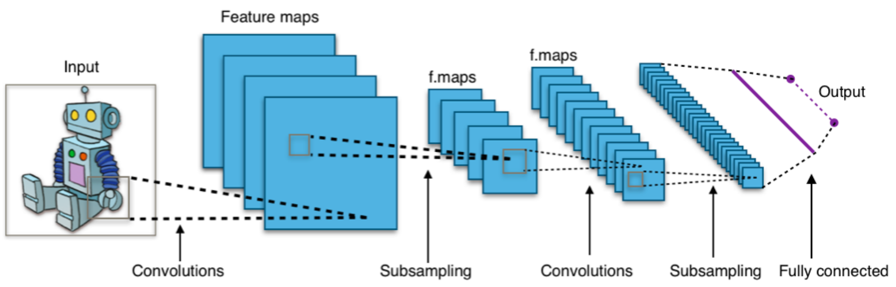

Figure 3.3: Typical architecture of CNN. Source: [3].

- *number of input channels* (usually defined by num. of output channels of previous layer)
- *number of output channels* – defined by number of *filters* (structures of multiple kernels stacked together)
- *stride* – describes the size of the convolutional step of the kernel, eg.: (1, 1), (2, 2), etc.
- *padding* – describes the type of padding of the input

Architectures of convolutional networks usually contain so-called pooling layers too. **Pooling layer** replaces the output of the network at a certain location with a summary statistic of the nearby outputs [11]. By this it reduces the dimensions of the data received from the previous layer. One of the most common types of pooling layers is *max-pooling*, which downsamples the input representation by taking the maximum value over the window which is shifted over the whole input. A max-pooling layer has following attributes:

- *pool size* – defines a shape of the pooling window, eg.: (2, 2), (2, 1), etc.
- *stride* – describes shifting of the pooling window, eg.: (1, 1), (2, 2), etc.

A typical architecture of CNN is portrayed in Picture 3.3. This network consists of two pairs of convolutional layer + pooling layer (subsampling) and final dense layer which produces output. Networks like these are usually used for classification or detection of objects in images.

Convolutional networks can be favourably used for the generation of music. Each musical piece can be represented as an „image“, which is referred to as *pianoroll* (see 4.3). Properties of pianorolls are very similar to properties of normal images. Notes/chords close to each other are more related than notes/chords further apart and notes/chords are invariant of their position in the pianoroll. The convolutional kernels are expected to detect different kinds of chords or notes and their alteration in time.

3.2 Recurrent neural networks

Recurrent neural networks (RNNs) are a family of neural networks used mainly for processing of sequential data. They are designed to recognize data's sequential characteristics and predict the next likely scenario [11]. These networks can be described as directed cyclic graphs with feedback loops, which allow information to persist (often described as a memory of the network). Classic RNNs, however, suffer from a so-called *vanishing gradient problem* which causes the drop of performance of the networks and their inability to „remember“ information for longer periods of time. One solution to this problem is usage of Long-short Term Memory units in the recurrent network.

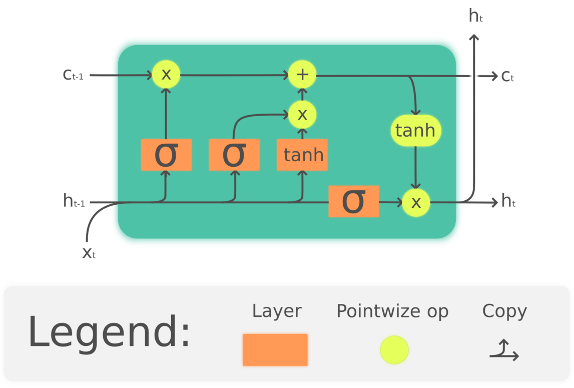

Figure 3.4: Long short-term memory unit. Source: [4].

Long-Short Term Memory networks

Long-Short Term Memory (LSTMs) RNNs belong to a special group of *gated RNNs*. Networks from this family are one of the most effective models for sequence modelling. A common LSTM unit (shown in Figure 3.4) is composed of 4 cells, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell [4]. The gates enable the network to figure out what data is important and should be remembered and looped back into the network, and what data can be forgotten.

Musical pieces are basically a type of sequential data, so the usage of a recurrent network for music generation is an obvious choice. Among many types of RNNs, LSTM networks are the most suitable for generation of musical pieces because of their ability to remember important information about notes/chords played in the past and use this knowledge to predict what is played in the next time step. It was also proven that these networks perform better than other types of RNNs.

Chapter 4

Overview of different music formats

This chapter provides a quick overview of different music formats which were used during the creation of a training dataset.

4.1 MIDI

MIDI (acronym for Musical Instrument Digital Interface) is a technical standard describing communications protocol and digital interface used for recording, playing, editing and synthesis of music in a computer. A MIDI file contains one or more tracks, each containing a list of events (messages) with time information for each event. Each MIDI event contains 3 basic parameters: time information (relative to previous event), track number, and type of MIDI event [12]. The most important MIDI events are:

- **Note On** and **Note Off** events: MIDI key press and release signalization. Both of them have 2 parameters:
  - **key (note) number** - from 0 (C_{-1}, about $8.18Hz$) to 127 (G_9, $12500Hz$)
  - **velocity** - force with which a note is played
- **Time Signature** meta-event: sets the time signature of a MIDI song, eg. 4/4, 3/4, 7/8, etc.
- **Key Signature** meta-event: sets the key of a MIDI song, eg. C dur, e moll, etc.

4.2 Guitar Pro format

Guitar Pro is a tablature editor software for guitar, bass and other fretted instruments. It allows creation of musical scores and backing tracks for other instruments eg. drums and piano, too [14]. Files composed in the Guitar Pro editor are saved as `.gp`, `.gpx`, `.gp5`, `.gp4` and `.gp3` file formats, each corresponding to different versions of software. It is worth noting that there exist other free, open source alternatives to Guitar Pro like Tux Guitar, that use Guitar Pro file formats. Example of a song opened in Guitar Pro editor is shown in Figure 4.1.

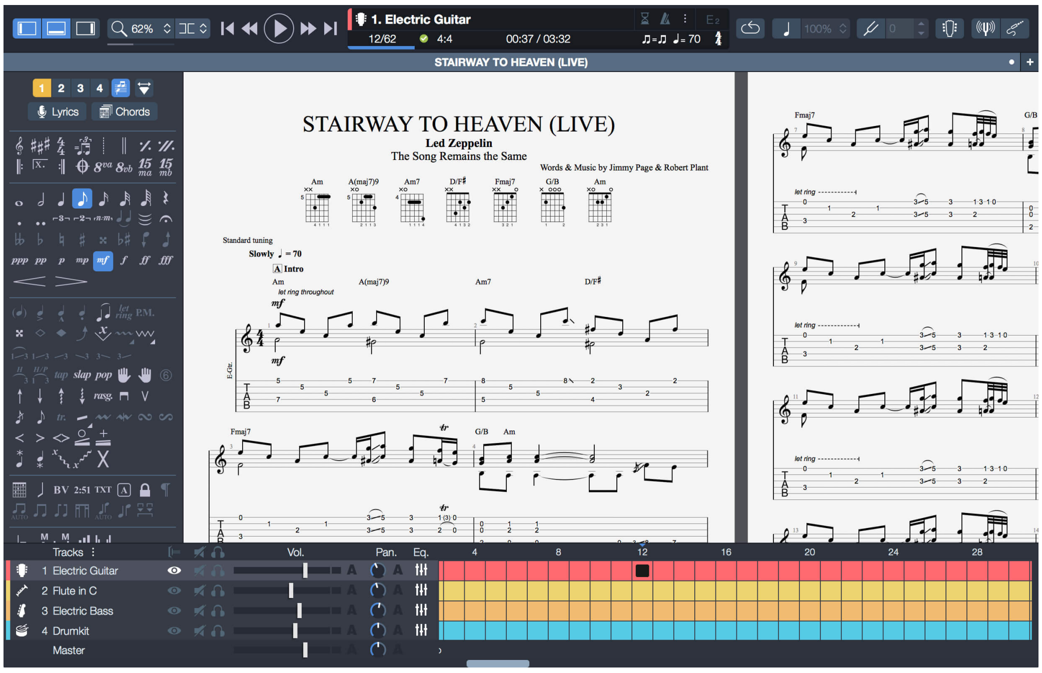

Figure 4.1: Song in GPX format opened in Guitar Pro editor. Source: [1].

Same as a MIDI file, a Guitar Pro file contains one or more tracks. Each track consists of measures and each measure consists of beats. A single beat contains notes that are played in that beat. A measure can also contain other meta information like: time signature, tempo, information about repetitions of measure, length of measure, lyrics or markers. Each of these information can be visualized or played using Guitar Pro editor (or other open source editor) as shown in Figure 4.1. Markers are used for providing additional text information and they usually contain name of the section of the song, eg. *Intro*, *Chorus*, *Verse*, etc.. Marker *Intro* can be seen in the Figure 4.1 in the beginning of the song.

Compared to MIDI, Guitar Pro divides song into measures, which then contain notes. Because of this, notes in Guitar Pro do not contain time information compared to notes in MIDI. The relative time of a note in a measure can be determined from type of the note (eg. half, quarter, etc.) and tempo with time signature of the measure.

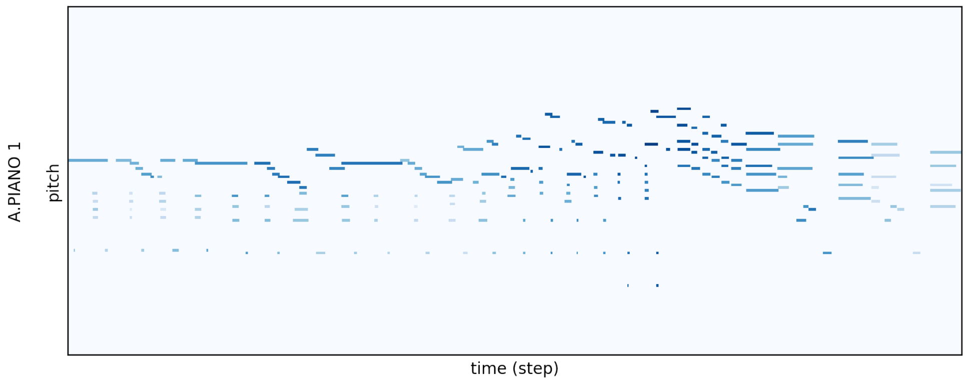

Figure 4.2: Example pianoroll. Source: [8].

4.3 Pianoroll

Pianoroll is a music storing format which represents a musical piece as a score-like matrix. The vertical axis represents pitch and horizontal axis time. Pitch has 128 different values, from $0 - C_{-1}$ to $127 - G_9$ (same range as MIDI). The values in pianoroll matrix represent the velocities of the notes. Velocity of a note can be simply interpreted as the loudness of the note when it is played. Time axis can be represented in *absolute timing* or *symbolic timing*. For absolute timing, the actual timing of note occurrence is used. For symbolic timing, the tempo information is removed and thereby each beat has the same length. [8] Example of a pianoroll is shown in Figure 4.2.

Chapter 5

Assessment of the current state and work plan

Suitable training dataset of musical pieces is one of the most important things for a successful training of a neural network. There already exist plenty of datasets with eg. classical songs, jazz songs, etc. in MIDI format – for example [20]. Songs in MIDI format usually contain a few tracks, each with a different instrument. Each of the tracks contain some notes which are played by the instrument. Aside from these information, songs in MIDI format usually do not contain any additional information which could be useful for algorithmic composition.

On the other hand, songs in Guitar Pro format often do contain some additional information. These information are for example: information about repetitions, detailed information about an instrument in a track and mainly information about song sections. The song sections specified in songs in Guitar Pro format are for example: *chorus*, *bridge*, *solo*, *outro*, etc.. The information about song sections could be favorably used to train a neural network (or some other method of algorithmic composition) to generate music which resembles certain song sections.

Almost every song in Guitar Pro format also contains one or more guitar tracks which are usually described as eg. *lead guitar*, *rhythm guitar* or *distortion*[1] *guitar* and *acoustic guitar*. This concrete description of the type of guitar could be also beneficial for the training of a network on a specific type of guitar track. Another advantageous aspect is that most songs in Guitar Pro belong to the following genres: Rock, Metal, Punk, Blues – which are all pretty similar to each other.

Because of these properties (of Guitar Pro songs) I initially wanted to create my own **dataset of guitar tracks with time-aligned description of song sections**. This turned out to be the wrong approach and a dataset like this would be impossible to create – described in Section 6.2. So instead of the dataset with time-aligned descriptions of song sections I decided to create **datasets of guitar tracks of specific song sections**. This approach was successful and is described in Section 6.3.

It is very important to select some mapping of music notation (of songs from a dataset) to input/output of a neural network. I decided to use one of the simplest: mapping of Guitar Pro songs to pianoroll matrices. Pianorolls can be fed directly into the neural network without any additional changes. This mapping provides a lot of variability and

[1]Distortion is a special type of guitar effect which changes the sound of electric guitar. It is used mainly in rock and metal songs.

does not restrict neural networks which could be both advantageous and disadvantageous. The whole dataset, which I created, is therefore saved in the form of pianorolls.

After the creation of the dataset, I planned to design and experiment with different neural networks, mainly convolutional neural networks and LSTM networks. These are the networks that I planned to train on my dataset:

- Dense-only NN [implemented – see 7.2]
- CNN with 1-dimensional output [implemented – see 7.2]
- CNN with 2-dimensional output (autoencoder) [not implemented]
- LSTM [implemented – see 7.2]
- LSTM Sequence-to-sequence (autoencoder) [implemented – see 7.2]

Chapter 6

Creation of a training dataset from Guitar Pro files

To create a dataset of guitar tracks with desired properties I required a sufficient amount of Guitar Pro songs from which I could filter out the suitable ones. The database of Guitar Pro songs that I eventually used can be accessed on this website: Guitar Pro database[1]. It contains **52512** Guitar Pro files that are sorted according to artists. It is one of the largest freely available databases of Guitar Pro songs that I found.

I used Python 3.7 together with Jupyter notebook for the analysis and the preprocessing of songs. Python 3.7 was an ideal choice because it contains libraries that are able to process Guitar Pro and MIDI files. The libraries, which I eventually used, are: NumPy [16], pretty-midi[2] and PyGuitarPro[3].

6.1 Analysis of Guitar Pro files

After acquiring the database of Guitar Pro songs I proceeded with analysis of the songs using Python library PyGuitarPro. My first objective was to find out how many songs in this dataset contain *markers* (see 4.2). Presence of markers in a song was essential for the creation of a dataset of guitar tracks with time-aligned description of song sections. Using a Python script (see `dataset_creation/stats.py`) I found out that out of **52512** total songs, **16747** contained at least one marker – 31.89% of songs contained a marker.

To simplify the analysis of Guitar Pro files, I also saved all available information about songs with at least one marker to a JSON file. The saved information were: title, artist, key, number of measures, tempo, information about tracks (name, actual play time), information about measures (serial number of measure, markers, time signature info, tempo info, repetition info). Actual play time (percentage) of a track was calculated by counting all the measures with at least one note and dividing it by the total number of measures in the song. These statistics were later used for analysis and filtration of songs.

From now on, I focused solely on songs that contained at least one marker. After examining the markers in songs I found out that there are many synonymous markers, which described the same section of the song eg.: *Chorus* and *Refrain*, *Solo* and *Guitar Solo*, etc., or synonymous markers with serial numbers eg.: *Verse* and *Verse 1* or *1st Verse*, etc..

[1]https://uloz.to/file/15ox6Hq7
[2]https://github.com/craffel/pretty-midi
[3]https://github.com/Perlence/PyGuitarPro

Section / Marker category	Num. of songs with the section[1]
Chorus	approx. 10500
Verse	approx. 10500
Intro	approx. 10000
Solo	approx. 7500
Outro	approx. 6500
Bridge	approx. 4500
Interlude	approx. 3000
Pre-chorus	approx. 2500

Figure 6.1: Different song sections / marker categories with the number of occurrences in the dataset of Guitar Pro songs. These song sections / marker categories are also referred to as *recognized markers*.

Because of that, I tried to put the most frequent markers into certain categories. Each of the markers was changed to lowercase and white spaces and other unnecessary characters were removed. Synonyms like eg.: *Chorus* and *Refrain* were united to the single category named *Chorus* using a simple logic. This way, I was able to determine the marker categories / song sections shown in Table 6.1. They are later referred to as *recognized markers*. Markers, which do not belong to one of these categories, are referred to as *unrecognized markers*.

Most of the Guitar Pro songs from my database consisted of more than one track and did not contain solely guitar tracks but also tracks with other instruments like: bass, piano, drums, etc.. Because of this, it was necessary to cleverly select the most suitable track in each song. I aggregated the track names to categories (very similarly to how I aggregated the markers) and acquired these guitar track categories:

- according to type of playing: *rhythm guitar* | *lead guitar*
- according to type of guitar: *acoustic guitar* | *electric guitar*
- according to used guitar effect: *clean guitar* | *distortion guitar* | *overdrive guitar*
- default category (no additional specification): *guitar*

6.2 Dataset of guitar tracks with time-aligned description of song sections

Before I tried to create a dataset of guitar tracks with a time-aligned description of song sections I defined a few conditions which would ensure that such a dataset would be suitable for training. A dataset needs to satisfy these conditions (in order according to their importance):

1. Each song in the dataset has **at least 4 song sections**. These song sections are indicated by time-aligned descriptions, each describing where the song section starts and where it ends. All of the song sections have to belong to the group of *recognized markers* described in Table 6.1.

2. Dataset consists of songs with a **guitar** track.

[1]Number of songs which contained at least one occurrence of the section.

3. Each guitar track in the dataset is played during the whole song from which it was extracted. This ensures that the guitar plays during each of the song sections and there is no silent part.

4. Dataset contains at least 5000 different songs. If a dataset contained less songs, it would not be large enough for training.

Analysis of **16747** songs which contained at least one marker led to the following results. To satisfy the condition No.1 we have to use only songs which contain at least 4 *recognized markers* but do not contain any *unrecognized markers* (see Table 6.1). Songs with unrecognized markers can not be included in the final dataset because it is impossible to determine what section does unrecognized marker represent. Number of songs which contain at least 4 recognized markers is **11356**. But the total number of songs with at least 4 recognized and 0 unrecognized markers is only **4753**. This shows that although there are a lot of songs which contain at least 4 recognized song sections lots of them contain some unrecognized markers.

To satisfy the condition No.2 we have to select only songs which contain a guitar track. Number of songs which satisfy condition No.1 and contain a guitar track is **3719**.

And finally to satisfy the condition No.3 we have to select only songs with a guitar track which plays during the whole song. Number of songs which satisfy condition No.1, No.2 and contain a guitar track which plays during the whole duration of song is only **1993**! This means that the condition No.4 is not satisfied and therefore a dataset of guitar tracks with time-aligned description of song sections, which would satisfy these demands, **cannot be created**.

Only 11.9% of Guitar Pro songs with a marker and only 3.8% of all Guitar Pro songs from my database satisfy the first 3 conditions. The database of Guitar Pro songs that I used was one of the largest freely available databases. Because the whole database contains such a large amount of Guitar Pro songs (52512 total), it is possible to assume that a different database of Guitar Pro songs would have a very similar percentage of songs (3.8%) which would satisfy the first 3 conditions. Therefore, I assumed that the analysis of a different database would lead to similar results.

6.3 Dataset of guitar tracks extracted from specific song sections

As it turns out, a dataset of guitar tracks with time-aligned description of song sections cannot be created. That is why I decided to create a dataset of guitar tracks extracted from specific song sections instead. Such a dataset would need to satisfy these conditions (in order according to their importance):

1. Dataset contains exclusively parts of **guitar** tracks from a selected section of songs. (eg. only choruses, only verses, etc.)

2. Dataset contains at least 5000 different songs. If a dataset contained less songs, it would not be large enough for training.

These conditions can be satisfied more easily and the whole process of creation of this dataset is described in the following paragraphs. Because *Verse* and *Chorus* were amongst the most frequently *recognized markers* I decided to extract solely verses and choruses and

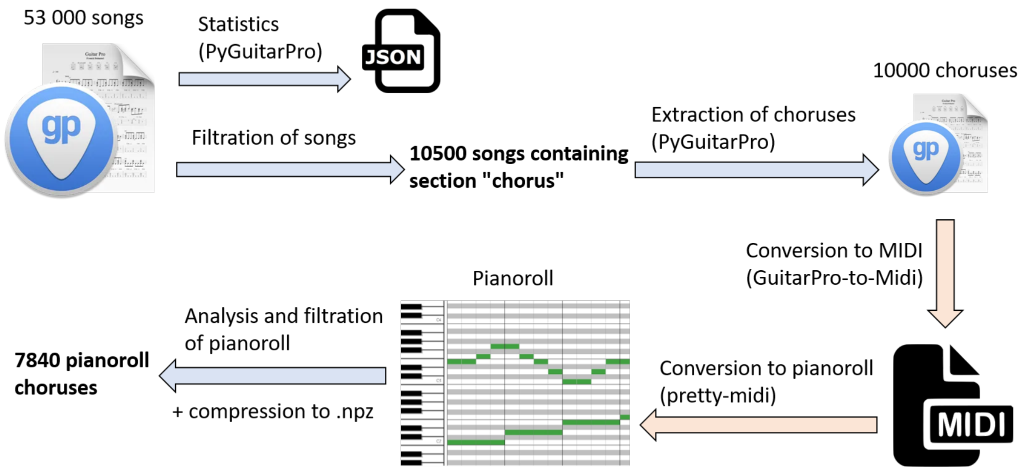
53 000 songs
Statistics (PyGuitarPro)
JSON
Filtration of songs
10500 songs containing section "chorus"
Extraction of choruses (PyGuitarPro)
10000 choruses
Conversion to MIDI (GuitarPro-to-Midi)
MIDI
Conversion to pianoroll (pretty-midi)
Pianoroll
Analysis and filtration of pianoroll
+ compression to .npz
7840 pianoroll choruses

Figure 6.2: Creation of dataset of guitar choruses in pianoroll format

create 2 different datasets. Dataset No.1 would consist of verses only and dataset No.2 would consist of choruses only. The following text will describe the creation of the dataset of choruses in detail – the dataset of verses was created very similarly and therefore it is not described in the next paragraphs. The whole process of creation of this dataset in pianoroll format is shown in Figure 6.2 and will be described more closely in the next paragraphs.

As mentioned previously, I focused only on the track category labeled: *guitar*, especially on the *rhythm guitar* category, because rhythm guitars usually have the most important role during chorus or verse – they often play a main riff. For extraction of chorus guitar tracks I used following algorithm (see `dataset_creation/extract_sections.py`):

For each song in dataset:

1. Check if the song contains a marker from category: *Chorus*. If not, continue with the next song.
2. Find index of measure marked as *Chorus* (this is the beginning of chorus). Let's call this index – `start`.
3. Find index of next closest measure with different marker after measure at index `start` (this is the end of chorus). If such a measure is not found use the index of the last measure in the song. Let's call this index – `end`.
4. Using function `find_best_guitar_track()` select the most suitable track, prioritizing *rhythm guitar* category over other *guitar* categories and prioritizing tracks that are playing most of the time from measure at index `start` to `end`.
5. Using function `extract_part()` cut only measures from `start` to `end` from the track, which was selected in the previous step, and save them as a new Guitar Pro file (`.gp5` format) with all the necessary information like tempo and time signatures.

Using this algorithm I obtained around **10000** choruses! The easiest way to convert these extracted choruses to pianorolls is: 1. convert a Guitar Pro chorus to MIDI and 2. convert the MIDI to pianoroll and save it as NumPy's compressed format: `.npz`.

[1] `https://github.com/alexsteb/GuitarPro-to-Midi`

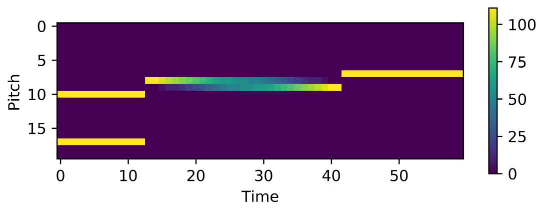

Figure 6.3: Fade-out and fade-in of notes.

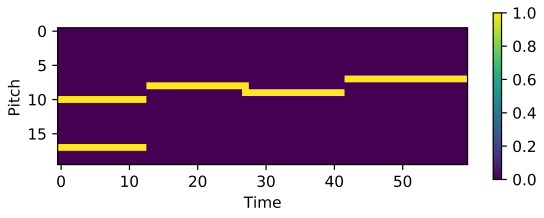

Figure 6.4: Thresholding with thresh.: 50.

Script `dataset_creation/convert_gp_to_midi.py` converts Guitar Pro choruses to MIDI using this GuitarPro-to-Midi converter[1]. This program is written in C# as a package for Unity. It is able to simulate the sound of different guitar effects (eg. different types of harmonics, vibrato, muted notes, bending, etc.) very precisely [19]. To be able to use it effectively I modified it slightly, allowing the execution from Python.

After the successful conversion of choruses to MIDI I used Python library: pretty-midi to convert each of the MIDI files to pianorolls. A pretty-midi's builtin function: `get_piano_roll()` computes a pianoroll matrix (NumPy array) of a given MIDI. This function computes a pianoroll with *absolute timing*, which means that the actual timing of the note occurrence is used. Its parameter `fs` sets the sampling frequency of the columns in pianoroll, i.e. each column is spaced apart by `1.0/fs` seconds. I decided to set the parameter `fs` to 20, which seemed to be sufficient sampling frequency after a number of experiments. This means that each 20 columns of the pianoroll describe 1 second of the song.

Pianorolls (NumPy arrays) produced by `get_piano_roll()` also contain information about velocity of notes (see 4.3). Velocity of a single note in pianoroll can also change in time, which simulates some guitar effect eg.: slide, hammer-on, pull-off, etc.. These effects usually look like a continuous fade-out of one note (the note becomes quieter until it stops playing) and a continuous fade-in of another note (volume of the note rises until it reaches a max value), shown in Figure 6.3.

To further simplify the pianorolls, I decided to introduce uniform velocity of notes. This means that a note is either played – value „1“ or is not played – value „0“. It was achieved by thresholding the pianorolls using a threshold with value 50, which was selected after examining the pianorolls and finding out that typical velocity of played notes is: 100. Result of thresholding of the section from Figure 6.3 is shown in Figure 6.4.

Figure 6.5 shows what pitches occur the most in the choruses from my dataset – x-axis shows different pitches, where 0 is C_{-1} and 127 is G_9 and y-axis shows how many choruses contain given pitch (in logarithmic scale). It is clear that the most occurring pitches are in range 40 to 80. Assuming that most of the choruses from my dataset are played by guitar and knowing that pitch range of guitar tuned to standard tuning is $\langle E_2, E_6 \rangle$[1] – which corresponds to range $\langle 40, 88 \rangle$ – I decided to discard all pianorolls which contained pitches out of this interval. Pianorolls that contained even a single note with pitch in the red-colored region in Figure 6.5 were discarded from my dataset.

Figure 6.6 shows the length of choruses from my dataset – x-axis shows length in seconds and y-axis shows how many choruses are in a certain length interval. It is clear that there are many really short choruses (only a few seconds long) but also choruses which were

[1]E_2 is empty lowest string and E_6 is 24th fret on the highest string in standard tuning. Standard tuning is the most common tuning of guitars.

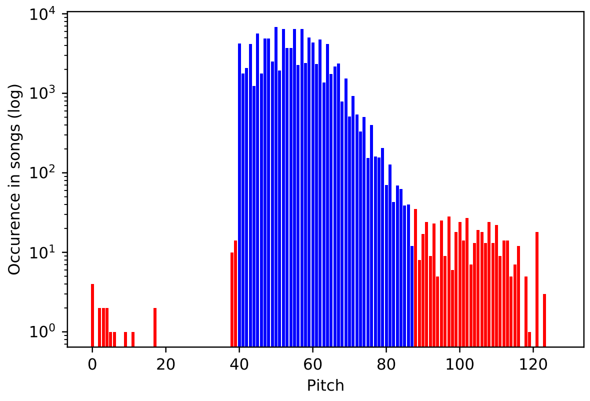

Figure 6.5: Pitch occurrence in choruses. Pitch value 0 corresponds to the pitch C_{-1} and pitch value 127 corresponds to the pitch G_9

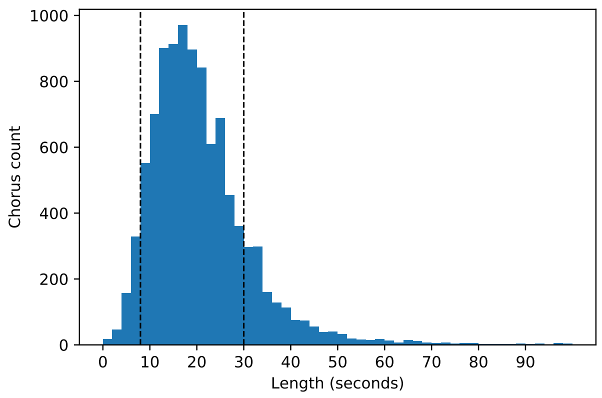

Figure 6.6: Number of choruses with certain lengths.

very long (almost a minute). With the assumption that choruses have usually under 30 seconds and over 8 seconds, I decided to discard all the choruses that have not satisfied these constraints. Choruses between the two dashed lines in the Figure 6.6 were included in my dataset – others were excluded from the dataset.

This whole process of filtering and processing of songs resulted in the dataset of **7840** guitar track choruses in pianoroll format. The whole dataset is located in directory: `datasets/chorus_npz`. This is a sufficiently large dataset and it is suitable for training of a neural network. All of the choruses in pianoroll format were compressed and saved as NumPy's zipped archives: `.npz`. A single zipped archive contains 500 choruses in pianoroll format. This number was chosen because of the memory limitations of Google Colaboratory[1], where the created neural networks were trained.

The dataset of verses was created in a similar way as the dataset of choruses. The dataset of verses contains **7030** guitar tracks in pianoroll format. The whole dataset is located in directory: `datasets/verse_npz`. All of the verses are saved in `.npz` format.

[1]Google Colaboratory is a hosted Jupyter notebook service that provides free access to Google's computing resources. URL: https://colab.research.google.com/notebooks/intro.ipynb

Chapter 7

Training of neural networks

At this point, 2 different guitar track datasets were at my disposal: dataset of choruses and dataset of verses. Because the dataset of choruses was larger, I decided to train all of the created neural networks on this dataset.

Implementation of this part of my work is written in Python 3.7 because of many well documented, easy to use libraries. Deep learning models were created using high-level machine learning library Keras, included in library Tensorflow 2.0 [5]. Preprocessing of input and postprocessing of output from neural networks was achieved using many libraries, mainly: NumPy, pretty-midi and midi2audio[1]. All models were trained in Google Colaboratory.

7.1 Preprocessing, training and generation of musical pieces

Preprocessing and training of NNs

Each of the pianorolls from the dataset of choruses is a matrix with shape: (120, *), where 120 is the pitch range $\langle C_{-1}, G_9 \rangle$ and „*“ represents the variable length of pianoroll (as each of the choruses has different length). Because pianorolls contain solely notes which pitch is in interval $\langle E_2, E_6 \rangle$, I decided to reduce the size of pianorolls to include only this interval of pitches. This resulted in the following shape of pianorolls: (48, *), where 48 is the new pitch range from $\langle E_2, E_6 \rangle$. Reduction of size of pianorolls is beneficial because it reduces the number of inputs of neural networks and therefore reduces the number of trainable parameters.

Created neural networks are trained to generate some length of music notes given some preceding notes. In order to create a training set I used a sliding window which cuts the pianorolls to smaller parts. I decided to use the sliding window with shape (48, 120), where 120 columns of pianoroll are actually 6 seconds of the song. Sections with shape (48, 120) created by this sliding window are used as input of neural networks during training. The column/s which follow an input section are a ground truth (or an expected output of the network) corresponding to the given input section. Expected output of dense-only NN 7.2, stacked LSTM 7.2 and CNNs 7.2 is a **single column** which follows the input section. Single column of the pianoroll transformed back to MIDI has actual length 1/20 of a second. Expected output of Sequence-to-sequence NN 7.2 are **20 columns** that follow the input

[1]https://github.com/bzamecnik/midi2audio

section. 20 columns transformed back to MIDI have an actual length equal to 1 second. Using this sliding window I acquired approximately **2 milion training samples**.

All of the created networks were trained in batches, each containing 200 input sections together with their ground truths. Since the music from my dataset is polyphonic (multiple notes being on at the same time) this is a multi-label classification problem and hence the use of *binary cross entropy* loss [13]. The neural networks were trained using *Adam* optimization algorithm. Each of the networks was trained for 20 epochs on the whole dataset of choruses. After each epoch a checkpoint of the network (a file which contains learned weights of the network) was saved.

Generation of musical pieces:

To generate a musical piece by a trained network we must prepare a so-called musical *seed*. Seed is a small section of a musical piece which is fed into the network at the beginning of generation. The network then produces an output – a predicted column/s of pianoroll which contains values between 0 and 1. This is caused by the *sigmoid* activation function in the last layer of each network. Each value in the output column/s may be interpreted as a „probability“ that a note corresponding to given value is played. Because of that, it is necessary to threshold the output from NNs to values: 0 (note is not played) and 1 (note is played). Value of the threshold affects how many playing notes are produced by the neural network. Adjusted output is then appended to the seed, and the network is once again fed with new input which now includes output of the network from the last step. By repeating this procedure it is possible to generate a music piece with any desired length.

After this procedure, the generated output is transformed to proper pianoroll by changing the shape of generated output from (48, *) back to (120, *) and changing the values „1“ in pianoroll back to velocity „100“. Pianoroll is then transformed using the utility function `piano_roll_to_pretty_midi()`[1] to MIDI format. Using the library midi2audio and a sound font, the produced MIDI is transformed to playable `.wav` format.

7.2 Architectures of neural networks

Baseline dense-only NN

This neural network serves as a baseline (a comparison) to other more complicated neural networks described later. Input of this dense-only NN is a tensor with shape (*, 120, 48), where „*“ symbolizes variable size of tensor – this is needed for training in batches with variable size. 120 is the number of columns in an input pianoroll and 48 is the pitch range of input pianoroll. Whole architecture of this neural network can be seen in Table 7.1. Column *Output shape* describes the shapes of tensors produced by each of the layers.

Input layer of the network is followed by a flatten layer which reshapes the input for use by dense layers. These initial layers are followed by 3 groups of dense, batch normalization and drouput layers. Each dense layer has *ReLU* activation function and is followed by batch normalization (for improvement of performance) and dropout layer with dropout rate of 0.3 (to avoid overfitting and to make the network more robust). Last layer is Dense with *sigmoid* activation function and output shape (*, 48), which represents a single predicted column of pianoroll (with actual length: 1/20 of a second). This model has exactly 8,855,238 trainable parameters.

[1]Source: https://github.com/craffel/pretty-midi/blob/master/examples/reverse_pianoroll.py

Layer	Output Shape
Input Layer	(*, 120, 48)
Flatten	(*, 5760)
Dense + Batch Norm. + Dropout	(*, 1440)
Dense + Batch Norm. + Dropout	(*, 360)
Dense + Batch Norm. + Dropout	(*, 90)
Dense	(*, 48)

Table 7.1: Architecture of the dense-only NN

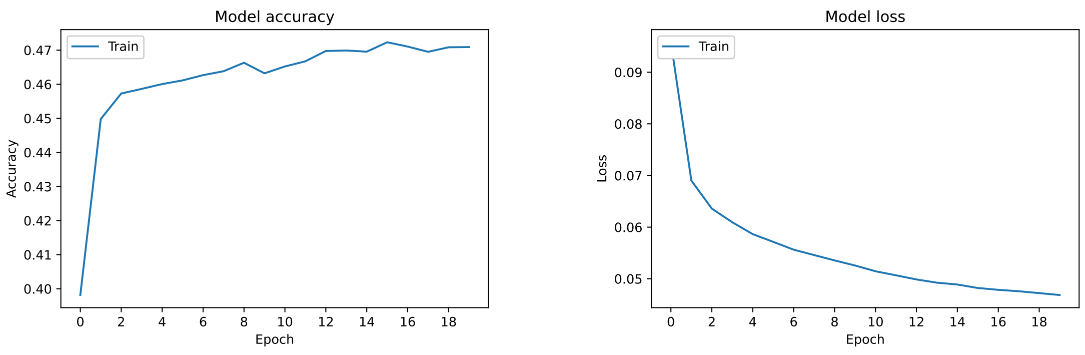

Figure 7.1: Training accuracy and loss of the dense-only NN

The model was trained for 20 epochs on the whole dataset of choruses. Model accuracy and model loss is shown in Figure 7.1. The best accuracy achieved by this model is: 47.2%. From the first figure it is clear that after the 15th epoch the accuracy of the network was no longer improving and it began to fluctuate. On the other hand, loss was improving during the whole training. Because the accuracy stopped improving at 15th epoch it is probable that it would not improve more with a longer training.

Convolutional neural networks

After creating the baseline dense-only network I experimented with different convolutional networks. At first I tried to use the architecture of CNNs which are usually used for image recognition. Architecture of these networks consist of convolutional layers with kernels shaped: (3, 3) or (5, 5), etc., alternated with max-pooling layers with the typical pool size: (2, 2). Last layer of these networks is typically a dense layer. Although these networks achieved training accuracy above 45% in 20 epochs, they did not produce very interesting results. Songs generated by these networks were very simple and contained only a few notes or even a single note which was played during the whole song.

Because of that, I tried to use different types of kernels in convolutional layers and different max-pooling layers. Architectures of the two most interesting networks which I created are shown in Table 7.2. These two networks are refered to as CNN Maxpool and CNN Stride in the following paragraphs.

CNN Maxpool was created with the following thoughts in mind: I decided to use a convolutional layer with 1 dimensional kernels with shape: (1, 5), with assumption that these kernels would be able to recognize different chords. After the convolutional layer of this type I decided to use a max-pooling layer with pool size: (2, 1), to reduce the length

CNN Maxpool			CNN Stride		
Layer	**Kernel**	**Filters**	**Layer**	**Kernel**	**Filters**
Input – shape: (*, 120, 48, 1)			Input – shape: (*, 120, 48, 1)		
Conv	(1, 5)	16	Conv	(1, 5)	16
Maxpool, size: (2, 1)	—	—	Conv, strides: (2, 1)	(3, 1)	16
Conv	(1, 5)	16	Conv	(1, 5)	16
Maxpool, size: (2, 1)	—	—	Conv, strides: (2, 1)	(3, 1)	16
Conv	(5, 5)	32	Conv	(5, 5)	32
Flatten			Flatten		
Dense – shape: (*, 48)			Dense – shape: (*, 48)		

Table 7.2: Architecture of the CNNs

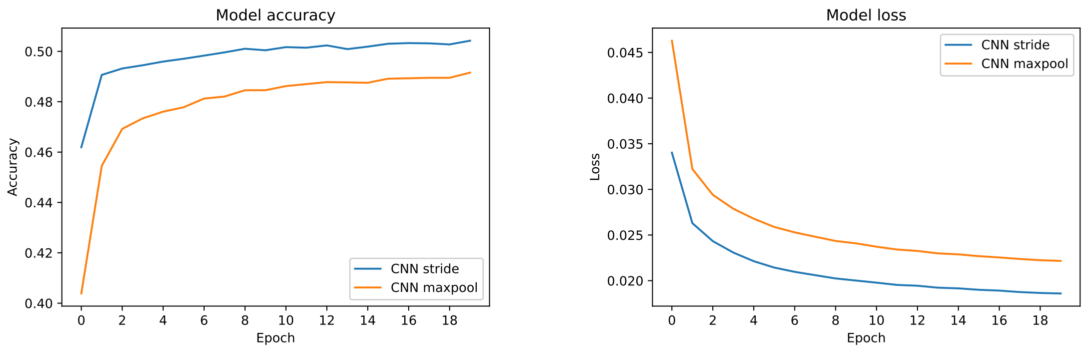

Figure 7.2: Training accuracy and loss of the CNNs

of the pianoroll to half. These conv. + max-pooling layers are repeated two times and followed by a classic convolutional layer with 2 dimensional kernel with shape: (5, 5). All of the convolutional layers have *ReLU* activation function. The last layer is dense with *sigmoid* activation function and output shape (*, 48), which represents a predicted column of pianoroll (with actual length: 1/20 of a second). This model has exactly 1,451,968 trainable parameters.

CNN Stride was derived from **CNN Maxpool** by the replacement of the max-pooling layers with convolutional layers with shape: (3, 1) and strides shaped: (2, 1). The use of convolutional layers with strides for the downsampling was inspired by this machine learning article [18]. The convolutional layer with strides also performs the reduction of length of pianoroll (same as a max-pooling), but unlike max-pooling it does not perform a fixed operation. This could be beneficial because the network can learn a mathematical operation that could be suitable for detection of chord transitions. This model has exactly 1,453,536 trainable parameters.

The training accuracy and loss are both shown in Figure 7.2. As expected, the CNN Stride achieved both better training accuracy and better training loss than the CNN Maxpool, in spite of having only a few thousand more parameters. CNN stride reached accuracy 50.42% while the CNN maxpool reached accuracy 49.15%. Both of them achieved better accuracy than the baseline dense-only model.

Layer	**Output Shape**
Input Layer	(*, 120, 48)
LSTM + Batch Norm. + Dropout	(*, 120, 64/128/256)
LSTM + Batch Norm. + Dropout	(*, 120, 64/128/256)
LSTM	(*, 64/128/256)
Dense	(*, 48)

Table 7.3: Architecture of the stacked LSTM network

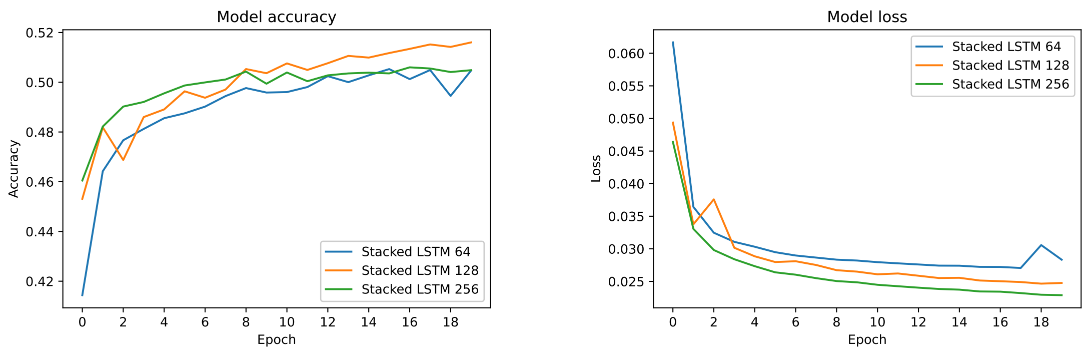

Figure 7.3: Training accuracy and loss of the stacked LSTM network

Stacked LSTM

This model consists of 3 stacked LSTM layers followed by a dense output layer shown in Table 7.3. Stacking LSTM layers makes the model deeper which is beneficial as depth of neural networks is generally attributed to the success of networks on a wide range of prediction problems. Input of the network is the same as the dense-only NN and the network also contains batch normalization and dropout layers for the same reasons as they were introduced in the dense-only NN. Output is once again a Dense layer with *sigmoid* activation function as in the dense-only NN. I trained 3 different version of this network differing in number of units[1] in LSTM layers:

Num. of LSTM units	Num. of trainable parameters
64	98,352
128	360,496
256	1,376,304

These 3 versions of the stacked LSTM were trained for 20 epochs on the whole dataset of choruses. Accuracy and loss of each version of the network is shown in Figure 7.3. As expected, all of the versions, including 64 units LSTM with only a fraction of trainable parameters compared to the dense-only network, reached higher accuracy on training data compared to the baseline dense-only network. Out of the three, the 128 units LSTM acquired the highest accuracy: 51.6%. It was unexpected that 256 units LSTM would reach the same accuracy as the 64 units LSTM in 20 epochs, in spite of having 14 times more trainable parameters. However, it is probable that 256 units LSTM would reach the same or higher accuracy than 128 units LSTM after a longer training.

[1]Increase of the number of LSTM units leads to the increase of the number of neurons in the LSTM cell.

	Layer	Output Shape
encoder	Input Layer	(*, 120, 48)
	LSTM + Batch Norm. + Dropout	(*, 120, 64/128/256)
	LSTM	(*, 64/128/256)
decoder	Repeat Vector	(*, 20, 64/128/256)
	LSTM + Batch Norm. + Dropout	(*, 20, 64/128/256)
	LSTM + Batch Norm. + Dropout	(*, 20, 64/128/256)
	Dense (Time Distributed)	(*, 20, 48)

Table 7.4: Architecture of the Seq2seq network

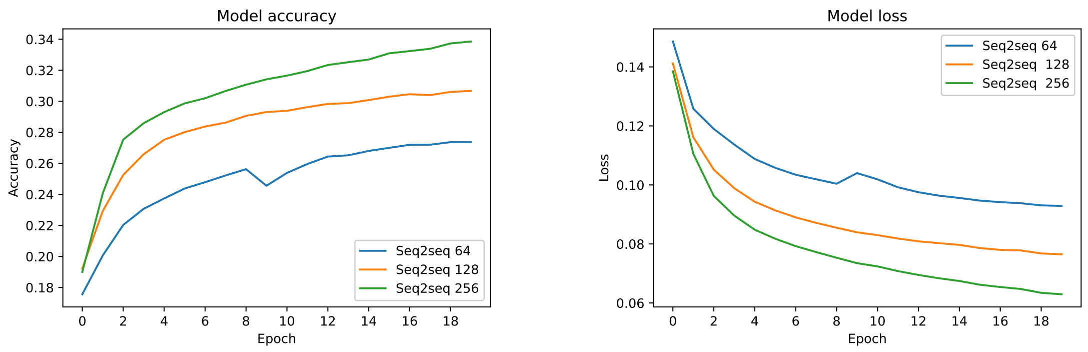

Figure 7.4: Training accuracy and loss of the Seq2seq network

Sequence-to-Sequence LSTM

This Sequence-to-Sequence (Seq2seq) model which uses multi-layered LSTM was adopted from Microsoft's Music Generation example [13]. In this example, the model was trained on pianoroll matrices of scale-chords dataset. Although the scale-chords dataset is much simpler and smaller than my dataset of choruses (it contains only 156 scale chords files in MIDI format), I wanted to try how this model scales to my more complicated task.

Architecture of Seq2seq consists of two main parts: *encoder* and *decoder* which are shown in detail in Table 7.4. Input of the encoder is the same as the input of the dense-only NN. Input layer of the encoder is followed by a group of LSTM, batch normalization and dropout layers – this LSTM layer returns whole sequences. Last layer of the encoder is LSTM layer which returns only the last output of LSTM cell, hence the shape (*, 64/128/256). Output of the encoder is an internal representation of an input pianoroll learned by the network. Decoder contains 2 groups of LSTM, Batch Normalization and Dropout and a single output Dense layer with *sigmoid* activation function. Output of the network is tensor with shape (*, 20, 48) which represents prediction of 20 columns of pianoroll (with actual length: 1 second).
I trained 3 versions of Seq2seq differing in number of units in LSTM layers:

Num. of LSTM units	Num. of trainable parameters
64	131,504
128	492,336
256	1,902,128

All of the 3 versions were trained for 20 epochs on the whole dataset of choruses and their accuracy and loss is shown in Figure 7.4. Highest accuracy was acquired by Seq2seq with 256 units model: 33.85%, which is less than the accuracy achieved by the baseline dense-only NN. This could be caused by the fact that the Seq2seq network is more complicated and takes more time to reach better accuracy or because this network predicts the next second compared to the dense-only NN which predicts only 1/20 of a seconds. The accuracy and loss of networks continued to improve steadily over the epochs. It is probable that longer training could lead to better results.

Chapter 8

Experiments and evaluation of results

This chapter describes my experiments and evaluates music pieces generated by created neural networks. To compare the networks to each other, I selected a few choruses from my dataset and used the first 6 seconds from each of them as a *seed*. These seeds were used to generate new 14 seconds of music by each of the neural networks. Each network was initialized to weights learned after the 20th epoch of its training. After a few experiments I decided to set the *threshold* (see 7.1) to value 0.1 as it produced the best results. After generating 14 seconds of new music from each seed, by each network, I plotted the pianorolls and saved them in MIDI and `.wav` formats.

8.1 Analysis of generated musical pieces

Figures 8.1, 8.2 and 8.3 displayed at the end of this chapter, show pianorolls generated by each network using 3 different seeds. Blue line separates the seed with a length of 120 time steps (equal to 6 seconds) and a new, generated part of the pianoroll which is 280 time steps long (equal to 14 seconds). All of the pianorolls are labeled with the type of network which generated given pianoroll and name of the song used as seed.

Figure 8.1 shows pianorolls generated from seed (chorus) from song Nothing Else Matters by Metallica, Figure 8.2 show pianorolls generated from seed (chorus) from song Layla by Eric Clapton and Figure 8.3 show pianorolls generated from seed (chorus) from song Hells Bells by AC/DC. These songs were selected because each of them shows a different style of guitar playing. Nothing Else Matters seed is a simple sequence of 4 chords whilst Layla seed is a more complicated sequence of chords with muted parts. On the other hand, Hells Bells seed does not contain chords – it is only a sequence of notes. These 3 songs should provide a good overview of capabilities of trained neural networks.

The plotted pianorolls contain these labels: label *Dense* refers to the baseline dense-only network, label *CNN Maxpool* refers to CNN which consists of convolutional and max-pooling layers, label *CNN Stride* refers to CNN which contains only convolutional layers with stride. Label *LSTM Stacked* refers to network with multiple LSTM layers – the version with 128 units, which achieved highest accuracy, and label *Seq2seq* refers to Sequence-to-sequence LSTM network – the version with 256 units, which achieved highest accuracy. All of these networks are described in Chapter 7.2.

Songs generated from Nothing Else Matters seed:

The pianorolls are shown in Figure 8.1 and MIDI files and WAV files of the generated songs are located in folder: `results/thesis_examples/n_e_m/`. Dense and LSTM Stacked networks continuously play the notes from the last chord in the input sequence, which is quite uninteresting and unwanted behavior. CNN Maxpool starts playing some additional notes but after a few seconds it stabilizes and continuously plays the same notes. CNN Stride and Seq2seq networks show more interesting behavior. CNN Stride plays a few quite harmonic chords/notes which do resemble the seed to some extent, whilst Seq2seq network plays many disharmonic notes without any structure. Both of these networks produce output without a sense of rhythm.

Songs generated from Layla seed:

The pianorolls are shown in Figure 8.2 and MIDI files and WAV files of the generated songs are located in folder: `results/thesis_examples/layla/`. Dense and LSTM Stacked networks produce output very similar to the previous one. They continuosly play the last notes from the seed. CNN Maxpool starts to play many disharmonic notes which do not resemble the initial input and after a while stabilizes once again. CNN Stride and Seq2seq networks both play some chords which were present in the seed but after a while they start to play random notes without a sense of rhythm.

Songs generated from Hells Bells seed:

The pianorolls are shown in Figure 8.3 and MIDI files and WAV files of the generated songs are located in folder: `results/thesis_examples/hells_bells/`. Dense, LSTM Stacked and CNN Maxpool continuously play a few notes (LSTM Stacked plays only a single note) and do not produce anything interesting. Seq2seq network plays a few harmonic notes with two muted parts. CNN Stride produces an interesting melody which however do not resemble the input seed.

Summary

The baseline dense-only network produced very simple and uninteresting music in each of the examples. Because the LSTM Stacked reached the highest training accuracy: 51.6% among the created networks, I assumed that it would produce the best and most musical results. Despite my expectations, LSTM Stacked produced musical pieces which were very similar to the baseline dense-only network. This could be caused by a short training (only 20 epochs) or because of the small number of LSTM layers. CNN Maxpool showed more variations in its output but overall played a lot of disharmonic notes and did not produce pleasing music pieces. Seq2seq network together with CNN Stride produced the best results. CNN Stride was arguably the most interesting music generator out of these networks.

However, even the tracks generated by these CNN Stride were not very good sounding. All of the generated tracks lacked a defined structure and a sense of rhythm. This was expected as the lack of structure and a sense of rhythm is common among many music generators [10]. Another problem was that networks often played disharmonic notes and the generated songs contained completely different notes and chords compared to their input seed. The output of the networks often stabilized and the network continued to play the same chord.

Further possible improvements:

Better results could *probably* be achieved by the following improvements. Deeper neural networks with more training parameters would probably generate better melodies, as the increase in depth of neural networks leads to better results in many machine learning tasks. On the other hand, such networks would require larger computational resources for the training. Different mapping of musical symbols to input/output of the network could also lead to better results. The mapping of musical symbols to pianoroll provides a lot of variability and the restrictions on what could be and could not be played are minimal. Mapping of musical symbols to a more strictly defined representation of music would probably be beneficial.

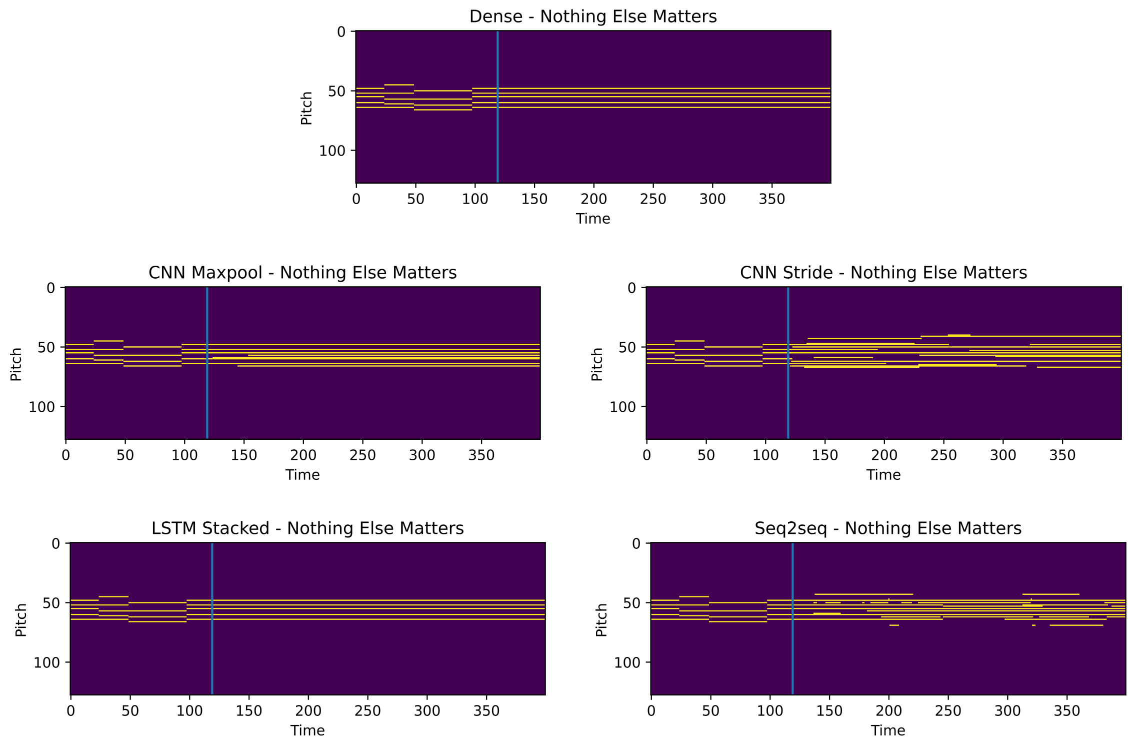

Figure 8.1: Music pieces generated from seed from song: Nothing Else Matters by Metallica. See folder: `results/thesis_examples/n_e_m/` with MIDI and WAV files of these songs.

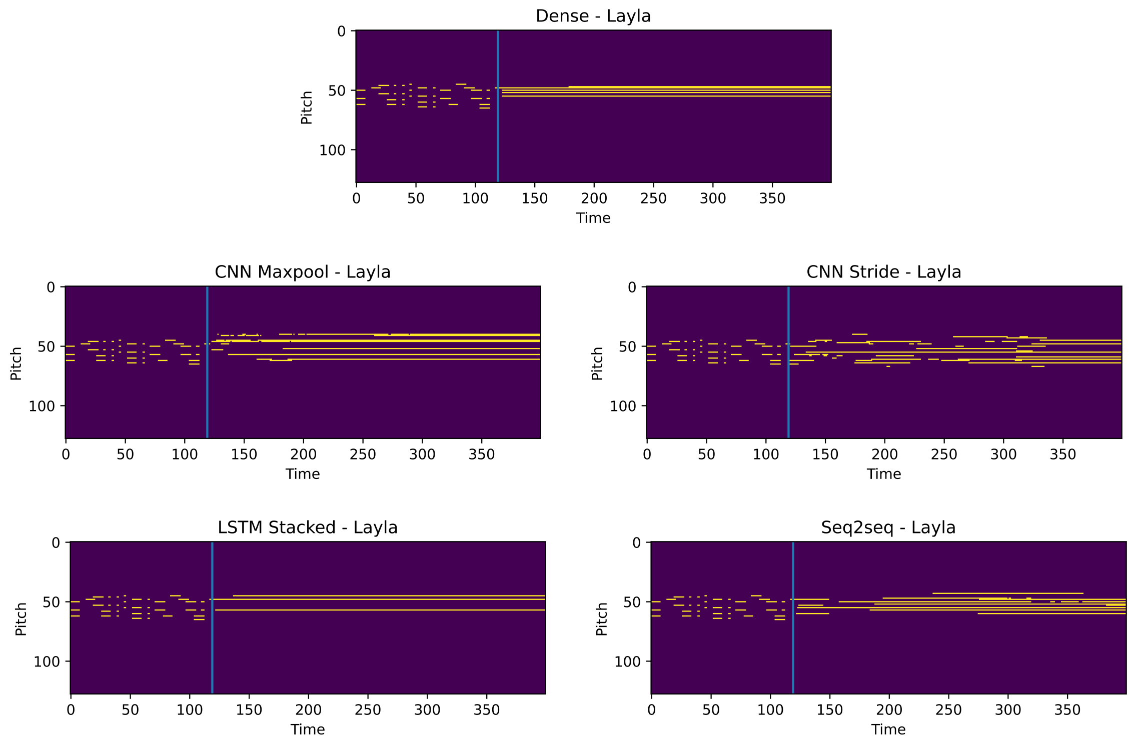

Figure 8.2: Music pieces generated from seed from song: Layla by Eric Clapton. See folder: `results/thesis_examples/layla/` with MIDI and WAV files of these songs.

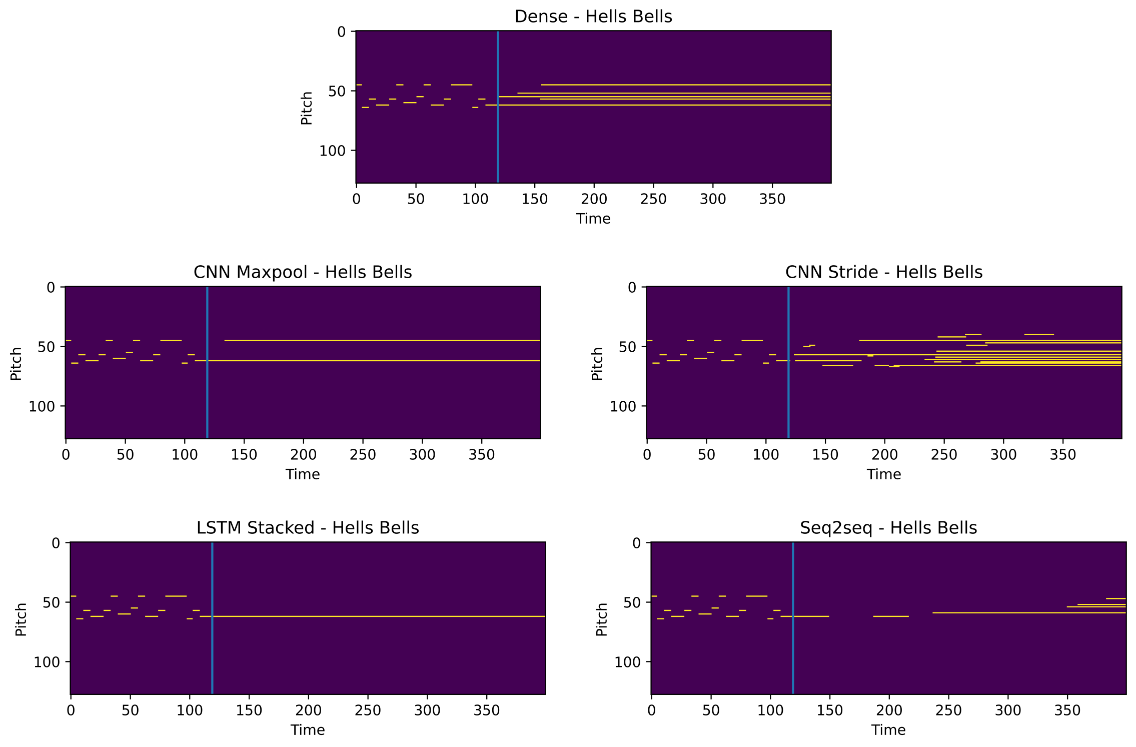

Figure 8.3: Music pieces generated from seed from song: Hells Bells by AC/DC. See folder: `results/thesis_examples/hells_bells/` with MIDI and WAV files of these songs.

Chapter 9

Conclusion

The goal of this thesis was to create a neural network which would be able to generate musical pieces. This task consisted of two different subgoals: creation of a dataset and training of a neural network.

The first subgoal was fulfilled only partially – I managed to create two separate guitar track datasets: dataset of choruses and dataset of verses in pianoroll format. Developed system of scripts is, however, capable of creating similar datasets of other song sections (eg. intro, solo, etc.) from any set of Guitar Pro songs.

The second subgoal was fully satisfied – I managed to train a few different types of neural networks – different LSTMs and CNNs – on the dataset of choruses. Each of the networks was trained for 20 epochs on the whole dataset. Among the created networks, the recurrent network with multiple stacked LSTM layers achieved the highest training accuracy: 51.6%. After the training, a few example tracks were generated by each of the networks. The generated tracks were evaluated and compared to each other. These tracks were, however, not as pleasing to the ear as expected. In spite of this, the results have shown that convolutional neural networks seem to be more suitable for the generation of polyphonic music than recurrent neural networks.

The results could probably be improved by using bigger and deeper neural networks, which would require more processing power, or by changing the mapping of musical symbols to input/output of the network.

Bibliography

[1] *Arobas Music: Guitar Pro 7 - Download* [online]. 2020 [cit. 2020-05-20]. Available at: https://www.long-mcquade.com/93892/Software/Recording-Software/Arobas-Music/Arobas-Guitar-Pro-7---Download.htm.

[2] *Artificial neural network* [online]. Wikipedia, the Free Encyclopedia, 2020 [cit. 2020-04-24]. Available at: https://en.wikipedia.org/wiki/Artificial_neural_network.

[3] *Convolutional neural network* [online]. Wikipedia, the Free Encyclopedia, 2020 [cit. 2020-05-20]. Available at: https://en.wikipedia.org/wiki/Convolutional_neural_network.

[4] *Long short-term memory* [online]. Wikipedia, the Free Encyclopedia, 2020 [cit. 2020-05-20]. Available at: https://en.wikipedia.org/wiki/Long_short-term_memory.

[5] Abadi, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Available at: https://www.tensorflow.org/.

[6] Caudill, M. Neural Network Primer: Part I. *AI Expert*. CL Publications. 1989.

[7] Cope, D. Panel Discussion. In: *Proceedings of the International Computer Music Conference*. 1993.

[8] Dong, H., Hsiao, W., Yang, L. and Yang, Y. *Lakh Pianoroll Dataset* [online]. [cit. 2020-05-02]. Available at: https://salu133445.github.io/lakh-pianoroll-dataset/representation.html.

[9] Dong, H., Hsiao, W., Yang, L. and Yang, Y. *MuseGan* [online]. Github, 2018. Available at: https://salu133445.github.io/musegan/.

[10] Fernández, J. D. and Vico, F. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research*. AAAI. 2013. Available at: https://arxiv.org/ftp/arxiv/papers/1402/1402.0585.pdf.

[11] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016 [cit. 2020-05-14]. Available at: http://www.deeplearningbook.org.

[12] Mareček, D. *Hudební improvizace v reálném čase*. Brno, CZ, 2010. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=117243.

[13] Menezes, E. *Music Generation with Azure Machine Learning* [online]. [cit. 2020-05-08]. Available at: https://docs.microsoft.com/en-us/archive/blogs/machinelearning/music-generation-with-azure-machine-learning.

[14] Music, A. *Guitar Pro* [online]. 2020 [cit. 2020-5-10]. Available at: https://www.guitar-pro.com/en/index.php.

[15] Nielsen, M. A. *Neural Networks and Deep Learning.* Determination Press, 2015. Available at: http://neuralnetworksanddeeplearning.com.

[16] Oliphant, T. E. *A guide to NumPy.* Trelgol Publishing USA, 2006.

[17] Papadopoulos, G. and Wiggins, G. *AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects.* AISB Symposium on Musical Creativity, 1999. Available at: http://www.doc.gold.ac.uk/~mas02gw/papers/AISB99b.pdf.

[18] Springenberg, J. T. et al. Striving for Simplicity: The All Convolutional Net. ICLR. 2015. Available at: https://arxiv.org/pdf/1412.6806.pdf.

[19] Stebner, A. *GuitarPro-to-MIDI* [online]. [cit. 2020-05-05]. Available at: https://github.com/alexsteb/GuitarPro-to-Midi.

[20] Teng, Y., Zhao, A. and Goudeseune, C. *MIDI Dataset* [online]. [cit. 2020-05-20]. Available at: https://composing.ai/dataset.