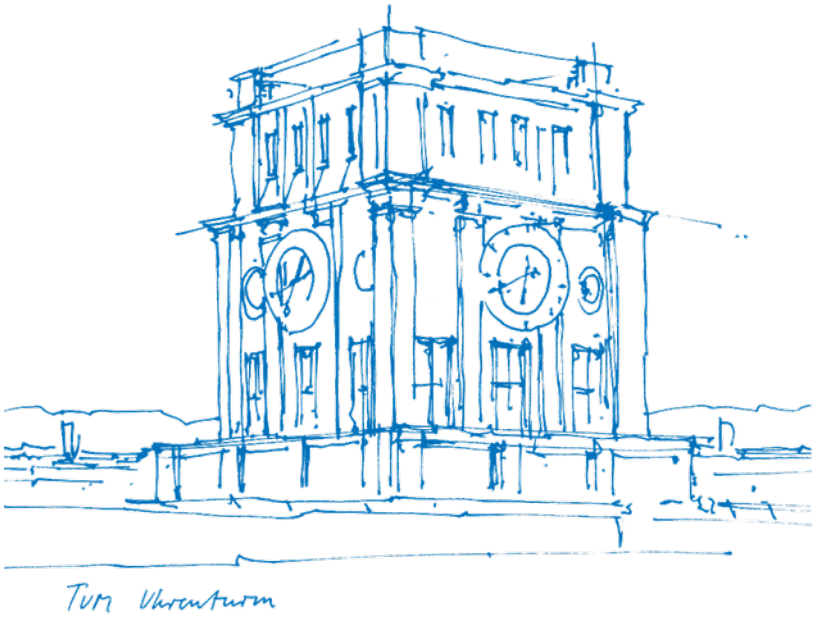
Technische Universität München
Department of Mathematics

Machine Learning Based Realtime Melody Response Generation

Master's Thesis by Patrick Wilson



Advisor & Supervisor: Prof. Dr. Dr. Jürgen Richter-Gebert

Submission Date: October 30^{th}, 2019

I hereby confirm that this is my own work, and that I used only the cited sources and materials.

München, October 30^{th}, 2019

(Patrick Wilson)

Acknowledgements

I want to thank Prof. Jürgen Richter-Gebert who gave me the opportunity to write my master's thesis about an interesting and challenging topic at his chair. Thank you for giving me the freedom to explore a field that I am passionate about and for guiding me through this project. Furthermore, I want to thank Sophia Althammer, Benedikt Mairhörmann and Fabio Raffagnato for fruitful discussions. I owe special thanks to my family and Lisa Van Noten whose support made this work possible.

Abstract

We develop a generative model for monophonic melody generation over chord progressions that is based on the long short-term memory neural network. The model takes a sequence of notes with harmonic context information – a melody – as input and outputs a note that continues the melody. The encoding of musical data for input and output of the model is designed to capture the connection between melody and harmony. An algorithmic approach to key detection is used to extend the model's harmonic context to compensate for the lack of key information in datasets. Moreover, harmonic, rhythmical and creative quality of generated melodies are evaluated with metrics that quantify musical characteristics. Based on the generative model, the implementation of a realtime application, that engages the user interactively, is developed.

Zusammenfassung

Wir entwickeln ein Machine Learning Modell für monophonische Melodiegenerierung über Akkordfolgen, das auf einer long short-term memory neural network Architektur basiert. Das Modell nimmt eine Melodie – eine Notensequenz mit harmonischem Kontext – als Input und generiert eine darauf folgende Note, die die Melodie fortführt. Die Kodierung von Input und Output des Modells hebt die Verbindung zwischen Melodie und Harmonie hervor. Der harmonische Kontext wird durch die algorithmisch ermittelte Tonart erweitert. Harmonische, rhythmische und kreative Qualität der generierten Melodien werden mit Metriken, die musikalische Eigenschaften quantisieren, evaluiert. Basierend auf dem generativen Modell wird die Implementierung einer interaktiven Echtzeitanwendung entwickelt.

Contents

1 Introduction **1**

2 Music theory **5**

3 Machine Learning Background **9**
3.1 Classification 10
3.2 Sequence classification 12
3.3 Fully Connected Neural Networks 12
3.3.1 Backpropagation 15
3.4 Recurrent Neural Networks (RNNs) 18
3.5 Vanishing and Exploding Gradient Problem 19
3.6 Long Short-Term Memory Networks (LSTMs) 19
3.7 Determinism of neural networks 21

4 Modelling melody generation **23**
4.1 Pitch-duration model 25
4.2 Pitch-only model 28
4.3 Adding tonality 28
4.4 Key detection with pitch class profiles 30
4.5 Pitch-duration-key model 32
4.6 Generating melodic sequences 32

5 Data **33**
5.1 Datasets 33
5.1.1 Wikifonia Dataset 33
5.1.2 Beatles Dataset 34
5.1.3 Lakh MIDI Dataset 36
5.1.4 Bach Dataset 37
5.2 Data preprocessing 37
5.2.1 Data augmentation 37

6 Training and Evaluation Methods **39**
6.1 Training methods 39
6.1.1 Early Stopping 39
6.1.2 Dropout 39
6.1.3 Gradient Clipping 41

6.1.4 Initialization of weights and biases . . . 41
6.2 Training setup . . . 41
6.3 Evaluation methods . . . 42
6.3.1 Harmonic Consistency (HC) . . . 43
6.3.2 Creativity metrics . . . 43
6.3.3 Mode collapse metrics . . . 44

7 Evaluation **45**
7.1 Evaluation of the Key Detection Algorithm . . . 45
7.2 Evaluation of the models . . . 45
7.2.1 Harmonic Consistency (HC) . . . 49
7.2.2 Creativity metrics . . . 49
7.2.3 Mode collapse metrics . . . 50
7.2.4 Evaluation results . . . 51

8 Interactive Melody Generation **53**
8.1 Timing . . . 53
8.2 Front-end . . . 54
8.3 Back-end . . . 55

9 Conclusion and Outlook **59**

Bibliography **61**

1 Introduction

The interest in computer-aided music generation is almost as old as the idea of computers. Ada Lovelace, who is known for her work on Charles Babbage's proposed mechanical general-purpose computer, already noted in 1843 that computers "might compose elaborate and scientific pieces of music of any degree of complexity", given an appropriate encoding of musical data [LM42]. Music lives in the space in which continuous time and the continuous frequency spectrum collide. The space of music entails such vast possibilities that it is difficult to know where to even begin to understand it. Abstractions of music have been designed so that we humans can write it down, communicate musical ideas and find certain structures in it. Many patterns have been found by analyzing music from different times and cultures and students of music theory can learn by applying those patterns. While music theory covers many concepts that can be described mathematically, a rigorous mathematical treatment of music is challenging, in particular when dealing with subjective concepts such as "interesting" or "nice-sounding" melodies. Therefore, we want to investigate the intersection of applied mathematics and music theory in this thesis.

Machine Learning is a field of research that is based on identifying patterns in data. The question arises whether Machine Learning methods can help us understand the complexities of music better. Generative music is an exciting task to study in this interdisciplinary research field. Recent research has applied Machine Learning to music generation [BHP17, RV14]. Examples for this include the Flow-Machines project [GPR16][1], the AI assisted drum sampler Atlas[2] and the neural audio synthesis tool NSynth [ERR+17].

Machine Learning research in melody generation was pioneered by Mozer [Moz91] with the model "Concert" which composes melodies based on chord progressions with the help of recurrent neural networks. Further, [ES02] used long short-term memory networks (LSTMs) for Blues melody generation and [Fra06] used LSTMs for Jazz melody generation. More recently, [CSG17] used LSTMs to generate Irish folk and Klezmer melodies. Melodies in the style of the soprano parts of the J.S. Bach chorale harmonizations were generated in [HN18]. "The Impro-Visor" [GTK10] is a generative jazz program based on probablistic grammars. In [TK18], another version of the Impro-Visor based on Generative Adversarial Networks was developed to predict monophonic

[1] Flow-Machines is an AI assisted music composing system that musicians can use to compose melody in different musical styles (`https://www.flow-machines.com/`)

[2] A software tool that maps drum samples so that similar are near each other in a 2D canvas (`https://www.algonaut.tech/`)

jazz melodies.

Moreover, the field of interactive melody generation has been researched. A Markov chain based interactive generator named "the Continuator" was introduced in [Pac10]. In the "Magenta AI duet" [WA16], an LSTM is used to generate melodies interactively based on human input. The Magenta AI duet is a web application that engages the user interactively while working with interesting Machine Learning technology on the back-end. It partially inspired the interactive application developed in this thesis. However, the Magenta AI duet lacks the employment of the concept of harmony as the melodies are standalone with no connection to a harmonic context.

A melody by itself is just a sequence of notes, but within a song, i.e. in a harmonical context, it can serve various purposes and evoke different emotions. Hence, some musical concepts can only be investigated if the encoding of melody in a melody generation model is based on this relationship. Still, many recent papers [HN18, CSG17] deal with melody generation without considering harmony. This further makes it difficult to evaluate the quality of generated melodies with metrics that build on music theory.

The main goal of this thesis is to model interactive melody generation with an emphasis on the musical connection between melody and harmony. We develop a monophonic melody generation model based on neural networks. It is designed to be deployed in an interactive web application in which melodies are generated based on user input and background harmony. Tonality information is added to the encoding as an extension by utilizing an algorithm for key-detection. To evaluate the musical quality of the model, generated melodies are analyzed with music theory based metrics. As our model emphasizes the connection between melody and harmony in the design, evaluation metrics are able to measure harmonic qualities. Our main objectives are:

Musical Encoding. To be able to feed data into a Machine Learning model, a suitable data representation is needed. Such representation should encode monophonic melodies and incorporate the connection between melody and harmony. Commonly used musical data formats should be able to be converted to this encoding.

Generative Model. A generative melody model should be able to continue a melody of any length for an arbitrary amount of notes given an inputmelody.

Measuring musical quality. The musical quality of melodies that the model generates should be measured. To do this, musical concepts need to be quantified. By quantifying harmonic, rhythmical and creative quality of the melodies, the strengths and weaknesses of the generated melodies can be analyzed.

Realtime Application. The generative model is deployable in an interactive realtime application. This application can be used for generating melodies based on various chord progressions.

The remainder of the thesis is structured as follows: Chapter 2 will cover basics of music theory that clarify the vocabulary and the concepts from music used in this thesis. Chapter 3 will introduce the Machine Learning background for this thesis. In Chapter 4, a melody generation model, based on the theory of the last two chapters is developed. Chapter 5 will elaborate on the datasets that are used and how they are processed to be used for the model. Chapter 6 discusses methods for training and evaluation of the model by developing metrics to make the melody quality measurable. Chapter 7 analyzes the evaluation of the model. Chapter 8 elaborates on interactive generation of melodies within the implemented realtime JavaScript application. Finally, in Chapter 9 we summarize our findings.

2 Music theory

In this Chapter we introduce basic concepts of music theory so that we are able to develop a music theory based model in a concise way. It also aims to serve as an introduction for readers with little to no background in music theory and is based on concepts from [Jun01] and [Sch11]. An introduction to basic musical concepts including the staff notation used in the illustrations can be found in [Sch11]. The visualizations in staff notation are created with the program *MuseScore3*. The mathematical plots are created with the Python libraries *matplotlib* and *seaborn* if not indicated otherwise.

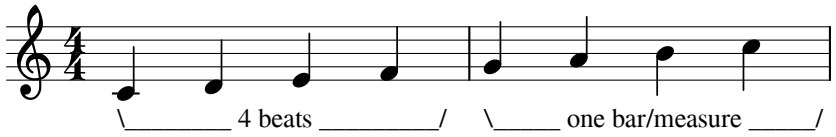


Figure 2.1: The C major scale using quarter notes in the 4/4 time signature in two measures.

The pulse of the music is called **beat** and establishes the tempo. Beats are usually divided into groups called **bar** (or measure). Bars are delimited in the staff notation by a vertical line. Most commonly, a bar equals 4 beats which is also called 4/4 time signature or *common time*. In 3/4 time signature, a bar equals 3 beats. In Figure 2.1, a 4/4 time signature is marked at the beginning of the illustration of 2 bars of music. A **pitch** is a sound of some frequency. Frequency is measured in Hertz (Hz) which is a measurement of cycles per second of the sound wave (1Hz = 1 cycle per second). High frequency sounds produce high pitch and low frequency sounds produce low pitch. A **note** is a musical sound that is characterized by pitch and duration. Durations are represented as fractions of a whole note. A whole note has the length of four beats. Examples include a quarter note, an eighth note and a sixteenth note. A *dotted* note is a note with a duration of 1.5 times the normal duration, for example a dotted quarter note has the length of a quarter note and an eighth note combined. A *triplet* is a rhythm playing three notes in the space of two. A common example is the eighth note triplet which has the duration of 1/12th of a whole note. Three eighth note triplets make up a quarter note which can be separated into two eighth notes. A beat is synonymous with a quarter note. Musical notes in this work are in the *twelve-tone equal temperament*. In this tuning, the frequency spectrum is divided by notes where the frequencies of each two adjacent notes have equal ratios of $\sqrt[12]{2}$.

This is the currently predominant system of musical tuning in Western music used for pianos, keyboards and guitars. It is an abstraction away from the real spectrum of possible notes which is a continuous spectrum from $[0, \infty)$. It originates from classical music in the 18th century. The twelve notes are C, C♯/D♭, D, D♯/E♭, E, F, F♯/G♭, G, G♯/A♭, A, A♯/B♭ and B. The ♯-symbol means one half-step up (augmented) while the ♭-symbol means one half-step down (diminished). Two notes with frequencies that are whole number multiples of each other are represented by the same letter but a different number, e.g. A4=440Hz and A5=880Hz=2×440Hz. These two notes with the same letter are said to be in the same *pitch-class* and can be considered as the same notes but with different pitches. To ease notation, we will leave out the number when talking about notes if it is not important in the context. Different to other tuning systems, in this case there are *enharmonic* (or enharmonic equivalent) notes such as F♯ and G♭. This means that they are spelled differently but are of the same frequency, so therefore have the same pitch.

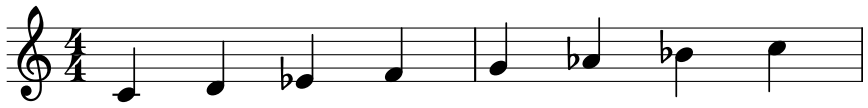


Figure 2.2: The C minor scale with the notes C4, D4, E♭4, F4, G4, A♭4, B♭4, C5.

A **scale** is any set of musical notes ordered by fundamental frequency or pitch. The scale containing all possible notes within one octave is called the *chromatic scale*. Scales can be defined in steps. The step from one note to the next in the chromatic scale, e.g. C to C♯ is called a *half-step*. The *full-step* is two half-steps, e.g. C to D. The most important scales in western music are the major and the minor scales. The major scale is defined by the steps: full-full-half-full-full-full-half which results for C major in the pitch-classes C, D, E, F, G, A, B, C as depicted in Figure 2.1. The minor scale is defined by the steps: full-half-full-full-half-full-full which results for C minor in the pitch-classes C, D, E♭, F, G, A♭, B♭, C, as depicted in Figure 2.2. The A minor scale consists of the pitch-classes A, B, C, D, E, F, G, A and therefore the same pitch-classes as C major. The corresponding minor scale to a major scale is called *relative minor* and has its root note 3 semitones below the major root note. C major and A minor scales are often used as easy examples because they represent the white keys on the piano keyboard. To ease notation, we will use pitch for pitch-class in the rest of the thesis if the context is clear. An **interval** is the distance between two notes. Intervals are notated by the quality (perfect, major, minor) and number (unison, second, third, etc.), e.g. perfect fifth or major third as shown in Table 2.1.

Interval name	Half-steps	Example
perfect unison	0	C4 - C4
minor second	1	C4 - C♯4
major second	2	C4 - D4
minor third	3	C4 - D♯4
major third	4	C4 - E4
perfect fourth	5	C4 - F4
tritone	6	C4 - F♯4
perfect fifth	7	C4 - G4
minor sixth	8	C4 - G♯4
major sixth	9	C4 - A4
minor seventh	10	C4 - A♯4
major seventh	11	C4 - B4
octave	12	C4 - C5

Table 2.1: The names of the intervals within one octave.

Consonance and **Dissonance** are categorizations of intervals where consonant means pleasant sounding and dissonant means unpleasant sounding. As these are subjective concepts that are influenced by different musical traditions, cultures, styles, and time periods we cannot define them in this context in a "correct", universally accepted or exhaustive way. Instead we will provide some characterizations of consonance that are reasonable in harmonic music theory.

Tonality (Key) is the arrangement of pitches and/or chords of a musical work in a hierarchy of perceived relations and stabilities. Tonality implies a tonal center (named "tonic"). Tonality and key are used interchangeably. Examples of tonalities are A minor or C major. In music theory the keys are usually ordered in the *Circle of Fifths* instead of chromatically, as depicted in Figure 2.4. Keys next to each other on the circle are similar in the sense that their scales only differ by one note, i.e. one note is moved a half-step up or down to change the key to a neighbouring one. Furthermore the fifth is often seen as the most consonant interval after the octave and the unison which brings another interesting view to the circle of fifths as an ordering of consonant pitches, as opposed to an ordering of similar tonalities. In some music the tonality is changed throughout the piece. In this work, however, we will assume that one piece of music has one tonality.

A **melody** is a sequence of $N \in \mathbb{N}$ notes. Musically, a melody by itself implies a tonality and can even imply various different, but somewhat related, harmonic contexts as described in [Jun01, Ch. 6]. Thus, it is unclear what harmonic context is intended by the melody and it is difficult to measure the consonance of a melody as in one harmonic context it sounds consonant and in another it does not. Therefore, we will always consider a melody within the context of a tonality in this thesis. To describe tonality more concisely we introduce chords.

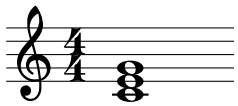


Figure 2.3: The C major chord.

Chords consist of multiple simultaneously played notes. Some of the terminology that we introduced for keys are also used for chords in a slightly different but very related context. A *major chord* is made up of the first, third and fifth note of a scale. Such a major chord can also be constructed by placing a minor third interval on top of a major third interval starting at a *root note*. The C major chord is made up of the pitches C, E and G where C and E are a major third and E and G are a minor third, as can be seen in Figure 2.3. A *minor chord* is built by stacking a major third on top of a minor third and an example for this is the C minor chord which consists of the notes C, E♭ and G. By combining these third intervals with different notes from one scale, one can create chords that fit well with the scale. In the C major scale we can for example make a D minor chord with the notes D, F and A. *Chord progressions* are sequences of chords and provide a harmonic context in a piece of music.

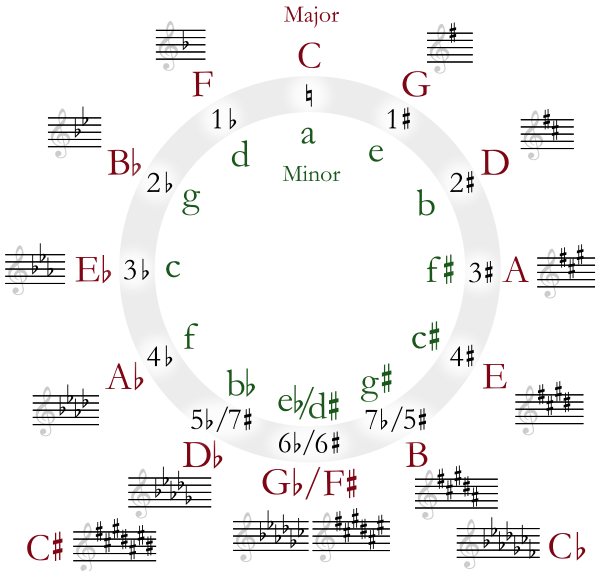


Figure 2.4: The Circle of Fifths visualizes an ordering of all pitch-classes. Two neighboring pitch-classes make up a perfect fifth interval. Its structure is also applicable to chords and keys. Image taken from [Wik08].

3 Machine Learning Background

Music is often said to be similar to language. Both music and language have a notation system and are used sonically. Both are in essence sequential and vary from culture to culture. Therefore, it is of interest to explore how techniques from the field of natural language processing, which relies heavily on Machine Learning, can be applied to melody generation. The following chapter introduces relevant concepts from Machine Learning with the main sources being the Deep Learning textbook by Goodfellow et al. [GBC16] and the textbook by A. Graves [Gra12]. The research area of Machine Learning studies machines and computer programs that are able to learn to perform a specific task without using explicit instructions, but relying on patterns and inference instead. A widely used definition is given by Tom M. Mitchell in [Mit97]:

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

Experience E is presented to the Machine Learning algorithm in form of datasets. The task T is based on discovering patterns and structure in the dataset. Common examples of the task T are to structure the dataset or to make predictions based on it. Machine Learning tasks with labelled training data, which consists of input-label pairs, are referred to as *supervised learning*. This is distinct from *reinforcement learning*, where scalar reward values are provided during training and *unsupervised learning*, where no additional information is provided. This thesis only discusses methods of supervised learning.

Let $\mathcal{X}$ be a non-empty set of training data and $\mathcal{Y}$ a set of labels. The learning algorithm takes a labelled training dataset (or training corpus) $S = \{(x_1, y_1), ..., (x_n, y_n)\} \subset \mathcal{X} \times \mathcal{Y}$ and outputs a function $f \in \mathcal{Y}^{\mathcal{X}}$, called *hypothesis*, that predicts the label for (unseen) data instances. A learning algorithm can therefore be seen as a map $\mathcal{A} : \cup_{n\in\mathbb{N}}(\mathcal{X} \times \mathcal{Y})^n \to \mathcal{Y}^{\mathcal{X}}$. It is assumed that the training data as well as the future (unseen) data is independent and identically distributed from some unknown probability measure P on the product space $\mathcal{X} \times \mathcal{Y}$. The performance of the hypothesis f is measured with respect to a suitably chosen *loss function* L, that measures "how far" $f(x)$ is from the true label y.

A loss function is defined as a function $L : \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$ with the property $L(y, y) = 0$ for all $y \in \mathcal{Y}$. The goal of learning is *generalization*, which is defined as the ability of the hypothesis to perform well on unseen test data. Generalization is assessed

by measuring the performance of the hypothesis w.r.t the loss function on a from S disjoint test dataset $S' \subset \mathcal{X} \times \mathcal{Y}$ which is not used for training. In some cases an extra *validation set* $S'' \subset \mathcal{X} \times \mathcal{Y}$ disjoint from both S and S' is used to validate performance during training while tuning the algorithm's *hyperparameters*. Hyperparameters are parameters for the model setup as well as the optimization algorithm setup that are set before the learning starts.

By minimizing the loss on the training dataset, the learning algorithm improves the *accuracy* of correctly predicting the labels from the training data.

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{number of predictions}}$$

The goal of learning is to improve the hypothesis' accuracy of predicting the correct test data labels as well as correct training data labels. Optimizing the parameters of the hypothesis on the training set does not necessarily mean improved accuracy on the test set. The phenomenon of high accuracy on training set and low accuracy on test set is called *overfitting*. In the case of overfitting, the hypothesis' parameters are optimized on the training set but do not generalize onto unseen data such as the test set. The other extreme is *underfitting*. Here, the hypothesis is not complex enough to capture structure in the data and performs bad on training and test data. One possibility to find a balance between overfitting and underfitting, is to restrict the class of functions from which the minimization picks the hypothesis, as exemplified in Figure 3.1. Instead of minimizing over $\mathcal{Y}^{\mathcal{X}}$ we choose a function class $\mathcal{F} \subset \mathcal{Y}^{\mathcal{X}}$ that is suitable for the task and the data.

3.1 Classification

This thesis deals with a subset of supervised learning problems called *classification* problems in which training data is chosen to be $\mathcal{X} = \mathbb{R}^d$ and the label space is $\mathcal{Y} = \mathbb{R}^K$. From the label space a discrete subset of labels $\mathcal{Y}' = \{C_1, ..., C_K\}$ is defined as the set of possible target labels, also called classes. Here, C_k denotes the k-th unit vector in $\mathcal{Y}$. The training dataset $S = \{(x_1, y_1), ..., (x_n, y_n)\} \subset \mathcal{X} \times \mathcal{Y}'$ consists of training samples with labels in $\mathcal{Y}'$. The *classifier* is a function $f : \mathcal{X} \to \mathcal{Y}'$. We model the classifier with conditional probabilities $p(C_k|x)$ for the K classes given the input x and the most probable class is chosen by the classifier $f(x)$:

$$f(x) = \arg\max_{k \in [K]} p(C_k|x) \tag{3.1}$$

This way, the output of the learning algorithm is a function in $p \in \mathcal{Y}^{\mathcal{X}}$ with each dimension $p(x)_k \coloneqq p(C_k|x)$ for $k \in [K]$. This is called *probabilistic classification*. Here, the relative magnitude of the probabilities can be used to determine the degree of confidence of the classifier.

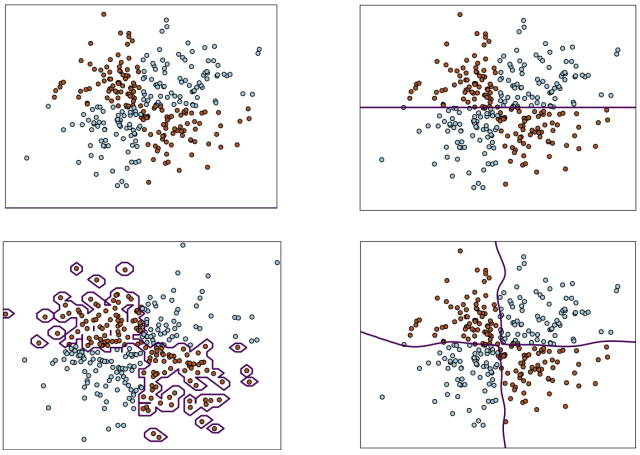


Figure 3.1: Overfitting and underfitting visualization. Assume we want to solve the famous X-OR problem of separating the points in the figure on the top left. The function class of linear functions on the top right cannot capture the complexity of the problem and underfits. On the bottom left, a hypothesis function, which is too complex, overfits the problem. It separates the points in the figure perfectly but would not generalize to new points added to the respective quarters. The bottom right shows a function that separates the data points as well as being able to separate newly added data points correctly.

Now let θ denote a set of parameters and f_θ be the probabilistic classifier dependent on those parameters θ. Then f_θ yields a conditional distribution $p(C_k|x,\theta)$ over the class labels C_k given input x. The product over the i.i.d. dataset S

$$p(S|\theta) = \prod_{(x,y)\in S} p(y|x,\theta)$$

represents the likelihood of the dataset given the hypothesis f_θ. The **maximum likelihood estimate** of this probability is given by

$$\theta_{ML} = \arg\max_\theta p(S|\theta) = \arg\max_\theta \prod_{(x,y)\in S} p(y|x,\theta)$$

To simplify notation we will from now on write $p(y|x)$ for $p(y|x,\theta)$ and implicitly assume y is predicted for f_θ dependent on θ.

For finding θ_{ML} it is convenient to minimize a maximum-likelihood loss function $L(S)$ defined as the negative logarithm of the probability assigned to S by the classifier

$$L(S) := -\ln p(S|\theta) = -\ln \prod_{(x,y)\in S} p(y|x) = -\sum_{(x,y)\in S} \ln p(y|x) \tag{3.2}$$

The loss function therefore decomposes as a sum over the training samples and we set $L(x,y) = -\ln p(y|x)$. This is a common characteristic of loss functions in the field of

Machine Learning that can be computationally beneficial for optimization [GBC16, Ch. 8]. The loss function L is specifically chosen to be differentiable. Note that minimizing of $-\ln p(S|\theta)$ is the same as maximizing $p(S|\theta)$ since the logarithm is a monotonically increasing function, i.e. the maximum of $p(S|\theta)$ occurs at the same value θ as the minimum $-\ln p(S\theta)$. The application of the logarithm makes it easy to derive the loss function. For

$$L(x, y) = -\ln p(y|x) \tag{3.3}$$

we have

$$\frac{\partial L(S)}{\partial \theta} = \sum_{(x,y)\in S} \frac{\partial L(x, y)}{\partial \theta} \tag{3.4}$$

3.2 Sequence classification

The goal of sequence classification is to assign labels to sequences of input data. What distinguishes such problems from the traditional framework of supervised pattern classification is that data points in sequences cannot be assumed to be independent. Instead, both the inputs and the labels form strongly correlated sequences [Gra12]. The training set S is independently drawn from the fixed distribution P on $\mathcal{X} \times \mathcal{Y}'$. Let $(\mathbb{R}^d)^*$ be the set of all sequences in $\mathbb{R}^d$ of any length $T \in \mathbb{N}$. The input space $\mathcal{X} = (\mathbb{R}^d)^*$ is the set of all sequences of d-dimensional vectors and the output space $\mathcal{Y}'$ is the set of all possible classes. The task is to use S to train a sequence labelling classifier $f : \mathcal{X} \to \mathcal{Y}'$ to label the sequences in the test set $S' \in \mathcal{X} \times \mathcal{Y}$, which is disjoint from S, as accurately as possible. For an input sequence $x = (x_1, ..., x_T)$ we want to predict a label $y \in \mathcal{Y}'$ and the output of the a classifier $f(x) = \arg\max_k p(C_k|x)$ is as before. Each vector x_t is said to be at time-step $t \in [T]$.

3.3 Fully Connected Neural Networks

State-of-the-art results in sequence learning tasks rely on *recurrent neural networks*, which are a class of *(artificial) neural networks* (NNs) [Gra13]. This section serves as a general introduction to fully connected neural networks before recurrent neural networks are explored in section 3.4. A neural network can be thought of as a function

$$f_\theta : \mathcal{X} \to \mathcal{Y}$$

of the function class $\mathcal{F}$ of neural networks that takes a data vector x and outputs a prediction $\hat{y}$ with $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}^K$. The goal of a neural network is to approximate a function $f^*(x)$. In our case this function is a classifier $f^*(x) = y$ where $y \in \mathcal{Y}'$ is a

category. This function f_θ is called a network because it is made up of concatenations of functions. These functions include *affine functions* of the form

$$g(x) = Wx + b$$

for $x \in \mathcal{X}, W \in \mathbb{R}^{h \times d}, b \in \mathbb{R}^h$ and *activation functions*, e.g. the hyperbolic tangent activation or the sigmoid activation functions:

$$\tanh(x) \coloneqq (\tanh(x_1), ..., \tanh(x_d))^T,$$
$$\sigma(x) \coloneqq 1/(1 + e^{-x}),$$

for $x \in \mathcal{X}$. Activation functions are, if not stated otherwise, applied element-wise. A *layer* consists of the concatenation of an affine function and an activation function, for example

$$\sigma(g(x)) = \sigma(Wx + b)$$

The values in W and b make up the set of learnable parameters θ (also called weights and biases). *Learning* is the process of optimizing the parameters θ so that f_θ best approximates f^*. If a neural network is for example composed as

$$f_\theta(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))),$$

then $f^{(1)}$ is called the *first layer*, $f^{(2)}$ the *second layer* and so on. The length of this chain of layers is called the *depth* of the network, which is where the term "deep learning" comes from [GBC16]. The final layer is called the *output layer*, while the previous layers are called *hidden layers*. The process of feeding the inputs through the neural network is also called *forward propagation*.

In classification tasks, a common choice for the nonlinear part of the output layer is the *softmax* function. $\text{softmax} : \mathbb{R}^K \to \mathbb{R}^K$ is defined for $k \in [K]$ as

$$\text{softmax}(z)_k \coloneqq \frac{e^{z_k}}{\sum_{i=1}^K e^{z_i}}.$$

This normalizes the K-dimensional vector and effectively turns it into a K-dimensional discrete probability distribution. A *1-of-K* encoding represents the label class y as a *one-hot vector* (binary vector) with all elements equal to zero except for element k, corresponding to the correct class C_k, which equals one. For example for $K = 3$ and the correct class C_2, the label y is represented as $(0, 1, 0)$. We want to maximize the likelihood for the correct prediction, so with

$$p(C_k|z) = \hat{y}_k = \text{softmax}(z)_k$$

we can rewrite the probability for the correct label $y \in \mathcal{Y}'$ as

$$p(y|x) = \prod_{k=1}^K \hat{y}_k^{y_k} \tag{3.5}$$

Substituting 3.5 into 3.3 yields the *cross-entropy* loss function

$$L(x, y) = -\sum_{k=1}^{K} y_k \ln \hat{y}_k \tag{3.6}$$

for a sample (x, y) with y being the one-hot label and $\hat{y}$ the output of the softmax function.

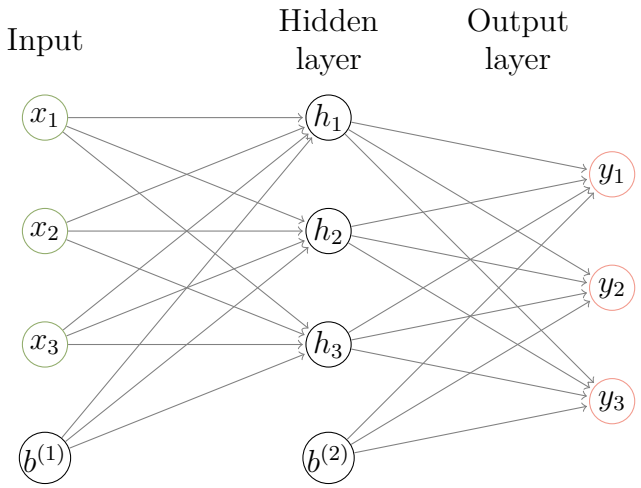


Figure 3.2: A fully connected neural network with 3-dimensional input, one 3-dimensional hidden layer and a 3-dimensional output layer.

For instance, consider a fully connected neural network with one 3-dimensional hidden layer and 3-dimensional input and output as sketched in Figure 3.2. The hidden states vector is calculated by an affine transformation of the input vector followed by a nonlinear activation function g

$$h = g(W^{(1)}x + b^{(1)}),$$

where $W^{(1)} \in \mathbb{R}^{3\times3}, b^{(1)} \in \mathbb{R}^3$. Here we can for example choose $g = \tanh$ as the (element-wise) activation function. The output $\hat{y}$ is obtained by another layer of affine transformation and with softmax activation

$$\hat{y} = \text{softmax}(W^{(2)}h + b^{(2)})$$

where again $W^{(2)} \in \mathbb{R}^{3\times3}, b^{(2)} \in \mathbb{R}^3$. The output layer again can be interpreted as a 3-dimensional discrete probability distribution. This network can be used for a 3-class classification task by choosing the class corresponding to the output unit with the highest probability. Here, the learnable parameters consist of $\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$.

By updating the parameters θ with a gradient descent based algorithm the loss can be minimized so that the neural network f_θ best approximates f^* based on the

examples from the dataset. Even though neural networks are not convex functions in general, because of their activation functions, minimization is applied in practice. Gradient descent relies on the observation that for a differentiable function L it holds that if

$$\theta^{(n+1)} = \theta^{(n)} - \lambda \nabla_\theta L_{\theta^{(n)}}(S)$$

for $\lambda > 0$ chosen small enough, then

$$L_{\theta^{(n+1)}}(S) \leq L_{\theta^{(n)}}(S)$$

The parameter λ is known as the *learning rate*. In standard gradient descent the gradient of the loss of each data sample from S with respect to each weight is computed before updating the weights. An adaptation of this is called *stochastic gradient descent with mini-batches* in which a random subset $S_1 \subset S$ is used to compute the gradient.

$$\theta^{(n+1)} = \theta^{(n)} - \frac{|S_1|}{|S|} \lambda \nabla_\theta L_{\theta^{(n)}}(S_1)$$

Note that the gradient is scaled by $\frac{|S_1|}{|S|}$ to compensate for using less samples. This method reduces the computation time of an update of the parameters and has shown to minimize the loss function faster [GBC16, Ch.8].

The loss function is differentiable as a concatenation of differentiable functions. Hence, the gradient $\nabla_\theta L_{\theta^{(n)}}(S)$ can be calculated by a method called *backpropagation*, which is an application of the chain rule. The term stems from backward propagation of losses which describes the fact that calculation of the gradients of the neural network proceeds backwards through the network starting from the output layer.

3.3.1 Backpropagation

The forward and backward propagation algorithms for a fully connected neural network with a softmax activation on the output layer and activation functions on each previous layer are depicted below. For simplicity the following algorithms are demonstrated on one sample (x, y). In practice one uses *minibatches*, fixed sized subsets of the training set, to speed up computation. The notation follows [GBC16].

Algorithm 1: Forward propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, ..., l\}$, the weight matrices of the model
Require: $b^{(i)}, i \in \{1, ..., l\}$, the bias parameters of the model
Require: x, the input vector
Require: y, the label vector
$h^{(0)} = x$
for $i = 1, ..., l$ **do**
  $z^{(i)} = b^{(i)} + W^{(i)} h^{(i-1)}$
  $h^{(i)} = g(z^{(i)})$
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y)$

After obtaining the loss J through forward propagation, the weights and biases are updated via backward propagation using the Algorithm 2. This computation yields the gradients of the outputs $z^{(i)}$ of each layer, starting from the output layer and going backwards to the first hidden layer. The gradients can then be used immediately as part of a gradient descent update.

Algorithm 2: Backward propagation

Require: Forward propagation
$G \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$
for $i = l, l-1, ..., 1$ **do**
  $G \leftarrow \nabla_{z^{(i)}} J = G \odot g'(z^{(i)})$
  $\nabla_{b^{(i)}} J = G$
  $\nabla_{W^{(i)}} J = G h^{(i-1)T}$
  $G \leftarrow \nabla_{h^{(i-1)}} J = W^{(i)T} G$

As an example we will calculate the partial derivative of the cross-entropy loss for the softmax layer. The partial derivative of the cross-entropy loss is

$$\frac{\partial L}{\partial \hat{y}_k} = -\frac{y_k}{\hat{y}_k}$$
$$\delta \hat{y} = \left[\frac{\partial L}{\partial \hat{y}_k}\right]_k = -y \odot \frac{1}{\hat{y}}$$

$\delta \hat{y}$ and y are in $\mathbb{R}^K$ and $\odot$ denotes the scalar product.

Suppose z is the K-dimensional input vector to the softmax layer of our network. Note that $\hat{y}_k$ for $k \in [K]$ depends on every input (into the softmax layer) z_i for $i \in [K]$.

$$\hat{y}_k = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}}$$

For $k \in [K]$

$$\frac{\partial L}{\partial z_k} = \sum_{j=1}^{K} \left(\frac{\partial L}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_k} \right)$$

If $k = j$, with $\mathbf{1}_{k=j} = \begin{cases} 1 & k = j \\ 0 & k \neq j \end{cases}$,

$$\begin{aligned} \frac{\partial \hat{y}_k}{\partial z_k} &= \frac{\partial}{\partial z_k} \left\{ \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \right\} = \frac{e^{z_k} \left(\sum_{i=1}^{K} e^{z_i} \right) - \left(e^{z_k} \right)^2}{\left(\sum_{i=1}^{K} e^{z_i} \right)^2} \\ &= \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} - \frac{\left(e^{z_k} \right)^2}{\left(\sum_{i=1}^{K} e^{z_i} \right)^2} = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \left(1 - \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \right) \\ &= \hat{y}_k \cdot (1 - \hat{y}_k) = \hat{y}_k \cdot (\mathbf{1}_{k=j} - \hat{y}_j) \end{aligned}$$

If $k \neq j$,

$$\begin{aligned} \frac{\partial \hat{y}_k}{\partial z_k} &= \frac{\partial}{\partial z_k} \left\{ \frac{e^{z_j}}{\sum_{i=1}^{K} e^{z_i}} \right\} = \frac{0 - e^{z_k} \cdot e^{z_j}}{\left(\sum_{i=1}^{K} e^{z_i} \right)^2} = \frac{-e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \cdot \frac{e^{z_j}}{\sum_{i=1}^{K} e^{z_i}} \\ &= -\hat{y}_k \cdot \hat{y}_j = \hat{y}_k \cdot (\mathbf{1}_{k=j} - \hat{y}_j) \end{aligned}$$

Together we have

$$\begin{aligned} \frac{\partial L}{\partial z_k} &= \sum_{j=1}^{K} \delta \hat{y}_j \cdot \hat{y}_k \cdot (\mathbf{1}_{k=j} - \hat{y}_j) = \sum_{j=1}^{K} J_{ij} \times \delta \hat{y}_j \\ \text{where } J_{ij} &= \hat{y}_k \cdot (\mathbf{1}_{i=j} - \hat{y}_j) \\ \delta x &= \left[\frac{\partial L}{\partial z_k} \right]_k = J \times \delta y \end{aligned}$$

In this layer $\delta x, y \in \mathbb{R}^K$ and $J \in \mathbb{R}^{K \times K}$.

This illustrates how derivatives are passed backwards to calculate the earlier derivatives in the process of backpropagation. As can be seen, the derivative of the softmax function has a convenient form that has the advantage of being computationally inexpensive to obtain. In practice, functions with minimal computational cost are preferred as activation functions in neural networks.

3.4 Recurrent Neural Networks (RNNs)

Musical data in form of melodies is sequential as one note is not independent of the previous notes in a melody. Fully connected neural networks, however, have not shown great capability to capture such structure. Moreover, the input dimension of a fully connected neural network is fixed, so that varying length inputs are only possible by padding techniques and are still limited by a maximum input length. Recurrent neural networks, or RNNs ([RHW+]), are a family of neural networks specifically designed for processing sequential data. The difference to neural networks in the previous section is that RNNs allow the network to contain cyclical connections. These networks allow input sequences to be of arbitrary length.

Let $n \in \mathbb{N}$ be the sequence length which is not fixed, let $d \in \mathbb{N}$ be the fixed dimension of one data sample, let $h \in \mathbb{N}$ be the fixed dimension of the hidden state. For an input sequence $(x^{(1)}, x^{(2)}, ..., x^{(n)})$ with $x^{(t)} \in \mathbb{R}^d$ for $t \in [n]$ we define *weight matrices* $U \in \mathbb{R}^{h \times d}$, $W \in \mathbb{R}^{h \times h}$, $V \in \mathbb{R}^{K \times h}$ and the *biases* $b \in \mathbb{R}^h, c \in \mathbb{R}^K$. The following equations define the forward propagation of a one-layer RNN at time-step $t \in [n]$:

$$\begin{aligned} a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)}, \\ h^{(t)} &= \tanh(a^{(t)}), \\ o^{(t)} &= c + Vh^{(t)}, \\ \hat{y}^{(t)} &= \text{softmax}(o^{(t)}), \end{aligned} \tag{3.7}$$

with tanh as an element-wise operation, $\text{softmax}(x)_j := \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}$ for $j \in [K]$ and $h^{(0)} = 0$, with 0 being the h-dimensional zero vector. Note that the weight matrices repeat at each time-step as can be seen in Figure 3.3.

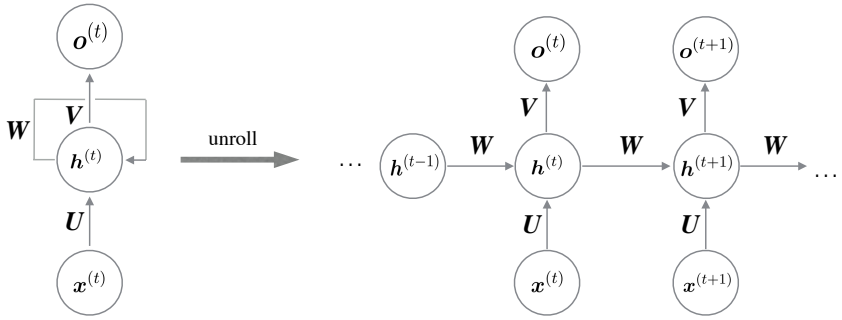


Figure 3.3: Unrolling of an RNN. In contrast to the fully connected neural network in Figure 3.2, each node in the figure represents a layer of network units at a single time-step with time going from left to right. Besides visualizing the cycles in the network graph (left) one can visualize the "unrolled" graph of the recurrent neural network for an arbitrary amount of time-steps (right).

3.5 Vanishing and Exploding Gradient Problem

A big problem with plain RNNs comes from reusing the same weight matrices for each time-step. The simplified case with

$$h^{(t)} = W^T h^{(t-1)}$$

can be seen as a recurrent neural network without nonlinear activation function and lacking inputs x. Suppose W has an eigendecomposition:

$$W = Q\text{diag}(\lambda)Q^{-1}.$$

Then the repeated multiplication of W to the signal

$$h^{(t)} = (W^t)^T h_0$$

is equivalent to

$$\begin{aligned} W^t &= (Q\text{diag}(\lambda)Q^{-1})^t \\ &= Q\text{diag}(\lambda)^t Q^{-1}. \end{aligned}$$

Any eigenvalues λ_i of magnitude less than one decay to zero and eigenvalues with magnitude greater than one explode. This turns out to be a problem as these matrices are used in the computation of backpropagation. A gradient of magnitude close to zero means that there is little to no signal in the gradient descent iteration so the weights cannot be updated in a significant way. A gradient of very large magnitude makes it difficult to fine-tune the parameters as the gradient descent iteration steps become too large. This problem is known as the *vanishing and exploding gradient problem*.

3.6 Long Short-Term Memory Networks (LSTMs)

To address vanishing gradients, we can introduce self-loops with gating mechanisms into the basic RNN propagation so that the gradient can flow for long durations. The most used type of gated RNN is the **long short-term memory** (LSTM) model initially introduced in [HS97] and extended to the current version in [GSC99], by making the weight of the self-loop conditioned on the context rather than fixed. The LSTM recurrent network replaces the simple RNN mechanism of applying an activation function on an affine transformation of the input and the hidden state by a whole *LSTM cell* that has more parameters to control information flow.

The *forget gate* $f^{(t)}$ regulates the self-loop weight by setting the value between 0 and 1. The idea of the forget gate $f^{(t)}$ is to create a vector of weights based on $x^{(t)}$ and $h^{(t-1)}$ that decide how much of the previous internal state $s^{(t-1)}$ should be kept and

how much should be discarded by scaling it down. For the input vector $x^{(t)} \in \mathbb{R}^d$ and the current hidden vector $h^{(t)} \in \mathbb{R}^h$, we have

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_{j=1}^{d} U_{i,j}^f x_j^{(t)} + \sum_{j=1}^{h} W_{i,j}^f h_j^{(t-1)}\right) \text{ for } i \in [h], \tag{3.8}$$

where b^f, U^f, W^f are biases, input weights and recurrent weights for the forget gates, respectively. This gate gives the model the chance to learn how much of the previous hidden state accounts to the current state [GBC16, Ch. 10]. As in the basic RNN the dimensions of weights and biases are $U^f \in \mathbb{R}^{h \times d}$, $W^f \in \mathbb{R}^{h \times h}$ and $b^f \in \mathbb{R}^h$. The *external input gate* $g^{(t)}$ unit is computed similarly to the forget gate but with its own parameters:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_{j=1}^{d} U_{i,j}^g x_j^{(t)} + \sum_{j=1}^{h} W_{i,j}^g h_j^{(t-1)}\right) \text{ for } i \in [h]. \tag{3.9}$$

The LSTM *internal state* $s^{(t)}$ is updated by

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \tanh\left(b_i + \sum_{j=1}^{d} U_{i,j} x_j^{(t)} + \sum_{j=1}^{h} W_{i,j} h_j^{(t-1)}\right) \text{ for } i \in [h], \tag{3.10}$$

with b, U, W respectively biases, input weights and recurrent weights into the LSTM cell with dimensions, analogous to before. To obtain the new internal state, first the external input gate creates a weight vector that represents the amount of how much each dimension should get updated. This weight vector scales the tanh gate which contains values that represent update candidates that could be added to the state. Finally, the update of the internal state is done by weighting the old internal state with the forget gate values and adding this to the weighted new candidate values [Ola]. As a last step, the *output gate* is implemented to scale the current internal state by a sigmoid activated concatenation of the previous hidden state $h^{(t-1)}$ and the input $x^{(t)}$. The output $h^{(t)}$ of the LSTM cell is gated by the output gate $q^{(t)}$ with

$$\begin{aligned} q_i^{(t)} &= \sigma\left(b_i^o + \sum_{j=1}^{d} U_{i,j}^o x_j^{(t)} + \sum_{j=1}^{h} W_{i,j}^o h_j^{(t-1)}\right), \\ h_i^{(t)} &= \tanh\left(s_i^{(t)}\right) q_i^{(t)}, \qquad \text{for } i \in [h] \end{aligned} \tag{3.11}$$

where b^o, U^o, W^o are the biases, input weights and recurrent weights respectively, with dimensions analogous to before. The initial values are generally chosen as $s^{(0)} = 0$ and $h^{(0)} = 0$ [GBC16, Ch. 10].

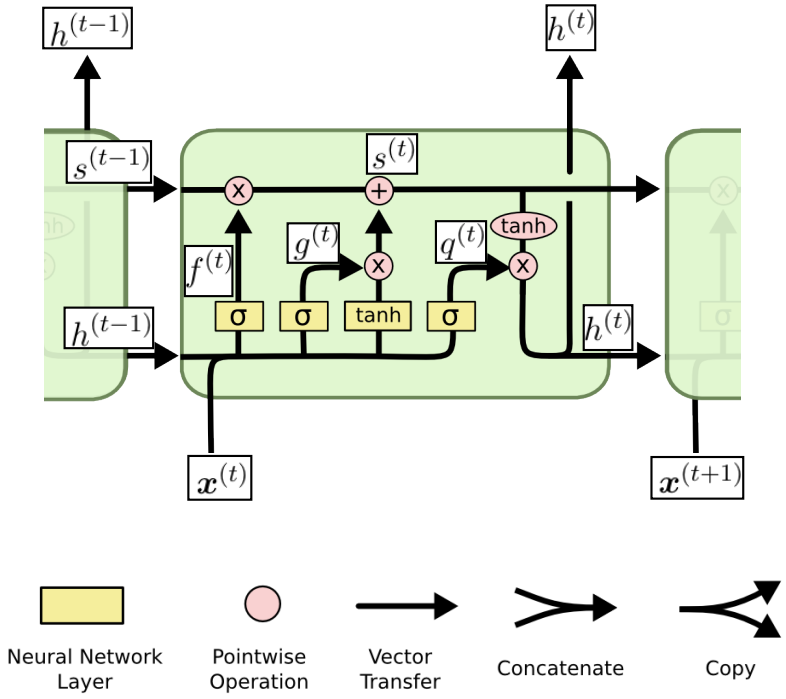


Figure 3.4: Visualization of the LSTM cell with gates as in the formulas. It shows the inputs $x^{(t)}$, the outputs $h^{(t)}$, the internal states $s^{(t)}$, the forget gate $f^{(t)}$, the external input gate $g^{(t)}$ and the output gate $q^{(t)}$. Image adapted from [Ola].

Multiple LSTM layers can be stacked by taking the output $h^{(t)}$ of one layer at each time step as input for the next LSTM layer. The partial derivatives of the loss with respect to the network weights are calculated with an extension to backpropagation algorithm called *backpropagation through time* [WZ95]. The basic idea is to unfold the recurrent neural network in time and apply backpropagation just as in fully connected neural networks. The difference to standard backpropagation is that the same parameters are updated over and over again as each time-step shares the same parameters. The network can then be trained with a stochastic gradient descent based algorithm just as a standard neural network. The equations for backpropagation of LSTMs are developed in [Gra13, Ch. 4.6].

3.7 Determinism of neural networks

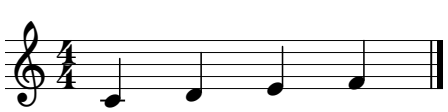
After training, the model is deterministic which means that for the same input, it will always predict the same output. In our case of training a generative model for music, for the same input melody the same output note is predicted. This stems from the

fact that we designed the final step of predicting the output label to output the most likely label. Another way to approach this, is to *sample* from the output vector by, once again, interpreting the output as a probability distribution. This results in more variation but also enables less likely and therefore perhaps less consonant sounding melodies. It is up to personal preference to decide between sampling and choosing the most likely next note. In this work we choose the most likely output.

4 Modelling melody generation

The task of melody generation can be framed in many ways. In this thesis, we aim to design a model that takes an melody, i.e. a sequence of notes, as input and outputs a sequence of notes that continues the melody. We will only discuss *monophonic* melodies which means at each point in time at most one note is played. This chapter will elaborate on modelling the input and output of the model and introduce three different variations of the model. Furthermore, there are different possibilities to encode rhythm into musical models. We will first discuss different encodings mentioned in [TK18] and then describe the encoding used in this thesis in detail.

In the **time-step encoding**, the piece of music is divided into relative time-steps and at every time-step a vector representing the possible pitches denotes which pitch is active at that time-step. Relative time-steps mean that the time-step are dependent on the tempo of the song, i.e. note fractions instead of milliseconds. Often the measurement of 16th notes is used here, as seen in figure 4.1.



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4.1: Example melody in staff notation (left) and time-step encoding (right). Each quarter note in the figure translates to four 16th time-steps in the vector encoding.

Notes are sustained by repeating pitches for multiple time-steps. Thus the difference between a sustained long note and multiple shorter notes immediately following each other is not well-defined. The encoded melody shown in Figure 4.1 could for example also be interpreted as only consisting of 16th notes as shown in Figure 4.2.

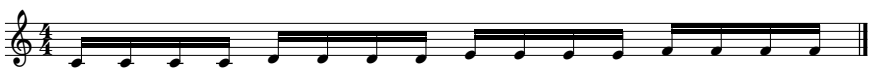


Figure 4.2: A different decoding of the example melody.

Therefore some studies include an attack bit, i.e. an extra row in the representation, to indicate the beginning of a new note [WA16]. Other studies double the resolution of the time-step grid to be able to indicate the end of a note [ES02, Lac16].

A model based on the **note-duration encoding** trains on sequences of notes, each note consisting of a pitch and a duration. This is the most natural way to think of a melody from a musicians perspective as music is notated in the staff notation in this way, too. The Figure 4.3 sketches the encoded melody on the right.

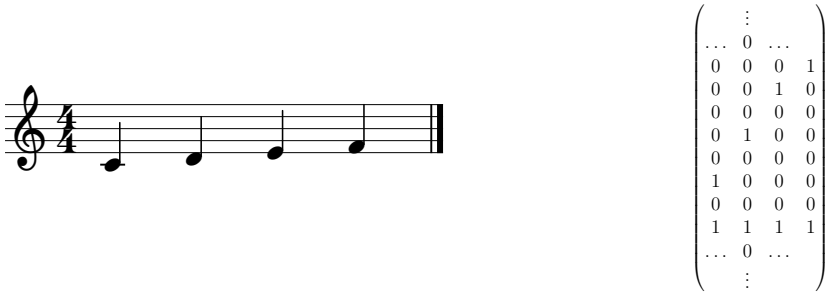


Figure 4.3: Example melody in staff notation (left) and note-duration encoding (right). Each quarter note translates to one column vector. Each column vector consists of two ones and zeros otherwise. The top 1 indicates the pitch and the bottom 1 indicates the note duration. In this case all note durations are quarter notes. The zeros on the top and the bottom indicate possibilities for higher pitches and other note durations.

Each note is made up of one vector in which the note's pitch is indicated in the top part with one 1 and the note's duration is indicated in the bottom part. Each note has the length of a quarter note so the bottom part of the matrix makes up a row of ones. A pause in this encoding can be indicated in the pitch part by an extra row, i.e. one row in the pitch section of the note vector denotes a pause instead of a pitch. The advantages of this encoding compared to the time-step encoding are that sequences are compressed in length and sparse attack sequences do not need to be guaranteed [TK18]. This encoding is used in [CSG17] and [Moz91].

In the **note-beat-position encoding**, the model learns to predict an ending beat position with each note. Note duration is calculated as the difference between ending beat positions. A possible disadvantage of the note-duration encoding compared to the note-beat-position encoding is that some rhythms tend to dominate the training set and the note-duration encoding is more susceptible to predicting the same duration (e.g. quarter note) over and over again. The note-beat-position encoding, however, does not recognize melodies that are shifted by a few time-steps as the same, while the note-duration encoding does.

This thesis will focus on the note duration encoding and leave the other encodings for future research. We will now describe the note duration encoding used in this work in detail.

4.1 Pitch-duration model

A sequence is an enumerated collection of objects. In our case the objects are notes which are defined as d-dimensional **feature vectors**. These vectors are split up into different parts, the first part being the pitch of the note. We restrict the pitch to be in the range C3 to B5. This range is large enough to capture most commonly used melodies. One note can have exactly one pitch or be a pause so this part of the feature vector is a one-hot vector with $36 + 1 = 37$ dimensions.

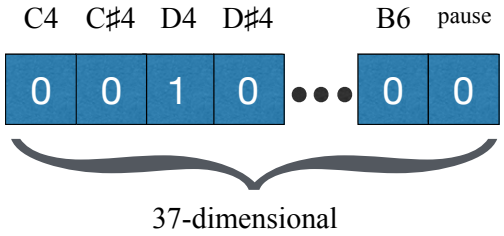


Figure 4.4: One-hot vector representation with pause indicating the pitch D4.

The next part of the feature vector is the duration of the note which is represented by a 48-dimensional one-hot vector, similar to [CSG17] and [TK18]. Each dimension in this vector represents a fraction of a whole note, e.g. the twelfth dimension represents a quarter note (12/48), the sixth dimension an eighth note (6/48), the fourth dimension an eighth triplet(4/48), etc. This design choice makes it possible to represent most common note durations and some combinations thereof.

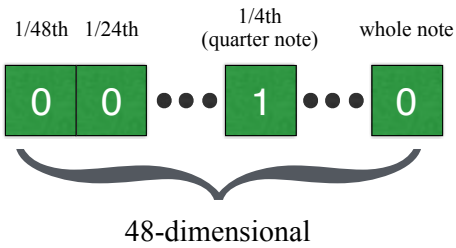


Figure 4.5: A one-hot vector representation of the duration indicating a quarter note.

With the pitch and duration vector we can represent melodies, if their pitches are in the range C4 to C6 and their note durations are representable with the duration vector.

Since we choose to emphasize this connection between melody and harmony in this thesis, we add harmonic context vectors to the encoding. We encode chords with a 12-dimensional multi-hot vector and a 12-dimensional one-hot vector representing the

root note of the chord. A *multi-hot* vector is a binary vector that, as opposed to a one-hot vector, can contain multiple ones. This way, all major and minor triads are uniquely mapped to and from the encoding. In Figure 4.6, the same C major chord as in Figure 2.3 is illustrated. Alternatively, we could have designed harmonic context as a 24-dimensional one-hot vector that represents all possible major and minor chords. This would, however, make the chord representations independent of each other. With our encoding we include the information about which notes are in which chords so that connections between chords can be learnt by the algorithm. The aim is that similarities between the chords, such as their structuring within the circle of fifths, are inferred by the learning algorithm.

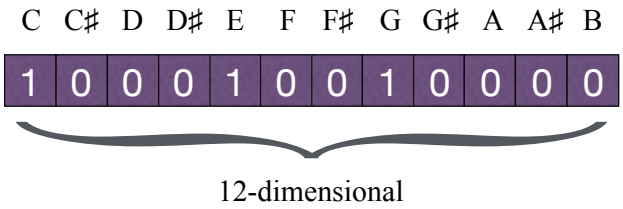


Figure 4.6: The C chord in vector representation.

The final context vector is made up of the current chord, the current chord's root note, the chord corresponding to the next note in the melody and the root note of that chord. By concatenating these vectors we obtain the 133-dimensional feature vector, as depicted in Figure 4.7.

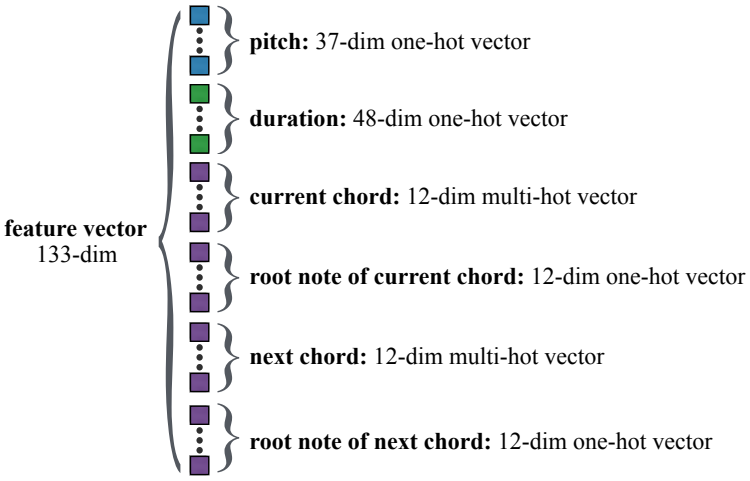


Figure 4.7: Complete feature vector in the note-duration encoding for the pitch-duration model.

A sequence of these feature vectors now represents a melody which can be fed the model. The output of the model is chosen to be one note consisting of pitch and duration. This way, the model is designed to take a sequence of notes as input and output, one note representing the next note in the sequence.

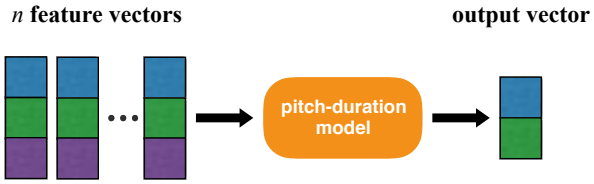


Figure 4.8: Pitch-duration model. One feature vector is made up of three parts. The blue part represents the pitch, the green part represents the duration and the purple part represents the harmonic context.

We will call the above model the **pitch-duration model**. The feature vectors now make up a sequence representing the melody that is fed into the model, as depicted in Figure 4.8.

The pitch-duration model's architecture is chosen to be a 2-layer LSTM with hidden dimension 128 in both hidden layers which results in 282.837 parameters. The output $\hat{y}$ is a vector with two sections. The first 37 dimensions make up the first section $\hat{y}_{pitch}$ and the latter 48 dimensions make up the second section $\hat{y}_{duration}$. The vector $\hat{y}_{pitch}$ represents a 37 dimensional probability distribution over the possible pitches including the pause marker and the vector $\hat{y}_{duration}$ represents a 48-dimensional probability distribution over the possible durations. The loss function is chosen as a convex combination of the cross-entropy losses of pitch and duration labels and the model's outputs, i.e. for $\lambda \in [0, 1]$.

$$L_{\text{overall}}(\hat{y}, y) = \lambda L_{\text{pitch}}(\hat{y}_{pitch}, y_{pitch}) + (1 - \lambda) L_{\text{duration}}(\hat{y}_{duration}, y_{duration}) \tag{4.1}$$

To emphasize the importance of pitch we set $\lambda = 0.8$. This loss function is differentiable as a linear combination of differentiable functions so it is straightforward to apply backpropagation.

The most similar architecture to the pitch-duration model found in literature is that of the DeepArtificialComposer [CSG17]. This differs in the splitting of the network into a separate pitch network and duration network. In their architecture, the duration network takes the previous pitches and durations as its inputs and predicts a pitch. Then the pitch network takes this prediction as well as the previous pitches and durations as input and predicts a pitch. In contrast to this, our pitch-duration model predicts both pitch and duration at the same time using one network. This means

that our model is more compact which is advantageous for deploying it in a realtime application.

4.2 Pitch-only model

For the final application of the melody generation model, we will also consider a **pitch-only model** which omits the duration vector in the encoding and simply generates a sequence of pitches. This model is useful for the interactive deployment in which we preserve the natural rhythm of the human input by copying the input rhythm.

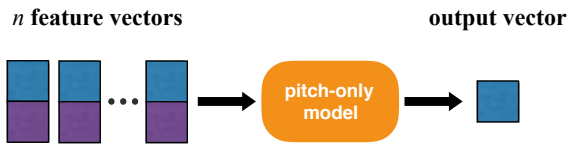


Figure 4.9: Visualization of the input and output for the pitch-only model. Feature and output vectors contain pitch and harmony information while leaving out duration information.

The architecture of the pitch-only model is a 2-layer LSTM with hidden dimension 128 in both hidden layers. The input is 85-dimensional and the output is 37-dimensional. The loss function is defined by the cross-entropy loss of the pitch-label and the pitch-only model output.

4.3 Adding tonality

To add information about the tonality of a piece to the harmonic context, we need to further investigate the concept of tonality first. Tonality is not a well-defined concept. Ideally, there would be a unique classification of a piece of music to one tonality. However, as we will explore in the following example, there is some level of uncertainty to a tonality of a piece. On top of that, most music datasets do not contain key information or even worse wrong key information. So first we need to investigate to what extent it makes sense to add tonality to the harmonic context.

We begin with an example of a chord progression of G major, C major, G major and C major. The accompanying melody is made of the notes D, D, F, G, F, D, D, F, G, F, as illustrated in Figure 4.10. The G major chord consists of the notes G, B, D and the C major chord consists of the notes C, E, G. In most western music the melody is meant to be coherent with the chords. While a melody by itself can sound consonant in itself, it is the harmonic context, given by the chords, that makes it sound coherent

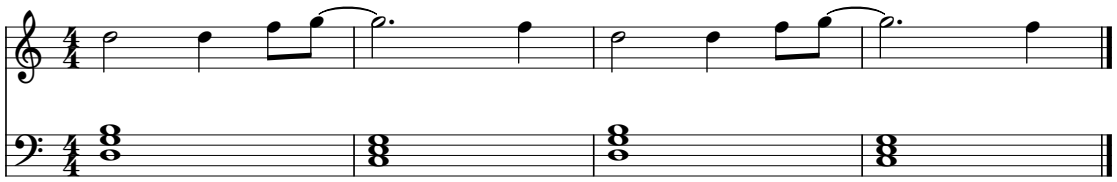


Figure 4.10: Example melody with chords. The bottom line shows the chord progression in which the G major and the C major chord alternate. The top line shows the melody consisting of the notes D,F and G.

within a harmonic structure. We will for the sake of simplicity assume that the chord always implies the corresponding scale, e.g. C major chord implies the C major scale. There are more complex scales and chords but this thesis focuses on major and minor tonality.

The harmonic structure is hierarchical. While each chord implies a tonality there is also an overall tonality of this excerpt of music. According to our example, this means that each bar has its respective tonality, i.e. the first bar is in G major, the second bar in C major, the third bar in G major and the fourth bar in C major. The overall tonality of this excerpt is marked as C major in the staff notation, but could in the context of the larger piece of music also be something different, e.g. G major. The melody and harmony together imply a hierarchy of what tonalities are more or less likely. That again is difficult to define in a clean way. The Circle of Fifths gives an indication for this hierarchy. Notes in the piece of music are part of certain scales, corresponding to the keys in the circle of fifths while they are not part of others. Scales that include all notes are more likely to be fitting candidates for the tonality of the key. The Circle of Fifths indicates this conveniently as neighboring keys only differ by one note in their scales. In most cases the tonality for this example would be categorized as C major. Only the C major scale contains all notes (D,E,F,G,B) used in the example. The G major scale only does not contain the F note but this F note is only played during or leading up to the C major chords. This means the F note could have been used here as a stylistic device to create some tension as it is not completely dissonant in the context. Therefore, G major is also a viable candidate for the tonality of the whole excerpt. This ambiguity makes it hard to come to clean conclusions about consonance. However, as seen in the circle of fifths, the difference between the scales of the two viable candidates C and G is only one note, i.e. F vs F♯, so they are harmonically very similar by that measure. Following this similarity, an algorithm that misjudges the example melody to be in G even if it is in C might still yield valuable information. Therefore, it is reasonable to assume that an algorithmically detected key is beneficial for the model. Moreover, we will evaluate in Chapter 7 whether the model benefits from the added key information based on the metrics.

4.4 Key detection with pitch class profiles

One of the most seminal approaches for key detection was introduced by Krumhansl in [Kru90]. Despite being long-established, it is at the time of writing still a widely used method for detecting keys in symbolic music data. The technique stems from the results of an experiment in which listeners rated how well pitch-classes fit in with a key based on their perception [KK82]. For the algorithm, a set of 12 dimensional vectors is defined, called *key-profiles*. These vectors represent the compatibility of each pitch-class with the key. The major and minor key-profiles used here are from [Aar03] and differ from the original ones in [KK82]:

$$\text{major} = \begin{pmatrix} 17.7661 \\ 0.145624 \\ 14.9265 \\ 0.160186 \\ 19.8049 \\ 11.3587 \\ 0.281248 \\ 22.062 \\ 0.145624 \\ 8.15494 \\ 0.232998 \\ 4.95122 \end{pmatrix}, \text{minor} = \begin{pmatrix} 18.2648 \\ 0.737619 \\ 14.0499 \\ 16.8599 \\ 0.702494 \\ 14.4362 \\ 0.702494 \\ 18.6161 \\ 4.56621 \\ 1.93186 \\ 7.37619 \\ 1.75623 \end{pmatrix}.$$

The key profiles are consistent with tonal music theory as the notes from the corresponding scale have higher values than the rest and the values from the main triad (major triad or minor triad) are highest, as illustrated in Figures 4.11 and 4.12.

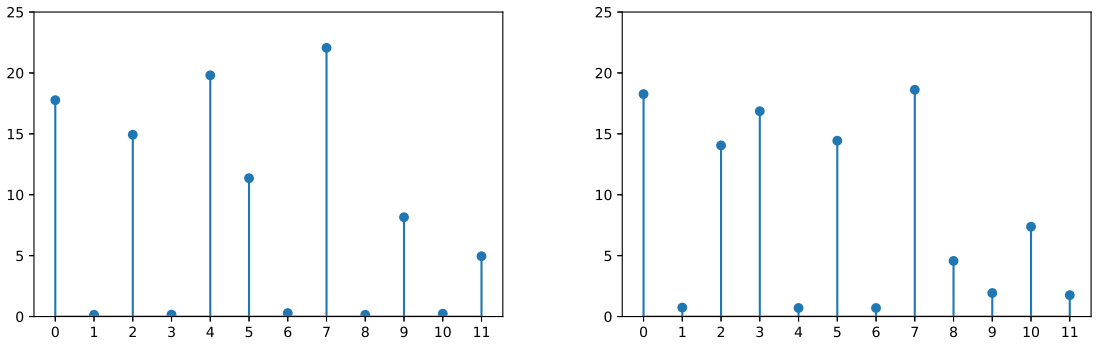


Figure 4.11: Aarden key profiles for major (left) and minor (right). The x-axis on the figure goes from 0 to 11 representing all pitch-classes. For the key of C, we can think of the root pitch-class C as 0 and C♯ as 1, etc. For the key of C♯, we can think of the root pitch-class C♯ at 0, and so on.

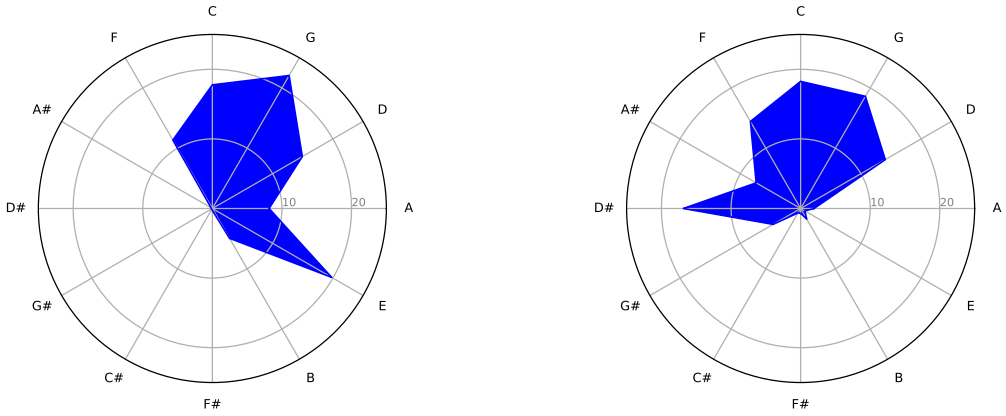


Figure 4.12: Key profiles plotted onto the Circle of fifths (C major left and C minor right). By plotting the key-profile values on the circle of fifths one can visualize the structure of the vectors. One side of the circle of fifths has small values while the other has large values. This shows a connection between the neighboring pitch-classes on the circle of fifths that is captured by the vector representation in the key profiles.

The KS algorithm is based on the *Sample Pearson correlation coefficient* which is defined for two vectors $x = (x_1, ..., x_{12})$ and $y = (y_1, ..., y_{12})$ with means $\bar{x} = \frac{1}{n}\sum_{i=1}^{12} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{12} y_i$ as:

$$r_{xy} \coloneqq \frac{\sum_{i=1}^{12}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{12}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{12}(y_i - \bar{y})^2}}. \tag{4.2}$$

Rearranging with $s_x \coloneqq \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ and $s_y \coloneqq \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2}$ yields:

$$r_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}\Big(\frac{x_i - \bar{x}}{s_x}\Big)\Big(\frac{y_i - \bar{y}}{s_y}\Big). \tag{4.3}$$

This shows that r_{xy} is the mean of the products of the standard scores $(\frac{x_i - \bar{x}}{s_x})$.

Finally, a piece of music is represented by another 12 dimensional vector, the *sample vector*, by accumulating the total duration of each pitch class in the piece. The Krumhansl-Schmuckler key-finding algorithm is obtained by calculating the Pearson correlation coefficient for the sample vector x and all key-profile vectors y and choosing the key with the highest coefficient. It reflects the strength with which the key is represented in the sample vector. The key with the maximum correlation value is the preferred key. The idea of the algorithm is to compute the most similar key to the sample vector according to this coefficient. We will evaluate the KS algorithm after we introduce the datasets.

4.5 Pitch-duration-key model

The harmonic context can be expanded by adding key information with the key-detection algorithm. This yields a new model which we will call **pitch-duration-key model**.

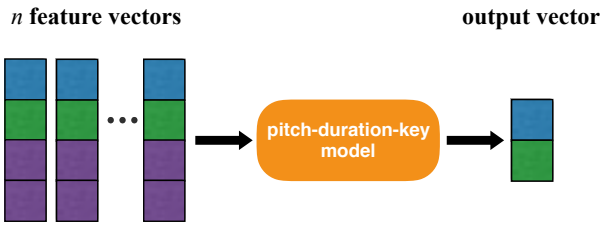


Figure 4.13: Pitch-duration-key model. The input contains pitch, duration, chord and key information and the output contains pitch and duration information.

The key information will be added to the feature vector just like the current chord with a 12-dimensional one-hot vector indicating the root of the key and 12-dimensional one-hot vector indicating the respective major or minor scale. The root key is added to make a differentiation between a major scale and its relative minor scale as they consist of the same pitch-classes. The input is of dimension $157 = 37+48+4*12+2*12$ and the output is of dimension $85 = 37 + 48$. Hence, the architecture of the pitch-duration-key model is the same as that of the pitch-duration model but with the added context of the key.

4.6 Generating melodic sequences

To obtain a melody of length $m \in \mathbb{N}$, we start by predicting one output vector from a *inputmelody* of length $n \in \mathbb{N}$. Here, n is not fixed as LSTM based model can take an input sequence of any length. The predicted output vector is appended to the inputmelody after a matching harmonic context is concatenated to it. The inputmelody now consists of $n + 1$ feature vectors and can be fed to the model to predict the next output vector. This procedure is repeated until the inputmelody is of length $n + m$ and the last m vectors in this melody yield the predicted melody of length $m \in \mathbb{N}$. This way, our model takes a melody of any length and can output a melody of any length as specified in the requirements.

5 Data

In this chapter, we introduce the datasets that are used for training and evaluation and discuss preprocessing.

5.1 Datasets

This thesis aims to predict a melody in twelve-tone equal temperament. To reduce the complexity of this task it was decided to use a symbolic representation of music as opposed to Audio data. Two commonly used file formats of symbolic music are the MIDI file format and the MusicXML file format, both of which are used in this work. The amount of songs used in other related research for training similar models differs greatly from paper to paper and ranges from only few minutes [Moz91], [ES02] in earlier experiments to 200 hours [HSR+19] in the Tensorflow Magenta dataset.

5.1.1 Wikifonia Dataset

The Wikifonia dataset was developed in a collaboration between several institutes of higher education in Ghent (Belgium). It consists of over 100 music pieces in the leadsheet file format MusicXML which is an XML based file format representing Western musical notation. The music is mostly contemporary/popular music by artists such as Elton John, Adele or Brian Wilson. 99.8% of the notes in the dataset are in the range C3 to B5 which is the range for the note-duration encoding.

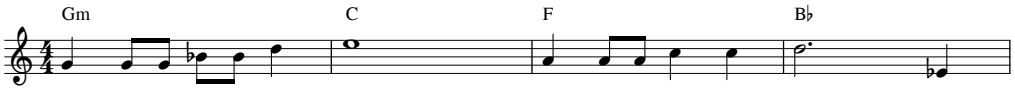


Figure 5.1: Excerpt from Elton John song from the Wikifonia dataset.

Models trained with this dataset will be marked as Wikifonia models. The used corpus from this dataset makes up 4005 bars of music which is comparable to similar research such as [TK18] which used 1700 bars of music.

This distribution of pitches in Figure 5.2 shows that the chosen range for pitch is appropriate for the data. Most pitches are concentrated in the center of the range and there are less on the outer margins of the range. Moreover, the distribution reveals

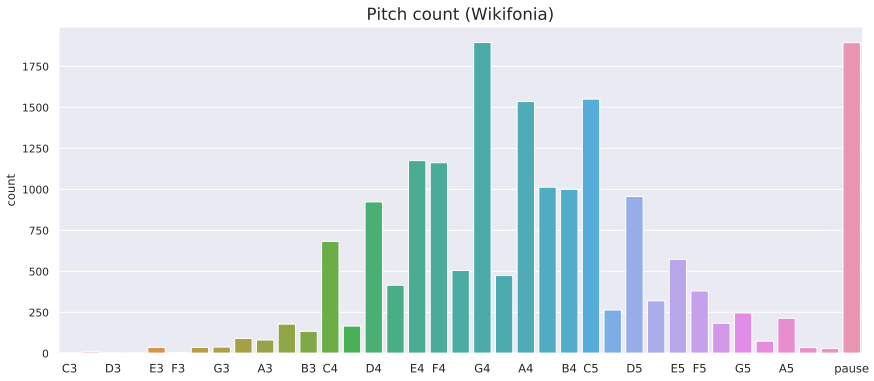


Figure 5.2: Histogram for pitches in the Wikifonia dataset. This plot shows how often each pitch occurs in the Wikifonia dataset.

that the notes from the C major scale occur visibly more often than the notes that are not contained in the C major key.

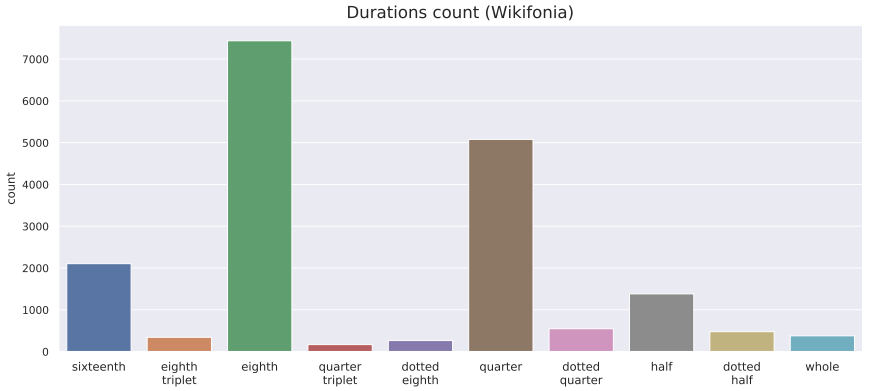


Figure 5.3: Histogram for durations in the Wikifonia dataset. This plot shows how often each duration occurs in the Wikifonia dataset.

The distribution of durations in the Wikifonia dataset, as plotted in Figure 5.3, shows that the dataset is dominated by eighth and quarter note durations.

5.1.2 Beatles Dataset

The Beatles dataset was created in [Lac16] by transcribing excerpts from 16 Beatles songs from the music book "Pop Classics For Piano: The Very Best Of The Beatles - Easy Arrangements for Piano" by Hans-Günter Heumann to MIDI files.

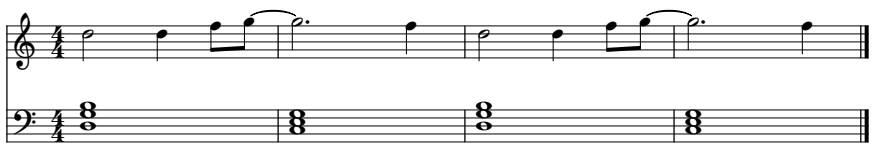


Figure 5.4: Example melody excerpt with chords from the Beatles dataset.

It consists of 68 MIDI files that are each 8 bars long which makes 544 bars of music. Melody pitches range from C4 to B5 while chord pitches range from C3 to B3. Models trained on this dataset will be marked as Beatles models.

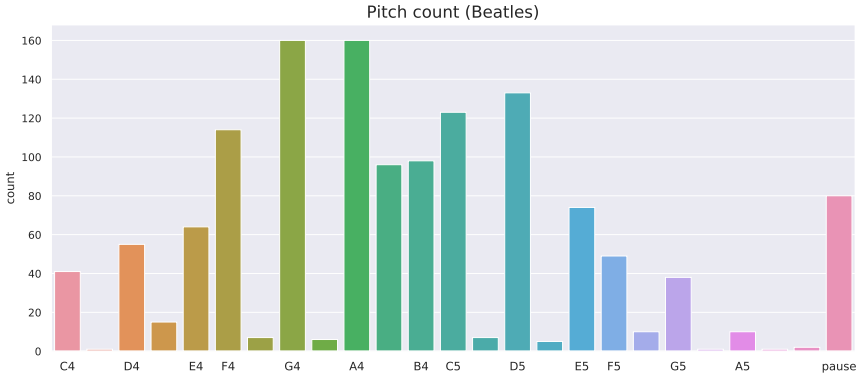


Figure 5.5: Histogram for pitches in the Beatles dataset. This plot shows how often each pitch occurs in the Beatles dataset.

The distribution of pitches in the Beatles dataset, as can be seen in Figure 5.5, is similar to that of the Wikifonia dataset. The distribution of durations in the Beatles dataset is also dominated by the eighth and quarter notes, just like in the Wikifonia dataset.

However, the the Beatles dataset consists of more occurring duration classes than the Wikifonia dataset. This can indicate that the Beatles dataset consists of more intricate rhythms than the Wikifonia dataset. An alternative explanation for this is that due to the encoding in MIDI the durations used in the Beatles dataset are less cleanly converted to the pitch-duration encoding than the XML dataset.

This points to a potential issue with datasets in the MIDI file format, as it is easily possible to create MIDI files with “non-clean ” rhythms. MIDI files are made of events that are timestamped by *ticks*. A common choice for discretization is that one tick equals 1/480th of a quarter note. This makes it easy to create MIDI files with slight delays in the rhythm. For example, it is possible to play a note a few ticks behind the beat. The complexity of rhythms is increased immensely by these subtleties. On the

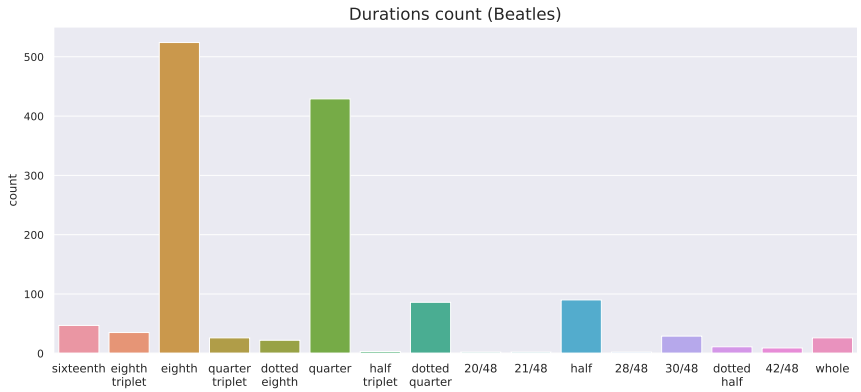


Figure 5.6: Histogram for durations in the Beatles dataset. This plot shows how often each duration occurs in the Beatles dataset.

one hand, MIDI files with such complex rhythms can be of very high quality, if they are, for example, created by recording professional musicians playing with a MIDI controller. On the other hand, MIDI files can easily contain "non-clean" rhythms that root in the creator of the MIDI file being sloppy in the creation process. In any case, such complex rhythms are difficult to encode in the rhythms into the note-duration encoding. The tick resolution is way higher than the duration resolution in our encoding which means the durations have to be rounded to the closest possible one. Contrarily, XML files make such examples more difficult to create as their creation is inspired by the staff notation where one defines pitches and durations similar to the pitch-duration encoding. Therefore, MusicXML datasets seem to contain "cleaner" rhythms than MIDI datasets.

5.1.3 Lakh MIDI Dataset

The Lakh MIDI dataset [Raf16, Raf] is a collection of 176,581 unique MIDI files. 45,129 of these MIDI files have been matched and aligned to entries in the Million Song Dataset. The Million Song Dataset is a freely-available collection of audio features and metadata for a million contemporary popular music tracks [BMEWL11]. The Lakh MIDI dataset aims to facilitate large-scale music information retrieval, both symbolic (using the MIDI files alone) and audio content-based (using information extracted from the MIDI files as annotations for the matched audio files). Though the quantity of this dataset is great, it could not easily be used to train our model. Each file is made up of multiple tracks that are inconsistently labelled, which makes it impractical to cleanly import a monophonic melody with a harmony. This dataset is used to benchmark the key detection algorithm.

5.1.4 Bach Dataset

The Bach dataset contains 96 fugues and preludes of Bach from the "Well-Tempered Clavier". This is a convenient dataset for testing key detection algorithms as each of the 24 major and minor keys are contained in it.

5.2 Data preprocessing

Before being able to train a model with data, the datasets need to be converted to the note-duration encoding described in Chapter 4. The Python package *music21* [Cc, CA10] was used to transform both MusicXML files and MIDI files into the encoding. To fit the encoding, only songs with maximum note durations of a whole note are considered. After preprocessing, the Beatles dataset consists of 16,055 data samples and the Wikifonia dataset consists of 205,524 data samples.

The datasets are converted to feature vectors by matching note pitch, note duration, current chord and next chord for each occurring note. A sequence length of 8 is chosen as the length for the input sequences. Each sequence of 8 consecutive feature vectors makes up an inputmelody and the label is set as the following note's pitch and duration. Hence, one training data sample consists of a sequence of 8 feature vectors, a one-hot label for the next note's pitch and a one-hot label for the next note's duration.

5.2.1 Data augmentation

The data is augmented by modulating the notes in the datasets into all twelve keys. Doing this removes the bias towards keys used more often in the dataset and expands the information in the data to all keys. We modulate the pitches downwards. If a pitch falls below C3 we transpose it up an octave. This changes the trajectory of the melody, but leaving it out would change the melody more drastically. Only 2% of the pitches in the Wikifonia dataset are below C4, so not many notes needed to be transposed up an octave. Nevertheless, depending on the use case, it can be advantageous to expand the possible pitch range to represent more than 36 pitches to prevent this issue. However, we deliberately choose to limit the range of the possible pitches to 36, because this amount of possibilities is plenty for the application of the interactive melody generation.

Figure 5.7: Transposing an excerpt down by one half-step (from left to right).

The changed distribution of pitches in the training sets can be seen in the Figures 5.8 and 5.9 below. The distribution of duration remains unchanged as each duration simply appears 12 times as often.

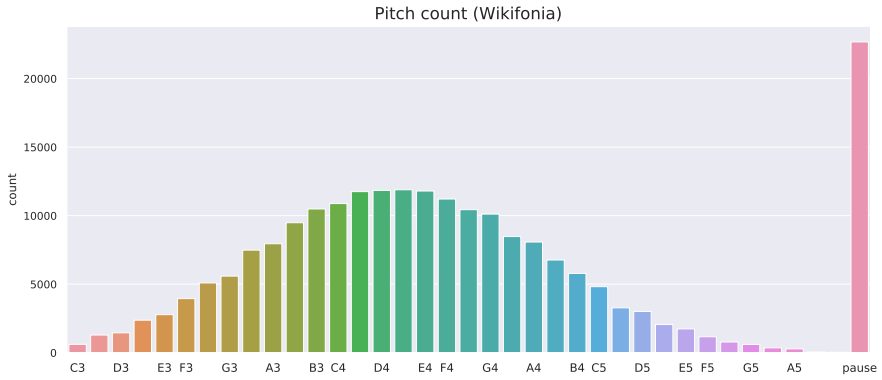


Figure 5.8: Histogram showing the distribution of pitches in the augmented Wikifonia dataset after transposing each piece of music into all twelve keys of the same mode. From the plot it may look like the number of pauses has increased but the ratio of pitch notes to pauses remains the same.

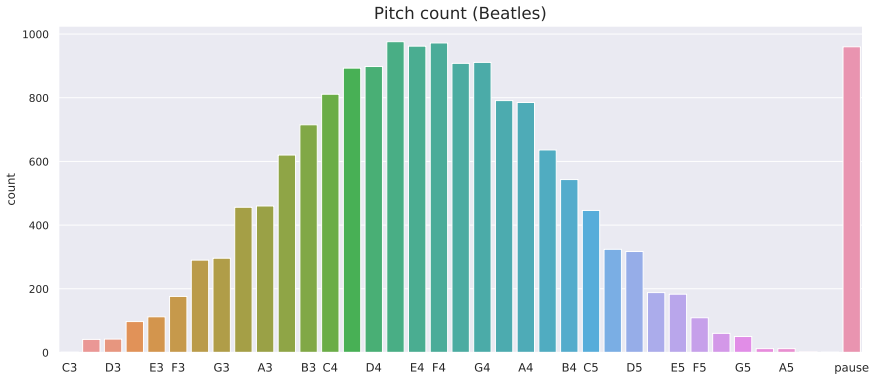


Figure 5.9: Histogram showing the distribution of pitches in the augmented Beatles dataset after transposing each piece of music into all twelve keys of the same mode.

6 Training and Evaluation Methods

This chapter discusses training and evaluation of the models introduced in Chapter 4. Training methods, model design details and hyperparameters for each model are given. Additionally, the methods for evaluation of the models are explained.

6.1 Training methods

Training means minimizing the loss function with stochastic gradient descent. More specifically, the training set is shuffled so that mini-batches can be drawn. Mini-batches of size *batch-size* are drawn at random without replacement. Each mini-batch is forward and backward propagated and the parameters are updated, as specified in 3.3.1. An *epoch* is defined as a single pass through the entire training set. The following techniques help generalization of the model (higher accuracy on unseen data) during training [GBC16].

6.1.1 Early Stopping

For early stopping, a validation set is drawn from the training set. The accuracy is evaluated on the validation set after each epoch and stopping criteria is applied to this evaluation. During training, the accuracy usually first increases on all sets and then plateaus out or decreases on the validation and test set, as the parameters are optimized only with respect to the training set. Training is stopped, when the accuracy on the validation set is maximal. This is the point where the neural network starts to overfit on the training data because performance does not improve on the validation set, as sketched in Figure 6.1. Stopping is achieved in practice with help of a *patience* value. The patience value is an integer that defines how many epochs we continue to train once the validation accuracy starts to increase. [LTN18]

6.1.2 Dropout

Another commonly used technique that aids generalization is Dropout [SHK+14]. With this technique, at each training step nodes in the network are "dropped out" of the network with a probability p_{drop} so that a reduced network is left. The reduced network is trained for one training step and the removed nodes are reinserted into the network with their last-known weights. Rates between $p_{drop} = 0.3$ and $p_{drop} = 0.5$ are common.

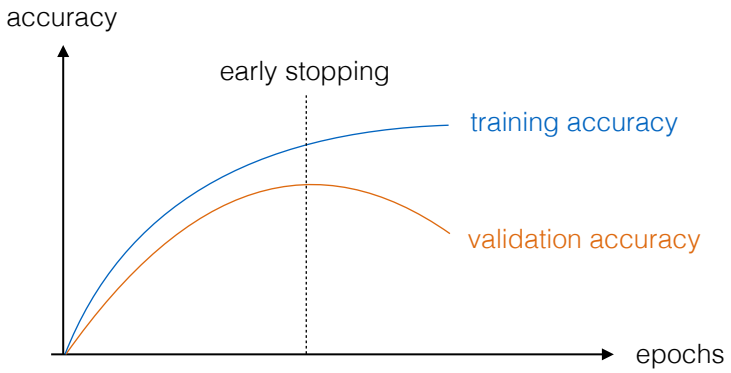
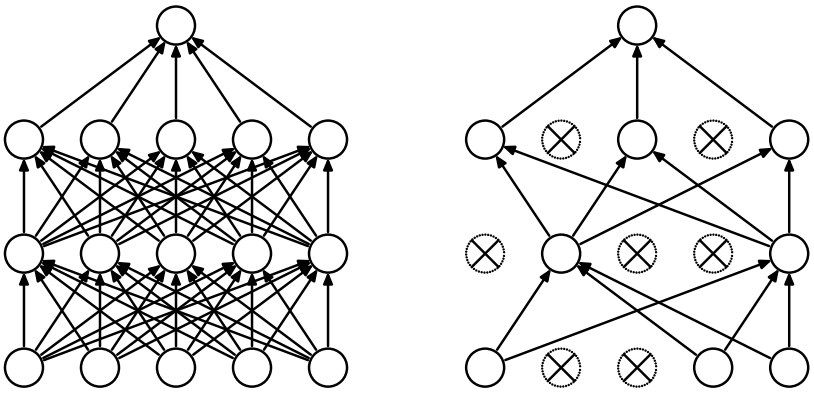


Figure 6.1: Visualization of early stopping and overfitting. The training accuracy and the validation accuracy are plotted. We want to stop training when the validation accuracy starts decreasing.

After training on different reduced networks the full network is combined by weighting all weights by the factor p_{drop} so that the expected value of each node is the same as in the training stages. With this we obtain an approximation of the sample average of all possible 2^n networks, which means we also only have to test the one combined network.



(a) Standard Neural Net (b) After applying dropout.

Figure 6.2: Dropout visualization on a neural network. A comparison between the connections of a fully connected neural network (left) and the connections remaining after the application of dropout in a training step (right). Image taken from [SHK+14].

6.1.3 Gradient Clipping

The gated mechanism of the LSTM only addresses the vanishing gradient problem and not the exploding gradient problem. If a parameter gradient is large, the parameter update could "throw" the parameters too far into a region where the objective function is larger, undoing much of the work that has been done to reach the current solution [GBC16]. A straightforward countermeasure to this problem is *gradient clipping*. Here, we set a threshold, e.g. $v = 1$ and if the gradient norm $\|g\| > v$, we update the gradient by $g_{\text{new}} = \frac{v}{\|g\|} g$.

6.1.4 Initialization of weights and biases

Neural networks are non-convex because of their nonlinear activation functions. Thus, gradient descent converges to a local minimum, if it converges. Therefore, the minimization achieved by gradient descent based algorithms is highly dependent on the initialization of the network's weights and biases. All models in this thesis were initialized with the Xavier Glorot initialization [GB10] which initializes the parameters with a uniform distribution in the range $[-\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}]$, where n_i is the number of units in the i-th layer.

6.2 Training setup

The models were trained using Tensorflow with Python on the Google Colab platform[1]. The hardware specifications are a Tesla K80 GPU with 12 GB of RAM and a single core Xeon Processor 2.3Ghz. Input and output of the models are described in section 4. From each dataset 10% of the data was left out as test data and the remaining 90% was split up in 80% train data and 20% validation data.

Different configurations of hyperparameters were tested on the pitch-duration model with the Beatles dataset and the configuration with the highest test accuracy was chosen to be used for all models. The Adam optimizer[GBC16, Ch . 8.5.3] was used as the parameter update rule with a learning rate of 0.001 and mini-batches of size 32. We used two hidden LSTM layers with layer size of 128. Dropout was applied with $p_{drop} = 0.3$. We used early stopping with a patience value of 3. The number of epochs varied between 10 and 50. Gradient clipping by norm with a threshold of 5 was applied.

[1] `https://colab.research.google.com`

6.3 Evaluation methods

The models were trained by optimizing their capability to predict the correct next note in a sequence of notes. On the one hand, the goal of the evaluation is to show whether the model is able to generalize in the sense of being able to predict the correct next note on the test dataset which represents unseen data. On the other hand, the aim is to teach the model how to create consonant melodies in general. As stated earlier, consonance is a subjective concept, but we will nevertheless define metrics to measure this concept. The metrics are based on the following assumptions about characteristics of consonant melodies:

Harmonic consistency. Melodies should consist of notes that fit the harmonic context that is created by the chords.

Creativity.

Variation. Melodies should consist of various pitches and durations. Furthermore, the generative model should not just copy melodies from the dataset.

Few pitch repetitions. Melodies should not repeat the same pitch over and over again.

Stability.

Not too much variation. Melodies should not consist of too much variation.

Conjunct melodic motion. There should be few large interval jumps from one note to the next. This is a feature of tonal music described in [Tym10, Ch. 1].

To begin with, we explain how melodies are generated for evaluation purposes. Then we define metrics that measure musical qualities of the melodies. The evaluation of the generated melodies will be discussed in Chapter 7. The goal of this evaluation is to get insight into how well the models learned to generate melodies similar to those in the dataset. By quantifying specific musical notions, we obtain a more meaningful insight into the quality of the model than the accuracy metric, that it is trained on, provides.

In this work, similar to [TK18] and [DHYY18], we generate 1.000 melodies (feature vector sequences) of length 9 with each model. The choice of the length 9 is motivated by the related work [TK18] enforcing their predicted sequences to be between half a measure and four measures. With our model mostly predicting eighth and quarter notes, a melody of length 9 is most likely in that range, too. The choice of fixed length sequences speeds up computations of the evaluation as they can be computed with matrix operations. Still, computation time remains a limitation to scalability of the evaluation approach. Each sequence is generated by priming the model with a randomly chosen melody of 8 feature vectors out of the test data that is left out at

training. A melody is then predicted successively until it consists of 9 feature vectors. We prime the model with test data to simulate the real case in which the model is primed with a melody that is most likely not in the training set.

6.3.1 Harmonic Consistency (HC)

The Harmonic Consistency (HC) metric from [TK18] measures how well predicted notes adhere to the given chord progression. It categorizes notes in relation to the chords into black, green, blue, and red notes. The note categorization is not clearly defined in [TK18] as blue notes are set as "approach notes" (by one half-step) to a chord note or to a note sympathetic to the chord and all notes can be considered approach notes to some note sympathetic to the chord. In this work, harmonic consistency is measured as follows.

Black notes are notes from the current chord. Green notes are notes from the scale belonging to the current chord. For major chords, the corresponding scale is the major scale and for minor chords the corresponding scale is the minor scale. The remaining notes are classified as blue, if they are part of the next chord or the next chord's scale and not longer than an eighth note. The leftover notes are red notes as can be seen by examples in Figure 6.3.

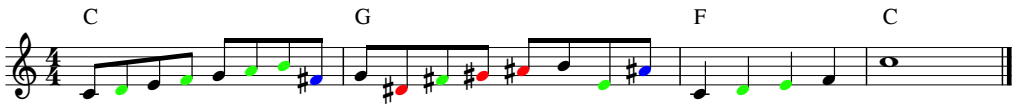


Figure 6.3: Example melody to visualize the note color scheme. Black notes are chord notes, green notes are chords from the scale belonging to the chord, blue notes are approach notes and red notes are notes that are classified as dissonant.

6.3.2 Creativity metrics

The following metrics are designed to measure whether the generated melodies are "interesting" in the sense that they vary in pitch and rhythm. Melodies can further be considered interesting if they vary from the melodies within the training corpus. These metrics were adapted from [TK18].

The **Pitch Variations (PV)** metric represents the ratio across all sequences of the number of distinct pitches to the total number of notes in the predicted sequence.

$$\text{ratio(sequence)} = \frac{\text{number of distinct pitches in sequence}}{\text{total number of notes in sequence}}$$

The average ratio over all predicted sequences is computed to be the PV value. The averages in the metrics are computed by the arithmetic mean.

Similarly, the **Rhythm Variations (RV)** metric measures the average ratio across all sequences of the number of distinct note durations to the total number of notes in the sequence.

$$\text{ratio(sequence)} = \frac{\text{number of distinct durations in the sequence}}{\text{total number of notes in the sequence}}$$

The average ratio over all predicted sequences is computed to be the RV value. The PV and RV metrics indicate if the variation in pitch and duration is too high or too low.

The **Rote Memorization (RM3,RM4,RM5,RM6)** metric measures how often a subsequence of respectively three, four, five or six pitches are copied exactly from the corpus. For fixed $s \in [T]$, a sequence $(x_1, ..., x_T)$ consists of $T - s + 1$ *subsequences* of length s of the form $(x_t, ..., x_{t+s-1})$ for $t \in [T - s + 1]$.

$$\text{RMx score} = \frac{\text{number of subsequences of length x contained in the corpus}}{\text{total number of subsequences of length x}}$$

We measure this metric for subsequences of lengths 3,4,5 and 6. This metric gives more insight into the generalization capability of the model

6.3.3 Mode collapse metrics

Mode collapse metrics measure whether the generative model is collapsing to a parameter setting that always emits the same output. The term *mode collapse* stems from the field of generative adversarial network research, a type of neural network, and describes the phenomenon of the network generating a limited diversity of samples [TK18]. In music generation, this could mean predicting the same pitch or duration over and over again which would imply less interesting melodies. These metrics were also adapted from [TK18].

The **Consecutive Pitch Repetitions (CPR)** metric measures the frequency of occurrences of two consecutive pitch repetitions, i.e. the number of times that one pitch is repeated three times consecutively. The CPR metric shows if the predicted melodies fulfill the requirement of few pitch repetitions.

The **Durations of Pitch Repetitions (DPR)** metric measures the frequency of a CPR that lasts at least a half note, i.e. the number of times that one pitch is repeated three times consecutively and their durations sum up to at least a half note. A high DPR score points to noticeable pitch repetitions that likely make a melody less interesting.

The **Tone Spans (TS)** metric measures the frequency of tone spans greater than an octave, i.e. the number of times a two consecutive notes are further than 12 half-steps apart from each other. A high TS score would indicate dissonance as the melody does not meet the requirement of conjunct melodic motion.

7 Evaluation

This chapter presents the evaluation results for the key detection algorithm and the evaluation of the models with the metrics described in section 6.3.

7.1 Evaluation of the Key Detection Algorithm

To measure performance of the Krumhansl-Schmuckler (KS) key detection algorithm from section 4.4 , it was analyzed how many pieces of music could be classified correctly on different datasets. The KS algorithm with original weights scored 81% overall correct on the 48 preludes of Bach's "Well-Tempered Clavier" when only given the first four notes of each piece as input [Kru90, Ch. 4]. We tested the KS algorithm with different key-profiles and on different datasets. On all tested datasets, the KS algorithm with Aarden key-profiles gave the best results. Specifically, on the 96 fugues and preludes of Bach, the KS-algorithm scored 83% correct with 14% classified in the relative minor key. The performance was further tested on the Lakh Midi Dataset [Raf16] which is a larger dataset than the Bach dataset. As noted in [RE16], some MIDI programs automatically annotate the key signature in the MIDI file to C major even though the piece is in a different key. Pieces with no key annotation or with annotated C major key signature were therefore omitted from this performance measure. On a subset of 169 pieces from this dataset the KS-algorithm scored 62% correct with 18% classified in the relative minor key. Though the KS-algorithm is not 100% accurate in detecting keys, these measurements indicate a reasonable performance. No better key detection algorithm was found, so we leave it up for future research to improve the algorithm. As explored in section 4.3, a wrongly detected key might still contain valuable information for the algorithm. Therefore, we choose to use this key detection algorithm for the pitch-duration-key model.

7.2 Evaluation of the models

We evaluated each model on the Beatles and the Wikifonia datasets, see Chapter 5, to get an indication of how well the models work on different types of original data formats. First, we give the accuracy scores of the models after training as basic indications for generalization capacity of each model.

The model overfits on the training set because the model is trained on it. The hyper-parameter tuning can also cause overfitting on the validation set. Thus, the test

Dataset	Model	pitch accuracy			duration accuracy		
		train	val	test	train	val	test
Beatles	pitch-only	0.46	0.34	0.32	-	-	-
Beatles	pitch-duration	0.62	0.31	0.28	0.57	0.43	0.53
Beatles	pitch-duration-key	0.53	0.33	0.28	0.55	0.43	0.53
Wikifonia	pitch-only	0.44	0.31	0.35	-	-	-
Wikifonia	pitch-duration	0.48	0.32	0.35	0.61	0.55	0.59
Wikifonia	pitch-duration-key	0.52	0.32	0.36	0.61	0.55	0.59

Table 7.1: Train, validation and test accuracies for each model trained on each dataset.

accuracies are the most important ones because it represents the real case of unseen data. Test pitch accuracies for the models trained on the Wikifonia dataset are 3-8% higher than for the models trained on the Beatles dataset, as can be seen in Table 7.1. Test duration accuracies are 6% higher in Wikifonia models. By this measure, the Wikifonia models show better generalization capacity. This result makes sense as the Wikifonia dataset is ten times as large as the Beatles dataset and therefore most likely contains more musical information.

In general, pitch test accuracies of 28% to 36% indicate that the models are capable of learning patterns in melodies. It means, that given 37 options of pitches to choose from at each prediction step, the correct one is chosen 28-36% of the time. On the contrary, a duration accuracy of 53% to 59% does not indicate much, as the majority of the datasets are made up of a few duration possibilities. Plotting the quantities of predicted pitches and durations for the 1.000 generated sequences visualizes this for the case of the pitch-duration model, see Figure 7.1 and Figure 7.2 for the Wikifonia model as well as Figure 7.3 and Figure 7.4 for the model trained on the Beatles dataset.

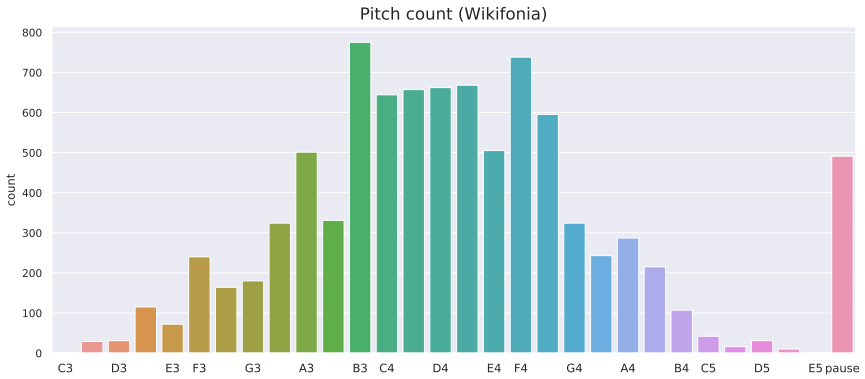


Figure 7.1: Quantitative analysis on the Wikifonia pitch-duration model. The plot shows how often each pitch is predicted in the 1.000 generated melodies. The distribution is similar to that of the dataset.

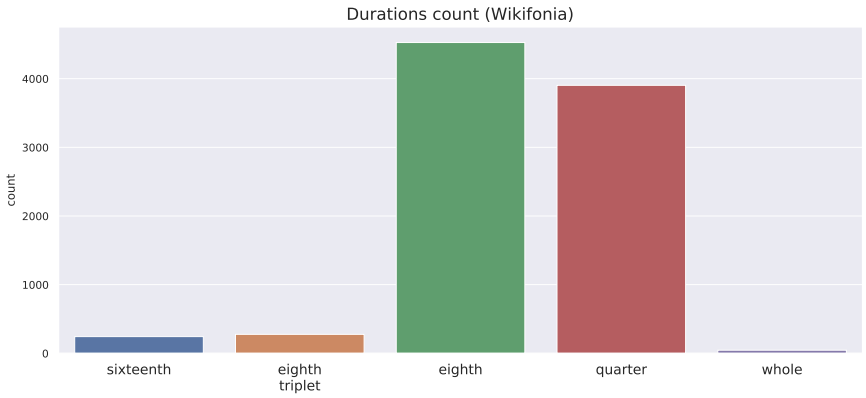


Figure 7.2: Quantitative analysis on the Wikifonia pitch-duration model. The plot shows how often each duration is predicted in the 1.000 generated melodies. The predicted durations are less diverse than in the dataset. The eighth and quarter notes make up the majority of durations just as in the dataset.

The prediction pitch count plot for the Wikifonia dataset shows a similar distribution of pitches as in the training dataset. The distribution is also symmetrical and bell-shaped with center around E♭4. It has an outlier at the pause marker. At the tails the prediction counts get less similar to the plot for the training data. This shape is similar to a Gaussian distribution with an outlier at the pause. It shows that our model learns structure in the dataset.

The predicted melodies consist of only 5 different types of durations in the Wikifonia model, see Figure 7.2, and of only 2 different types in the Beatles model, see Figure 7.4. In contrast to this, the dataset contains a higher variety in durations, as can be seen in 5.3. The quarter note and the eighth note are being predicted the majority of the time, just like in the datasets. The duration distribution of the dataset is therefore not perfectly approximated but shows similar features. The lack of different rhythms might indicate that the generated melodies are not very interesting in their rhythms. This can be a result of unbalanced classes. Some durations dominate the dataset naturally as they occur more frequently in the dataset. A common approach to counteract this is to scale the error depending on the class, which would make all classes equally as important. The balancing of classes can overcompensate easily though. The model should not learn to predict all durations with equal likelihood, some durations should remain more frequent than others. Further research is needed to improve on the model's capability to learn more complex rhythms while staying close to the dataset's duration distribution.

To get a deeper insight into the musical qualities of the model, we now investigate the model's predictions with metrics that are based on musical concepts. If possible, the dataset corpus for each source dataset is evaluated with the metrics. Then we can compare the scores of the models to those of the respective datasets to be able to get an

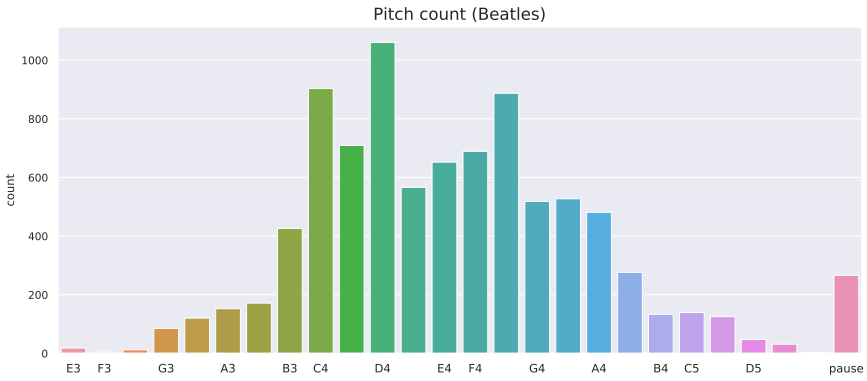


Figure 7.3: Quantitative analysis on the Beatles pitch-duration model. The plot shows how often each pitch is predicted in the 1.000 generated melodies. The distribution is resembles the distribution of the Beatles dataset pitches, yet it contains a few out-liers at C4, D4 and F♯4.

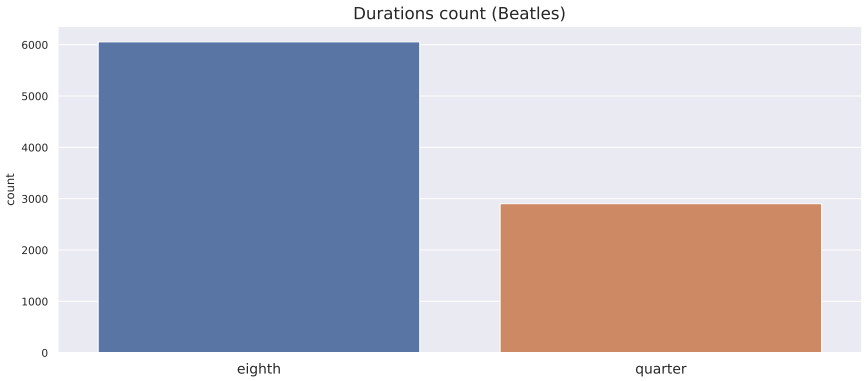


Figure 7.4: Quantitative analysis on the Beatles pitch-duration model. The plot shows how often each duration is predicted in the 1.000 generated melodies. The predictions only contain eighth and quarter notes while the dataset contains more variety of durations.

indication of how well the model is able to pick up musical concepts. The comparison between the scores of the same model on different datasets has its limitations, as these datasets vary in size. Hence, we will concentrate on the comparison of each dataset-model combination to the respective corpus. The following sections evaluate the generated melodies for each model with the introduced metrics from section 6.3.

7.2.1 Harmonic Consistency (HC)

This metric measures the fraction of black, green, blue and red notes, as described in subsection 6.3.1. Pauses are a separate category. As the pitch-only model does not contain information about the durations, the blue notes cannot be measured for this model which results in more red notes.

Dataset	Model	Black	Green	Blue	Red	Pauses
Beatles	corpus	0.619	0.252	0.01	0.055	0.064
Beatles	pitch-only	0.724	0.246	-	0.03	0.018
Beatles	pitch-duration	0.611	0.317	0.02	0.023	0.03
Beatles	pitch-duration-key	0.647	0.3	0.014	0.029	0.011
Wikifonia	corpus	0.638	0.172	0.008	0.093	0.089
Wikifonia	pitch-only	0.71	0.245	-	0.045	0.112
Wikifonia	pitch-duration	0.667	0.248	0.02	0.011	0.055
Wikifonia	pitch-duration-key	0.657	0.256	0.022	0.009	0.056

Table 7.2: Harmonic Consistency metric scores for each model.

Table 7.2 shows the evaluation on the predictions of each model-dataset combination. The values in the table denote the share of each category for the predictions. All models predicted an amount of pauses within 6% of the amount in their respective training corpus. Thus, the models emulated the pause frequency from the music pieces in the corpus. The lower the share of red notes, the more consonant the melody sounds in the harmonic context of the chords. Black notes sound very consonant while green and blue notes sound mostly consonant and can make the melody interesting. All models, except the Beatles pitch-duration model, predicted a higher share in black notes than there are in the corpus. The Beatles pitch-only model even predicted 10.5% more black notes than the Beatles corpus consists of. This indicates that the model learnt a strong connection between the predicted notes and the current chord notes. It likely stems from the fact that black notes dominate the datasets and the path of least resistance was to learn to play chord notes. Playing mostly the notes from the chord makes the melodies harmonically consistent but may lead to little variety. The pitch-duration and pitch-duration-key models trained on the Wikifonia dataset scored lower amounts of red notes than the rest, even though the Wikifonia corpus consists of twice the amount of red notes as the Beatles corpus. All models predict an amount of green notes within 10% of the corpus frequency. Low red note frequencies combined with high black and green note frequencies suggest the model predicts consonant melodies.

7.2.2 Creativity metrics

The Pitch Variation (PV) and Rhythm Variation (RV) values denote the average variation ratio of generated sequences. The higher the ratio the more unique notes

are in the sequence on average. The Rote Memorization's RMx values denote how many of the subsequences of length x are also contained in the training data corpus. The RM values are lower for the models including durations because in this case two subsequences with the same pitches and different durations are regarded as different subsequences.

Dataset	Model	PV	RV	RM3	RM4	RM5	RM6
Beatles	corpus	0.574	0.404	-	-	-	-
Beatles	pitch-only	0.247	0.111	0.914	0.762	0.638	0.532
Beatles	pitch-duration	0.35	0.124	0.832	0.521	0.38	0.288
Beatles	pitch-duration-key	0.292	0.115	0.871	0.605	0.48	0.388
Wikifonia	corpus	0.581	0.367	-	-	-	-
Wikifonia	pitch-only	0.212	0.111	0.995	0.964	0.881	0.75
Wikifonia	pitch-duration	0.243	0.134	0.967	0.851	0.689	0.513
Wikifonia	pitch-duration-key	0.249	0.135	0.967	0.836	0.652	0.459

Table 7.3: Scores for each model on the creativity metrics: Pitch Variation (PV), Rhythm Variation (RV) and Rote Memorization (RMx).

RV values are significantly lower for all models than in the corpus which means the models produce sequences with less unique note durations, than there are in the training corpus. Less variation is an indication for less creative melodies. This can also be a consequence from the imbalance of classes in duration stated earlier. The PV values of all models are also significantly lower for the pitch-duration models, around 20-30% lower than in the corpus. This shows that the predicted melodies are not as diverse as the training corpus. In all models the RM3 value is highest and the trend for all models is that, the longer the subsequence, the less likely it is contained in the corpus. The Beatles models have lower RM values than the Wikifonia models which may be due to the fact that the Wikifonia dataset consists of more data. In general, the RM values seem very high, with all of the Wikifonia models having a RM6 value of over 45%, which means that many created melodies are copied from the dataset. Thus, the models seem to lack creativity.

7.2.3 Mode collapse metrics

The CPR and DPR values in table 7.3 shows the average amount of pitch repetitions within a predicted sequence as defined above. The higher the value, the more pitch repetitions occur in the sequences. High TS values mean many jumps from one note to the next within the melody of over an octave which tends to not sound consonant. Both datasets consist of under 10% CPR while all models score above 30% CPR. The Wikifonia models score over 64% CPR and over 28% DPR. This indicates that pitches are repeated a lot more often in the prediction than they are repeated within

Dataset	Model	CPR	DPR	TS
Beatles	corpus	0.08	0.044	0.107
Beatles	pitch-only	0.548	-	0.03
Beatles	pitch-duration	0.346	0.107	0.055
Beatles	pitch-duration-key	0.454	0.136	0.016
Wikifonia	corpus	0.094	0.05	0.116
Wikifonia	pitch-only	0.754	-	0.011
Wikifonia	pitch-duration	0.657	0.288	0.04
Wikifonia	pitch-duration-key	0.644	0.306	0.03

Table 7.4: Scores for the mode collapse metrics: Consecutive Pitch Repetitions (CPR), Durations of Pitch Repetitions (DPR) and Tone Spans (TS).

the dataset. Melodies with many pitch repetitions tend to be less interesting than melodies with more movement. All models score low TS values of under 6%. Hence, the predicted melodies consist mostly of small intervals. Small intervals in melodies is named *conjunct melodic motion* in [Tym10, Ch. 1] and is characterized as an important feature of tonal music. Thus, a small TS value can be seen as a requirement of a consonant melody. However, as some small intervals are still very dissonant, for example the tritone, a small TS value alone does not imply consonance.

7.2.4 Evaluation results

There are a few findings from the evaluation. First and foremost, the low HC Red notes and high amount of HC Black and Green notes indicates that the models are able to generate mostly consonant melodies within the harmonic context of the chord progression. Moreover, the pitch test accuracy scores show reasonable generalization capacity of the models. The Rote Memorization scores suggest that the models are creating melodies distinct from the dataset to some extent, albeit insufficiently. The Pitch Variation, Consecutive Pitch Repetition and Duration Pitch Repetition scores signify that the generated melodies are not varying as much as the melodies in the dataset. Furthermore, the Rhythm Variation score and the quantitative analysis of the predicted durations imply that the model's capability of generating rhythms are restricted. In Conclusion, the models are capable of learning to generate consonant melodies within a harmonic context but the melodies lack creative qualities.

The pitch-duration-key model is an extension of the pitch-duration model with the added information of key in the harmonic context. The key was inferred with the Krumhansl-Schmuckler key detection algorithm. In the HC metrics, the pitch-duration-key models scored very similar to the pitch-duration models only differing up to 0.6% in HC red. In creativity metrics, the pitch-duration-key models have similar to lower variation scores. In the category DPR, the pitch-duration-key models score worse than the pitch-duration models and in CPR they only score slightly better for the Wikifonia

case by 1.3%. In TS, they score slightly better. In conclusion, there is no strong indication that the added harmonic context of the key improves the pitch-duration model by our metrics. It remains to be investigated whether the model's performance can be improved by a better key-detection algorithm or a dataset containing correct keys. Another possibility is that the model learns the connection between key and melody alone from the melody and does not benefit from extra information. As simpler models are preferred at the same performance, we recommend choosing the pitch-duration model.

These evaluations only scratch the surface. More refined metrics based on these and more advanced concepts of music theory are required to analyze the strengths and weaknesses of melody generation models more precisely.

8 Interactive Melody Generation

The final challenge of this thesis is to deploy the model in a realtime environment. This chapter elaborates on our approach to create an interactive application that generates melodies based on user input. The idea for the application is partially inspired by the "Magenta AI duet" [WA16], which also generates melodies interactive, though without a harmonic context. The application is realized in JavaScript so that it can run on a webpage and is easily accessible and usable via an internet browser. The front-end, as well as most of the back-end of the program, is designed in the CindyJS framework [vGKRGS16], which runs on top of JavaScript. The Machine Learning model is deployed with Tensorflow.js. [1]

The idea for the melody response application stems from the jazz music concept *Trading Fours*. This term indicates a pattern in which two solo instruments alternately play four bars each. This concept is not fixed on four bars, it can also be e.g. one, two or eight bars. The use of melody generating LSTMs for trading fours with the user is also developed in [Fra06]. Our realization of this concept focuses on the connection between melody and harmony. The harmony is created by setting a chord progression that is played in the background in an infinite loop, for a fixed amount of bars. One loop means one iteration through the chord progression. Before being able to play a sound within a loop, an implementation of timing is required.

8.1 Timing

As a basis for any music application, a discretization of time has to be implemented. For this, the *cstick* script is used. The cstick script is a section in the CindyJS script that is evaluated around about every 20 milliseconds depending on the hardware. A variable *timestamp* is updated every time the *cstick* script is evaluated by the following update rule.

$$\text{timestamp} = \texttt{floor}\left(\frac{12 * \texttt{seconds()} * \text{bpm}}{60}\right)$$

Time is evaluated with the CindyJS function `seconds()` which is based on the JavaScript function `Date.now()` and returns the current time to millisecond accuracy. The variable *bpm* stands for beats per minute, e.g. 120. The function `floor`

[1] An implementation of the interactive web application can be found at `https://pdywilson.github.io/jambuddy.html`.

rounds the input down to the next whole number. This results in 12 timestamps per beat and with four beats per bar it results in 48 timestamps per bar, as in the note-duration encoding. Based on the timestamp variable all timed events can be triggered, such as playing a note, starting a new loop and capturing the input of the user for the inputmelody. Events need only to be stored with their own timestamp that marks at what time the event should be triggered. If the global timestamp passes that value, the event is triggered. These events will be explained after a brief overview of the application's front-end.

8.2 Front-end

The program starts by clicking the play button. This initiates the background music loop that is only stopped, if the user clicks the pause button or leaves the application. As seen in Figure 8.1, the user interface displays the current bar and beat similar to other music software. Depending on the setting, the beat counts from 1 through 3 or 1 through 4 per bar representing 3/4 or 4/4 time signature. The bar counts 1 through 1, 2 or 4 depending on the user's setting. In each bar one chord is played in the background with a piano sound. Chords are chosen by the user out of all 24 major and minor chords. With the settings chosen in the top bar depicted in Figure 8.1, the program plays four bars with one chord per bar in the background.

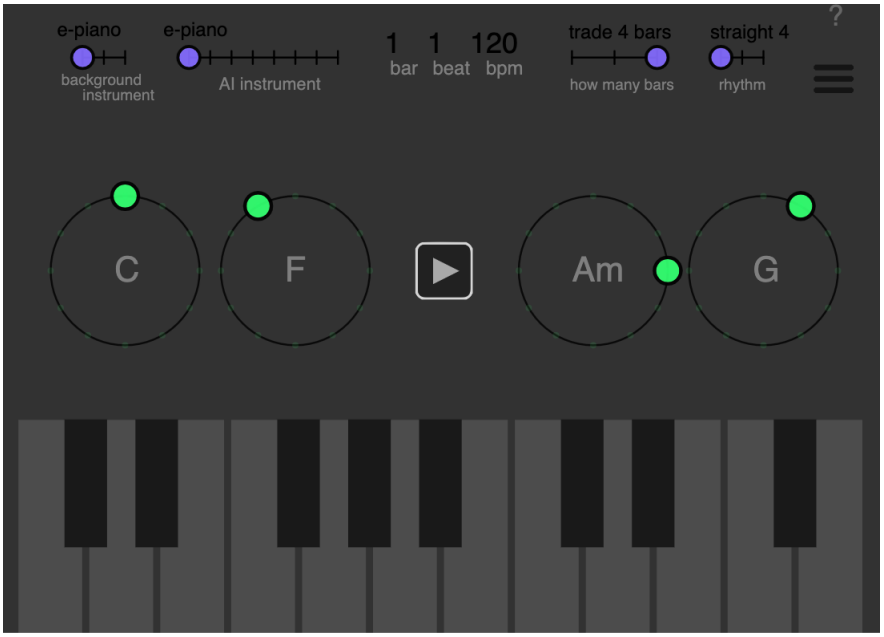


Figure 8.1: View of the user interface of the Jam Buddy. The Chords can be adjusted by rotation the green dots and clicking the chord name toggles between major and minor chords.

In the bottom of the interface, a piano is plotted that can be either clicked with the mouse or played with corresponding keys on the computer keyboard. This way, the user can play notes that make up a melody together with a harmonic context created by the chords played in the background. These chords can also be adjusted with the four circles that are plotted in the middle of the user interface. The User Input is played back by triggering the `playsin` function in CindyJS which creates a sound based on a sine wave. For the background chords and the AI generated melody, the soundfonts [Gle] and [sal] were used. The top bar of the user interface enables the user to choose between different background instrument tones and response melody tones. The number of bars that are traded and the rhythm in which the background chords are played is also set in the top bar. Whenever the user inputs a melody, the program responds with a melody in the next iteration of the background music loop. The calculation of this response melody will be described in the next section.

8.3 Back-end

We will discuss two variations of using the melody generation models in a realtime application. The basic concept is to collect notes as an inputmelody during one loop and playing the response melody in the following loop. A first algorithm deploys the pitch-duration model and a second algorithm utilizes the pitch-only model. To explain the back-end, we introduce the following terminology:

- An `Event` is triggered by an action from the user or by a `timestamp`.
- A `noteEvent` is triggered, if the user activates a note through the UI. The `note` stores the pitch of the played note so that it can be appended to the `melody` list.
- A `nextLoopEvent` is triggered, when the global timestamp reaches a new loop of the background music sequence.
- The boolean variable `firstNoteEventInLoop` evaluates to `true`, if `noteEvent` is the first note event in the current loop and to false otherwise.
- The boolean variable `melodyTooShort` evaluates to true if the melody's duration adds up to the length of one loop or more and to else otherwise.

The following algorithm for computing the response melody collects all of the user input notes as an inputmelody and calculates a response melody at each `nextLoopEvent`. Note that the length of a note in the inputmelody is defined as the difference in time-steps between two consecutive notes. One time-step is defined as 1/48th of a whole note to match the note-duration encoding. The length of the final note of the loop is set to be the difference between the final time-step in the loop and the note's trigger event. The function `getNoteDurationEncoding` is used to combine the user

inputmelody values with the timestamps for the note-duration encoding of the melody. The function `predictNext` triggers the pitch-duration model to predict the next note from the melody. It internally stores the prediction by adding it to the inputmelody so that it is ready to predict the next note again. The function `play` sends the predicted note to the MIDI sequencer so that it triggered at the right times in the loop.

Algorithm 3: Predicting with the pitch-duration model.

```
melody = []
timestamps = []
while true :
    if noteEvent :
        melody.append(note)
        timestamps.append(timestamp)
    if nextLoopEvent :
        timestamps.append(timestamp)
        inputmelody = getNoteDurationEncoding(melody,timestamps)
        while melodyTooShort :
            play(predictNext(inputmelody))
        melody = []
        timestamps = []
```

Algorithm 3 predicts a melody at the start of each loop and then plays it. The length of the melody is tailored to the length of the loop, by producing response notes until the melody is long enough, i.e. fills the loop. The last note is cut to the right length.

Ideally, the computation time would be minimal and the response melody could be played instantly. Unfortunately, a forward pass with the model in Tensorflow.js can take around 30 milliseconds depending on the hardware. Each forward pass generates a new note with pitch and duration that can be queued to be played. The generated note is further appended to the inputmelody for the next forward pass. Thus, multiple separate forward passes are needed to compute the complete response melody.

The delay that is introduced by the prediction is an issue, because it means we cannot play a note on the first beat. The best approach to overcome this computation time issue is to use a faster implementation of the neural network forward pass. Assuming we cannot decrease computation time, one approach to overcome this problem is to play only the last part of the response melody after the computation by bypassing the notes from the response melody that were missed during the computation delay. This way, the beginning of each loop is silent and the melody response kicks in after the computation is done. This is not a great solution because the first part of the melody is left out.

Another approach would be to calculate the response melody before the next loop

starts by starting to calculate a response before the earlier loop has finished. This puts inconsistencies into the melody though. If the user still inputs notes after the prediction has begun, the response melody is only continuing the first part of the user's melody. This is not intended in the design of the model as the model is trained to predict next notes, i.e. continue a melody. However, we can still obtain melodies that are consonant within the harmonic context from this approach because the background melody is still enforces harmony.

Changing the usage of the model in this way introduces new possibilities. We can utilize the pitch-only model for an interesting application in which the duration prediction is replaced by copying the rhythm of the user's input.

Algorithm 4: Predicting with the pitch-only model on each noteEvent.

```
melody = []
predictions = []
timestamps = []
while true :
    if noteEvent :
        timestamps.append(timestamp)
        melody.append(note)
        prediction = predictNext(melody)
        predictions.append(prediction)
    if nextLoopEvent :
        playmelody(predictions,timestamps)
        melody = []
        predictions = []
        timestamps = []
```

Algorithm 4 predicts a next note every time the user inputs a note. During one loop, the predicted notes and the timestamps of each `noteEvent` are stored. The function `playmelody` now triggers MIDI note events at the saved timestamps from the previous loop, so that the predicted notes are played back at the same time, as the user played the notes in the previous loop. A subtlety in this application is that the information about the chord of the following note in the melody. In contrast, the whole melody of the previous loop is passed into `getNoteDurationEncoding` in Algorithm 3. For each note, the chord of the following note is available and the next chord of the final note in the melody can be set as the first chord in the loop. We choose to set the next chord of each note for Algorithm 4 as the following chord in the chord progression.

As stated earlier, the neural network is deterministic, i.e. the prediction is the same for the same input. In the case of Algorithm 4, the first prediction will always be the same if the user starts with the same note. Therefore, the generated melodies can

become repetitive and the application might become dull quite quickly. An adaptation of the algorithm that waits for a few input notes or a few beats before starting to predict, is a good way to improve on variation in the generated melodies.

9 Conclusion and Outlook

This thesis investigated how to model interactive melody generation with Machine Learning methods. Our approach models melodies in the context of harmony in a similar way to the staff notation system that is used by musicians. We discussed the differences between datasets in the MIDI and MusicXML file formats and preprocessed datasets from both formats to represent them in our melody encoding. Melody generation was modelled as a sequence classification task with a classifier based on long short-term memory recurrent neural networks. Different variations of the neural network were trained with the datasets and evaluated with metrics based on music theoretical assumptions for consonant melodies. The best model scored 36% test accuracy for pitch prediction and 59% test accuracy for duration prediction. The evaluation showed that the models were able to capture musical aspects from the data, such as harmonic consistency and melodic stability. However, the evaluation also indicated that the models struggled with creativity. We found that adding tonality information to the harmonic context did not improve performance by our metrics. Finally, we discussed different applications of the neural networks in an interactive realtime environment. Even though the model was designed to be compact, computation time limited the application of the pitch-duration model. Still, a web application producing harmonically consistent melodies could be deployed with the pitch-only model.

There are various further extensions to the musicality of our model that should be investigated. As mentioned in Chapter 2, there are different ways to encode musical data for a Machine Learning approach. Research has to be done to find out which way works best and maybe the ideal encoding has yet to be found. Adding information about the beat-position of the note to the note-duration encoding could improve the model's capability of learning rhythms. A factor that plays an important role in music is dynamics, i.e. loudness between notes. This factor is difficult to incorporate as our datasets contain inconsistent information about it. It would be interesting to see, what a Machine Learning model can learn from dynamics information regarding other aspects of the music. On top of that, our approach only deals with symbolic music data and a lot more information can be found in audio data. Audio data contains a vast amount of information so that filtering out the useful parts is a whole challenge in itself. Additionally, the harmony encoding presented in this thesis only considers the basic music theoretical concepts of major and minor keys and chords. An extension would be to integrate more sophisticated musical concepts from fields such as Jazz music theory. A high quality dataset as well as more detailed metrics for the evaluation

of the model are needed to make the advanced music theory measurable. It remains to be investigated whether our models can learn specific styles of music. For this, more nuanced datasets as well as metrics that can measure attributes of styles are needed.

We believe it is crucial to focus on the music theory based evaluation to improve the quality of melody generation models. More research has to be done to improve the measurability of the quality of generative models in fields such as music or art. On the one hand, the subjective concepts of these fields, such as consonance, need to be quantified precisely, for example by categorizing different types of styles of melodies that groups of people can agree on being consonant or dissonant. On the other hand, theoretical concepts in music theory need to be implemented in a quantifiable manner. Finally, it would be desirable that the research community agrees on a well-defined standard of evaluation of melody generation models so that models are more comparable.

Bibliography

[Aar03] B. J. Aarden. Dynamic Melodic Expectancy, PhD Thesis. 2003.

[BHP17] J. Briot, G. Hadjeres, and F. Pachet. Deep learning techniques for music generation - A survey. *CoRR*, abs/1709.01620, 2017.

[BMEWL11] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[CA10] M. Cuthbert and C. Ariza. Music21: A toolkit for computer-aided musicology and symbolic music data. *Proceedings of the 11th International Society for Music Information Retrieval Conference, ISMIR 2010*, pages 637–642, 01 2010.

[Cc] M. S. Cuthbert and cuthbertLab. Music21 code (excluding content encoded in the corpus) is free and open-source software, licensed under the Lesser GNU Public License (LGPL) or the BSD License. *http://web.mit.edu/music21/*.

[CSG17] F. Colombo, A. Seeholzer, and W. Gerstner. Deep artificial composer: A creative neural network model for automated melody generation. In J. Correia, V. Ciesielski, and A. Liapis, editors, *Computational Intelligence in Music, Sound, Art and Design*, pages 81–96, Cham, 2017. Springer International Publishing.

[DHYY18] H.-W. Dong, W.-Y. Hsiao, L.-C. Yang, and Y.-H. Yang. MuseGAN: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[ERR+17] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, and M. Norouzi. Neural audio synthesis of musical notes with wavenet autoencoders. *CoRR*, abs/1704.01279, 2017.

[ES02] D. Eck and J. Schmidhuber. A First Look at Music Composition using LSTM Recurrent Neural Network. 2002.

[Fra06] J. A. Franklin. Recurrent neural networks for music computation. *INFORMS J. on Computing*, 18(3):321–338, January 2006.

[GB10] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. *MIT Press*, 2016. `http://www.deeplearningbook.org`.

[Gle] Gleitz. FluidR3_GM Soundfonts released under CC BY 3.0 US.

[GPR16] F. Ghedini, F. Pachet, and P. Roy. Creating Music and Texts with Flow Machines. pages 325–343, 07 2016.

[Gra12] A. Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*. 2012.

[Gra13] A. Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[GSC99] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *IET*, 1999.

[GTK10] J. Gillick, K. Tang, and R. M. Keller. Machine Learning of Jazz Grammars. *Computer Music Journal*, 34(3):56–66, 2010.

[HN18] G. Hadjeres and F. Nielsen. Anticipation-rnn: enforcing unary constraints in sequence generation, with application to interactive music generation. *Neural Computing and Applications*, Nov 2018.

[HS97] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation, MIT Press*, 9(8):1735–1780, 1997.

[HSR+19] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck. Enabling factorized piano music modeling and generation with the MAESTRO dataset. In *International Conference on Learning Representations*, 2019.

[Jun01] A. Jungbluth. Jazz-Harmonielehre. *Schott*, 2001.

[KK82] C. L. Krumhansl and E. J. Kessler. Tracing the dynamic changes in perceived tonal organization in a spatial representation of musical keys. *Psychological review, American Psychological Association*, 89(4):334, 1982.

[Kru90] C. L. Krumhansl. Cognitive foundations of musical pitch. *Oxford University Press*, 17, 1990.

[Lac16] K. Lackner. Composing a melody with long-short term memory (LSTM) Recurrent Neural Networks. *Bachelor's Thesis TUM*, 2016.

[LM42] A. Lovelace and L. Menabrea. Sketch of the Analytical Engine invented by Charles Babbage Esq. *Scientific Memoirs. Richard Taylor: 694*, 1842.

[LTN18] P. Leal-Taixé and P. Niessner. Introduction to Deep Learning Course, TUM. 2018.

[Mit97] T. M. Mitchell. Machine Learning. *McGraw-Hill, Inc.*, 1997.

[Moz91] M. C. Mozer. Neural network music composition and the induction of multiscale temporal structure. In W. Brauer and D. Hernández, editors, *Verteilte Künstliche Intelligenz und kooperatives Arbeiten*, pages 448–458, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[Ola] C. Olah. Understanding LSTM Networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/` , date accessed: 1.9.2019.

[Pac10] F. Pachet. The continuator: Musical interaction with style. *Journal of New Music Research*, 32:333–341, 08 2010.

[Raf] C. Raffel. The Lakh MIDI Dataset v0.1. `https://colinraffel.com/projects/lmd/`, date acessed: 2.7.2019, CC-BY 4.0 license.

[Raf16] C. Raffel. *Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching*. PhD thesis, Columbia University, 2016.

[RE16] C. Raffel and D. P. Ellis. Extracting Ground-Truth Information from MIDI Files: A MIDIfesto. In *ISMIR*, pages 796–802, 2016.

[RHW+] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

[RV14] J. D. F. Rodriguez and F. J. Vico. AI methods in algorithmic composition: A comprehensive survey. *CoRR*, abs/1402.0585, 2014.

[sal] Salamander C5 Light Soundfonts. CC-A-3.0 Licence. `https://rytmenpinne.wordpress.com/sounds-and-such/salamander-grandpiano/`, date accessed: 20.7.2019.

[Sch11] P. Schmeling. Berklee Music Theory. *Berklee Press*, 2011.

[SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[TK18] N. Trieu and R. Keller. JazzGAN: Improvising with Generative Adversarial Networks. 08 2018.

[Tym10] D. Tymoczko. A geometry of music: Harmony and counterpoint in the extended common practice. *Oxford University Press*, 2010.

[vGKRGS16] M. von Gagern, U. Kortenkamp, J. Richter-Gebert, and M. Strobel. CindyJS. *International Congress on Mathematical Software. Springer, pp. 319–326.*, 2016.

[WA16] D. R. A. Waite, E.; Eck and D. Abolafia. Project magenta: Generating long-term structure in songs and stories. 2016. `https://experiments.withgoogle.com/ai-duet`.

[Wik08] WikimediaCommons. Circle of fifths deluxe 4.svg. 2008. `https://commons.wikimedia.org/wiki/File:Circle_of_fifths_deluxe_4.svg`, licensed under GNU Free Documentation License v1.2 and CC BY-SA 3.0, date accessed: 1.8.2019.

[WZ95] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications, Psychology Press*, 433, 1995.