FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Generating music using neural networks
Student:	Jan Šimerda
Supervisor:	Mgr. Jan Tyl
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

This thesis aims to use NLP techniques for the task of music generation/transformation.
Sub-goals are the following:
- Research existing solutions
- Obtain suitable datasets for the task
- Preprocess dataset to be usable as a neural network-based training algorithm input, experiment with different representations
- Implement one or more models using modern neural network architectures such as LSTM, Transformer, etc...
- Discuss/compare model performance

Electronically approved by Ing. Karel Klouda, Ph.D. on 30 December 2021 in Prague.

Bachelor's thesis

GENERATING MUSIC USING NEURAL NETWORKS

Jan Šimerda

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Mgr. Jan Tyl
May 12, 2022

Contents

List of Figures

I would like to thank my supervisor Mgr. Jan Tyl for the leading of my bachelor's thesis and for his advices when working on this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.
I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act

In Prague on May 12, 2022 .....

Abstract

This thesis aims at using artificial neural networks in machine music generation. We emphasized using natural language processing techniques based on attention mechanisms. The work describes the whole music generation pipeline from data selection through *tokenization* (transforming music from raw data into a format digestible by selected models) up to model training.

We used state-of-the-art models based on the *Transformer* architecture commonly used in NLP to answer whether these well-performing models in domains like text generation or text translation can also be used to generate music. We also tested some proposed enhancements to the Transformer model and the attention mechanism and compared them to the vanilla Transformer model.

We used the MAESTRO dataset for the training process that contains hundreds of hours of classical piano pieces. The songs used for training the models are in symbolic MIDI representation.

We found out that the original Transformer is not suitable for the music generation task an it's better to use Music Transformer that reaches 25.13 % accuracy on test set.

Keywords music generation, MIDI, transformers, natural language processing, artificial neural networks, PyTorch, Python

Abstrakt

Tato bakalářská práce zkoumá využití umělých neuronových sítí v oblasti strojového generování hudby. Zaměřili jsme se na využití technik používaných ve zpracování přirozeného jazyka založených na attention mechanismu. Práce popisuje celý proces generování hudby od výběru dat, přes *tokenizaci* (transformace hudby ze surových dat do formátu vhodného pro vybrané modely) až po trénování modelu.

Použili jsme nejmodernější modely založené na *Transformer* architektuře, běžně užívaných v NLP, abychom získali odpověď na otázku, jestli lze tyto modely, které mají skvělé výsledky v doménách generování textu a strojového překladu, použít také pro generování hudby. Vyzkoušeli jsme také několik navrhovaných vylepšení Transformer modelu a attention mechanismu a porovnali je s původním Transformer modelem.

Pro trénování neuronových sítí jsme využili dataset MAESTRO, který obsahuje stovky hodin klasických klavírních skladeb. Skladby použité pro trénování modelů jsou v symbolické MIDI reprezentaci.

Zjistili jsme, že originální Transformer není pro tvorbu hudby vhodný a je lepší zvolit Music Transformer, který dosahuje přesnosti 25,13 % na testovací sadě.

Klíčová slova generování hudby, MIDI, transformers, zpracování přirozeného jazyka, umělé neuronové sítě, PyTorch, Python

Lists of abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
GAN	Generative Adversarial Networks
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MAESTRO	MIDI and Audio Edited for Synchronous TRacks and Organization
MIDI	Musical Instrument Digital Interface
NLP	Natural Language Processing
RNN	Recurrent Neural Network

Introduction

From the beginnings of modern humanity to this day and age, music has played an ever-present part in our lives. Up until the upswing of the computer era, music composing was carried out mainly by humans, though there were some experiments with algorithmic music generation even before the first computer[1]. However, after scientists developed the first computers, people became interested in whether computers could also perform complex and creative tasks like self-driving or music generation.

Lately, *artificial* intelligence has crept into our lives more than ever before. From fraud detection, personalization, and content recommendation to image enhancement and facial recognition, AI helped drive all those things forward. The inventions in artificial intelligence and machine learning techniques, along with advances in computer hardware performance, helped tremendously with the ability to perform complex tasks like those mentioned above.

Even music did not escape this evolution, and there have been attempts at using *RNN*s, *GRU*s, and *LSTM*s for automated music generation with promising results. With the *Transformer* being one of the newer models, not many papers exist on utilizing it for music composition, in contrast to recurrent neural networks.

This thesis aims to research possible ways of generating musical compositions using *artificial neural networks*. Specifically, this work focuses on leveraging neural network architectures used in *natural language processing* (GRU, LSTM, Transformer, …). The goal of the practical part of this work is to create a functioning music generation model that would be able to generate songs from scratch or generate continuation for an existing song. This solution could help music composers with the creative part of music composition.

[1] the musical piece was composed using defined musical segments selected by dice roll [1]

Chapter 1

Music theory

Since our goal is to research music and see how we can compose it algorithmically, it is reasonable to start with some fundamentals of music theory. This chapter will cover the basics of sound, music, and different music terminology.

1.1 Sound

We can define *sound* as an auditory sensation we perceive when exposed to certain types of atmospheric disturbances (sound waves). Sound waves are produced by a vibration of some source, like human vocal cords, an instrument, or a loudspeaker. [2]

1.2 Music

Music can be defined as "organized sound" in the broadest possible sense. This open-ended and safe definition is coherent regardless of era, style, culture, or the mechanics of musical organization. Each successive historical era produces musically artistic expressions of its own time and musical aura. The study of *Music Theory* is how we investigate this. [3]

1.3 Music theory

As described by [3], Music Theory is a scientific study of music and its organizational characteristics. Its purpose is to examine questions like how we perceive music aurally, how we experience music aesthetically, and how we can symbolize it visually. We can learn to associate sounds with symbols to aid our ability to perceive music at levels of increasing depth and better our comprehension.

Moreover, [3] states that while studying music, we employ two approaches:

- *Analysis* – we learn to employ commonly accepted techniques and specialized language to describe the musical organization. These techniques share analytical language throughout the community of musicians. *This is conceptual knowledge and evaluation.*

- *Composition* – either by actively creating our own works or (as is the case of this thesis) imitating or emulating the works of earlier composers. *This is active knowledge and procedure.*

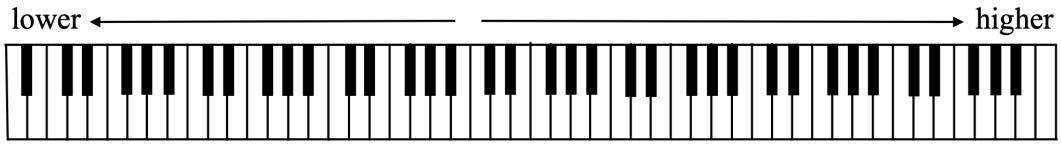


Figure 1.1 A piano with 88 keys and indicated pitches [4]

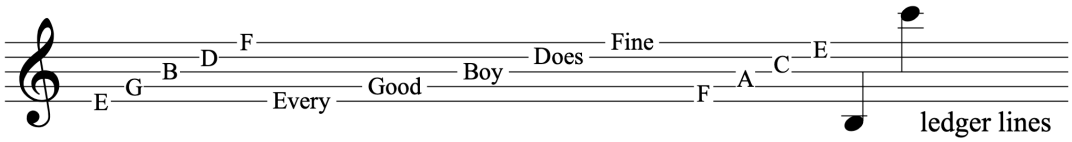


Figure 1.2 The staff with a treble clef [4]

1.4 Pitch

We can describe *pitch* as a perceived highness or lowness of a sound; this directly corresponds to the frequency of the sensed sound. On a piano, there are 88 *notes*. Each of the notes corresponds to a different pitch. Notes placed on the left side of a piano correspond to a lower pitch, and as we go to the right side of the piano, the pitch gets progressively higher 1.1. [4]

1.5 Notation

Notes are written on a five-line *staff* (figure 1.2). A *clef* orients the lines to a reference point. For example, when placed on a five-line staff, the *G clef* becomes the *treble clef*, the most well-known *clef*. In treble clef, the notes on the lines are E–G–B–D–F from lowest to highest, often remembered through the traditional mnemonic[1]. The spaces are F–A–C–E from lowest to highest. *Staves* (the plural of "staff") are extended by the ledger lines (figure 1.2). [4]

1.6 Octave registers

The note names used in music are *ABCDEFG* (known as the "musical alphabet"). After G, note A returns, and *ABCDEFG* occurs again. An octave is a distance from any note to the same note in the next or previous register. A piano also contains so-called *accidentals* (special keys that raise or lower a note's pitch). The piano keyboard with 88 notes consists of seven octaves (composed of seven notes and five accidentals) along with three extra notes and one extra accidental. [4]

When learning about octave registers, we focus on note C for reasons that will soon become clear after learning about the major scale. We use octave registers (C4, D5, …) to specify the note's exact register. The note C4 is known as **"middle C"** and is a vital reference point. See the keyboard in the figure 1.3. [4]

Notice that the register number changes after note B each time (e.g., B4 is followed by C5). In the treble clef notation, middle C is placed on the *ledger line* below the staff. In the bass clef, the *middle C* is placed on the *ledger line* above the staff. Both notations are visible in figure 1.4. [4]

[1] Every Good Boy Does Fine

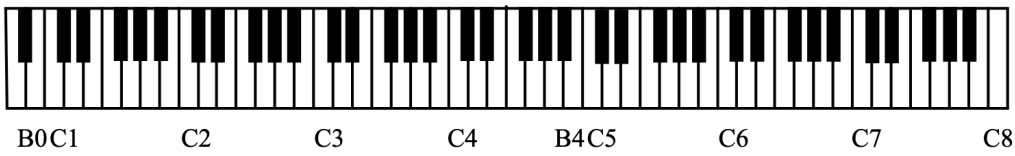


Figure 1.3 A piano with octave registers denoted [4]

Figure 1.4 Middle C (C4) in treble clef and bass clef [4]

When we join the treble and the bass clef together by a bracket, we create the so-called **grand staff**, how piano music is written. An example of the **grand staff** is shown in figure 1.5. [4]

1.7 Accidentals

Accidentals are characters that we use to modify the following note (either raise or lower the note pitch). The following five symbols exist [4]:

- *Sharp* symbol (♯) raises pitch half a step
- *Flat* symbol (♭) lowers pitch half a step
- Double *sharp* symbol (𝄪) raises pitch two halves a step (a whole step)
- Double *flat* symbol (𝄫) lowers pitch two halves a step (a whole step)
- *Natural* symbol (♮) cancels out accidentals previously applied in a measure of Major Key Signatures or Minor Key Signatures

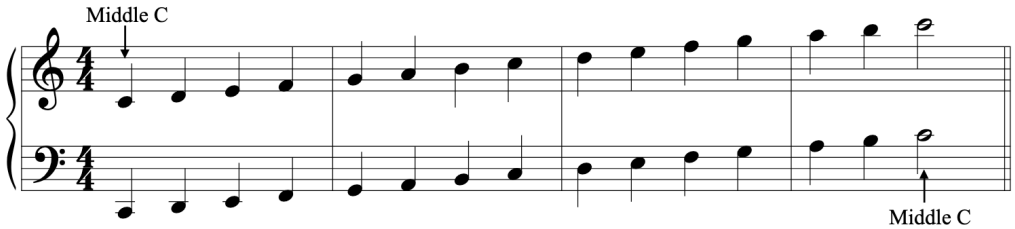


Figure 1.5 The grand staff [4]

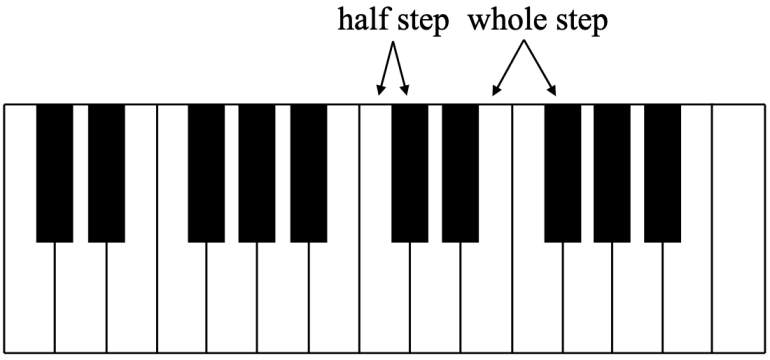


Figure 1.6 The half step and whole step [4]

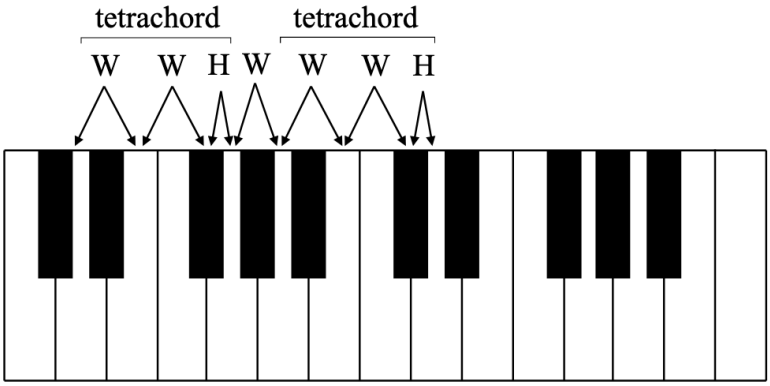


Figure 1.7 The D major scale on a keyboard [4]

1.8 Half Steps and Whole Steps

A half step on a piano keyboard is the distance from one note to the following immediate note. A whole step composes of two half steps (figure 1.6). [4]

1.9 The Major scale

A specific sequence of whole and half steps is called a *major scale*. It is helpful to think of the pattern as consisting of two *tetrachords*[2] and a single whole step. The lower *tetrachord* is of the following pattern: whole step, whole step, half step. A whole step then joins both *tetrachords* together. The upper *tetrachord* consists of the same pattern as the lower one: whole step, whole step, half step. If we use W for the whole step and H for the half step, we can write the major scale pattern as W–W–H, Whole–step connection, W–W–H. [4]

Note that all *major scales* use the notes of the musical alphabet in order; no notes get skipped, and no note occurs twice. In figure 1.8, the first four notes are D–E–F$^{\sharp}$–G, not D–E–G$^{\flat}$–G. In D–E–G$^{\flat}$–G, G incorrectly occurs twice, and the F$^{\sharp}$ between E and G gets skipped. [4]

[2] *"a tetrachord is a four-note scale segment" [4]*

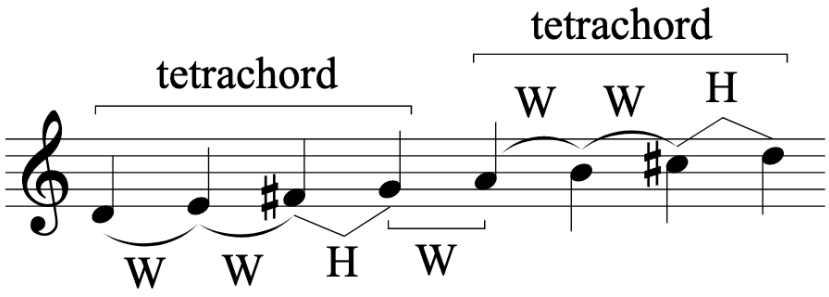


Figure 1.8 The D major scale in treble clef [4]

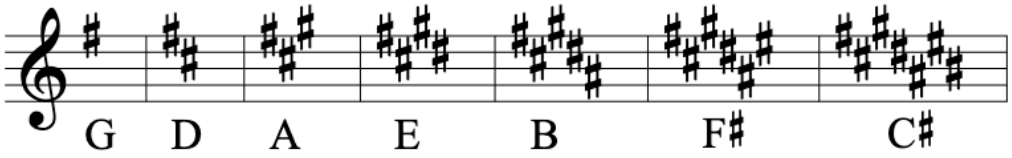


Figure 1.9 Major key signatures in sharps [4]

1.10 Major key signatures

The key signature is a notation positioned next to the clef at the beginning of a piece or section. We use it to hint which sharps or flats are in the piece's scale to prevent the composer from writing every sharp/flat from the scale each time it occurs. [4]

There are **15** major *key signatures*. The key of *C major has no sharps or flats* in the key signature, while the other key signatures can have between *1 to 7 sharps* and *1 to 7 flats*, resulting in the other 14 key signatures. Notations of the major key signatures can be seen in figures 23 and 24 for sharps and flats, respectively. [4]

"A helpful learning device to remember the order of keys in relation to the order of sharps and flats is the ***circle of fifths****. As you ascend in fifths (clockwise), key signatures get one degree 'sharper.' (C to G is a fifth because C=1, D=2, E=3, F=4, and G=5.) As you descend in fifths (counterclockwise), key signatures get one degree 'flatter.'"* [4] (figure 1.11)

Notice the overlapping keys at the bottom of the circle. B major is enharmonically[3] the same as C♭ major, F♯ major is enharmonically the same as G♭ major, and C♯ major is enharmonically the same as D♭ major. [4]

[3] pitches that are the same notes on a piano but are written differently on the staff

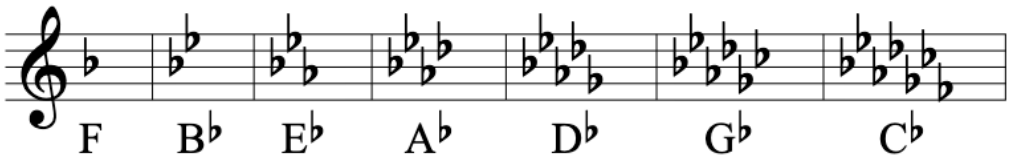


Figure 1.10 Major key signatures in flats [4]

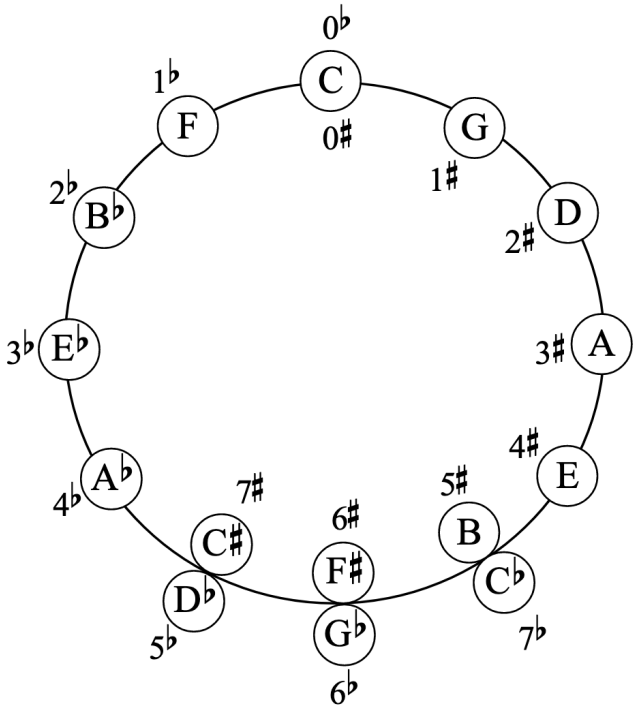


Figure 1.11 Circle of fifths in major keys [4]

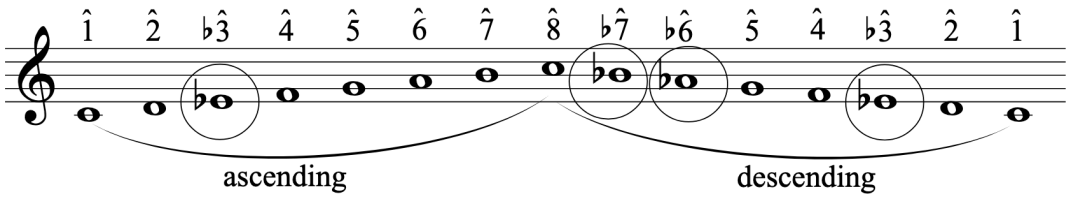


Figure 1.12 Melodic minor scale [4]

1.11 Minor scales

Alongside the *major scale*, there are also three *minor scales*: the *natural minor scale*, the *harmonic minor scale*, and the *melodic minor scale*. The *melodic minor scale* has an ascending version, and a descending version that is the same as the *natural minor scale*. Both can be seen in figure 1.12.

1.12 Minor key signatures

Minor key signatures agree with the notes of the natural minor scale. Since the C natural minor scale had E♭, A♭ and B♭ accidentals, the key signature of C minor has three flats, written in the order of flats (B♭, E♭, A♭). [4]

A *minor key signature* will have three lowered notes—the third, sixth and seventh—related to the corresponding major key signature. We use the term *parallel minor* when referring to a minor scale (e.g., the parallel major of F minor is F major) with the same first scale degree (in this case C) as the major. One method of figuring out a minor key signature is to add three flats

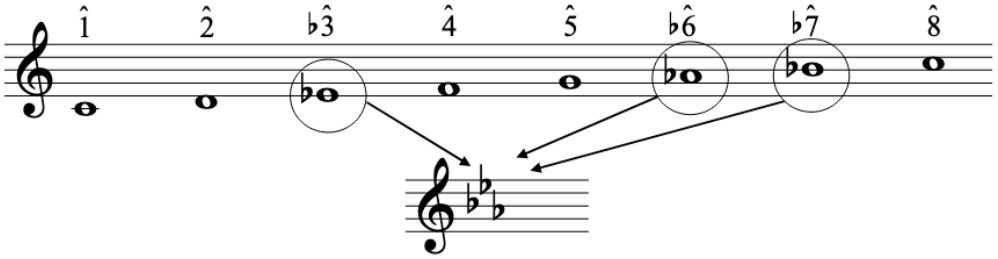


Figure 1.13 Natural minor scale in major key signature [4]

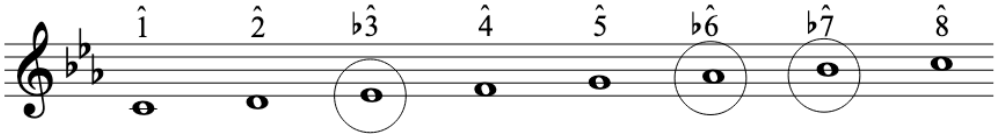


Figure 1.14 Natural minor scale in minor key signature [4]

(or subtract three sharps) to the parallel major key signature. When writing below the five-line staff to designate keys, we use upper case for major keys and *lowercase for minor keys.* [4]

We also add figures of minor key signatures (figure 1.16) and circle of fifths (figure 1.17) for minor scale for completeness.

1.13 Time signature

The staff can also contain a *time signature* next to a clef. We denote it as two stacked numbers; the lower number is typically a number corresponding to a power of 2 and tells us the relative duration of a note, while the upper number hints at how many pulses (or *beats*) we can expect per *bar*[4]. [4]

1.14 Durational symbols

The most common *time signature* is $\frac{4}{4}$ (also known as "common time"). It makes sense to introduce *durational symbols* in the context of $\frac{4}{4}$ time signature because a whole note takes up a full measure in $\frac{4}{4}$, a half note takes up half a measure of $\frac{4}{4}$, a quarter note takes up $\frac{1}{4}$ of a measure, and so on.

Meter describes the number of beats in a measure (also called a bar) and how we typically divide the beats. A beat is a basic pulse measured in music and thus the unit in which we think about music. Pulse and beat are interchangeable. The speed of a beat is called tempo. We can state tempo in beats per minute (*bpm*), such as 60bpm (where the rate of the beat would be equal

[4]specified segment of time corresponding to the number of beats

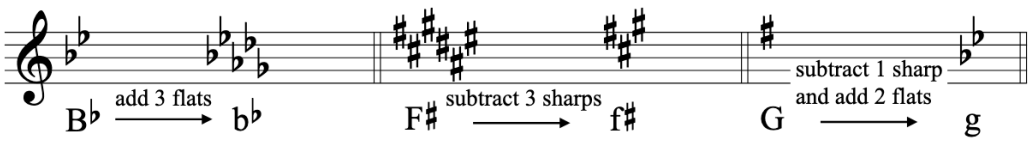


Figure 1.15 Parallel minor keys signatures [4]

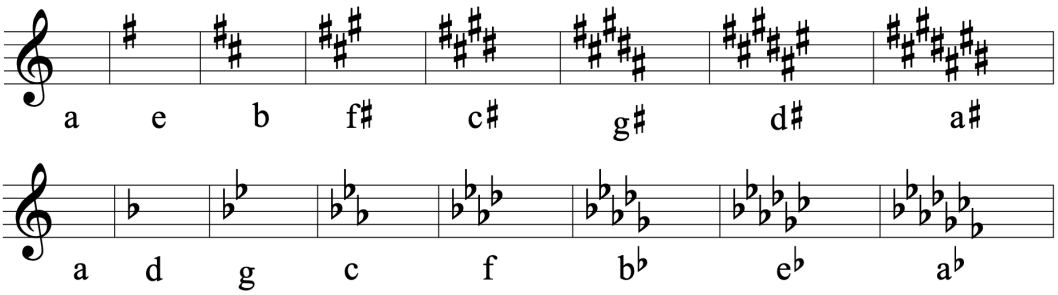


Figure 1.16 Minor keys signatures [4]

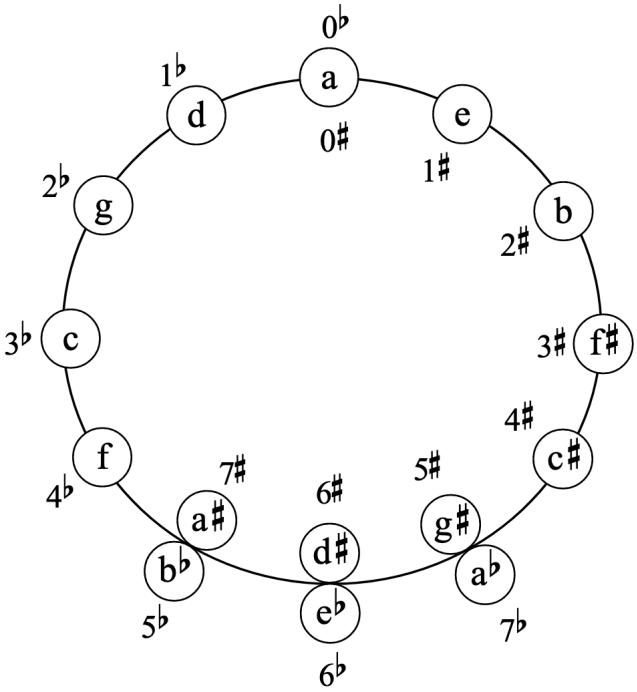


Figure 1.17 Circle of fifths in minor keys [4]

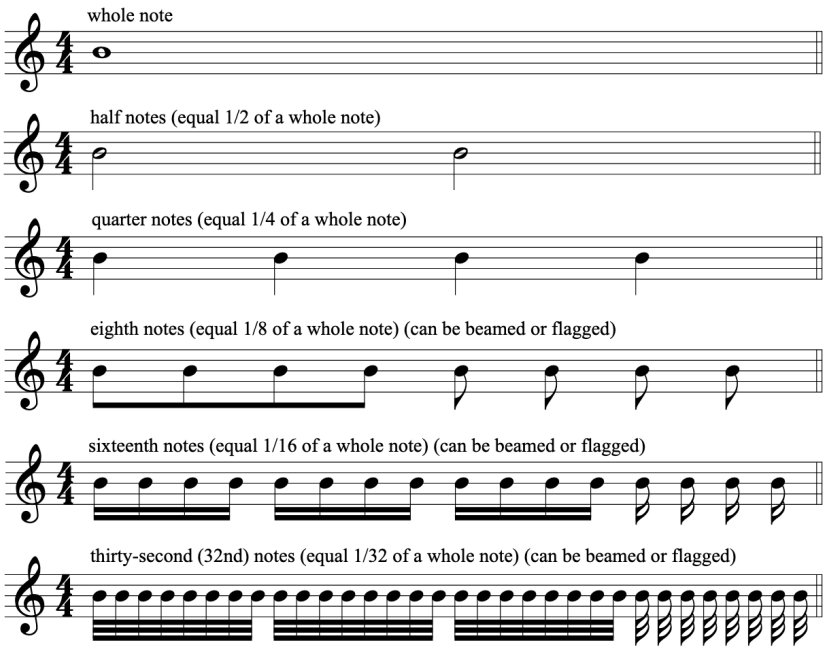


Figure 1.18 Different notes used for a common time [4]

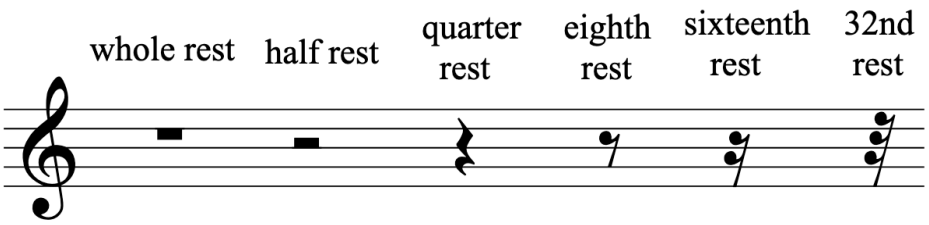


Figure 1.19 Durational symbols for rests [4]

to a second), or, in classical music, with terms like Allegro, Andante, and Adagio, sometimes in combinations with "M.M." for Maelzel's Metronome. [4]

Some meters have a special name; meters with two beats in a bar are called *duple*, three beats in a bar *triple*, and four beats in a bar *quadruple*. Meter is described as *simple* if the beats are normally divided into two parts and *compound* if the beats are normally divided into three parts. [4]

Chapter 2

Automated music generation

When talking about automated music generation, we may want to distinguish between composition assistance (software that is not generating the music from scratch but instead helps the composer incrementally with suggestions, auto-completion, etc.) and autonomous music generation (which takes over the whole music composing process and users are restricted just to parametrization of the generation process). In this chapter, we will focus on the latter. We will briefly go over the history of the music generation and then move on to modern techniques utilizing machine learning techniques. [1]

2.1 Pre-computer techniques

First experimentations with *algorithmic composition* took place in the late 15th century by employing **"canonic composition."** [5]

"The prevailing method was to write out a single voice part and to give instructions to the singers to derive the additional voices from it. The instruction or rule by which these further parts were derived was called a canon, which means 'rule' or 'law.' For example, the second voice might be instructed to sing the same melody starting a certain number of beats or measures after the original; the second voice might be an inversion of the first or it might be a retrograde [etc.]" [6]

These *rules* of imitation and manipulation form an *algorithm* by which performers unfold the music. In this automatical process, we see a clear removal of the composer from a large portion of the compositional process: the composer himself only invents a core of the music, a single melody or section. [5]

Wolfgang Amadeus Mozart experimented with automated composition techniques using the so-called *Musikalisches Wurfelspiel* ("musical dice game"). The game worked by joining several predefined musical segments selected by dice roll. This simple form of *stochastic* algorithmic composition left the creative decisions in the hands of chance, letting the dice roll decide what notes to use. [5]

2.2 Use of computers

There are three possible approaches when using a computer to generate a composition:

- Stochastic
- Rule-based

	Sunny	Windy	Rainy
Sunny	0.6	0.3	0.1
Windy	0.7	0	0.3
Rainy	0.5	0.2	0.3

Figure 2.1 Markov chain represented by transition matrix [7]

Artificial intelligence

The *stochastic* way involves *randomness* and can be as simple as Mozart's *Musical dice game* we already briefly touched upon; however, we can also use more complex methods like *statistical theory* and *Markov chains*. Many creative decisions are merely left to chance when generating a composition using the stochastic method. Another example of non-computer-oriented stochastic composition can be found in Karlheinz *Stockhausen's Klaveirstucke XI*, in which the sequence of various fragments of music is to be performed by a pianist in random order. [5]

2.2.1 Markov chains

So-called *Markov chains* are a major technique for generating musical compositions using stochastics. We define the Markov chain using a simple sequence of random variables $X_1, X_2, X_3, \ldots, X_i$[1]; we call this sequence a *stochastic process*. For this process to be the Markov chain, the following equivalence must be true:

$$[X_i|X_1, \ldots, X_{i-1}] \sim [X_i|X_{i-1}]$$

In this context, the equivalence means that the *probability distribution* on the left-hand side is equivalent to the probability distribution on the right-hand side. For a fixed value of i, X_i is called the state of the Markov chain. The equivalence implies that the value of i^{th} state is purely dependent on the immediately previous state, a trait also called *memoryless*. [7]

Markov chains are primarily represented in two ways a *transition matrix* (figure 2.1) or a *directed graph* (figure 2.2). The transition matrix M is a matrix with dimensions n by n, where n is the number of different states the Markov chain maintains. The $M_{a,b}$ value then represents $\mathcal{P}(b|a)$, the probability of transition from the state a to state b. This representation is practical for use in computers. The directed graph is a good representation for visualization; each *vertex* represents a state, and the *directed edges* represent the probability of a transition between two states. [7]

2.2.1.1 Generating music

Once we have the Markov chain defined, we can use it to generate music in a simple manner; we want to create a model that will contain sound objects (notes or chords and their duration) as states and probabilities of transitions between them. We do this by using existing music pieces and using them as training input. We compose the states by extracting all different sound objects occurring in the training pieces. We can compute the transition probabilities by gathering all possible *bigrams* (sequences of two adjacent objects). The probability of transition from the state a to state b is then computed by following division $\mathcal{P}(b|a) = \frac{\#ab}{\#ac}$, where ab represents bigrams of sound object a followed by sound object b, and ax represents *all bigrams* starting with the object a. Using this method we compose the whole transition matrix.

With the transition matrix available, we can move on to the music generation itself. In order to utilize the matrix for generating transitions, we first need to select the first sound object the

[1] the index i in this context is sometimes referred to as the time

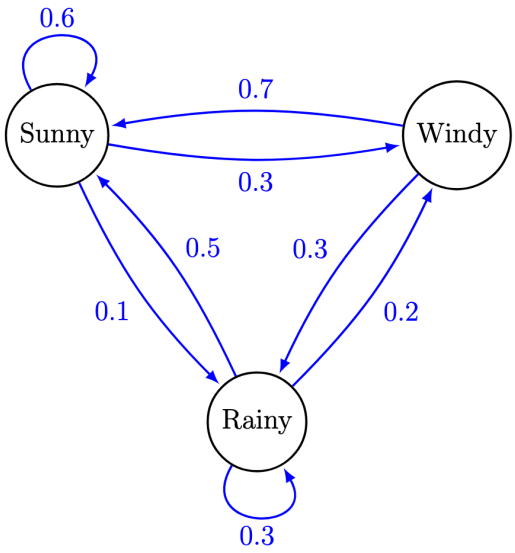


Figure 2.2 Markov chain represented by directed graph [7]

musical piece will start with. We can do that by manually picking the desired sound object, or we can generate it randomly by composing a vector of probabilities of starting sound objects, where the probability of each object being the starting object is the number of times it was starting object in training musical pieces divided by the total number of training musical pieces. So to generate the piece, we select starting object from the initial vector (the likelihood of selecting an object is determined by its computed probability). Then for every other state, we receive the vector of probabilities by using a row of the transition matrix corresponding to the current state. We iteratively continue until we are satisfied with the length of a piece. [7]

2.2.2 Rule-based music generation

Music theory traditionally describes rules that help to direct the compositional process. While composers regularly break those rules, they can be successfully used to implement a system for generating music. One of the examples would be the *Illiac Suite*, composed in 1957 by professors *Lejaren Hiller* and *Leonard Issacson*, where the rule-based system was used to help generate the first two movements. [8]

"The general idea is to use screening rules to accept or reject randomly generated pitches and rhythms. Probability distribution and Markov processes can also be found in the suite." [9]

2.2.2.1 Formal Grammars

In the 1950s, *Noam Chomsky* introduced the concept of *Generative Grammars*, a tool for analyzing language that became highly influential in linguistic studies. In a Generative Grammar, two alphabets of *terminal* and *non-terminal symbols* are used, along with a set of *rewriting rules* given over the union of these two alphabets that allow transforming non-terminal symbols (or string of non-terminal and terminal symbols) into other symbols (both terminals and non-terminals). The generated *language* is the set of all possible strings of terminal symbols generated from a special starting variable (usually called S) and applying any number of rewriting rules in sequence. These Grammars can be seen as an implementation of the beforementioned rule-based systems. [8]

Lindenmayer Systems (L-Systems) are a variant of Generative Grammars used for music generation; the difference from Chomsky's Grammars is that they implement *parallel rewriting*, applying all the rewriting rules at once instead of only one at a time. This characteristic makes these systems less inclined to generate sequential data, like simple melodies, and have been used to generate stunning visual effects. When applied to music generation, a common approach was to map visual data generated by *L-systems* to score information or to a sequence of musical segments. [8]

One of the most influential researchers of rule-based music generation is *Ebcioğlu*, who implemented a custom logic language that he used to create *CHORAL*, a system for the generation of Bach-like chorales that uses some 350 rules for harmonization and generation of melodies [10]. The hardship of designing such a system lies in the complexity of explicitly coding enough rules, many of which often do not have a formal definition in musicology literature. [8]

2.2.3 Artificial intelligence

"Artificial Intelligence (AI) is the property of machines, computer programs and systems to perform the intellectual and creative functions of a person, independently find ways to solve problems, be able to draw conclusions and make decisions." [11]

AI is a buzzword that contains two main branches, the original *symbolic AI* and *machine learning*. The symbolic AI are systems where we capture knowledge using formal mathematical logic, genetic algorithms, state-space search, automated planning, … . The machine learning AI builds models with a set of hidden internal parameters we are trying to fine-tune so that the model performs well on predefined metrics; this optimization process is called learning. In this thesis, we will specifically focus on *artificial neural networks*, which are subset of ML techniques.

The increased computational power of computers and the widespread general-purpose GPU programming recently made *deep learning*[2] techniques extremely popular, with applications spanning from *NLP* to *image processing* to *music generation*.

While the interest in these algorithms grew exponentially in the last decade, the first music generation system to use ANNs was that of Peter M. Todd[12], who used a three-layered *Recurrent Neural Network* (RNN) to generate monophonic melodies. Recurrent Networks reuse the results of the computations from previous steps every time a new input is fed, allowing them to encode temporal sequences. Still, standard feed-forward networks are also an option for music generation. There is also room for standard feed-forward networks: in 1991, J. P. Lewis trained a network[13] with musical patterns ranging from random to well-constructed to learn a measure of "musicality" used by his music generation system to select pleasing compositions.

As mentioned, RNNs are a popular choice for music generation. In particular, *LSTMs*[14] are a special variant of recurrent networks that use gates to decide the amount of information taken from novel input and what is maintained from older inputs, hence the memory. The first LSTM used for music generation was applied to blues improvisation[15]. Another deep learning approach is that of Generative Adversarial Networks (GANs)[16]; the concept behind this method is to train two networks at the same time, one generates musical compositions imitating what is learned from real-world examples, and the other tries to discriminate between original and imitated compositions. As one network gets better, the other must improve as well in order to "beat" the other network (therefore making them "Adversarial").

[2] use of ANNs with more than three hidden layers

Chapter 3

Digital representation of music

In reality, sounds manifest as continuous waves propagating through a medium (like air). On the other hand, personal computers are digital by design and cannot easily represent continuous information; therefore, when storing audio in a computer, we have to perform conversions for the computer to accommodate our audio track. This chapter will look over possible representations of music representation in a computer.

3.1 Wave representation

The first possible option when representing an audio track in a computer is to try and capture characteristics of the beforementioned *sound wave*. In order to manipulate this *continuous signal* via a digital computer, the signal must be digitalized with an *analog-to-digital converter* (A/D). The converter repeatedly samples the instantaneous *voltage amplitude* of the *analog* input signal at a given sample rate, commonly thousands or tens of thousands of times per second. In the audio signal, the measured voltage of the input signal is proportional to the sound pressure measured by a device such as a *microphone*. The final discrete representation of the converted signal created by the converter consists of a sequence of numeric values (hence the term digital) representing the amplitude of the original waveform at evenly spaced points in time. [17]

3.2 Symbolic representation

Another option is to represent music *symbolically*. Unlike the wave representation, which represents music (any audio generally) in a low-level physical fashion, the symbolic representation uses *special symbols* to represent different musical elements like notes and rests.

The downside of this representation is that it can only depict specific musical instruments, unlike wave representation, which can also depict vocals and arbitrary sounds. However, this approach is much better suited for our application, as it expresses the essential musical properties. Therefore, it will be much easier for our neural nets to learn the intrinsic properties of compositions provided as learning data and thus will presumably be more successful at imitating them. We will now go over the most common symbolic formats for music.

3.2.1 MIDI

The MIDI abbreviation stands for the *Musical Instrument Digital Interface*. It is a protocol developed in the early 1980s to standardize the *exchange and storage* of musical information. It

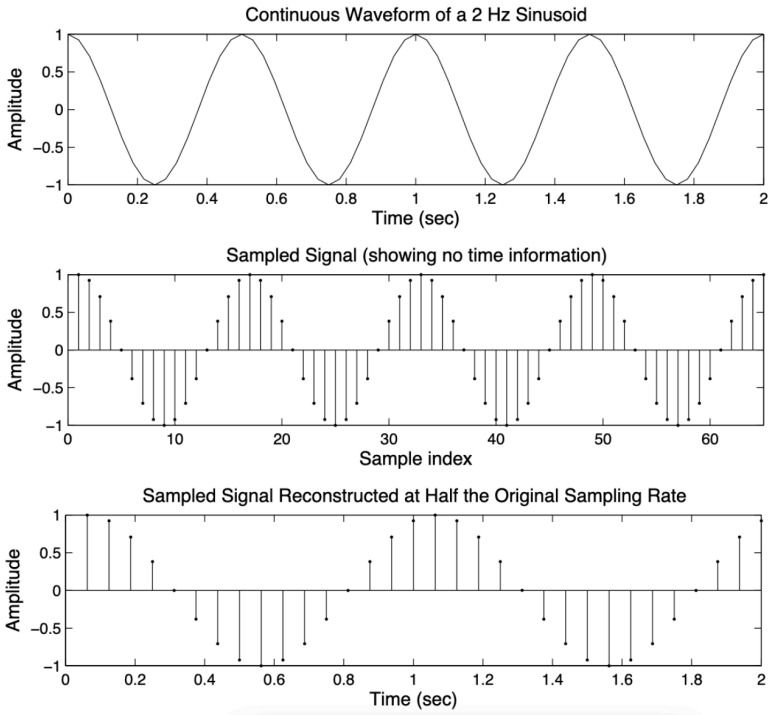
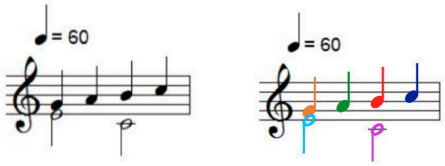


Figure 3.1 Digitization of signal via sampling [18]

is the most widespread *binary* communication protocol intended to connect electronic musical instruments such as synthesizers and other electronic music equipment to a computer for recording, editing, and programming. MIDI protocol appeared as a response to the need to standardize the communication between the synthesizers and other musical equipment and was created when electronic music was developed by a consortium of Japanese and American manufacturers of Synthesizers (Sequential Systems, Roland Corporation, Yamaha, Kurzweil, …). It is used to transmit data using serial ports but can also be written to a file and be used as a means of storage. MIDI is an extensive standard consisting of hardware, drivers, communication channels, messages, modes, controllers, visual effects control, and a file format standard. In this work, we will only concern ourselves with the *MIDI file format*, as it is the thing that we will need during the implementation part. Since the protocol was developed in the 80s, it was designed to have low overhead and, therefore, *low-level*. We do not need to go into the detail of binary encoding used to describe different MIDI events; instead, we will present an abstracted high-level overview, which is sufficient for our application. [19]

3.2.1.1 MIDI file format

The MIDI file starts with a header, where the format type (single track, multiple synchronous tracks, or multiple asynchronous tracks), the number of tracks, and time division (ticks per beat). The file body composes of an array of MIDI *messages*. Each message denotes the message content and Δ (delta) time; the amount of ticks the event is shifted after the preceding event. There are some *meta-messages* that specify additional information like song lyrics, tempo, state end of the track, and others. There are multiple message types like `control_change` for specifying pedal/slider position or program change to choose one of 128 instruments. However, the most substantial messages are `note_on` and `note_off`. These take arguments pitch (0-127) and velocity (0-127) and specify the pressing and releasing of a note specified by pitch number. The velocity controls the force of a note being pressed. Zero velocity has the same meaning as the `note_off`



(a) score (b) colored score

Figure 3.2 Example of a musical score [19]

message, so the release of a key. Please take a look at figure 3.2, which contains an example of a score. [19]

This would translate into following messages in MIDI:

midi message	pitch	note value
note_on	64	E4
note_on	67	G4
note_off	67	G4
note_on	69	A4
note_off	69	A4
note_off	64	E4
note_on	60	C4
note_on	71	B4
note_off	71	B4
note_on	72	C5
note_off	72	C5
note_off	60	C4

Chapter 4
Neural networks

Artificial neural networks (ANNs), or just neural networks, are a class of mathematical models used for various tasks, including data classification, self-driving, chatbots, sentiment analysis, time series prediction, computer vision, art generation, and many more. As the name suggests, ANNs are a set of artificial *neurons* specially arranged into so-called layers. Neural networks are somewhat inspired by biological neural nets; hence we will now briefly examine them. [20]

The brain consists of many neurons (figure 4.1) consisting of dendrites, a cell body, and an axon. Dendrites are branched connections of a neuron that received propagate the electrochemical stimulation received from other neurons and send them to the cell body, where the signals are summed up, and once the triggering threshold is reached, the signal propagates through the axon. The last part of a neuron is the axon, the long connection leading from a neuron that transmits a signal to different neurons, muscles, or glands. Neurons can communicate with other neurons' dendrites and other body parts via these connections, so-called synapses, and pass on their electrochemical potential. A depiction of the triggering threshold and voltage output is illustrated in figure 4.2. [20]

4.1 Neurons

Now let's look at a mathematical model of an *artificial neuron*. This model contains three input nodes: X_1, X_2, and X_3, that channel their output values multiplied by their respective weights w_{11}, w_{12}, and w_{13}, into the neuron "body." We denote n dendrites in the input layer nodes as x_1, x_2, x_3, …, x_n and their corresponding m weights as w_{11}, w_{12}, …, w_{nm}, where w_{ij}

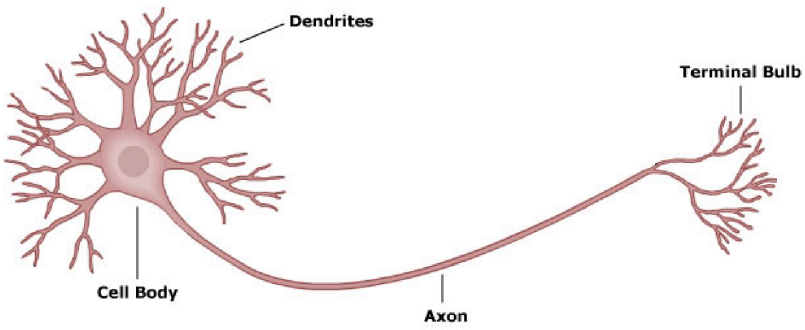


Figure 4.1 A typical biological neuron [20]

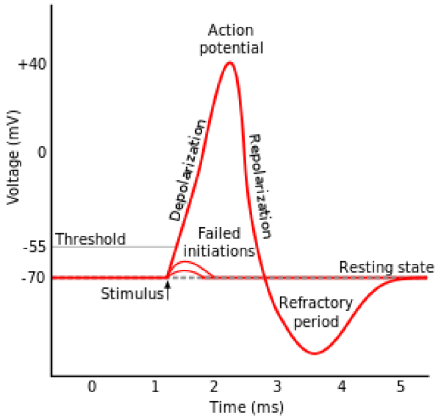


Figure 4.2 Representation of action potentials and the triggering threshold needed to propagate a signal [20]

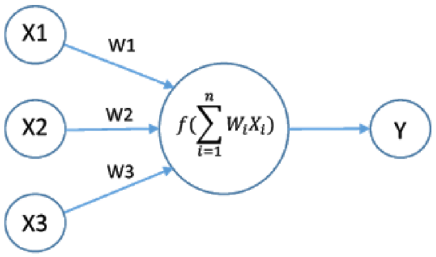


Figure 4.3 A mathematical depiction of an artificial neuron [20]

refers to the weight taking x_j to the node i. The body of an artificial neuron works simply by summing input values multiplied by the connection *weights* along with the neuron *"bias"* term b (this can be thought of as neuron resting-state potential) and passes the summed value to an $activation function$. The activation function is usually one of the nonlinear transfer functions described later. This value is either fed into the following layer of neurons or outputted out of the model. A simplified model of the artificial neuron can be seen in figure 4.3. [20]

A mathematical formula for neuron internal potential is following:

$$y = (w_{11}, w_{12}, \ldots, w_{1n}) \times \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ \vdots \\ x_{1n} \end{bmatrix} = w_{11}x_1 + w_{12}x_2 + \ldots + w_{1n}x_n + b = \mathbf{w} \times \mathbf{x} + b$$

Later we apply an *activation function* (figure 4.4) σ to the node's *internal potential*, $\sigma(y)$ is the output of a single neuron. The activation function corresponds to the activation state of a node. As mentioned earlier, a voltage potential must build up enough signal in the cell body to send a signal down the axon. The activation function simulates this biological effect in a neural network for the node and output signal. Overall, this is how we model a single neuron. A *neural network* is a collection of single neurons arranged into layers. Therefore, by understanding how a single neuron works, we can better grasp how a neural network functions. [20]

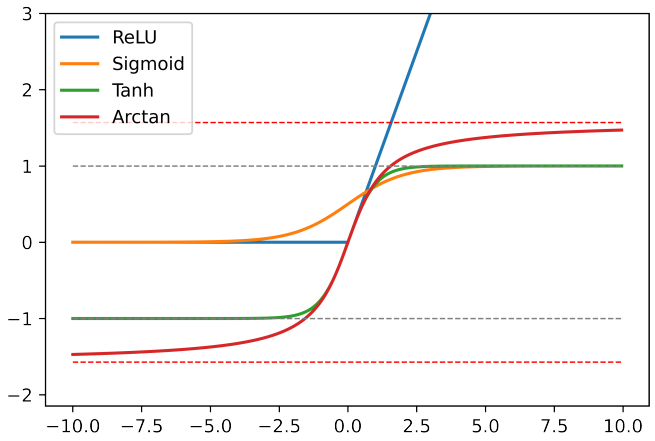


Figure 4.4 Commonly used activation functions

4.2 Activation functions

The *activation function*, also known as the *transfer function*, corresponds to the activation state of a neuron. It simulates the biological effect of overcoming a voltage potential to propagate to an axon. This function manipulates internal potential (pre-state) and transforms it into the output coming from a node. Mathematically, the transfer function $\sigma(r)$ has to be differentiable (to allow backpropagation), increasing, and has to have horizontal asymptotes. r corresponds to a pre-state for which the activation function generates an output. A couple of typical functions for $\sigma(r)$ will be discussed. [20]

These are the formulas for the most common functions:

$$\sigma(r) = arctan(r) = tan^{-1}(r),$$

$$\sigma(r) = sigmoid(r) = \frac{1}{1 + e^{-r}},$$

$$\sigma(r) = \frac{e^{2r} - 1}{e^{2r} + 1},$$

$$\sigma(r) = reLU(r) = max(0, r)$$

You can see these four activation functions plotted in a graph in figure 4.4.

4.3 Network architecture

A neural network consists of a sequence of *layers*. The first layer is called the *input layer*, and the number of neurons (or nodes) in the input layer is derived from the *dimensionality* of the input: $x_i \in \mathbb{R}^n$, the layer has n nodes. The final layer is called the output layer, and the number of neurons in the output layer is derived from the dimensionality of the output: $y_j \in \mathbb{R}^m$, the layer has m nodes. In Figure 4.5, $x_i \in \mathbb{R}^3$ and $y_j \in \mathbb{R}^1$ so, there are three neurons in the input layer and one neuron in the output layer. In between the two layers mentioned before are a number of the hidden layers, each containing some number of k neurons. We define the neural network's *architecture* by the number of nodes in each layer. For example, figure 4.5 depicts a neural

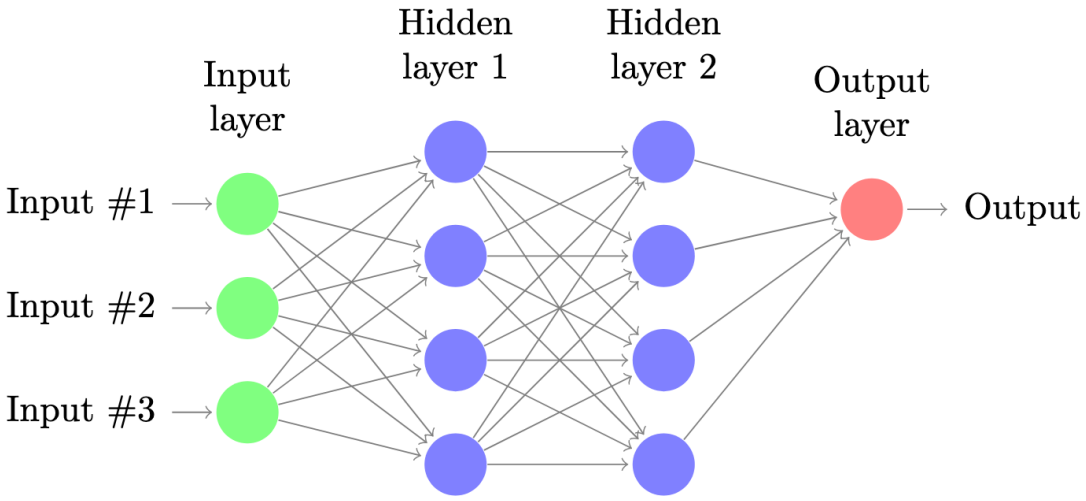


Figure 4.5 A 3-4-4-1 neural network [20]

network with three nodes in the input layer, four nodes in the first hidden layer, four nodes in the second hidden layer, and one node in the output layer. This network could be described as a 3-4-4-1 neural network. [20]

4.4 Training and application of ANNs

A neural network is a model that, when trained, *recognizes patterns* in data sets. Once a neural net is *trained*, given enough simulation data to recognize the patterns, it can predict outputs in future data. We can think of training a neural network as *estimating a function* between a given domain and range. Once trained, any data within the domain we provide can be mapped to the range of the function. A simple example of a neural network in action is data *classification*. We are given a data set containing six characteristics of 200 wines (the input would be a 6×200 matrix) and knowing the properties of 5 different types of wine. We can train the neural network on 50 different wines, and then the generated function will be able to classify the other 150 wines into the five types of wine (the output would be a 5×200 matrix). ANNs can be a powerful tool for *analyzing*, *predicting*, or *generating* data. [20]

There are two types of learning: *supervised* learning and *unsupervised* learning. Supervised learning is when the output or target values are known. That was the case in the beforementioned example about the wine classification. In classifying wine into the five types, we knew the correct wine types for the 200 bottles when used as a learning dataset. Contrary to that, unsupervised learning does not "know" the outputs or target values. The learning process finds *patterns* within the data in order to output values. Unsupervised learning is used in many complex systems, including data processing, modeling, and classification. The goal of the training process is to find weight and bias values that produce the most accurate function approximation. That is easier said than done as there are many caveats when using and training neural networks, like choosing unsuitable network architecture. [20]

4.5 Feed-Forward Neural Networks

A *feed-forward* neural network is one of the most straightforward ANN architectures. It is a part of supervised learning and creates a mapping $\mathbb{R}^n \to \mathbb{R}^m$. The mapping consists of an initial

signal x, pre-states P_j, activation function $\sigma(r)$, and states S_j. In order to compute the neural network's final output, we have to calculate all these states for each layer. We start at the input layer and continue forward towards the output layer because each layer depends on the previous one. [20]

These are the formulas for calculating pre-states P_j and states S_j:

$$P_i = W_i S_{i-1} + b_i, S_i = \sigma(P_i)$$

4.6 Backpropagation

The goal of a neural network is to approximate a function between a given range and domain. Our aim is to build the function so that the determined outputs equal the given target values, $F(x_i) = \hat{y}_i$, where F is the function created by ANN, x_i the inputs, and $\hat{y}_i$ the target values. The typical to go about this is to create a *loss function* that computes the error between our prediction $\hat{y}_i$ and the actual output y_i. We then find the values of parameters that minimize it. The loss function depends on what type of problem we are solving. **Mean Squared Error** is mainly used for regression and **Categorical Cross-Entropy** is most commonly used for classification. [20] Since we know the targets y_i and the outputs $\hat{y}_i = F(x_i)$ (as we just described earlier), our error functions will be the following:

Mean Squared Error:

$$L = (y_i - \hat{y}_i)^2$$

Categorical Cross-Entropy (given M classes):

$$L = \sum_{i=1}^{M} y_i \log \hat{y}_i$$

The loss function is dependent on the weight matrices W_i and the bias vectors b_i for each layer of the neural network. In order to decrease the error of the neural net, we will use the *gradient* of the error function. We calculate the *derivative* of the loss function, compute the *direction of the gradient* and change the weights and biases to move in the *opposite* direction of the gradient. Moving in the direction of the gradient achieves the fastest ascend, so moving in the direction opposite to the gradient results in the fastest descent. [20] Unsurprisingly this method is called *gradient descent*, where the parameter u (the actual parameters of the function are the weights W_i and biases b_i) is updated by:

$$u_{new} = u_{old} - \alpha \frac{dL}{du}$$

Where is the learning rate, controlling how big the leaps are taken when updating the weights and biases to reduce error. *"Generally, a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final set of weights. A smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train."* [21] We use backpropagation in order to determine the term $\frac{dL}{du}$. While there are other techniques (like genetic algorithms), *backpropagation* of error is an efficient way of computing the change of the error in a network. For backpropagation, we first run a forward pass through the network to determine each node's state conditions. We determine the partial derivatives through the network to get each node's error term Δ_m^l when going backward. [20] Following is the formula to update the weight W_{mn}^l, which connects node n in layer $l-1$ to node m in layer l:

$$new\, W_{mn}^l = W_{mn}^l + \varepsilon \frac{dL}{dW} = W_{mn}^l + \varepsilon \Delta_m^l S_n^{l-1}$$

Δ_m^l here means the error term that measures how much node m in layer l was responsible for overall errors in our output, and S_n^{l-1} is the state of node n in layer $l-1$. Δ_m^l is defined explicitly as $\delta_j^L = (\hat{y}_j - y_j)\sigma'(P_j^L)$ for the output layer L and recursively for preceding layers $l = 1, 2, \ldots, L-1$ as $\Delta_m^l = \sigma'(P_m^l)\sum_{j\in l+2}\Delta_j^{l+1}W_{jm}^{l+1}$. [20] For updating the biases, following formula is used:

$$new\, b_k^l = b_k^l + \varepsilon\frac{dL}{db} = b_k^l + \varepsilon\Delta_k^l$$

Chapter 5

The Transformer architecture

The Transformer is a unique ANN architecture researched at Google Brain and proposed in 2017 in the paper "Attention Is All You Need[22]." The model was a massive success in the natural language processing field. It was first designed for machine translation but has since been adapted to many other NLP tasks like text classification, text generation[23, 24, 25], text summarization, question answering, and was even extended to work in computer vision[26, 27, 28].

5.1 Overview

The Transformer model takes advantage of an *encoder-decoder* structure. The encoder maps the input sequence $x = (x_1, \ldots, x_n)$ to a sequence of continuous representations $z = (z_1, \ldots, z_n)$. We then pass representation z to the decoder that generates the final output sequence $y = (y_1, \ldots, y_m)$ one symbol at a time. At each step, the model is *auto-regressive*, meaning it consumes previously generated symbols as additional input for the following step. The Transformer implements this architecture using stacked point-wise, fully connected layers and self-attention for both the encoder and the decoder. Please see the overall structure of the Transformer in figure 5.1. [22]

5.2 Encoder

The *encoder* is formed of a stack of $N = 6$ identical layers. Each layer has two sub-layers; a *multi-head self-attention* mechanism and a simple, position-wise, fully connected *feed-forward network*. The sub-layers use a *residual connection*[1] around each of the two sub-layers, followed by layer normalization[30]. To sum it up, the output of each sub-layer is $LayerNorm(x+Sublayer(x))$, where $Sublayer(x)$ is the function implemented by the sub-layer itself (self-attention/feed-forward ANN). To enable these residual connections, all sub-layers in the model and the embedding layers produce outputs of dimension $d_{\text{model}} = 512$. Visualization of the encoder module can be seen in figure 5.2. [22]

[1] residual connection means that not only consecutive layers are connected, but there are also additional connections bypassing a certain number of layers[29]

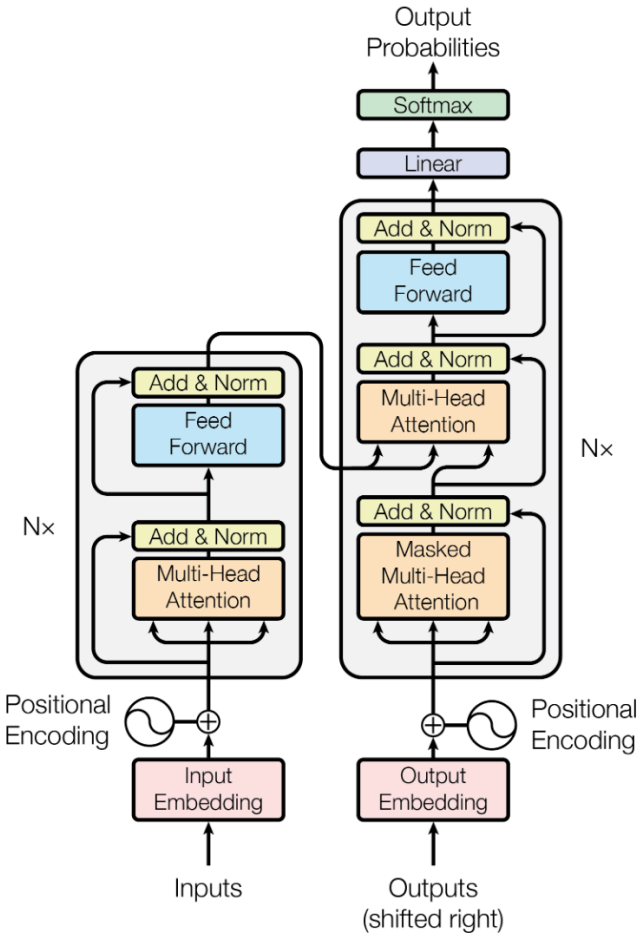


Figure 5.1 The Transformer - model architecture [22]

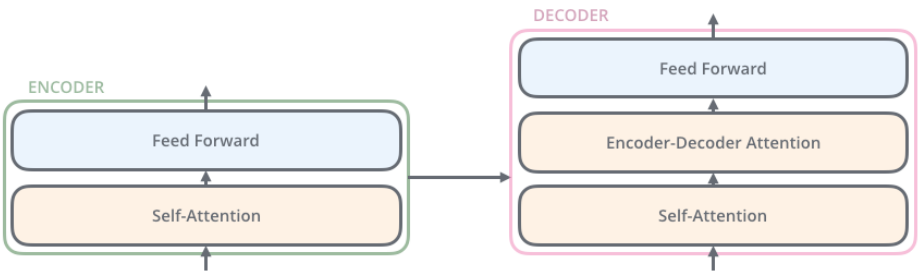


Figure 5.2 The encoder and decoder module detail [31]

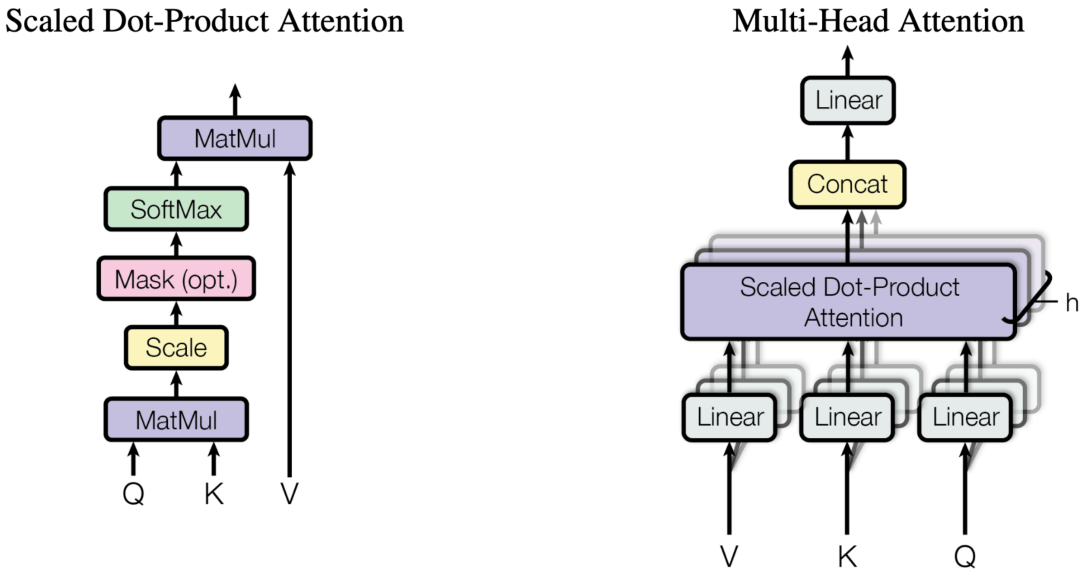


Figure 5.3 Depiction of Scaled-Dot-Product Attention (left) and Multi-Head Attention (right) [22]

5.3 Decoder

The *decoder* comprises a stack of $N = 6$ identical layers as well. The decoder module uses the same two sub-layers (just like the encoder) but, in addition to that, introduces a third sub-layer, which performs multi-head attention over the encoder's stack output. Analogous to the encoder, we use layer normalization and residual connections when connecting sub-layers. The *self-attention* sub-layer in the decoder is modified to forbid positions from attending to successive positions. Using this masking (also called teacher-forcing) and the fact that the output embeddings are shifted by one position ensures that predictions for an i^{th} position depend exclusively on known positions less than i. [22] The decoder and its relation to the encoder can be seen in figure 5.2.

5.4 Attention

Attention is the heart of the Transformer architecture; it maps a *query* and a set of *key-value* pairs to an output. The output is computed as a sum weighted by a compatibility function of the query with the corresponding key. The query, keys, and values are all represented as vectors. [22] The representation of attention can be seen in figure 5.3.

5.4.1 Scaled Dot-Product Attention

The input for "Scaled Dot-Product Attention" consists of *queries*, *keys*, and *values*. The keys and queries take up dimension d_k, and the values take up dimension d_v. The weights of values are computed by calculating the dot products of the query with all keys, each divided by $\sqrt{d_k}$, and finally, a *softmax functio* is applied. In practice, the attention function is computed on a set of queries, values, and keys simultaneously by packing them into *matrices* $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively. [22] The matrix of outputs is computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

The authors suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions with minimal gradients. Hence, to counteract this effect, the dot products are scaled by $\frac{1}{\sqrt{d_k}}$. [22]

5.4.2 Multi-Head Attention

The authors found that instead of using *Scaled Dot-Product Attention* isolated, it is beneficial to *linearly project* queries, keys, and values h times with different learned linear projections to d_k, d_k, and d_v dimensions. The attention function is computed on all these projected versions of Q, K, and V in parallel, yielding d_v-dimensional output values. Those are then concatenated and projected again, resulting in the final values. Multi-head attention allows the model to attend to information from different representation subspaces at different positions at once. [22] Following is the formula for *Multi-Head Attention*:

$$\begin{aligned} MultiHead(Q, K, V) &= Concat(head_1, \ldots, head_h)W^O \\ \text{where } head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

The projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

5.4.3 Use of Attention inside the Transformer

Multi-head attention is used in three different places inside the Transformer. The first occurrence is in the *encoder-decoder layers*. The queries come from the previous decoder layer, and keys and values come as outputs from the decoder. Another way the multi-head attention is used is *self-attention* (figure 5.4)), that is, when queries, keys, and values are all taken from the same place, in this case, from the output of the previous layer in the encoder. The decoder also contains self-attention layers to allow all positions in the decoder to attend to any of the previous positions. It must be prevented from attending a position that is after a given position to retain *auto-regressivity*. This is implemented inside the scaled dot-product attention by masking out all following values in the input. [22]

5.5 Feed-forward networks

Aside from attention sub-layers, each layer in our encoder and decoder module contains a *fully connected feed-forward* network applied to each position separately and identically [22]. The operation composes of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = max(0, xW1 + b1)W2 + b2$$

The input and output dimensions are $d_{model} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$ [22].

5.6 Embeddings and Softmax

The model uses learned *embeddings*[2] to convert the input tokens and output tokens to vectors of the d_{model} dimension. The *softmax function* converts the decoder output to predicted *next token*

[2] *"Embeddings are functions that map raw input data to low-dimensional vector representations, while preserving important semantic information about the inputs."*[32]

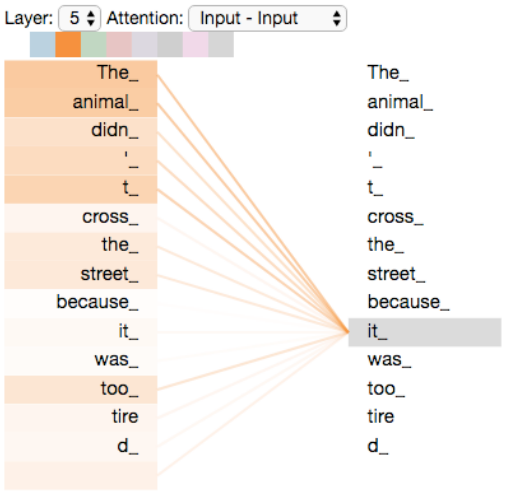


Figure 5.4 High-level depiction of self-Attention [31]

probabilities. The Transformer model shares the same weight matrix between the two embedding layers and the pre-softmax linear transformation. The weights are then multiplied by d_{model} in the embedding layers. [22]

5.7 Positional encoding

The model does not use any *recurrent* or *convolutional* layers, so for the model to understand the positions of all elements, we must provide additional information about the *relative or absolute position* of the tokens in a sequence. We add *positional encodings* to the input embeddings at the bottoms of the encoder and decoder stacks to provide this information. The positional encodings have the same dimension d_{model} as the input embeddings to make summing possible. [22] Position encodings in the Transformer are calculated using the following formulas:

$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d_{\text{model}}})$$

Where i is the dimension and pos is the position. This results in each dimension of the positional encoding corresponding to a *sinusoid*. The *wavelengths* create a geometric progression from 2π to $10000 \cdot 2\pi$. The authors chose the sinusoidal version as they hypothesize it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training. [22]

5.8 Music Transformer

The Music Transformer is an *auto-regressive* model based on the original Transformer. A decoder-based[3] model introduces modifications to the attention mechanism to enable *relative* positional encoding and decrease memory requirements. [33]

[3] the encoder blocks from the original Transformer model are not used

5.8.1 Relative positional encoding

"Music has multiple dimensions along which relative differences arguably matter more than their absolute values; the two most prominent are timing and pitch. To capture such pairwise relations between representations, Shaw et al.[34] introduce a relation-aware version of self-attention which they use successfully to modulate self-attention by the distance between two positions. We extend this approach to capture relative timing and" [33]

Shaw et al.[34] introduced *relative position encoding* as an alternative to the Transformer's absolute position realized via positional sinusoids. The relative positional encoding allows attention to be informed by how far two positions are apart in a given sequence. This is done by training a separated relative position embedding E^r of shape (H, L, D_h), embedding every possible pairwise distance $r = j_k - i_q$ between a query and key in position i_q and j_k, respectively. The embeddings are then ordered by their distance from $-L+1$ to 0 (where L means sequence length) and are learned separately for each head. In the "Self-attention with relative position representations,"[34] the relative embeddings interact with queries and form a S^{rel}; $L \times L$ dimensional logits[4] matrix that modulates the attention probabilities for each head as follows[33]:

$$\text{RelativeAttention} = \text{Softmax}\left(\frac{QK^T + S^{rel}}{sqrtD_h}\right)V$$

The *Music Transformer* uses the same approach to introduce *relative distance* information to the attention computation while introducing a novel way of computing the S^{rel}, resulting in a significant memory footprint decrease. In the "Self-attention with relative position representations,"[34] there is an intermediate tensor R of shape (L, L, D_h), instantiated for each head containing the embeddings corresponding to the relative distances between all keys and queries. Q is then reshaped to an $(L, 1, Dh)$ tensor and $S^{rel} = QR^T$. This burdens the embedding computation to $O(L^2D)$ space complexity, restricting use for longer sequences. [33]

5.8.1.1 Efficient implementation

The Music Transformer improves the implementation of relative attention by reducing its *intermediate memory* requirement from $O(L^2D)$ to $O(LD)$. The authors observed that all of the terms needed from QR^T are already available by directly multiplying Q with Er, the relative position embedding. After the QE^{rT} is computed, the (i_q, r) entry contains the dot product of the query in position i_q with the embedding of relative distance r. Nevertheless, each relative logit (i_q, j_k) in the matrix S^{rel} from the previous equation should instead be the dot product of the query in position i_q and the embedding of the relative distance $j_k - i_q$, to match with the indexing in QK^T. Therefore, we need to "skew" QE^{rT} to move the relative logits to their correct positions, as is illustrated in figure 5.5. The time complexity for both methods is $O(L^2D)$, but in practice, the authors report their method to be 6× faster at length 650. [33]

5.8.2 Relative local attention

Local attention is not a new concept; it has been used, for example, in Wikipedia and image generation[35]. It is realized by *chunking* the input sequence into *non-overlapping* blocks; each block attends to itself and the one before, as shown by the smaller thumbnail on the top right corner of figure 5.6. To extend the concept of relative attention to this local scope, we note that the right block has the same configuration as in the global case (see figure 5.5) but smaller: $(\frac{L}{M})^2$ (where M refers the number of blocks, and N be the resulting block length) instead of L^2. The left block is unmasked with relative indices going from -1 (top right) to $-2N + 1$ (bottom left). Because of that, the learned E^r for the local scope has shape $(2N - 1, N)$. Like the global case,

[4] output layer values, before passed to softmax and becoming probabilities

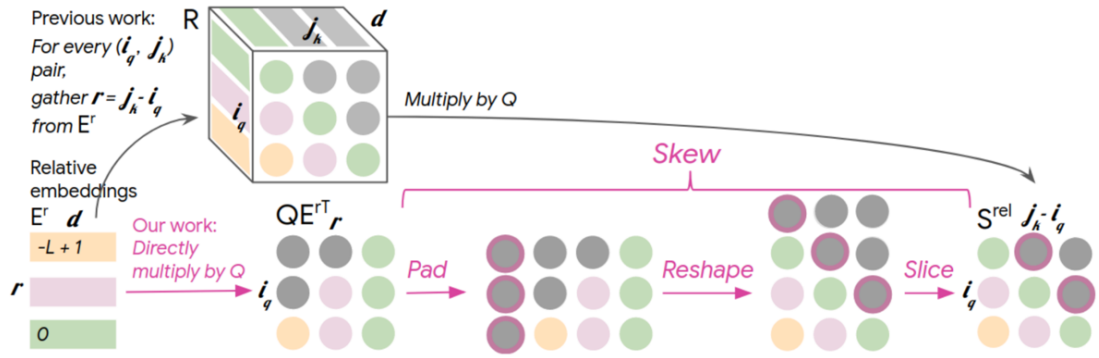


Figure 5.5 Relative global attention: the bottom row describes the memory-efficient "skewing" algorithm. Gray indicates masked or padded positions. Each color corresponds to a different relative distance. [33]

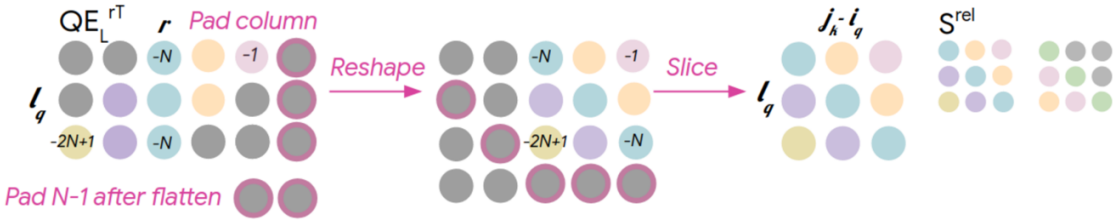


Figure 5.6 Relative local attention: the thumbnail on the right shows the desired configuration for S^{rel}. The "skewing" procedure is shown from left to right. [33]

we first compute QE^{rT} and afterward use the following procedure to "skew" it to have the same indexing as $QK^T>$, as shown in figure 5.6. The procedure is as follows[33]:

1. Pad dummy column vector with length N after the rightmost column.
2. Flatten the matrix and then pad with a dummy row having length $N - 1$.
3. Reshape the matrix to the shape $(N + 1, 2N - 1)$.
4. Slice obtained matrix to retain only the first N rows and last N columns, resulting in a matrix of shape (N, N).

Chapter 6

Music generation pipeline

This chapter will follow up on the research done in previous chapters and implement the original Transformer model and the Music Transformer. We will then obtain a suitable dataset and describe the steps leading to a fully trained model.

6.1 Dataset

In order to train a machine learning model, we first need some dataset on which the model will train. For this purpose, we used the **MAESTRO** (MIDI and Audio Edited for Synchronous TRacks and Organization v3.0.0) dataset[36], distributed by *Magenta* – a research project from Google that researches applications of AI for creating art. The dataset consists of mostly *classical pieces*, including composers from the 17th to early 20th century. The dataset was composed utilizing a MIDI capture system integrated into concert-quality acoustic grand pianos used in the **International Piano-e-Competition**[1]. Each piece was captured into a MIDI format during the competitions as the pianists played, producing 200 hours of high-quality musical audio. The dataset contains a train/validation/test split configuration to prevent the same composition from appearing in multiple subsets even if played by multiple contestants. [36] Below is a summary of this dataset:

Split	Performances	Duration (hours)	Notes (millions)
Train	962	159.2	5.66
Validation	137	19.4	0.64
Test	177	20.0	0.74
Total	**1276**	**198.7**	**7.04**

6.2 Tokenization

For the model to digest tracks from the dataset, we first have to convert it from MIDI representation into a *vector of integers* (vector of tokens). Before we get into tokenization, we first preprocess the dataset by splitting the track if the piece contains a rest longer than 3 seconds and removing tracks shorter than 10 seconds. This process is called tokenization. We have three reserved tokens with special meanings:

- START – denotes the start of the track
- END – denotes the end of the track

[1] `https://piano-e-competition.com/`

- PADDING – used as padding after the END token if the tokenized track is shorter than the vector length

We propose two different tokenization methods; the first focuses on the `note_on` and `note_off` events themselves, while the second aggregates all notes pressed simultaneously.

6.2.1 Note-centric tokenization

Note-centric tokenization method closely maps MIDI events onto different tokens; we have 128 tokens for `note_on` events and 128 tokens for `note_off` events. Their values correspond to different note pitches (0-127 is the MIDI pitch range). Besides note tokens, we also need to encode the *rest* information. To encode the rests, we first decrease the MIDI resolution down to 10 ms; this reduces the sequence length while still not being noticeable to humans. Since the rest periods are variable in time, we have to be able to tokenize *arbitrary rest length*. We do this by introducing rest tokens with values of powers of two. So, for example, let us say we have 530 ms rest. That is 53 rest units since the resolution is 10 ms. This number has a binary representation `110101`, implying four rest tokens with powers 0, 2, 4, and 5. Now, these powers are turned into tokens by shifting them after special tokens and tokens representing the notes.

6.2.2 Chord-centric tokenization

The *chord-centric* tokenization solves this problem in another way. Instead of tokenizing single `note_on`s and `note_of`s, it encodes whole chords (more accurately currently pressed keys) and the duration for which this specific combination of keys is pressed. The pressed keys are represented by a single intermediate number obtained again by binary representation; the i^{th} bit is `1` if the note corresponding to pitch i is pressed, `0` otherwise. The obtained intermediate chord representation can be any value from 0 to $2^{128} - 1$. Once we have this chord representation, we use it to create a mapping between chord representation and tokens by assigning a unique token value to each new chord representation discovered. The final sequence is then encoded like this: `<chord_representation_1>`, `<duration_representations_1>`, `<chord_representation_2>`, `<duration_representations_2>`, …. Durations are also represented as powers of two, like in the note-centric tokenization.

6.3 Training

For the final training, we will use two models mentioned in this thesis's research part: the Transformer and the Music Transformer[2]. Also, both tokenization methods will be used and compared. Following hyper-parameter (parameters set manually, instead of being discovered through optimization) are used:

- Transformer – note-centric:
  - seq-len: 4096
  - vocabulary-size: 270
  - d_{model}: 512
  - d_{ff}: 2048
  - enc-layers: 6
  - attention-heads: 8
  - dropout: 0.1

[2]this model implementation was used: `https://github.com/jason9693/MusicTransformer-pytorch`

- Transformer – chord-centric:
  - seq-len: 1024
  - vocabulary-size: 543892
  - d_{model}: 256
  - d_{ff}: 1024
  - enc-layers: 6
  - attention-heads: 8
  - dropout: 0.1
- Music Transformer – note-centric:
  - seq-len: 4096
  - vocabulary-size: 270
  - d_{model}: 512
  - dec-layers: 6
  - dropout: 0.1
- Music Transformer – chord-centric:
  - seq-len: 512
  - vocabulary-size: 543892
  - d_{model}: 256
  - dec-layers: 6
  - dropout: 0.1

Note: the hyperparameters for models were decreased for chord-centered tokenization, as this tokenization method came with greater memory requirements, and the models otherwise would not fit into GPU memory.
Meaning of the hyperparameters:

- seq-len: Length of the tokenized vector – dataset is trimmed to this value
- vocabulary-size: Number of different tokens
- d_{model}: Embedding dimension
- d_{ff}: Feed-forward network dimension
- enc-layers: Number of encoding layers
- dec-layers: Number of decoding layers
- attention-heads: Number of attention heads
- dropout: Percentage of randomly dropped nodes used to reduce over-fitting[3].

[3] state when training loss decreases, but test loss increases

6.3.1 Metrics

Two metrics are used to evaluate model performance, specifically *cross-entropy loss*, and *categorical accuracy*. Following is the cross-entropy loss formula:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = -\omega_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

[37] The categorical accuracy simply calculates the percentage of predicted tokens that match the sequence's actual values fed to a model.

$$\text{acc}(x, y) = \frac{\#\text{tokens matched}}{\text{sequence length}}$$

6.3.2 Environment

Python 3[4] programming language is used for implementation along with the ML framework *PyTorch*[5]. For training, cloud environment *Google Colab*[6] is used as it is a free (with limitations) service that allows executing *Jupyter Notebooks*[7] and is very useful for training of ANN models since the environment also allows allocation of powerful GPUs like NVIDIA Tesla P100, NVIDIA Tesla V100, and NVIDIA A100.

6.4 Evaluation

All four model/tokenization combinations were trained for 50 *epochs* (one epoch is counted as an iteration over the whole training dataset). The progression of test accuracy of all trained models can be seen in figure 6.1.As can be seen, the Music Transformer outperforms Vanilla Transformer for both tokenization methods. This, however, comes at the cost of significantly longer training time, as can be seen in figure 6.2. The problem with the Vanilla Transformer is that the model generates the first note and then keeps generating rests only, resulting in an empty composition. We hypothesized that this is because of low note density in the tokenized sequences (most of the tokens are rests) and believed that the chord tokenization might mitigate this issue. It seems, however, that the problem prevails even with the chord tokenization.

On the other hand, this is fixed with the Music Transformer, where the model always generates a non-empty composition. Instead, it has the opposite problem. When listening to the generated compositions, it can be heard that there are too many notes being played simultaneously, which does not sound very pleasant. This problem is more significant for the chord-centered tokenization than the note-centered tokenization. We provide a table with validation results for all of the models.

Model	Tokenization	Validation accuracy	Validation loss
Vanilla Transformer	note	30.34 %	2.83
Vanilla Transformer	chord	28.23 %	4.58
Music Transformer	note	21.60 %	3.74
Music Transformer	chord	25.13 %	5.77

[4] https://www.python.org/
[5] https://pytorch.org/
[6] https://colab.research.google.com/
[7] https://jupyter.org/

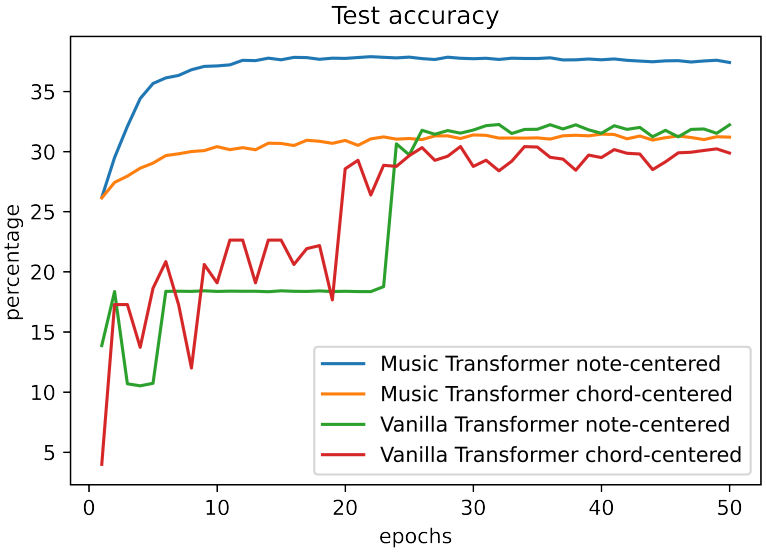


Figure 6.1 Accuracy of models on test set

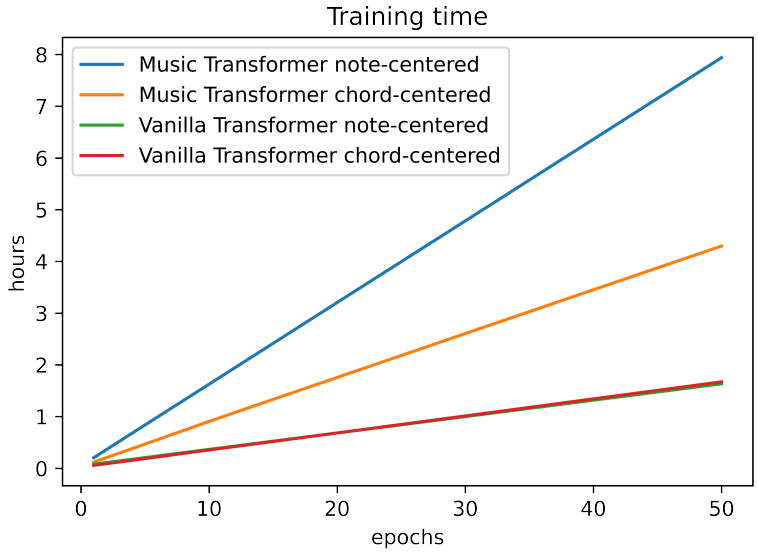


Figure 6.2 Models training time

Appendix A

Appendix

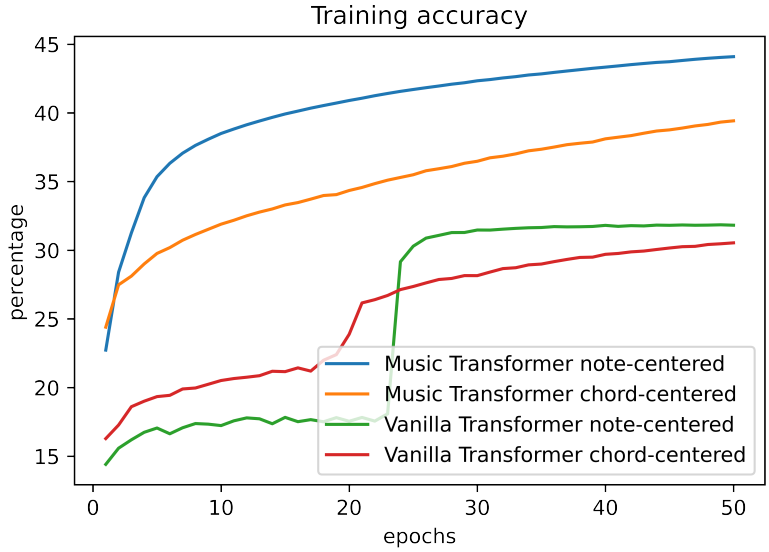


Figure A.1 Accuracy of models on training set

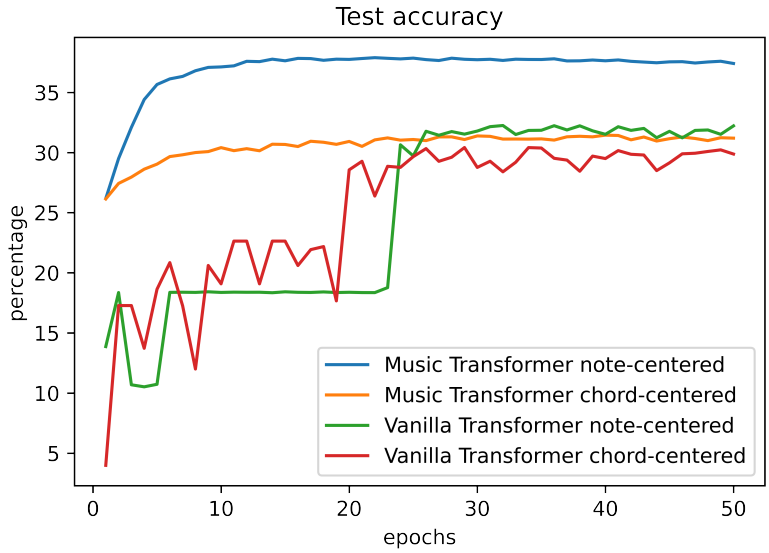


Figure A.2 Accuracy of models on test set

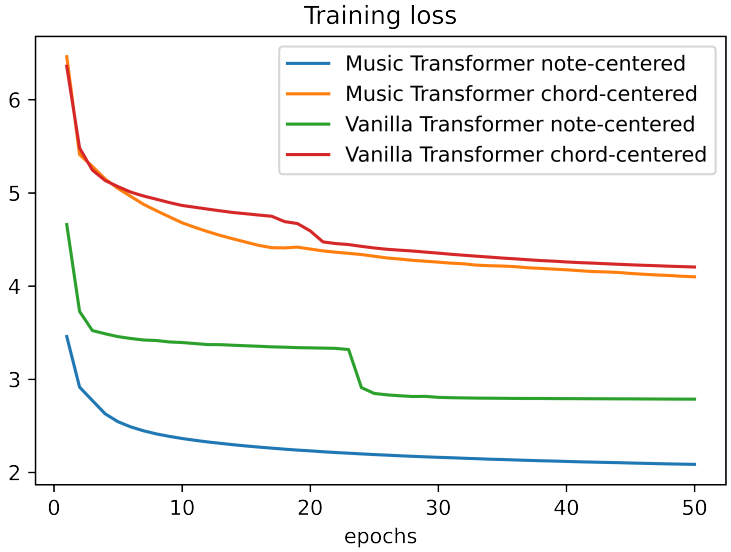


Figure A.3 Loss of models on training set

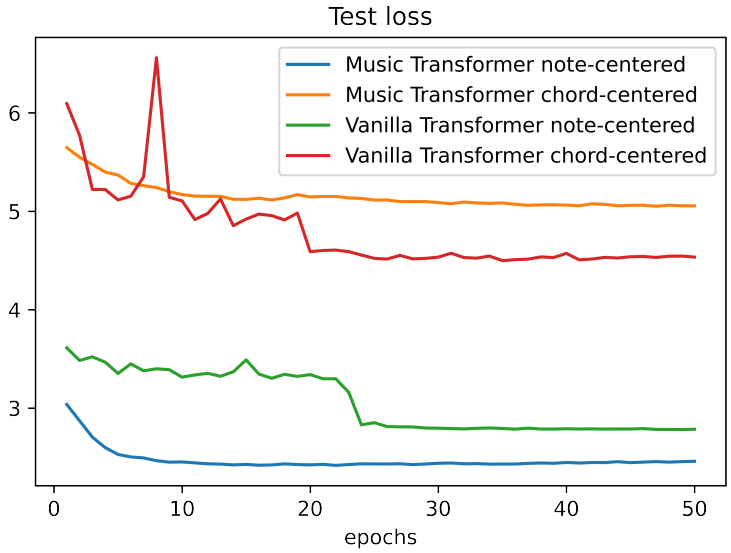


Figure A.4 Loss of models on test set

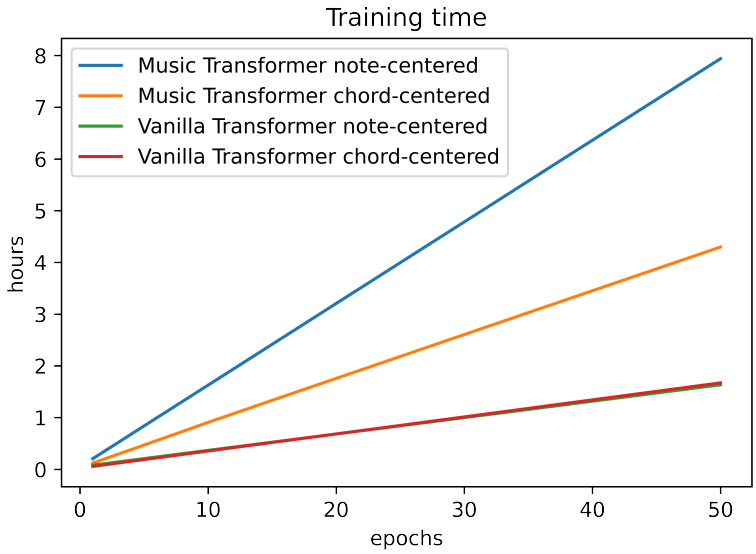


Figure A.5 Models training time

Bibliography

1. BRIOT, Jean-Pierre. *From Artificial Neural Networks to Deep Learning for Music Generation – History, Concepts and Trends*. arXiv, 2020. Available from DOI: 10.48550/ARXIV.2004.03586.
2. FARNELL, A. *Designing Sound*. MIT Press, 2010. The MIT Press. ISBN 9780262288835. Available also from: https://books.google.cz/books?id=eMPxCwAAQBAJ.
3. COOK, Mark Andrew. *Music Theory* [online]. v. 1.00th ed. 2012 [visited on 2022-04-21]. Available from: https://2012books.lardbucket.org/pdfs/music-theory.pdf.
4. HUTCHINSON, Robert. *Music Theory for the 21st-Century Classroom*. August 2020 version. Tacoma, Washington: University of Puget Sound, 2020. Available also from: https://musictheory.pugetsound.edu/hw/MusicTheory-Aug-2020.pdf.
5. MAURER, A. John. *A Brief History of Algorithmic Composition* [online]. Stanford, California: Stanford University [visited on 2022-04-23]. Available from: https://ccrma.stanford.edu/~blackrse/algorithm.html.
6. BURKHOLDER, J. Peter; GROUT, Donald Jay; PALISCA, Claude V. *A History of Western Music*. Hardcover. W. W. Norton & Company, 2019. ISBN 978-0393668179. Available also from: https://lead.to/amazon/com/?op=bt&la=en&cu=usd&key=0393668177.
7. SHAPIRO, Ilana; HUBER, Mark. Markov Chains for Computer Music Generation. *Journal of Humanistic Mathematics*. 2021, vol. 11, no. 2, pp. 167–195. Available from DOI: 10.5642/jhummath.202102.08.
8. CARNOVALINI, Filippo; RODÀ, Antonio. Computational Creativity and Music Generation Systems: An Introduction to the State of the Art. *Frontiers in Artificial Intelligence*. 2020, vol. 3. Available from DOI: 10.3389/frai.2020.00014.
9. SANDRED, Örjan; LAURSON, Mikael; KUUSKANKARE, Mika. Revisiting the Illiac Suite - A rule-based approach to stochastic processes. *Sonic Ideas/Ideas Sonicas*. 2009, vol. 2, pp. 42–46.
10. EBCIOGLU, Kemal. An Expert System for Harmonizing Four-Part Chorales. *Computer Music Journal*. 1988, vol. 12, no. 3, p. 43. Available from DOI: 10.2307/3680335.
11. SHABBIR, Jahanzaib; ANWER, Tarique. *Artificial Intelligence and its Role in Near Future*. arXiv, 2018. Available from DOI: 10.48550/ARXIV.1804.01396.
12. TODD, Peter M. A Connectionist Approach to Algorithmic Composition. *Computer Music Journal* [online]. 1989, vol. 13, no. 4, pp. 27–43 [visited on 2022-05-05]. ISSN 01489267, ISSN 15315169. Available from: http://www.jstor.org/stable/3679551.
13. TODD, Peter M; LOY, Gareth. *Music and Connectionism*. London, England: MIT Press, 1991.

14. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-term Memory. *Neural computation*. 1997, vol. 9, pp. 1735–80. Available from DOI: 10.1162/neco.1997.9.8.1735.

15. ECK, Douglas; SCHMIDHUBER, Jürgen. Finding temporal structure in music: blues improvisation with LSTM recurrent networks. *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*. 2002, pp. 747–756.

16. GOODFELLOW, Ian J.; POUGET-ABADIE, Jean; MIRZA, Mehdi; XU, Bing; WARDE-FARLEY, David; OZAIR, Sherjil; COURVILLE, Aaron; BENGIO, Yoshua. *Generative Adversarial Networks* [online]. arXiv, 2014 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1406.2661.

17. K. LISA YANG CENTER FOR CONSERVATION BIOACOUSTICS. *Raven Pro 1.4 User's Manual* [online]. 2010. Revision 11 [visited on 2022-05-05]. Available from: https://ravensoundsoftware.com/wp-content/uploads/2017/11/Raven14UsersManual.pdf.

18. SMYTH, Tamara. *Music 270a: Fundamentals of Digital Audio and Discrete-Time Signals* [online]. University of California, San Diego, 2019 [visited on 2022-05-05]. Available from: http://musicweb.ucsd.edu/~trsmyth/digitalAudio/digitalAudio_4up.pdf.

19. OLIVEIRA, Hélio de; OLIVEIRA, Raimundo. Understanding MIDI: A Painless Tutorial on Midi Format [online]. 2017 [visited on 2022-05-05]. Available from: https://www.researchgate.net/publication/316955785_Understanding_MIDI_A_Painless_Tutorial_on_Midi_Format.

20. OKEN, Adam. *An Introduction To and Applications of Neural Networks* [online]. 2017 [visited on 2022-05-05]. Available from: https://www.whitman.edu/Documents/Academics/Mathematics/2017/Oken.pdf.

21. BROWNLEE, Jason. *How to Configure the Learning Rate Hyperparameter When Training Deep Learning Neural Networks* [online]. 2019 [visited on 2022-05-05]. Available from: https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/.

22. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N; KAISER, Łukasz; POLOSUKHIN, Illia. Attention is All you Need. In: GUYON, I.; LUXBURG, U. Von; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (eds.). *Advances in Neural Information Processing Systems* [online]. Curran Associates, Inc., 2017, vol. 30 [visited on 2022-04-23]. Available from: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

23. RADFORD, Alec; NARASIMHAN, Karthik; SALIMANS, Tim; SUTSKEVER, Ilya. Improving Language Understanding by Generative Pre-Training. In: [online]. OpenAI LP, 2018 [visited on 2022-05-05]. Available from: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

24. RADFORD, Alec; WU, Jeffrey; CHILD, Rewon; LUAN, David; AMODEI, Dario. Language Models are Unsupervised Multitask Learners. In: [online]. OpenAI LP, 2018 [visited on 2022-05-05]. Available from: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

25. BROWN, Tom B.; MANN, Benjamin; RYDER, Nick; SUBBIAH, Melanie; KAPLAN, Jared; DHARIWAL, Prafulla; NEELAKANTAN, Arvind; SHYAM, Pranav; SASTRY, Girish; ASKELL, Amanda; AGARWAL, Sandhini; HERBERT-VOSS, Ariel; KRUEGER, Gretchen; HENIGHAN, Tom; CHILD, Rewon; RAMESH, Aditya; ZIEGLER, Daniel M.; WU, Jeffrey; WINTER, Clemens; HESSE, Christopher; CHEN, Mark; SIGLER, Eric; LITWIN, Mateusz; GRAY, Scott; CHESS, Benjamin; CLARK, Jack; BERNER, Christopher; MCCANDLISH, Sam; RADFORD, Alec; SUTSKEVER, Ilya; AMODEI, Dario. *Language Mod-*

els are Few-Shot Learners [online]. arXiv, 2020 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.2005.14165.

26. CARION, Nicolas; MASSA, Francisco; SYNNAEVE, Gabriel; USUNIER, Nicolas; KIRILLOV, Alexander; ZAGORUYKO, Sergey. *End-to-End Object Detection with Transformers* [online]. arXiv, 2020 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.2005.12872.
27. DOSOVITSKIY, Alexey; BEYER, Lucas; KOLESNIKOV, Alexander; WEISSENBORN, Dirk; ZHAI, Xiaohua; UNTERTHINER, Thomas; DEHGHANI, Mostafa; MINDERER, Matthias; HEIGOLD, Georg; GELLY, Sylvain; USZKOREIT, Jakob; HOULSBY, Neil. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* [online]. arXiv, 2020 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.2010.11929.
28. PARMAR, Niki; VASWANI, Ashish; USZKOREIT, Jakob; KAISER, Łukasz; SHAZEER, Noam; KU, Alexander; TRAN, Dustin. *Image Transformer* [online]. arXiv, 2018 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1802.05751.
29. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. *Deep Residual Learning for Image Recognition* [online]. arXiv, 2015 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1512.03385.
30. BA, Jimmy Lei; KIROS, Jamie Ryan; HINTON, Geoffrey E. *Layer Normalization* [online]. arXiv, 2016 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1607.06450.
31. ALAMMAR, Jay. *The Illustrated Transformer* [online]. 2018 [visited on 2022-05-05]. Available from: https://jalammar.github.io/illustrated-transformer/.
32. SONG, Congzheng; RAGHUNATHAN, Ananth. *Information Leakage in Embedding Models* [online]. arXiv, 2020 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.2004.00053.
33. HUANG, Cheng-Zhi Anna; VASWANI, Ashish; USZKOREIT, Jakob; SHAZEER, Noam; SIMON, Ian; HAWTHORNE, Curtis; DAI, Andrew M.; HOFFMAN, Matthew D.; DINCULESCU, Monica; ECK, Douglas. *Music Transformer* [online]. arXiv, 2018 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1809.04281.
34. SHAW, Peter; USZKOREIT, Jakob; VASWANI, Ashish. *Self-Attention with Relative Position Representations* [online]. arXiv, 2018 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1803.02155.
35. LIU, Peter J.; SALEH, Mohammad; POT, Etienne; GOODRICH, Ben; SEPASSI, Ryan; KAISER, Lukasz; SHAZEER, Noam. *Generating Wikipedia by Summarizing Long Sequences* [online]. arXiv, 2018 [visited on 2022-05-05]. Available from DOI: 10.48550/ARXIV.1801.10198.
36. HAWTHORNE, Curtis; STASYUK, Andriy; ROBERTS, Adam; SIMON, Ian; HUANG, Cheng-Zhi Anna; DIELEMAN, Sander; ELSEN, Erich; ENGEL, Jesse; ECK, Douglas. Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset. In: *International Conference on Learning Representations* [online]. 2019 [visited on 2022-05-05]. Available from: https://openreview.net/forum?id=r1lYRjC9F7.
37. FACEBOOK'S AI RESEARCH LAB [online]. [N.d.] [visited on 2022-05-05]. Available from: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

Contents of attached media

- `src`
  - `impl` ..... source codes of implementation
  - `thesis` ..... source code format LaTeX of thesis
  - `models` ..... trained models
- `text` ..... text of the thesis
  - `thesis.pdf` ..... thesis in the PDF format