Generating Live Interactive Music Accompaniment Using Machine Learning

A model for polyphonic multi-track music generation for practicing improvisation

Benjamin Kløw Askedalen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
The Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Generating Live Interactive Music Accompaniment Using Machine Learning

A model for polyphonic multi-track music generation for practicing improvisation

Benjamin Kløw Askedalen

Abstract

Practicing musical instruments can be experienced as repetitive and boring and is often a major barrier for people to start playing music. With the addition of digital tools for the composition, production, practice and sharing of music, it has become much more accessible, and with the rapid advances in machine learning technology, it is natural that these techniques are also introduced to musical tools. This project had the goal of creating the basis for an application that can help musicians practice improvisation and musical interplay by generating live interactive musical accompaniment to a human player.

A deep learning model was developed, which uses two Long Short-Term Memory (LSTM) networks to generate polyphonic accompaniment for several instruments to one input melody. This compound model consists of one network trained to generate a fitting chord progression to the melody, and one network that uses the chord progression along with the melody to generate polyphonic music.

The model was tested at a very low tempo with live music input and showed clear signs of adapting to what the user was playing. The implemented model was not fast enough to test the application at full speed, so musical analysis was performed on samples of accompaniment to static melodies, generated by the model. The music generated by the model was somewhat monotonous, likely due to data imbalance issues, but some interesting passages generated by the model are described in this thesis.

Additionally, a baseline LSTM model was used to determine whether the proposed solution was better at generating music than a single, straight-forward LSTM. The models performed similarly when evaluated objectively through model accuracy, but through musical analysis it was concluded that the compound model generated more meaningful and functional music.

Contents

List of Figures

List of Tables

Preface

This project was carried out as the final part of my master's degree in Informatics at the University of Oslo. The project idea was my own and appeared mainly out of a lifelong passion for music. Combining two of my biggest interests, music and software development, has made the project very motivating, and it has been intriguing to learn about music technology and machine learning.

First and foremost, I would like to thank my supervisor, Carsten Griwodz, for believing in my project idea and helping me realize it by continuously following up my progress and arranging outside assistance when necessary.

I also want to thank Stefano Fasciani from the Department of Musicology at UiO for giving me an introduction to machine learning for music and guiding me to many resources including books, open source projects and datasets. This really helped me get started with the project.

Additionally, I would like to thank my fiancé, family and friends for their continued love and support, and for always believing in me.

Benjamin Kløw Askedalen
Oslo, May 2022

Chapter 1

Introduction

1.1 Motivation

Music has for thousands of years been an important method of communication and expression in nearly all human cultures. Music is used to describe, document, and share the human experience and imagination, and has become exceedingly complex to the point where music is both an art and a science. During the last 70 years, digital tools have been an increasingly large part of music production and sharing. Technological advances have helped us create many new instruments with which to express ourselves, and tools to help us create and share musical ideas. With the large interest and advancements in artificial intelligence and machine learning techniques during this time, it is only natural that these techniques are being introduced to music. So far, there have been a few experiments using machine learning to generate music and aid in music composition or create entire musical pieces, but there has been little focus on using machine learning technology as a tool for musicians to practice their craft.

Learning to play or improving one's skill on an instrument can be experienced as static, repetitive, and tedious for many people, and is often the main barrier stopping people from becoming proficient at an instrument. It is important to practice scales and technique, but for most musicians it is equally as important to practice listening and playing along with other musicians. In styles like jazz and blues, there is much emphasis on the musical interplay and improvisational skills of the performers. To practice these skills, musicians need to be able to play together with others.

The process of learning any art is very individual. Many digital tools exists to help with this, especially in music. There are many digital resources to learn music theory and playing instruments, including online courses, games that include or simulate playing an instrument, and interactive music creation tools. There are, however, few digital tools that focus on interplay and improvisation. As there seems to be no formal definition of what musical interplay is, we choose to define it broadly as the process of performing music with an ensemble, and the processes and communication that enable it.

Improvisation in music is a technique in which one or more musicians play freely without composing the music in advance. Improvisation can occur in several forms. Some improvisation is completely free, with a whole ensemble listening to each other and complementing each other freely, but most improvisation consists of one musician playing over a previously agreed chord sequence and rhythm. In such improvisation, the musician usually plays notes and sequences that fit the current key and chords, but they are also free to experiment and use dissonance to their advantage in order to create tension or express specific attitudes. This method of improvisation can be very difficult to learn and requires years of dedicated practice. One must be aware of the underlying music theory as well as have a broad vocabulary of motives and sequences and an understanding of how to utilize them properly. One of the most common ways to practice improvisation today is to play over recordings of chord sequences, practicing scales and improvising melodies. However, a recording is completely static and does not react to the player in any way, which makes the situation unrealistic, and does not allow the player any freedom to deviate from the recording or gain an understanding of how interplay feels. The best way to practice improvisation is to have a band one can play with, in which musicians interact and listen to each other.

This project was undertaken to create the basis for an application that musicians can use to practice musical interplay and improvisation without the need of a full band. While such an application could never replace the experience of playing with other humans, it could still be an effective tool to practice these skills. This idea is partially inspired by similar applications such as Band-in-a-box, Jammer Professional and the Rocksmith 2014 Session Mode, which allow the user to play freely while the program adapts in some way to what the user is playing. These tools are mostly based on static musical rules, and not actual music, which makes playing with them over time boring and repetitive. This project attempts to create the basis for a similar experience using machine learning to create more diverse and interesting sequences and adapt better to what the user is playing.

1.1.1 Use case

To explain the goals of this project and what the intended use of this solution is, we have written a simple use case that explains how this solution would be used in a practical application. This use case describes how "Steve", a 23-year-old synth player, would use this solution to practice his improvisational skills.

Steve is at home and wants to practice his improvisational skills. He does not currently play in a band, and he is tired of playing the same songs over again. Steve decides to play with the virtual band application. He opens the application and is met with a starting screen similar to a video game. Steve has the option to adjust the synth sounds of the playback instruments and his own playing as he wishes, and he can use custom synths and instruments in the application. Before he can start a jam session, Steve must decide some initial parameters like tempo, time signature and key. He enters the jamming screen, which contains informational fields like what chord is currently playing, what chord comes next, a visual metronome, and some visual representation of what he is playing. Until Steve starts playing, all the fields are empty. Steve starts playing a few notes and finds a sequence that he repeats for a while. The program slowly starts to accompany him with some piano, and eventually adds more instruments as it gets more confident. The display shows what chord the virtual band is playing, and what chord comes next. If Steve does not like the next chord, he can play something else to make the network change its mind and play something else.

After about 8-16 bars, the program has settled on a chord sequence that repeats with minor variations, and Steve starts to experiment more with what he is playing. If he wants to change the chords being played, he changes his playing to lead to different chords. He starts to improvise over the chord sequence, and the program adapts to his playing both dynamically (loudness/feel) and tonally. Steve continues like this for about 10 minutes, occasionally changing what he is playing, making the virtual band follow him. After a while, Steve wants to end the session, and starts to reduce the amount he plays, signalling to the program that he wants the music to come to a stop. Steve and the application eventually finish the playing.

Steve liked a lot of the ideas in the music he just played and wants to save it so he can listen to it later. After the session, he gets the option to save the session, storing the entire jam session as a MIDI file. Now Steve can import what he and the virtual band played to other music software, where he can listen, edit, and use the music in other compositions.

1.1.2 Research question

This thesis explores the use of machine learning techniques to generate live, interactive music, with the goal of creating the basis for a solution that can be used by musicians to practice improvisation and interplay. These are important abilities that can be difficult to practice, either requiring other musicians to play with, or playing over static recordings many times. This is challenging for many musicians because it might be difficult to find others to play with, and playing the same songs gets very tedious over time.

To help approach this subject in a structured manner, this thesis asks the following research question:

Is it possible to create a solution for generating live, interactive musical accompaniment using machine learning techniques, that can be used by musicians to practice improvisation and interplay?

1.2 Earlier work

The area of interactive music generation with machine learning is lacking, and at the time of writing, only a few articles have been published that have looked into this. Jiang et al. [20] explored this topic using Reinforcement Learning (RL) techniques to generate live musical accompaniment and showed that samples of classical music accompaniment generated by their model were somewhat preferred to samples generated by a baseline supervised Recurrent Neural Network (RNN) model. Benetatos et al. [2] implemented and tested an RNN model that generates a counterpoint melody to what the user is playing in real time. Video demonstrations show that the model is able to quickly adapt to what is being played and plays suitable notes that fit in the context of western classical music.

Garoufis et al. [15] described an interactive system where the user gets several choices for what chord should be played next and selects one of them using an Augmented Reality controller, thus deciding the direction of the chord progression. This system uses a Long Short-Term Memory (LSTM) network trained on chord progressions of hit songs. Through qualitative testing, they found that the chosen architecture provided "relatively satisfying" results, however, the network had issues with long-term coherence.

Generating chord progressions or harmonies for static melodies has been attempted with several different techniques, with varying degrees of success [14, 24, 25]. The most successful method of achieving this using machine learning techniques seems to be Bidirectional Long Short-Term Memory (BLSTM) networks, with which Lim et al. [25] and Faitas et al. [14] achieved better results than other models, both quantitatively and qualitatively.

A BLSTM network uses both past and future timesteps by analysing the sequence from both ends simultaneously. In music, the notes or chords being played often depend on both the previous and next notes. Compared to uni-directional LSTMs, Faitas et al. [14] experienced that the chord sequences generated by BLSTM were rated to be much more likely to be created by a human composer by human subjects. However, it is not possible to use BLSTM in live music generation, because the input sequence is created live.

There have also been attempts at generating music using Generative Adversarial Networks (GANs). Dong et al. [13] designed and tested three different methods of music generation using GANs, and found that they had "desirable properties".

Apart from generating accompaniment or chord sequences for a given melody, there are also projects such as the JamBot by Brunner et al. [5], which tries to generate meaningful musical sequences from scratch by selecting randomly from a probability distribution. They use an alternative network structure that separates chord and music generation. By using one network for generating a chord sequence, and one for generating polyphonic accompaniment for that chord sequence, the model is able to generate music with long-term structures.

The model proposed in this thesis is based on the JamBot model described by Brunner et al. [5]. Due to the model using regular LSTM networks, it was possible to adapt it to working with live sequential input, instead of creating an entirely novel approach from scratch. While JamBot is designed to generate single-track polyphonic musical pieces, our model generates multi-track accompaniment to a lead melody, so it had to be adapted to take this melody as input and output several instrument tracks.

1.3 Research method

This project has been performed using the design paradigm, which as described by Comer et al. [11] consists of iterations of requirement specification, design, implementation, and testing. This is a highly practical approach that uses experimentation through trial and error to optimize a solution. By applying this method, we were able to focus on the practical implementation and use of the solution rather than the mathematics and specifics of the machine learning techniques that were applied.

1.4 Thesis outline

This thesis is divided into eight chapters.

Chapter 2 contains background information about the machine learning techniques, tools, and data formats used and discussed in this thesis and related work.

Chapter 3 provides a simplified introduction to western music theory, and contains explanations of musical terms used throughout this thesis.

Chapter 4 describes the solution designed and implemented during this project.

Chapter 5 describes the implementation process including how we arrived at the proposed solution.

Chapter 6 contains objective and subjective evaluations of the music generated by the proposed solution and the simpler baseline model.

Chapter 7 discusses the results from the previous chapter, as well as the challenges faced in this project and in machine learning research in general. This chapter also looks into the philosophy of machine generated art.

Chapter 8 summarizes the findings in this thesis and tries to answer the research question, and suggests future work based on the work performed in this project.

Chapter 2

Background

This chapter includes necessary background information about machine learning techniques, tools, and data formats, and serves as a reference point for expressions and terms used throughout the thesis.

2.1 Machine learning

Machine learning (ML) is a set of data analysis techniques that use real data to automatically improve the accuracy of their output compared to some ground truth. Machine learning techniques are applied to a vast number of fields, including business management, autonomously driving cars, speech recognition and machine translation, among others. Machine learning encompasses many different techniques used for different purposes. This chapter will serve as a basic introduction to machine learning, and explains concepts used in this thesis, mostly focusing on the algorithmic and practical perspective. If the reader wishes to get a greater understanding of machine learning algorithms and their use, we recommend reading Marsland's book about the topic [26], on which most of the explanations in this section are based.

Machine learning techniques are generally divided into three different categories: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning techniques depend on learning from labelled data to create a model for meaningful predictions and classifications. Unsupervised learning techniques are static analysis techniques mostly used for classifying or finding connections in the data and have very limited use in music applications. Reinforcement learning techniques use agents that learn how to correctly interact with their environment by trial and error. Although there have been attempts at generating music using reinforcement learning techniques [4, 20], it seems like most research done in the field of machine generated music has used supervised techniques. Because of this, supervised techniques will be the primary focus of this thesis.

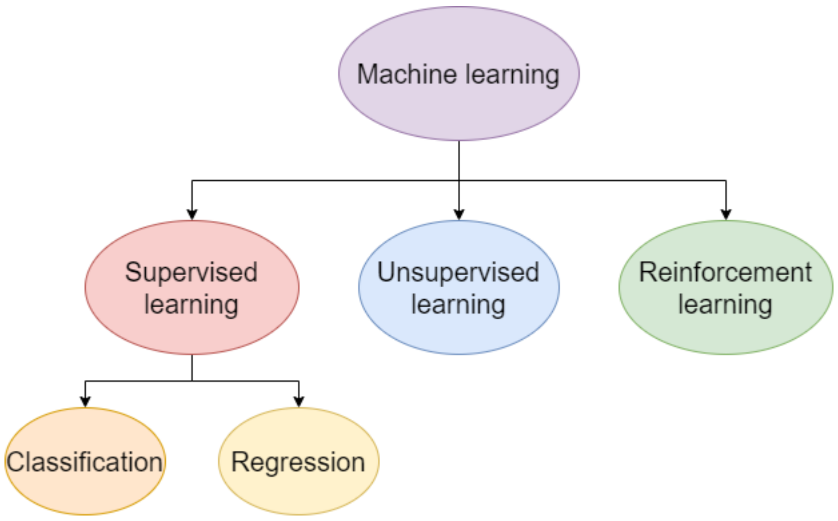


Figure 2.1: Hierarchy of machine learning

2.2 Supervised learning

Supervised learning consists of algorithms that use already existing, labelled data to train a model to make accurate predictions for unlabelled data. The term "prediction" in the context of machine learning is usually used to describe the output of a model. The prediction does not have to be about the future. Supervised learning techniques can be applied to a great number of fields and can be very useful in practical settings. Most supervised learning techniques are very complicated, and it is usually impractical to try to fully comprehend the connections made by the model in order to predict accurately. Because of this, many of the algorithms work as a sort of "black box" where it is either impossible or impractical for humans to understand what happens internally.

All supervised learning techniques apply a process of trial and correction, where the model gets some input, generates some output, calculates the error, and adjusts its calculations accordingly. These calculations are adjusted in small increments, so the model is generalized for different input values. The goal is that the model should predict accurately not only for the dataset it is being trained on, but also real unlabelled data examples.

Supervised learning problems are generally split into two categories: regression and classification. Some algorithms are mostly suitable for one of these categories, and some algorithms are frequently used for both. The algorithms used in this project are based on neural networks, which can be used for both regression and classification problems.

2.2.1 Regression

Regression problems involve finding connections between attributes to predict a numerical target attribute. The input of a regression model is usually a number of numerical attributes, and the model is gradually adjusted with the goal of finding a generalized function that can predict one numerical attribute as accurately as possible based on the input values. Regression is used to predict continuous attributes like temperature, price, or sales. Regression is ultimately about finding the dependencies between one dependent variable (the output variable), and a set of independent variables.

Typical algorithms used for regression problems include linear regression and polynomial regression. Linear regression tries to find a straight line that fits the training data as well as possible. A linear regression model can be represented by a simple mathematical formula in the format $y = ax + b$. Polynomial regression tries to find the polynomial function that fits the data best. This usually involves testing different degrees of polynomials to find the one best suited to the data. More complex problems that cannot be represented by a function can use for example neural networks to predict a numerical attribute.

Regression is used for problems like weather forecasts, stock market prediction, house price prediction etc., where the goal is to predict a numerical attribute like rainfall, wind speed, stock value or price.

2.2.2 Classification

Classification problems are based on sorting data entries into classes based on contextual attributes. A classification model takes a number of attributes as input, and outputs one or more categorical values. There are three main types of classification: binary classification, multi-class classification and multi-label classification. Binary classification problems contain two possible classes, and the output of the model will either be 0 or 1. This is for example used for spam email detection. Multi-class classification problems contain several mutually exclusive classes, where the model predicts a probability value for each class, adding up to 1. We can either select the class with the highest value or do a probability selection. Multi-class classification is used for problems like image classification and other problems where an entry can only belong to one of several classes. Multi-label classification problems, contrary to the other two types, contain several classes that are not mutually exclusive. Each entry can be assigned to several classes/labels, and each class is treated as a separate binary classification problem. Usually, all the classes that have a value over a certain threshold are selected. This can for example be used for object recognition in images where more than one object may appear.

Algorithms normally used for classification problems include logistic regression, K-nearest neighbours, neural networks, and others. These use different methods to assign the inputs into classes based on one or more attributes. The solution presented in this thesis uses neural networks for multi-class and multi-label classification.

2.2.3 Challenges with supervised techniques

The main challenges with supervised learning are what is referred to as overfitting and underfitting. Overfitting is when a model makes better

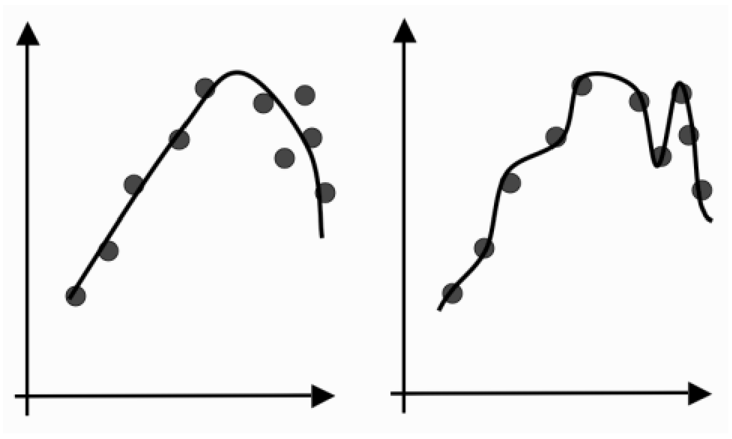


Figure 2.2: The effect of generalizing (left) and overfitting (right). Source: [26, pp. 20]

predictions on the training data than on the real data. This happens when the model is trained for too long and its predictions are no longer generalized but specialized to the training data. Underfitting occurs when the model is not able to make accurate predictions. This is usually because of a lack of data or too short training period, but it might also occur if the algorithm is not fit for the problem.

In order to combat the challenges of over- and underfitting it is normal to split the dataset into three parts: training, validation, and testing. The training data is the dataset that is used to train the model and is usually the largest. The validation data is used during training to see whether the model is overfitting to the training data. If the training accuracy keeps increasing, but the validation accuracy does not, the model is overfitting. While the training and validation datasets are used during model design and parameter optimization, the testing data should not be used until the final model is finished, to check that the model is not specifically adjusted to the validation data.

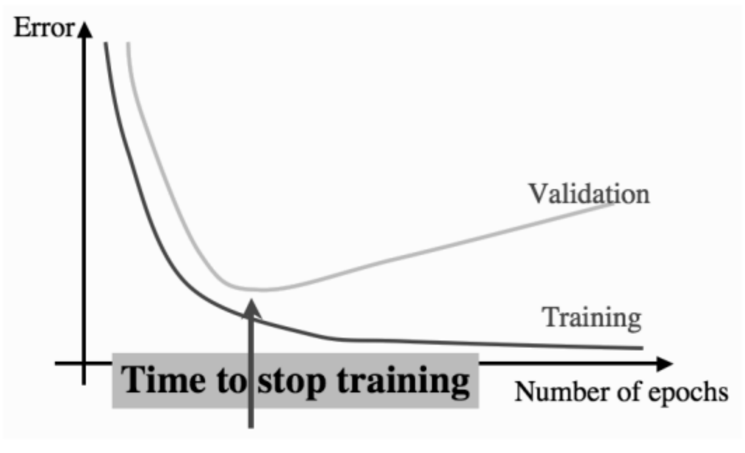


Figure 2.3: The graph shows the model overfitting over time. Source: [26, pp. 88]

2.2.4 Neural networks

The algorithms mostly used for music generation are based on neural networks, which is a general-purpose method of replicating connections observed in real data. Neural networks are based loosely on how the human brain learns, by training neurons to make specific calculations based on what it has previously observed. Obviously, the human brain is much more complicated than it is possible to replicate in a modern computer, but if we simplify this principle, it can be used to make meaningful predictions based on experience from real data.

The simplest neural network is called the perceptron, which was invented in 1957 by the psychologist Frank Rosenblatt [36]. Perceptrons can be explained as a graph of input and output nodes, where all input nodes are connected to all the output nodes. Each output has an activation function, which is the same for all inputs, and a weight associated with each input. These weights are tuned through training on real data. For each iteration of training, the input values are run through the network, the activations are calculated, and the resulting outputs are compared with the expected outputs. Usually, each output node is either fired or not fired depending on its activation function.

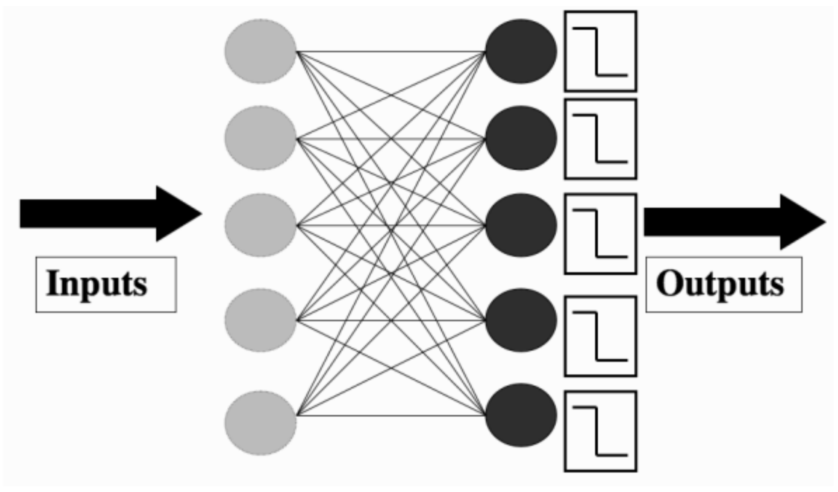


Figure 2.4: The perceptron network with input and output nodes. Source: [26, pp. 44]

If a node fires when it is not supposed to, or does not fire when it is supposed to, the weights related to that node are changed. How much the weights are changed depends on the learning rate. If the weights are changed too much, it might skip over the optimal weight, and if the weights are changed too little, the model might get stuck during training in what is called a "local optimum", where small changes made to the weights only make the model worse, but a larger change could get better results. When the model is stuck in a local optimum, the adjustments made to the weights are so small it never explores outside the bounds of the local values, making it unable to find more optimal weights.

While the perceptron can be effective for simple, linear problems, it has difficulties tackling more complex problems that are not linearly separable. Eventually, there were attempts to include a third layer, the "hidden layer", to the perceptron to help it learn more advanced connections in data. However, there was no way to calculate the error of the hidden weights separately from the output weights. If we just calculate the error at the end, it is not possible to tell whether the hidden layer or the output layer needs to adjust their weights, or how much. In 1986, Rumelhart, Hinton and McCelland [37] proposed a method called backpropagation to solve this issue, and introduced the multi-layer perceptron, which is the basis of most neural networks today.

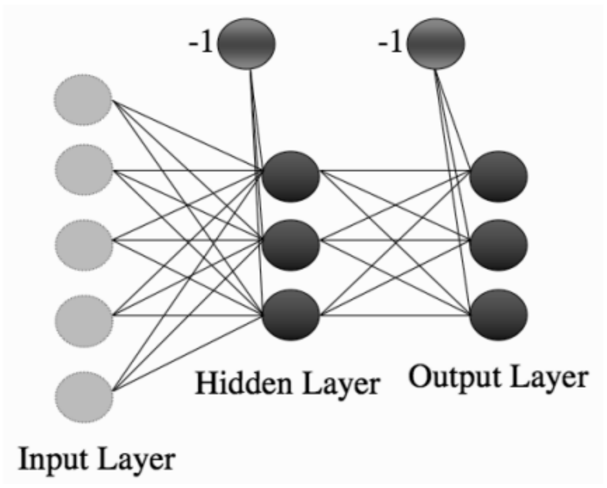


Figure 2.5: The multi-layer perceptron with one hidden layer. Source: [26, pp. 72]

While machine learning encompasses many different techniques, the most interesting techniques for generating music are currently in the field of deep learning [4]. Deep learning is a subset of machine learning techniques that are mainly based on multi-layered artificial neural networks, including Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), Convolutional Neural Networks (CNNs) and other architectures. The most used architecture for music generation is RNNs, which are able to process sequential data and learn temporal dependencies.

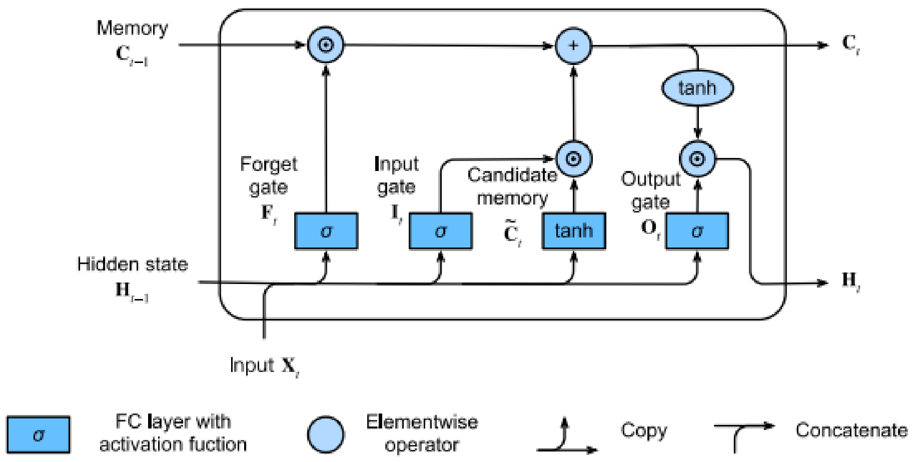
Recurrent Neural Networks

Traditional neural networks use one static data point as input. This is typically a set of attributes with numerical values, or a static image. The data is fed to the network in random order during training, and the model does not consider the relationship between different points. This type of network is not able to process sequential data like music. Recurrent Neural Networks (RNNs) allow the processing of sequences where information from previous time steps is required to make predictions about the current timestep. RNNs utilize a hidden state that is passed recursively in the hidden layer of the network for each time-step of the sequence. This allows the model to take previous data points into account, making it able to analyse and predict sequential data like text, speech, and music.

However, a shortcoming of RNNs is vanishing gradients; information is not able to persist for many time steps, only providing "short-term memory" and not "long-term memory" [18]. This is important in several applications such as text generation and music composition. A pronoun might depend on some context in a previous sentence, and a chord might depend on the entire chord progression and melody leading up to it.

Long Short-Term Memory networks

As a solution to the issue of vanishing gradients, Sepp Hochreiter [18] proposed the Long Short-Term Memory (LSTM) network, which uses more advanced network nodes that selectively forget and pick up data. This approach allows the model to decide what data are important for the rest of the sequence, ignore the rest, and even remember data just until it is needed, and then drop it. This approach and variants of it have proven relatively effective in music generation.



Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$
FC layer with activation fuction
Elementwise operator
Copy
Concatenate

Figure 2.6: LSTM cell. Source: [47, pp. 355]

LSTM cells have three different gates that regulate information flow:

The **forget gate** is used to decide what information should be thrown away and what should be kept. The previous data and the current input are passed through a sigmoid function, giving each number a value between 0 and 1. Values closer to 0 mean that the data should be forgotten, and values closer to 1 mean that it should be kept. These values are multiplied with the cell state.

The **input gate** decides which values should be updated. This is done by passing the previous and current input through a sigmoid function, to decide which values are important, and through a tanh function, which gives them a value between -1 and 1, to regulate the values. These values are then multiplied, so the values with a sigmoid close to 1 are changed a lot more than the values close to 0. These numbers are added to the cell state.

The **output gate** decides what the new hidden state should be. The new cell state is passed through the tanh function and multiplied by the sigmoid of the previous hidden state and the current input. Both the new cell state and the hidden state are passed to the next time step.

Bi-directional LSTM

Bi-directional LSTM is used to improve LSTM performance, adding a second LSTM that takes the reversed sequence as input. This allows the model to use both previous and future sequences to predict a single sequence, which can be very useful in musical chord generation, where a chord often depends on both what comes before and what comes after. Based on the results of Faitas et al. [14] and Lim et al. [25], BLSTM seems to currently be one of the best deep learning techniques for generating music. However, because it generates a full sequence simultaneously, it is not possible to generate live interactive music with BLSTM. BLSTM is dependent on a full sequence that is being analysed from both ends, whereas an interactive model must predict the next chord and accompaniment without knowing how the user is going to continue the sequence, or what chords come after.

Sequence to Sequence Learning

Sequence-to-sequence learning is a technique used to transform a sequence into a different sequence, using an encoder-decoder architecture. This technique is often used for machine translation but can also be used to create chords or harmonic melodies to a melody [14]. Like BLSTM, this technique requires a full sequence to be generated simultaneously and cannot be used to generate live music.

Hidden Markov Model

The Hidden Markov Model is another supervised method that can be used for music generation. It uses the Markov property to predict states based on indirect observations, and as shown by Lim et al. [25] it can be used to generate relatively good chord progressions for melodies. However, it scored much worse than BLSTM in both objective and subjective tests.

Generative Adversarial Networks

Generative Adversarial Networks (GANs) consist of two networks; a *generator* and a *discriminator*, working against each other. The discriminator is trained to tell whether a sample is from the training data or if it is generated by the generator, and the generator is trained to fool the discriminator [16]. To our knowledge, there have currently been no attempts at using GANs for live music generation. This might be because GANs require a lot of data for training, and it is difficult to obtain enough data to train a model to work in a live setting.

Embeddings

Embeddings are low-dimensional representations of high-dimensional vectors, which are used to map the relationships between categories. The technique was originally introduced by Bengio et al. [3]. The most common use for embeddings is in text analysis and generation, where the embedding is trained to map the words in a lower-dimensional space, both to reduce the data size and to group semantically similar words. Brunner et al. [5] used this technique to map the relationships between the chords in their training data, and experienced that their embedding was able to extract important music theoretical concepts from the data.

2.3 Reinforcement learning

Reinforcement learning is a type of machine learning different from both supervised and unsupervised learning. While supervised learning algorithms work on labelled data and unsupervised algorithms work on unlabelled data, reinforcement learning does not use any prior data, but lets an agent train a model through exploring, perceiving, and interpreting its environment, being rewarded/punished for doing things correctly/incorrectly, with the goal of maximizing its reward over time. Reinforcement techniques are usually applied when there is no prior data to train a model with, or when we want to train an optimal model instead of one relying on existing data, such as Google's AlphaZero bot [39].

So far, some of the most practical usage of reinforcement learning is in the field of robotics, where some applications have been successful [22]. There have also been attempts at generating music with reinforcement learning techniques. Jiang et al. [20] argue that supervised RNNs trained to minimize loss suffer from two major issues. The first is that there becomes a mismatch between the conditions in which the models are trained and used. When training, the model uses an entirely human-made sequence, but when being used, the model relies on what the model itself has previously generated, which might cause accumulative deterioration of the music. The second issue is that the way the problem is modelled is not representative of the goal. The models minimize the cross-entropy loss at each step to select the note or chord with the highest probability, while the coherence and function of each step actually relies on the entire piece.

Reinforcement learning techniques could potentially work very well for interactively generating musical accompaniment, however, they are usually much more complex than supervised techniques, and would require more time and experimentation to achieve. Because of this, and the fact that there is very little prior research in the field of RL generated music, this thesis focuses on supervised techniques.

2.4 Machine learning tools

Machine learning techniques, especially deep learning, have until recently been very difficult to implement in real projects. Neural networks can be very complex, and for most projects it is not possible to implement and train a model from scratch in a reasonable amount of time. In order to save time and avoid needing expert knowledge of the mathematics involved, there are several available frameworks made for machine learning. Currently, two of the most popular ML frameworks are TensorFlow [27] and PyTorch [32]. They are both described as great ML frameworks, and each have strengths and weaknesses when compared. TensorFlow is developed by Google, while PyTorch is developed by Meta. Both are Python frameworks. In this project, we ultimately decided to use TensorFlow, mostly because it seemed to have a more comprehensive user base and documentation, and because of the high-level API Keras [7], which allowed us to implement the networks with a higher level of abstraction than using PyTorch. TensorFlow also seems to be much more frequently used for music generation, as nearly all similar applications we have found were implemented using it.

2.4.1 TensorFlow

TensorFlow [27] is a low-level framework for machine learning. It is developed by Google and is currently the most popular ML framework. TensorFlow has a large user base and comprehensive documentation, most likely making it far easier to learn and use than other frameworks. Additionally, it comes with Keras, which is a high-level API meant for rapid development.

Keras

Keras [7] is a high-level API for neural networks and is mainly designed for rapid development. It can handle complex neural networks with its functional API. Keras is an integrated part of TensorFlow but can also be used on top of other frameworks. Keras' simplistic interface allowed us to implement the networks at a much faster pace without having to simplify them, which was particularly important to the success of this project.

2.5 Data formats

There are several ways to represent music digitally. Music is usually stored digitally by converting sound waves to binary numbers. However, this can be very challenging to work with using ML, as the size of the data is large, and audio files contain no musical structure. By representing musical notes in a symbolic format, we can use features in the music, which require a lot less processing power and storage space. Therefore, this project uses MIDI and chord sequences extracted from the MIDI music to represent the music for the models.

2.5.1 MIDI

The Musical Instrument Digital Interface (MIDI) is a standard for digital music communication, mainly intended for synchronizing digital instruments. The standard was first published by Dave Smith and Chet Wood [40] in 1981, and the improved version has since been used to digitally represent music of all genres in many different systems. Everything from synths, digital drum sets, to Digital Audio Workstations (DAWs) use MIDI as a convenient way to represent musical information and to control other devices.

MIDI data contains metadata about the song and instrument tracks, notes, and other events. Each note has a pitch, velocity, start time and end time. The pitch is what note is being played, and velocity describes how hard the key was pressed.

Several other projects have also used MIDI to represent the music for their models. MIDI datasets can be easy to obtain, but they can be very noisy and require pre-processing [13]. However, as MIDI is relatively easy to manipulate, this can be automated. Through the MIDI metadata it is also possible to obtain information about tempo, instrument types and other events.

2.5.2 Chord sequences

Chord sequences only describe each chord in a song and how long it is held. It contains no information about melody, rhythm, a chord's relation to other chords, or even what notes each chord includes. Chord sequences have been used by for example Garoufis et al. [15], who used the Mc-Gill Billboard dataset [6], which contains chord progressions from a number of pop songs, to train their model. Chords usually help define the structure of a song and create the foundation and direction of the music. Usually, chords are predetermined even during improvisation.

2.5.3 Piano rolls

While MIDI is a great format for representing music digitally through instructions, it is not really suitable as raw input to a neural network. Because the music is supposed to be quantized to a certain time grid, the model should output whether each note is being played at each timestep. MIDI does not organize the notes in timesteps but contains timestamps for each note.

The most normal method of representing MIDI data in a machine learning network is the use of the "piano roll" format. This is a 2D matrix representation of the music that shows the velocity of each note for each sample. The number of samples is decided by the sample rate, which is usually given as the number of samples per second or the time between each sample.

A piano roll can easily be converted into a chroma representation, which condenses all the notes into the twelve semitones instead of all the separate notes in a MIDI track. This is useful for visualizing the music and can also be used to reduce the input size of a model. However, it does not contain information about in which octave each note is being played.

The piano roll representation fits well with RNNs because each sample can be used as one timestep, and if the data is converted to binary rather than integer velocity values, the notes are already one-hot encoded. One issue with this format is that it might require a lot of space, especially at higher sample rates. However, our model uses 16th notes as timesteps, so the sample rate is relatively low. This drastically reduces the size of the data, allowing us to load thousands of songs into memory at once.

2.6 Validation techniques

In order to evaluate the performance of the system, it is possible to use objective or subjective measurements. Objective methods can help provide an idea of how well the program works algorithmically, such as evaluating accuracy on a test dataset, doing cross validation, or looking at loss. However, humans' subjective interpretation of music cannot be measured this way. If the numbers tell us that the algorithm generates logical passages, this does not necessarily mean they sound good.

2.6.1 Objective methods

To evaluate the accuracy of the predictions, it is normal to split the data into training, validation, and testing data, with the training data being the largest set. Doing this can help avoid overfitting of the data by stopping training when the accuracy on the training set gets better than the accuracy on the validation set. The test set is used at the very end of the experiment and tells us whether our model is overfit to the training and validation datasets. This is the most common way of partitioning the data for training and evaluation.

The predictions are measured in a confusion matrix, which counts how many true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn) the model predicted. In multi-class and multi-label classification, this is measured for each class or label. From these measures, we can calculate accuracy, precision and recall as follows:

$$accuracy = \frac{tp + tn}{N}$$

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

where N is the number of predictions

Accuracy tells us how many percent of the predictions were correct. This lets us easily see whether the predictions are random guesses, or if the model is actually learning. However, only looking at accuracy can be misleading, and does not tell us anything about what mistakes the model is making.

Precision measures the proportion of positive predictions that were correct. If accuracy is 75%, but there were no false positives, precision will still be 100%. In situations where it is more important to avoid false positives, low accuracy can be accepted if precision is high.

Recall measures the proportion of positives that were correctly predicted. If precision is 100%, recall can still be low if many positives were identified as negatives. Recall is often reduced as precision is increased, because we often have to accept many false negatives in order to avoid false positives.

This method of splitting the data and measuring accuracy was used by Garoufis et al. [15] and Lim et al. [25]. Dong et al. [13] used other calculated measures to evaluate different attributes.

2.6.2 Subjective methods

Objective methods of evaluation can be useful to see whether the model is training properly, but it does not actually say anything about the musical quality. The perception of music is highly subjective, and in order to evaluate the actual musicality of the generated predictions, we need subjective evaluations generated by human testers. In other projects this has been done a number of ways.

Faitas et al. [14] asked participants to evaluate how much they liked the music and how much it harmonized on a scale from 1 to 5, and whether it sounded human-made when they had to choose between two different samples. This form of evaluation separates the quality of the music from how convincing it is as human music.

Dong et al. [13] had participants listen to samples generated by their model and rate the samples' harmony, rhythm, structure, coherence, and overall rating on 5-point Likert scales. They also separated what they called "pro users" and "non-pro users" based on their musical background. Similarly, Garoufis et al. [15] had twelve subjects, both musicians and non-musicians, test the system and rate the musical coherency and variety on 5-point Likert scales.

In this project, it was not possible to perform subjective testing with external people. Subjective evaluations of the generated music were therefore performed through musical analysis. This might help evaluating the functional harmonic and rhythmic aspects of the music, but we recognize that it is challenging to be objective in the analysis when evaluating our own model. Therefore, the results of the musical analysis are only meant for the purpose of explaining to non-musicians why parts of the music generated by the model is interesting, and what issues are observed in the music.

Chapter 3

Music theory

This chapter serves as a simplified introduction to western music theory, which was used in this project to set up the models and analyse the generated music.

3.1 Western music theory

Music is a broad term used to describe sounds organized in a specific way, usually with the goal of invoking some feeling or impression in the listener and/or performer. Music is an art, and thus is not defined by a set of rules by which all music conforms. There are, however, several tools one might employ to better understand, analyse, and create music, with the ultimate goal of understanding how humans react emotionally to music.

Different cultures have since the dawn of human civilization had different approaches to performing, experiencing, and analysing music, with different focuses and purposes. The European approach to music theory, commonly called western music theory, has through the past 200 years become the unofficial international standard for music analysis. Modern western music theory is based on a twelve-tone scale where the distance between consecutive notes is equal, called the twelve-tone equal temperament. This scale was originally based on the harmonic series, in which the frequency of each note is a multiple of the fundamental frequency being played. The twelve-tone equal temperament was normalized in western music during the 18th century and forms the basis of all western musical analysis today. Other cultures might use different tuning systems, such as the modern Arab tone system, which is based on a 24-tone equal temperament. In this thesis, however, the western system is used to explain musical ideas.

To gain a proper understanding of what we mean by the term "music", we might refer to the philosophy of music. Whether or not a machine learning model is able to create musical accompaniment might be very much dependent on how we define what music is. In the Merriam Webster online dictionary, music is defined as "vocal, instrumental, or mechanical sounds having rhythm, melody, or harmony" [29]. This is a very broad definition, which depending on how we define rhythm, melody, and harmony, may allow any collection of sounds to be called music. While this definition may be correct for the everyday use of the word, it is not especially useful when trying to distinguish musical and non-musical noise.

A different way of viewing the problem is by considering intention. Maybe any collection of sounds can be music if it is intended as such. This does however imply that music must be intentional, which is not necessarily the case. This would certainly mean that a machine, which cannot have any intention without having thoughts of its own, could never create music. Then what about automatically generated sounds and patterns already being used in a large portion of modern music? One could argue that there is intention in how automatically generated music is used, and how the algorithms that create the sounds are created, but where would we draw the line? If we created a hyper-intelligent AI that in turn created an AI to generate music, would that be music? A different approach is to consider how the sounds are perceived. If we say that any collection of sounds perceived as music is music, then the definition of music will be very individual. One person may hear music where another hears only noise, and any piece of organized sound can be both music and not music.

For the purpose of this thesis, music will be defined as a set of organized sounds intended to be heard as music and experienced as music by the listener. We might say that while the machine itself does not have any intention, the creator of the program intended the model to generate music, and as long as the listener perceives it as music, it will be classified as such.

3.2 Musical terms

This section contains explanations of a number of musical terms that will be used throughout the thesis. Most of these terms are very basic musical concepts and are fundamental to analysing and describing music.

Note

The term "note" refers to a specific frequency being played for a specific duration.

Tone

"Tone" in the context of western music usually refers to the pitches' placement in the 12-tone equal temperament. The tones are named A to G following the order of the modal A minor scale, and the tones in between (black keys) are either "sharp (#)" or "flat (b)".

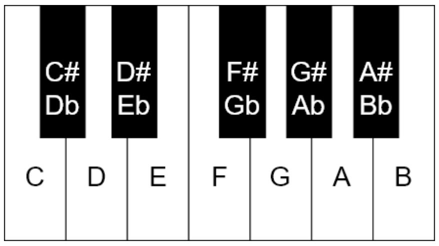


Figure 3.1: Note names on a piano.

Intervals

An interval refers to the distance between two notes being played together. Table 3.1 describes the most used intervals.

Name	Example	Distance
Minor second	C - Db	1
Major second	C - D	2
Minor third	C - Eb	3
Major third	C - E	4
Perfect fourth	C - F	5
Tritone	C - Gb	6
Perfect fifth	C - G	7
Minor sixth	C - Ab	8
Major sixth	C - A	9
Minor seventh	C - Bb	10
Major seventh	C - B	11
Octave	C - C	12

Table 3.1: Tonal interval names

Chord

A chord is a combination of several notes played together, usually in what is known as triad formations, meaning they are built up in thirds. The C major chord, for example, consists of the notes C, E and G. E is one major third up from C, and G is one minor third up from E. Chords come in different types that are built up the same way. C major and G major, for example, both consist of a major third and a minor third.

Different chord types are written differently. Major chords are just written with the name of the base note (C), minor chords are suffixed with "m" (Cm), and chords with more than three notes contain the number of the last step. C7 for example, contains the notes C, E, G and Bb, while C11 contains C, E, G, Bb, D and F. Cmaj7 contains a major seventh interval, so it has B instead of Bb.

Chords have different functions in the scales. The three most important chords in a scale are called the tonic, dominant and subdominant. These are the chords built up of triads from the 1st, 5th and 4th step, respectively. In the C major scale, the tonic is C, the dominant is G, and the subdominant is F. These are the most commonly used chords and are closely connected.

Consonance/dissonance

An interval that sounds/feels good and "right" to humans is called consonant. Typical consonant intervals are the octave, perfect fifth, perfect fourth and major third. In the harmonic series, the respective frequency ratios of these intervals are $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$ and $\frac{4}{5}$. Although these ratios are different in the 12-tone equal temperament system, we still hear them as consonant.

An interval that sounds/feels weird or "bad" to humans is called dissonant. Dissonance is important in music to create and release tension. The interval usually considered the most dissonant is the tritone, which has a ratio of $\frac{45}{32}$.

Harmony

"Harmony" describes how different pitches sound when played together. Harmony plays a very important part in nearly all music and is used to create tension and release by combining consonant and dissonant intervals. Harmony is highly subjective, and changes throughout time and based on cultural background.

Beat

A beat can generally mean two things: (1) a rhythm being played continuously (usually by a drummer), or (2) a single step in the rhythm and tempo being played. Beats are the rhythm being counted. Everything smaller than one beat are subdivisions.

Tempo

The tempo of a piece of music is the number of beats that occur per minute. The most commonly used tempos are in the range of 80 bpm to 140 bpm (beats per minute).

Bar

A bar in music is a short collection of beats making up a unit of time. The number of beats in a bar is decided by the time signature.

Time signature

The time signature is the number of beats per bar, usually denoted as a fraction such as $\frac{4}{4}$, $\frac{3}{4}$ or $\frac{6}{8}$.

Subdivision

A subdivision is a measure of time shorter than a beat, and is created by dividing the beat by a certain number, usually in base 2. In most songs, one beat is what is called a quarter note (because it is one quarter of a bar in $\frac{4}{4}$ time), and can be divided into two eighth-notes, four 16th notes, eight 32nd notes etc. The names of the subdivisions describe how many there are in one $\frac{4}{4}$ bar. It is also possible to divide the beats by other values like three or five, by which we get subdivisions called "triplets" and "quintuplets".

Key signature

The key signature is the starting key or tone of the scale.

Scales and modes

The scale or modality describes the tones that are used in the current music. The simplest scales are the major and minor pentatonic scales, which contain five notes. The steps in these scales equal the black keys on the piano, starting at F# and C#, respectively. Most scales contain 7 notes. The most used scales in western music are the major scale and the natural minor scale. These scales are two of the seven diatonic modes traditionally based on the major scale. Each mode uses the same sequence of notes, but starting at a different step. For simplicity, this is usually explained using the C major scale as starting points, because then all the modes only use the white keys on a keyboard. The first mode is the C major scale, or Ionian. Starting from D and using only the white keys is D Dorian, starting from E is E Phrygian, etc. This way we get the Ionian, Dorian, Phrygian, Lydian, Mixolydian, Aeolian and Locrian modes. The other modes apart from Ionian and Aeolian are rarely used in popular music and are mostly utilized in jazz improvisation. In this project, five of the most common scales were

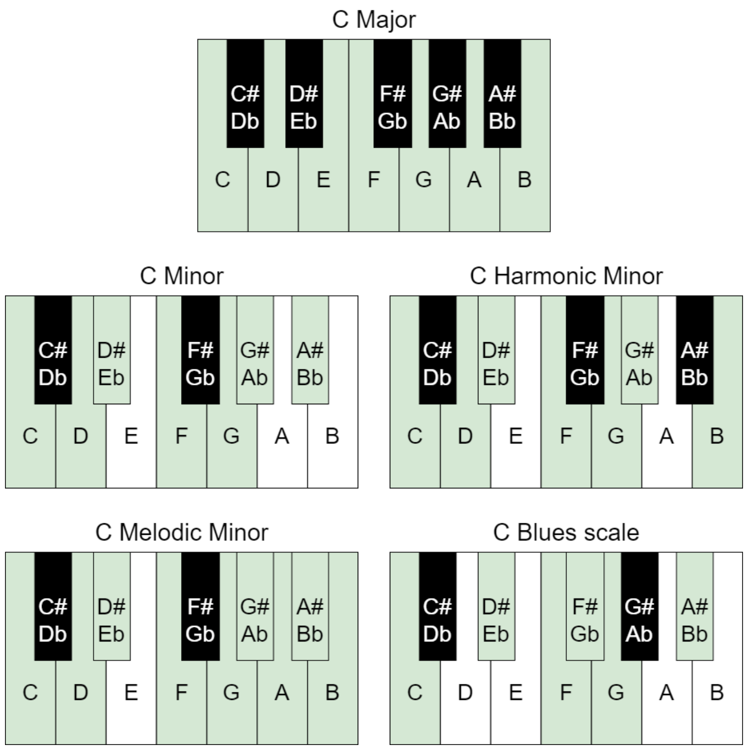


Figure 3.2: The notes included in the different scales in the key of C.

used to normalize the data to the same key. These are major (Ionian), minor (Aeolian), harmonic minor, melodic minor and the hexatonic blues scale. The harmonic minor scale is similar to Aeolian, but it has a natural 7th note. The melodic minor scale uses both flat and natural 6th and 7th notes, and the hexatonic blues scale consists of the minor pentatonic scale with the tritone included.

Counterpoint

Counterpoint refers to two or more equally dominant melodic lines that play simultaneously, creating meaningful harmonies while being rhythmically and melodically independent. It is a phenomenon often occurring in classical music, and it has been the focus of several previous attempts at generating music using ML.

Jam

A "jam session" in music is a phenomenon that emerged in the early 20th century where musicians would improvise together with minimal prior planning. This allows the musicians to be very free in their playing, often taking turns soloing and constantly adapting to each other. Jamming is often used both to practice improvisation and to compose new music. A large amount of music is created by exploring ideas through jamming.

Circle of fifths

The circle of fifths is a model used in music theory to explain the relationships between different chords and scales. It contains all the tones of the 12-tone equal temperament, clockwise in the order that they occur going upwards in fifths.

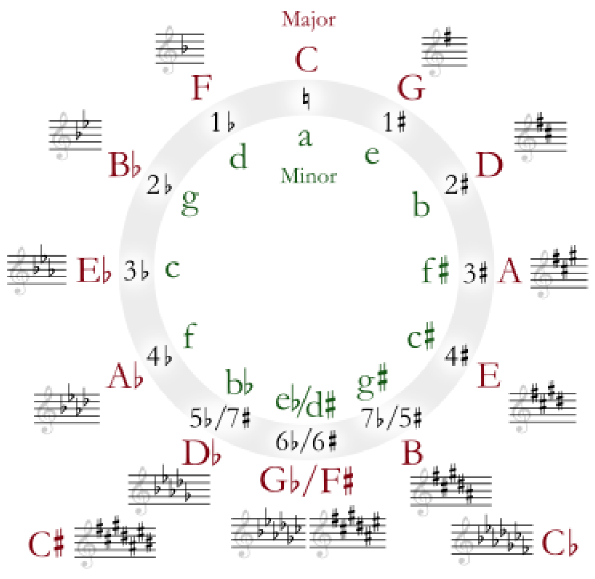


Figure 3.3: The Circle of fifths. Source: [45]

Monophonic/polyphonic

The term "monophonic" describes music that only plays one note at a time, and "polyphonic" music plays several notes simultaneously. Most music is polyphonic; however, an instrument can still be monophonic although it is playing with others. A monophonic instrument just means that it is not possible to play several notes simultaneously with that instrument.

Velocity

Velocity describes with how much force a note is being played. It is mostly used in the MIDI format to describe how hard a note should be played. This is separate from the volume of the music, as the velocity also affects how the instrument sounds.

3.3 Music theory in this thesis

In this thesis, some of the simpler concepts from music theory are used to describe the models and the generated music. Some musical understanding is required to create an ML model that can generate music. Understanding concepts like notes, pitches and rhythm is essential to being able to model them and understand how the models work.

No prior knowledge of music theory is required to read this thesis, and the terms and concepts that are relevant to this thesis are explained in the above section. If the reader wants a more in-depth explanation of western music theory, they are referred to works like Laitz' The Complete Musician [23] or McGrain's Music Notation [28].

Chapter 4

Design

This chapter describes the final design of the proposed solution. This includes an explanation of how the data is pre-processed, how the models are designed and how the solution was implemented and evaluated.

4.1 Overview

The solution proposed in this thesis is to train two separate LSTM networks with real music in the MIDI format and use these models to generate live accompaniment for piano, guitar, bass, and drums to music played by a user. This accompaniment will be played back to the user in real-time, simulating the experience of jamming freely with a band. The accompaniment will mainly adapt to the user's playing, but a certain level of listening and adaptation is also required from the performer, as is also true when playing with other people.

The solution is divided into two parts: training the models and using the models to generate live music. The first part includes data preparation, network design and training, and the second part consists of getting live MIDI-input, generating accompaniment in real-time using the pre-trained models, and playing it back to the user. The entire solution is depicted in figure 4.1. First, the dataset is prepared for training the models by normalizing and extracting the required data from the MIDI music. Then, the models are created and trained. The chord model is trained first, as the models use the same chord embedding. The models are exported and used in the jamming application. Here, a human user plays something on piano, which is transformed into input for the models. The polyphonic model is used each timestep to generate the next timestep of polyphonic music, while the chord model is called periodically to predict the next chord, used as input for the polyphonic model. The output from the polyphonic model is processed and split into different digital instruments that play what the model predicted back to the human user. This cycle continues until the user stops the session.

In order to be able to implement the networks in the timeframe of this project, we used a high-level machine learning API, which has some limitations regarding speed performance, and turned out not to be suitable

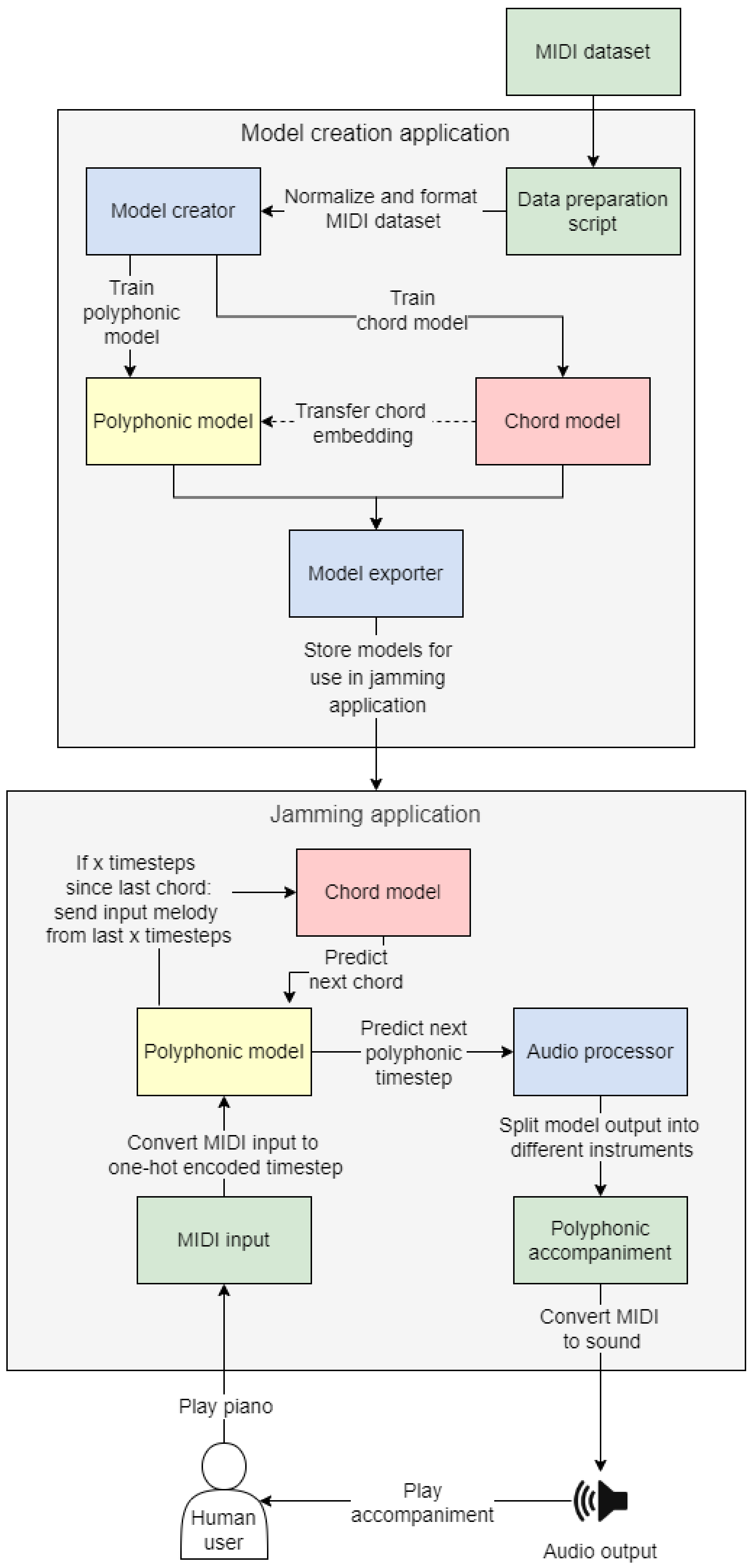


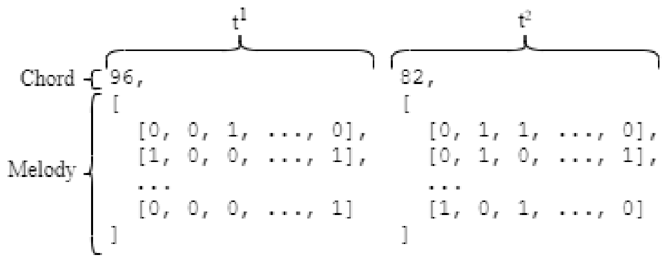
Figure 4.1: The solution is split into two separate applications: one that prepares the MIDI dataset, trains, and exports the models, and one that uses the models to generate live polyphonic accompaniment based on notes played by a user.

for live generation with the implemented models. Therefore, the solution presented in this chapter is suboptimal in terms of the project's initial goals. An interactive version of the application was implemented, but the models themselves are too slow to be used in a live setting. This project did not yield a working demo of the intended application, but instead music samples generated using static melodies were used to control the fitness of the models. These samples, along with the integration of an application that works, but only at a very low tempo, serve as a proof of concept and shows that the proposed network structure and application is feasible for the purpose of interactively generating musical accompaniment.

4.2 Model design

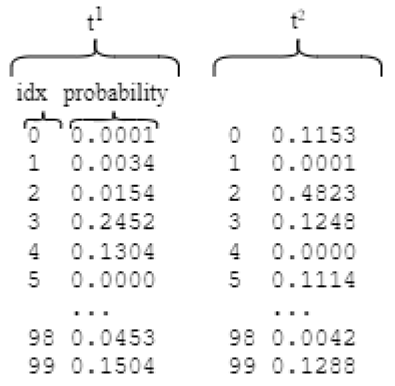
The proposed solution is a compound model that uses two different LSTM networks; one to generate a chord sequence for a melody, and one to generate polyphonic accompaniment based on the melody and chord sequence. This approach is based on the solution proposed by Brunner et al. for the JamBot model [5] but adapts the models to generate multi-track polyphonic accompaniment to a melody instead generating single-track polyphonic music from scratch. Brunner et al. [5] argue that separating chord generation and polyphonic music generation into two networks that use different time intervals achieves better long-term structure because the chords are used as a structural basis for the music. This is similar to how chords work in most music. Additionally, by letting the model learn the relations between different chords and notes instead of using the chords as rules for what notes to predict, the chords are used more like they are in real music. In the JamBot model [5], the chord model first generates a whole chord sequence, and the polyphonic model generates music based on that. In this project, the models were designed to be executed continuously. The chord model predicts the next chord, which is used for the next 16 timesteps, and the polyphonic model uses it to generate polyphonic music for the next 16 timesteps.

The first network (chord network) has two different inputs: the melody and the initial chord. For each step, the model uses the chords previously generated, and the continuation of the melody. The melody consists of one-hot encoded segments of 16 timesteps, which go through a 1D convolutional layer. Each timestep in the data is one 16th note, and each chord is held for exactly one bar (16 16th notes), so the model is only called once for each bar. By running the melody input through a convolution, the model can use the whole melody of the bar while considering the order of the timesteps. The chords are mapped using an embedding layer, allowing the model to map the relations between the chords. This is a concept often applied in natural language processing and Brunner et al. [5] is to our knowledge the first to use embeddings to map chords. They showed that the chord embedding was able to extract a mapping similar to the circle of fifths, which describes relationships between different scales and chords in western music theory. This clearly shows that the chord



```
96,                          82,
[                            [
  [0, 0, 1, ..., 0],           [0, 1, 1, ..., 0],
  [1, 0, 0, ..., 1],           [0, 1, 0, ..., 1],
  ...                          ...
  [0, 0, 0, ..., 1]            [1, 0, 1, ..., 0]
]                            ]
```

(a) Input shape of the chord network.



```
0  0.0001      0  0.1153
1  0.0034      1  0.0001
2  0.0154      2  0.4823
3  0.2452      3  0.1248
4  0.1304      4  0.0000
5  0.0000      5  0.1114
   ...            ...
98 0.0453      98 0.0042
99 0.1504      99 0.1288
```

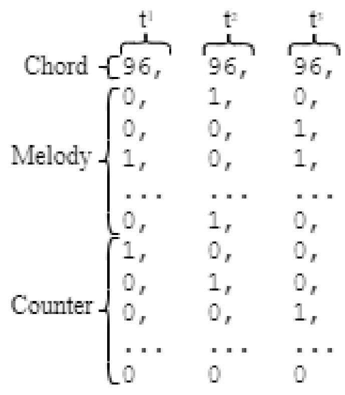
(b) Output shape of the chord network.

Figure 4.2: Input and output shapes of the chord network.

embedding is meaningful. The embedded chords and convoluted melody are concatenated and fed into an LSTM layer with 512 hidden states. Using a dense layer and a softmax activation, the model outputs a probability distribution for all 100 chords. The next chord is then selected randomly from this distribution.

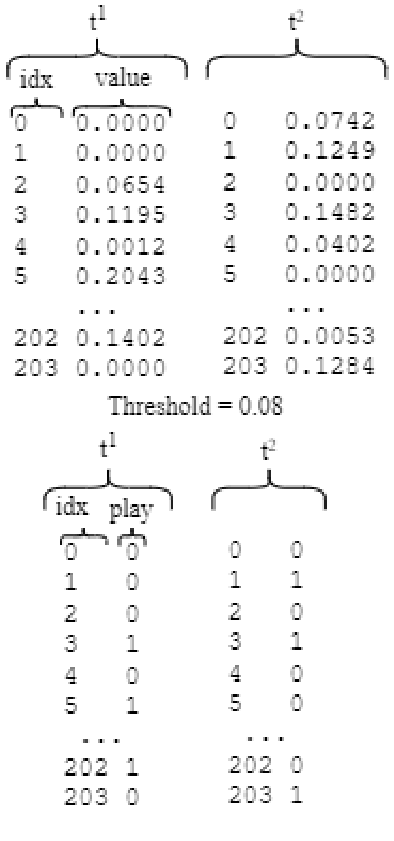
For training the model, we used the Adam optimizer [21], which is included in TensorFlow, with a learning rate of 0.0001. The model was trained for 50 epochs, and we selected the epoch with the highest validation accuracy.

The second network (polyphonic network) also takes the chords and melody as input, but one 16th note at a time, so each chord is repeated for 16 timesteps. Additionally, the input includes a one-hot encoded counter that describes the current step in the bar. This tells the model where in the bar it is, and when the next chord change is, most likely making better transitions between chords. The chords are run through the same embedding that was trained in the chord model. The embedding is non-trainable in the polyphonic network. Contrary to Brunner et al. [5], we did not include the next chord, because this information is not available as the melody and chords are generated sequentially. The chord embedding is concatenated with the one-hot encoded melody and counter and fed into an LSTM layer with 1024 hidden states. Finally, the model has a dense layer and a sigmoid activation, giving a value between 0 and 1 for each of the 204 notes. All



```
96,   96,   96,
0,    1,    0,
0,    0,    1,
1,    0,    1,
...   ...   ...
0,    1,    0,
1,    0,    0,
0,    1,    0,
0,    0,    1,
...   ...   ...
0     0     0
```

(a) Input shape of the polyphonic network.



```
0   0.0000    0   0.0742
1   0.0000    1   0.1249
2   0.0654    2   0.0000
3   0.1195    3   0.1482
4   0.0012    4   0.0402
5   0.2043    5   0.0000
    ...           ...
202 0.1402    202 0.0053
203 0.0000    203 0.1284
```

```
0   0     0   0
1   0     1   1
2   0     2   0
3   1     3   1
4   0     4   0
5   1     5   0
 ...       ...
202 1     202 0
203 0     203 1
```

(b) Output shape of the polyphonic network before (top) and after (bottom) threshold selection.

Figure 4.3: Input and output shapes of the polyphonic network.

notes over a certain threshold are played by the computer. The threshold used was 0.08, which was the value found to generate the best sounding and most coherent music.

The polyphonic network was also trained using the Adam optimizer with an initial learning rate of 0.0001. The model was trained for 10 epochs, and the epoch with the highest validation accuracy was used for objective and subjective evaluation.

The models are trained and used separately because they operate at different time intervals. The chord model predicts a chord for every bar, which is equivalent to 16 timesteps in the polyphonic network. There is no way to use some of the layers only once every 16 timesteps [5], and if the model were to predict a chord for every timestep it would either have a much less meaningful selection of chords, or it would predict the same chord every timestep. Separating the models allows them to work at different time intervals.

4.2.1 Baseline model

In order to evaluate whether the addition of chords is better for generating polyphonic accompaniment than just using a straight-forward LSTM network, we created a simpler LSTM network that does not integrate chords, to use as a baseline for the model performance. This network only contains one LSTM layer with 1024 hidden states, as well as a dense layer and a sigmoid activation. Our hypothesis was that the baseline model is too simple to learn the subtle harmonic connections in the music and would not be able to generate meaningful polyphonic music in the way that the compound model can.

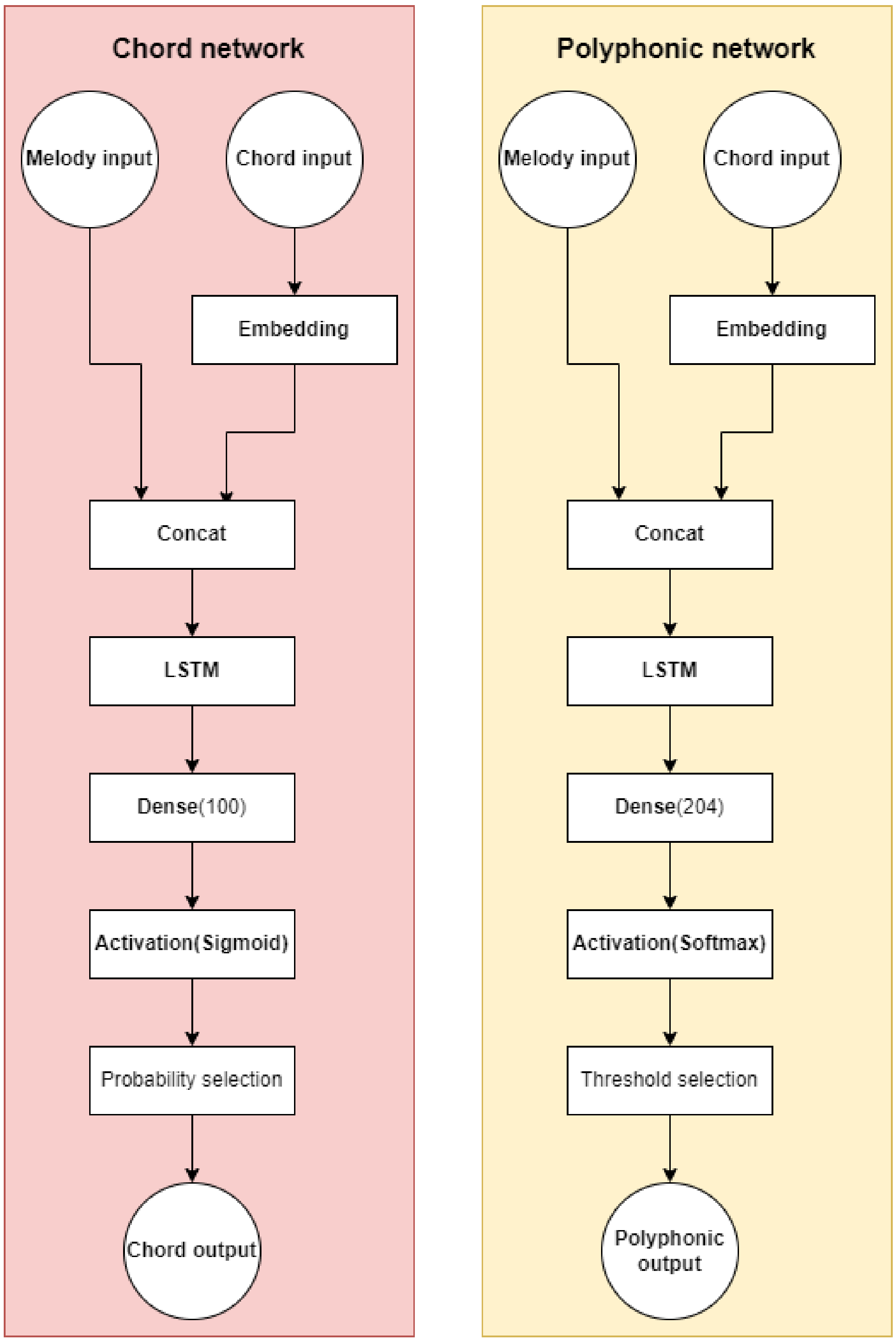


Figure 4.4: Network architecture of the chord network (left) and the polyphonic network (right).

4.3 Data

The data used to train the model is the Lakh MIDI Dataset [34], which consists of about 180 000 songs in many styles and genres in the MIDI format. This seems to be the most comprehensive and cleanest dataset of MIDI music available, and a subset of this dataset was used by Brunner et al. to train the JamBot model [5]. The MIDI files are processed with Python using a package called pretty_midi [35]. The data preparation consists of four steps: normalization, chord extraction, lead melody extraction and accompaniment extraction.

4.3.1 Key normalization

For the model to be able to learn the relationships between the notes properly, it is important that the music is normalized to the same starting key. Depending on the key and scale being played, the notes have completely different roles and functions, however, the model has no way of extracting this context out of the samples. This would likely require much more data, which is not available. If all the songs have the same starting key, the model is much more likely to make meaningful connections between the notes, making it able to generate much better music. The scale of a song is extracted by comparing the most commonly occurring notes with 60 scales. These scales are the five scales major (Ionian), minor (Aeolian), harmonic minor, melodic minor and the blues scale, for each of the 12 starting keys.

As the notes in a song might match more than one of these scales, and the scales contain varying amounts of notes, the most common scales are checked first. If the seven most common notes match one of the major scales, the song is either in major or minor, because they are two of the diatonic scales and contain the same notes. If so, the program checks whether the 6th note occurs more frequently than the 1st note, in which case the song is likely in minor. If the notes do not match any of the diatonic scales, the seven most common notes are compared with the harmonic minor scales, which have a natural 7th step. If harmonic minor does not fit either, the program checks the nine most occurring notes against the melodic minor scales, which use both flat and natural 6th and 7th steps. Lastly, if none of the other scales match, the program compares the six most occurring notes to the hexatonic blues scale, which is a minor pentatonic scale with a tritone included. The notes contained in each of these scales are shown in figure 3.2.

Each song is modulated to the same starting key, but they may have different scales. Songs that do not match any of the scales are removed from the dataset. The songs are also set to the same tempo by using the first tempo found in the song's metadata to set the sample rate when converting to the piano roll format. This way, one timestep is the same length for all the songs, based on the actual note values instead of their timestamps. This method assumes most of the songs are in $\frac{4}{4}$ time and do not have tempo changes.

4.3.2 Chord extraction

The chord sequence is especially important to the musical accompaniment, often even more so than the melody. Instead of using only chord sequences as input, like JamBot [5], this solution uses a combination of chord sequences and melodies. This way, it can learn to generate chord sequences based on a melody and learn voicings and rhythms complementing those chords.

The model uses chords extracted from the data. The chord schemes are created by finding the three most occurring notes in each bar. The 99 most common three note combinations are enumerated in a dictionary, while the rest are given the label 'UNK'. This is done so that the chords can be used in the embedding layer in the chord network. Each chord is held for one bar, which is typical for most music. The ten most occurring chords are displayed in table 4.1.

Chord	Occurrences
C	1 835 301
G	962 911
F	786 590
Am	538 739
Cm	477 943
Csus2	450 175
Csus4	394 112
DFG	277 831
Dm	271 265
CGA	229 818

Table 4.1: Most commonly occurring chords

4.3.3 Melody extraction

The melodies used for training are the instrument tracks included in the MIDI that are most likely to be a lead or melodic instrument. If several tracks are potential candidates, one is selected randomly for each song. The program tries to find a track that is mainly monophonic, plays most of the time, and is not a bass instrument. The melodies are represented in the piano roll format, with 60 notes for each timestep. The piano rolls are binary, so each note is either playing or not playing. As many of the 128 notes in the MIDI format are rarely used, the highest and lowest octaves are removed before the data is used.

4.3.4 Accompaniment extraction

The polyphonic tracks used as target output for the polyphonic network are separated into four parts: piano, guitar, bass, and drums. Each part is selected by finding all the MIDI tracks containing keywords related to the instrument and combining them into one track. The four tracks are then concatenated into one large track with 204 attributes, which is the target output of the polyphonic network. The three first instruments have 60 notes each, while the drum track only has 24 notes, as it normally uses only a few of the notes.

4.4 Application

The solution was implemented using Python. The application was split into two parts: a model creation application and a jamming application. The model creation application prepares the data, and creates and trains the models, and the jamming application uses the models to generate accompaniment based on live input.

4.4.1 Model creation application

The models were created and trained using the high-level API Keras [7], with TensorFlow [27] as back end. Although Keras has limitations regarding speed performance, it drastically reduced the time it took to implement and test the models. The application makes use of Keras' functional API, which is used to create a graph containing all layers in order. The models are easily trained and evaluated using built-in functionality. Once the models are trained, they are stored as TensorFlow graphs, and can be loaded back into a different program without having to re-create or re-train the models.

The model creation, training and evaluation is done by a custom Python class, which allowed us to easily reuse the code for model creation and training to test different configurations and model parameters. The same class also stores the models using Keras' *Model.save()* function and creates the training graphs using the Python library Matplotlib [19]. These graphs were used to see whether the models were overfitting, and how many epochs were necessary for training the models.

4.4.2 Jamming application

The jamming application imports the models using Keras' *models.load_model()* function. The models are imported and used by a custom class containing a function for performing one timestep, which returns the polyphonic output. This function is called periodically by the main application, which uses the library PYO [1] to receive and process MIDI input from the user, output sound based on what the polyphonic model generates, and to run the program loop itself. The PYO library contains a class called *Pattern*, which is used to call the models at a constant time interval, corresponding to one 16th note.

The input played by the user is obtained by using two binary arrays with length equal to the number of MIDI notes. We use PYO's MIDI event handlers to record what notes are being pressed and released in the arrays, and these are used to create the model input. With this method, the notes played by the user will always be quantized to the 16th note grid, which is the same grid used as timesteps for the model.

4.4.3 Implementation

The solution was implemented during the course of the master's project starting in August 2021. The following chapter describes how the solution was implemented, in addition to going into more detail about the algorithms and the applications.

Chapter 5

Implementation

This chapter contains descriptions of the development processes and experimental approach to the project. As the project is design based, the reader might want to look at the source code. The code used for creating, training and using the models, as well as preparing the data, will temporarily be available at Google Disk through this url: ht-tps://drive.google.com/drive/folders/1we16zbsSCxhgGn3_GPrM9Xqn Dd4vUhdX?usp=sharing

5.1 Model

We initially experimented with different models and implementations of neural networks for music generation. The first thing we wanted to attempt was using a sequence-to-sequence model, often used for machine translation. However, sequence-to-sequence models take a complete sequence and transform it into a different sequence, so it cannot work with live input. In order to generate continuous accompaniment to live input, the model needs to predict each timestep separately.

We eventually decided to implement a normal LSTM model, even though earlier work had found that this type of model was too simple to generate meaningful music with long-term structures. Our plan was to experiment with adding or removing different layers to find a network structure that worked better. However, before doing this, some more time was spent searching for alternative models, during which we found the project called JamBot [5], which used a somewhat different approach.

JamBot was created to generate new polyphonic music by first generating a chord sequence with one network, and then generating polyphonic music with a second network. We decided to try to adapt this model to generating polyphonic accompaniment to a melody by including the melody as input to both networks. We experimented with different data formats and sources like the Million Song Dataset but ended up using a method similar to Brunner et al. [5], extracting key signatures, scales, and chords from the MIDI data itself. The results we got for the most common chords and scales were similar to those of Brunner et al. [5], although we additionally included the melodic and harmonic minor scales in the dataset, and not just the diatonic scales. Additionally, the model was restricted to use a window of 128 timesteps instead of training over the whole song. This means that long-term musical structure is not preserved in the same way, but the model processes the music with a window of eight bars. The session is supposed to be continuous and not end after a certain time, so using the same structure as a normal song would not suffice. It does however implore the exploration of different window sizes, which might have different results.

After some trial and error, we were able to successfully implement a model that could generate a chord sequence for an existing melody. At first, we built the networks with Keras' sequential model, but this turned out to be too restricted for our networks. Because the networks had to take the chords and melody as separate inputs, and concatenated them later, we ended up using Keras' functional API, which was much more versatile than the sequential model.

The model was initially terribly slow, so we spent some time optimizing the model's performance, mostly working on the data pipeline. We initially created batch generators that processed the data simultaneously as training, so not all the data would have to be loaded into memory. This later turned out to be a major bottleneck when training, so we took steps to reduce the size of the data and loading it all into memory before training.

To use the melody as input to the chord model, we initially made it predict the next chord for each timestep, which meant that in the training data, each chord was held for 16 timesteps. This seemed like a logical way to predict the chords because the model would work with live input, and the chords should depend on the input melody as well as the previous chords. We also wanted to see if the model was able to select chords for different time periods depending on the melody. However, this method caused the model to be very repetitive, often predicting the same chord for lengthy periods. We tried creating an onset and sustain value for each chord, describing whether there is a chord change, but this did not yield better results. Because the model was supposed to generate the next chord only every 16 timesteps, the model was changed so that each timestep is one bar. Instead of one 16th note, the model gets a whole bar of the melody, which is run through a 1D convolutional layer. This allows the model to preserve the order of the notes while processing the whole bar in a single timestep. This method seems to work as intended and is also similar to how Brunner et al. [5] uses their chord model.

The code in listing 1 contains a function that returns the chord network. The network structure is the same as shown in figure 4.4. The network has two inputs: the chords and the melody. The shapes of the inputs reflect the shape of the input data. The network gets the last eight bars, with one chord and one 16-note melody with 60 one-hot encoded notes, for each bar. The embedding layer gets the chord input, and outputs a 10-dimensional vector. The melody is input to a 1-dimensional convolution, which is concatenated with the chord embedding, and used as input for the LSTM. The output of the LSTM, with 512 hidden states, is put through a dense layer, which reduces the output size to 100. Lastly, the softmax

```python
import tensorflow as tf

def create_chord_network():
    #Network inputs
    chord_input = tf.keras.layers.Input(shape = (8, 1))
    melody_input = tf.keras.layers.Input(shape = (8, 16, 60))
    #Chord embedding
    embedding = tf.keras.layers.Embedding(
        input_dim = 100,
        output_dim = 10
    )(chord_input)
    #1D Convolution
    conv = tf.keras.layers.Conv1D(
        filters = 60,
        kernel_size = 16,
        input_shape = (16, 60)
    )(melody_input)
    #Concatenate embedding and convolution
    concat = tf.keras.layers.concatenate([embedding, conv])
    #LSTM layer
    lstm = tf.keras.layers.LSTM(
        units=512,
        return_sequences=True
    )(concat)
    #Dense layer and activation
    dense = tf.keras.layers.Dense(100)(lstm)
    output = tf.keras.layers.Activation("softmax")(dense)
    #Build model
    chord_network = tf.keras.models.Model(
        inputs=[chord_input, melody_input],
        outputs=output
        )
    return chord_network
```

Listing 1: Sample code used to create the chord network using Keras' functional API. Some reshaping steps have been left out for better readability.

activation gives each output a probability. The graph is turned into a model with the *tf.keras.models.Model* class.

The polyphonic network was much faster to implement than the chord network, as we were more familiar with TensorFlow and Keras, and most of the data formatting was already done. We still needed to extract the different instrument parts from the MIDI music, but we were able to do so fairly quickly. As input the model uses the same training data as the chord model, and it is trained on the correct chord sequences and not the ones generated by the chord model. It was a bit challenging to obtain the instrument tracks as target data for the polyphonic network, but we eventually created a method that worked relatively well, described in chapter 5.2.

The code in listing 2 contains a function that returns the polyphonic network. The structure of this network is also the same as in figure 4.4. This

```python
import tensorflow as tf

def create_poly_network(embedding_weights):
    #Initialize embedding
    embed_init = tf.keras.initializers.Constant(embedding_weights)
    #Network inputs
    chord_input = tf.keras.layers.Input(shape = (128, 1))
    melody_input = tf.keras.layers.Input(shape = (128, 60 + 16))
    #Chord embedding with non-trainable weights
    embedding = tf.keras.layers.Embedding(
        input_dim = 100,
        output_dim = 10,
        embeddings_initializer = embed_init,
        trainable = False
    )(chord_input)
    #Concatenate embedding and melody
    concat = tf.keras.layers.concatenate([embedding, melody_input])
    #LSTM layer
    lstm = tf.keras.layers.LSTM(
        units = 1024,
        return_sequences = True
    )(concat)
    #Dense layer and activation
    dense = tf.keras.layers.Dense(204)(lstm)
    output = tf.keras.layers.Activation("sigmoid")(dense)
    #Build model
    poly_network = tf.keras.models.Model(
        inputs = [chord_input, melody_input],
        outputs = output
    )
    return poly_network
```

Listing 2: Sample code used to create the polyphonic network using Keras' functional API. Some reshaping steps have been left out for better readability.

network has the same inputs as the chord network, but in a different shape. Instead of 8 bars, the polyphonic network gets 128 16th notes. Each chord is repeated 16 times. And instead of 16-timestep segments, the polyphonic model gets 128 timesteps of melody in order. The shape of the melody input is equal to the number of notes (60) plus the size of the counter (16). Like in the chord network, the chord input is used in an embedding layer, however, the polyphonic network uses the pre-trained embedding from the chord network. The embedding is concatenated with the melody and used as input for the LSTM layer with 1024 hidden states. The LSTM output is put through a dense layer, reducing the size to 204, which is the number of outputs for the polyphonic network. The sigmoid activation assigns each output a value between 0 and 1.

After the networks were implemented and tested, the training speed was optimized in order to be able to test different configurations and tune the models. We had originally intended to test about 30 different combinations of parameters for the networks and running each test several times to see what parameters made statistically significant changes to the accuracy of the polyphonic network, but this would require training the network several hundred times in order to get enough data. As this turned out not to be a viable option, we still tested many different configurations to see what parameters had the biggest effect on accuracy and time.

Near the end of the project, we increased the amount of data from 14 000 from the LMD_Matched dataset used by Brunner et al. [5], to the 130 000 songs of the full Lakh MIDI dataset. Training the models with the additional data took a lot more time, and it was not possible to train them as often. Instead we used the experience from testing the models with different parameters and concluded what parameter values were most likely to get the best accuracy for the polyphonic network. We also reduced the size of the training dataset for the polyphonic network, as training this on all 130 000 songs would have taken too long.

Because we were not able to perform user testing with the generated music, we decided to use a much simpler LSTM network as a baseline for the model performance. This way, we could compare both the objective analysis and the generated music, to see whether the addition of the chord network actually helps the model generate better music. The baseline model was easy to implement, as the network structure is very simple.

5.1.1 Model tuning

In order to find the best parameters for training we ran the model with many different configurations and measured the validation accuracy. We looked at several measurements to select the best parameters, and tested different values for initial learning rate, LSTM sizes, training batch sizes and number of epochs.

After testing several values for each of these parameters, we created a configuration that balances training speed with accuracy, so the training does not take too long, while it keeps the best possible validation accuracy during training. The configuration can be found in table 5.1. In general, higher LSTM sizes had a large effect on the outcome of the models. The polyphonic model has many output features, so the LSTM size needs to be large. However, this also makes the model very slow.

	Learning rate	LSTM size	Batch size	Epochs
Chord	0.0001	512	128	50
Polyphonic	0.0001	1024	256	10

Table 5.1: Configuration used for the compound model.

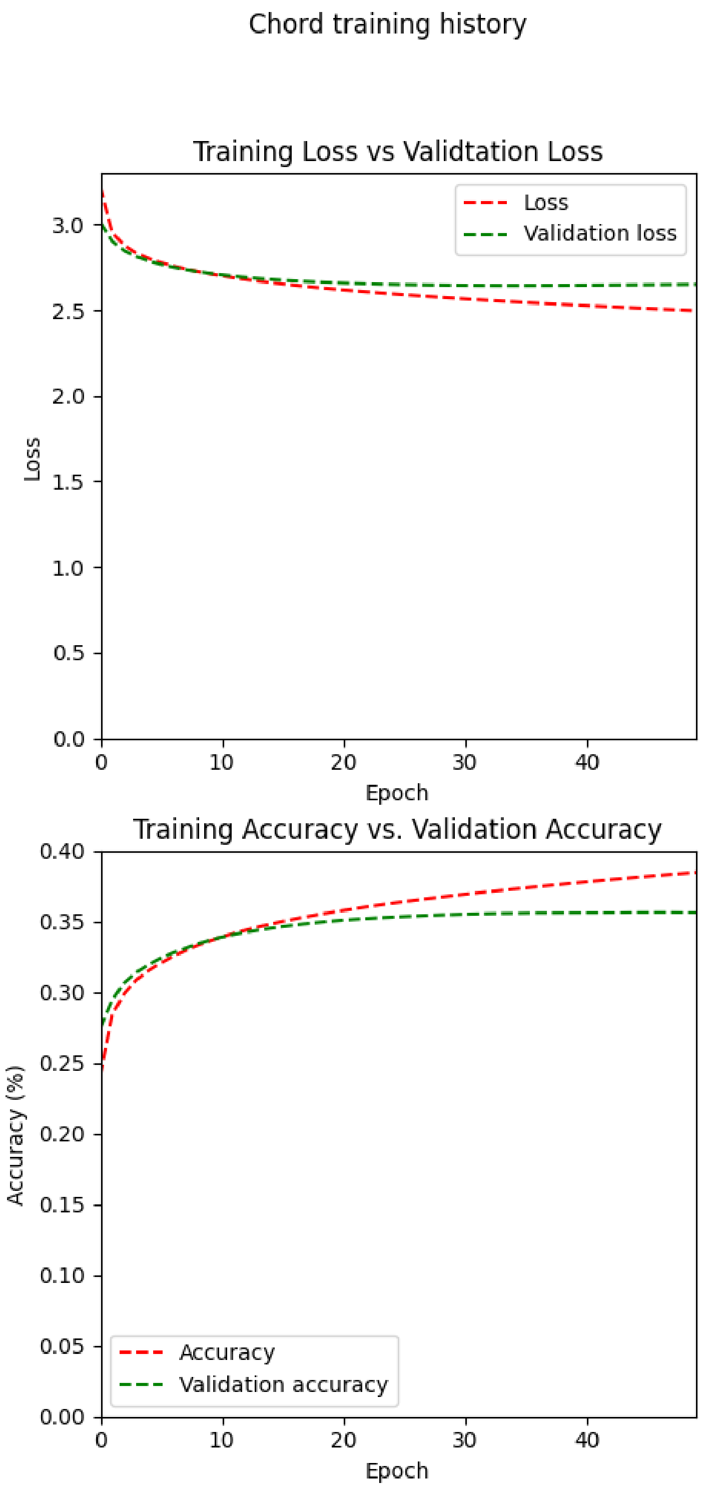


Figure 5.1: Training graph of the chord network using the optimized configuration. The model is overfitting after about ten epochs.

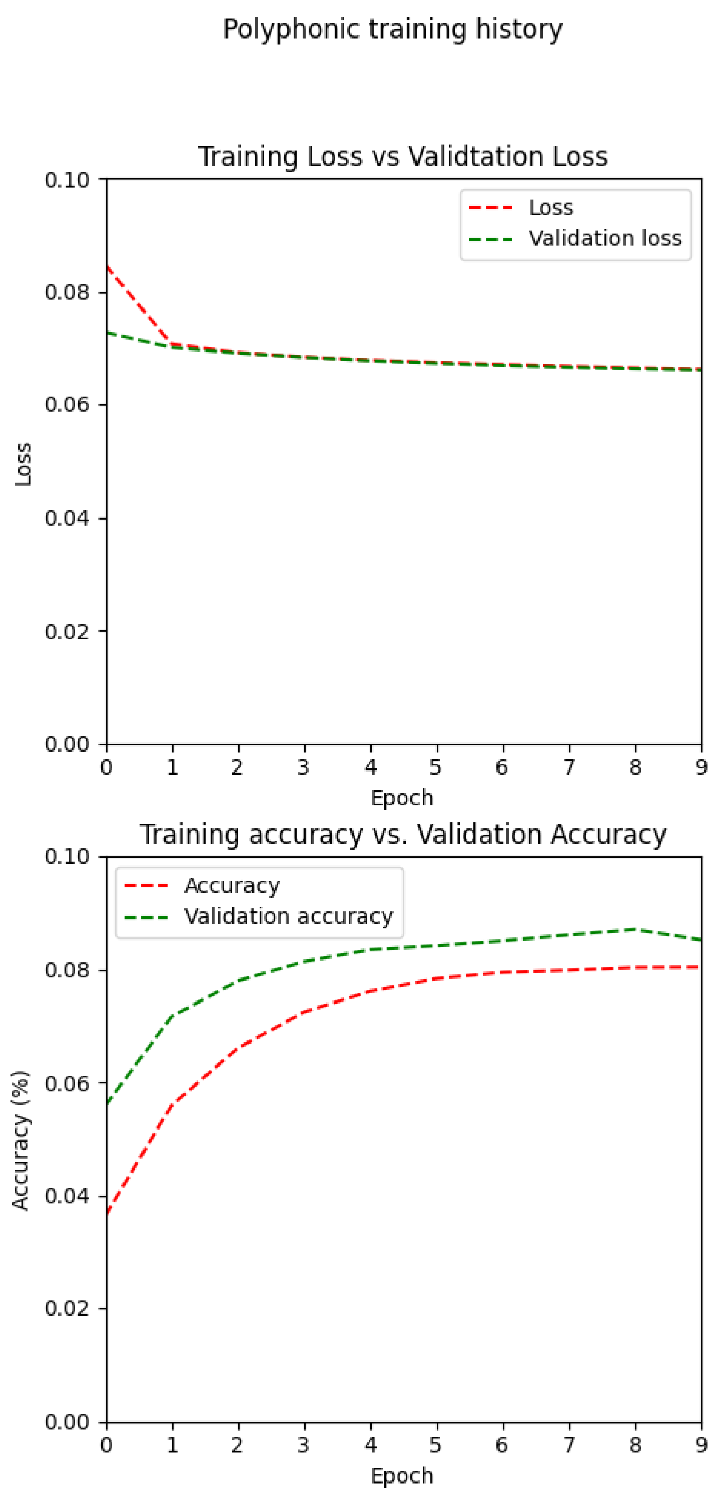


Figure 5.2: Training graph of the polyphonic network using the optimized configuration.

5.2 Data

In any machine learning or data analysis project, finding and formatting data usually takes a considerable amount of time. This project was no exception, and a lot of time was spent writing and testing scripts to format and adapt the data to the models. We used the Lakh MIDI Dataset [34], which was also used by Brunner et al. [5].

The dataset contains about 180 000 MIDI files, many of which are split into different instrument tracks and contain metadata about the songs' notes, tempos, instrument names etc. About 130 000 of the songs were used for training the chord network, and 60 000 for the polyphonic network. The polyphonic network was trained on less data because it would take an unreasonably long time to train the network with the full dataset. To load the MIDI data into Python we used a utility package called pretty_midi [35], which contains functions for loading, writing, and manipulating MIDI. It also has a function for converting the MIDI data to the piano roll format.

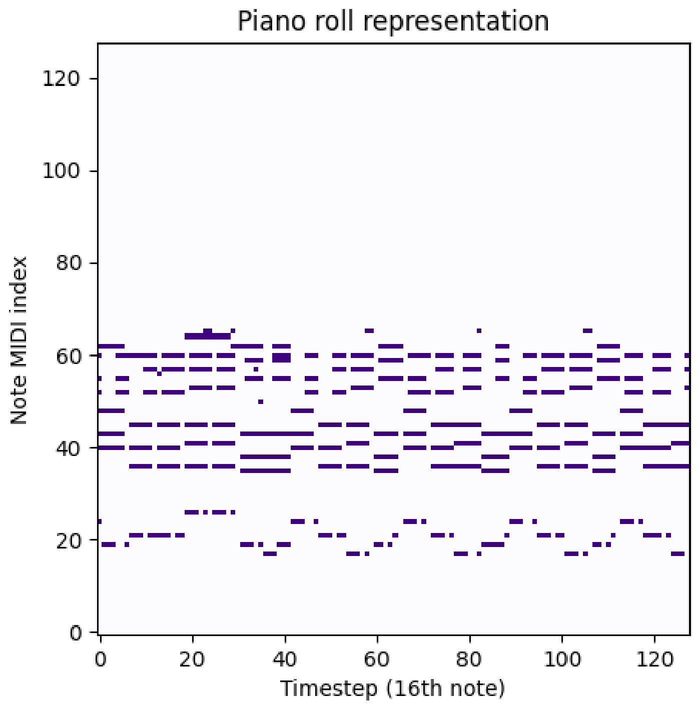


Figure 5.3: Example of a piano roll representation

We tested several formats before deciding to use the piano roll format. First, we tried using the representation created by Colombo and Gerstner [10], called BachProp, but their code for transforming the data required an unavailable Python package. Creating a custom script for converting MIDI to this format would likely have been a waste of time, so we decided to avoid that. We attempted to use the raw MIDI as input, but this did not yield any meaningful predictions, as the model would have to predict both pitch and length for each note, and it would not be quantized to 16th notes, but just generate notes in a sequence. This issue would also have been the case with BachProp. We eventually tried using the piano roll format, which uses more space, but is neatly organized into uniform timesteps, and the notes are already one-hot encoded. As the pretty_midi [35] package has a function for converting MIDI to piano roll, this was relatively easy. We tried doing this with a custom script first, but it was much slower, so we ended up using the built-in function *get_piano_roll* in pretty_midi. This function takes a sample rate, which is the number of milliseconds each sample (timestep) lasts. We calculated the sample rate based on the tempo of the song, which is included in the metadata of most songs. This makes the method relatively accurate for quantizing the songs to 16th notes, but it does not account for tempo changes during the song, or other subdivisions such as triplets or quintuplets. These are quantized to a 16th note grid.

5.2.1 Data preparation

The method used to normalize the songs to the same key is the same as the one used by Brunner et al. [5]. The scales are extracted by creating a histogram of the chroma representation of the song, which takes all 128 notes condensed into the 12 tones, and creating a scale based on the most common notes. The scale and key it is matched with is used to determine the key, and the entire song is modulated to C by adding or subtracting a specific index to all the notes in the MIDI data. Contrary to Brunner et al. [5], we used all songs that got assigned a scale using this method, and not just the diatonic scales. Using only the diatonic scales might make the music less diverse and functional because the Aeolian minor scale does not have a dominant chord.

Using this method, about 135 000 of the songs were assigned a scale, with the majority being assigned the major scale. Many of the songs got assigned the natural minor scale, but the other scales were very few compared to major and natural minor. Regardless, the addition of these scales could help the model learn more interesting relationships it can use in the music.

The chords used for training the models are extracted from the MIDI music by creating a histogram of all 12 notes for each bar of each song and putting the three most common tones of each bar together as a chord. This method assumes that all the songs are in the $\frac{4}{4}$ time signature, which is probably not the case. The MIDI metadata does not contain information about the time signature, and this is difficult to extract from just analysing the notes. By extracting the chords from the actual music, the model uses actual three-note combinations that occur in the music, and not only chords from music theory.

The melodies used for training are extracted by for each song creating piano rolls of all the tracks included in the MIDI file and finding the track that is most likely playing the melody. This is done by finding all the tracks in a song that are candidate melodies and selecting randomly from these. The candidates are selected based on a few criteria. Firstly, if the track metadata includes keywords related to "melody" or "lead", it is automatically a candidate, and if there are keywords related to "bass", it is automatically excluded. Otherwise, a track is selected as a candidate if it plays more than 50% of the time and is mostly monophonic. If none of the tracks match these criteria, a random polyphonic track that pays more than 50% of the time is selected instead. If that also fails, the song is removed from the dataset. Only about 5000 songs did not get a melody track using this method.

To get the target data for the polyphonic network, the instrument tracks, the metadata of each track in each song is searched for keywords, and similar instruments are added together. This way, if there are more than one of each instrument, e.g., Piano 1 and Piano 2, they are combined into one single track. Finally, the chords, melodies and instrument tracks are padded with zeros so that the number of steps for each song is divisible by the number of steps used by the model, which is 8 for the chord model and 128 for the polyphonic model. The melodies and instruments are also converted to binary to save space. The data is stored using a binary Pickle format, which was found to be the quickest way to load the data.

The output of the model is also in the piano roll format, but it contains several concatenated tracks. After the model outputs a timestep, it is converted back into MIDI by splitting the tracks, creating a *PrettyMIDI.Instrument* object for each track, and using a function found in the "Examples" module in the pretty_midi package, called *piano_roll_to_midi*, to convert the piano roll back into MIDI. The tracks can be combined into a MIDI track, and exported to a MIDI file, or played as music by a module like PYO [1].

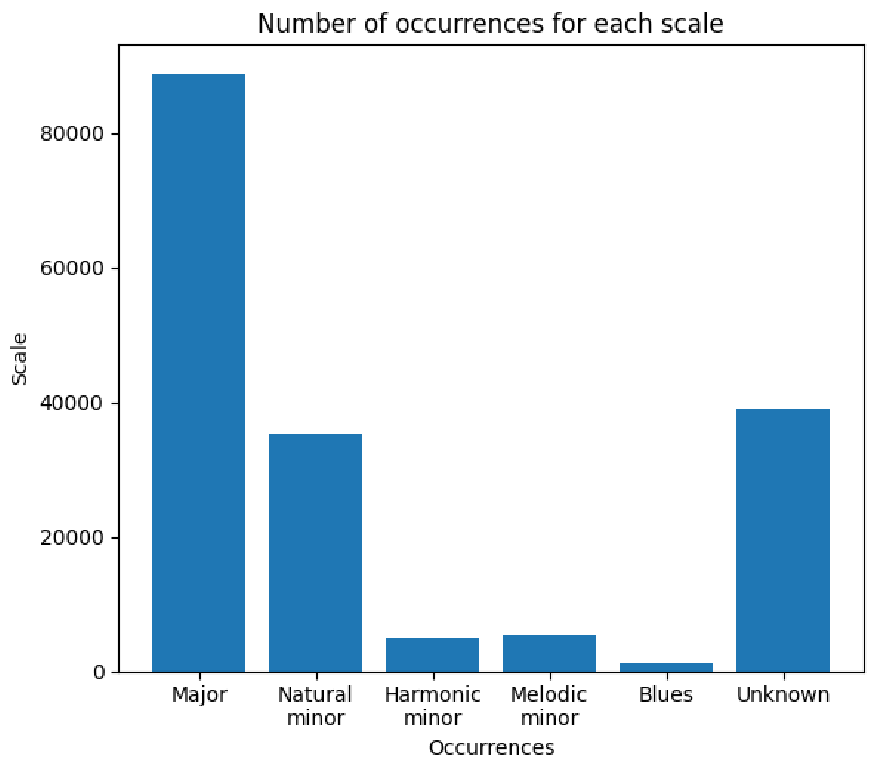


Figure 5.4: Most commonly occurring scales in the dataset.

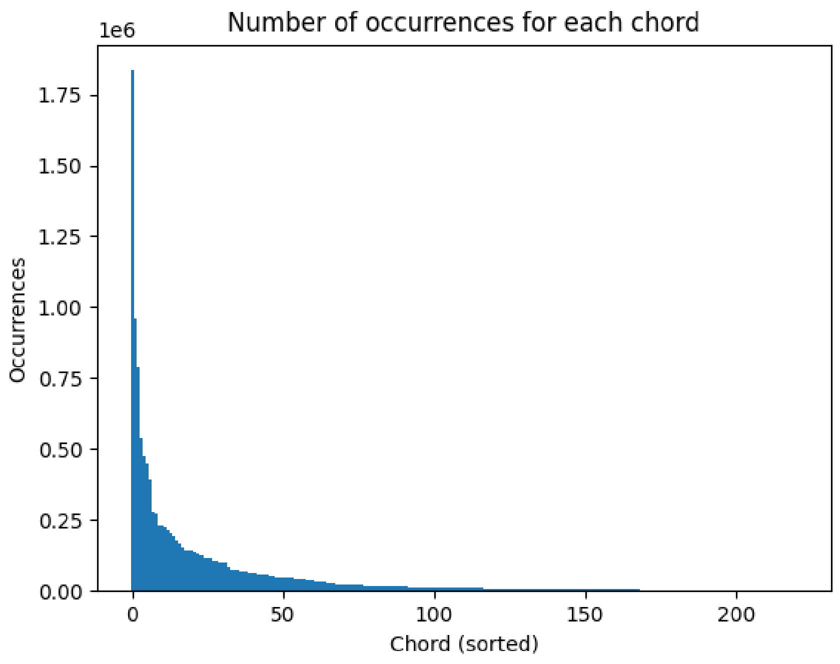


Figure 5.5: Most commonly occurring chords in the dataset.

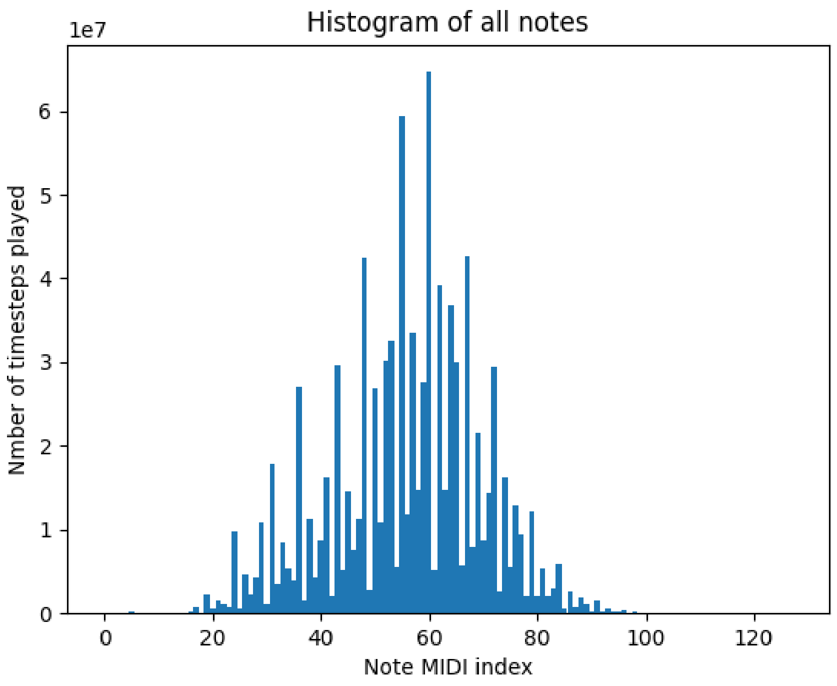


Figure 5.6: Histogram of all 128 notes over the whole dataset.

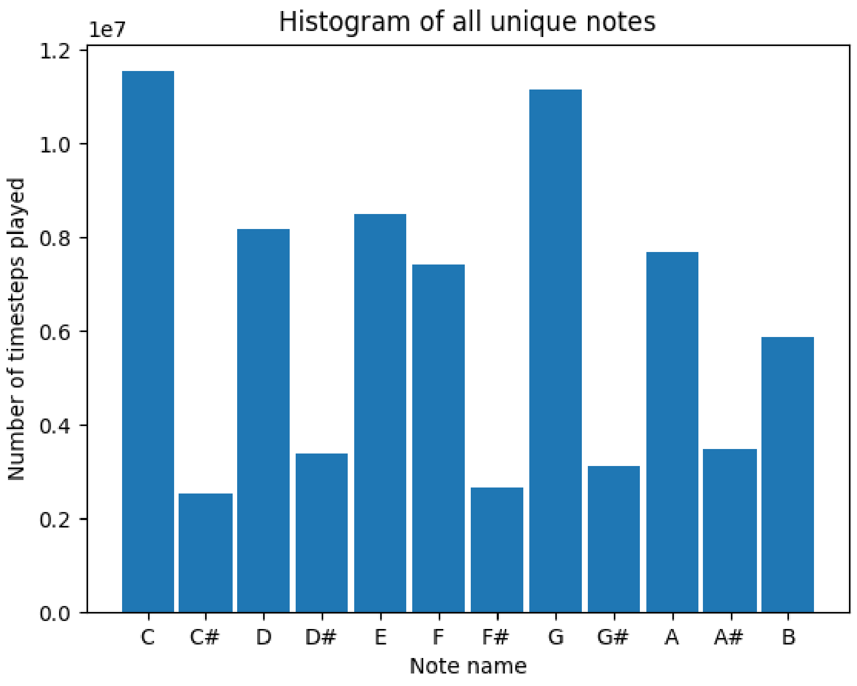


Figure 5.7: Histogram of each unique note over the whole dataset.

5.3 Application

In the actual implementation of the application, we used TensorFlow [27] and Keras [7] to implement the models, and the package PYO [1] to implement the MIDI interface, simple GUI, and sound engine. There were other options that that were either tested or considered before implementing the final solution. This section describes the processes and evaluations of alternative solutions that could be used for implementing the application.

5.3.1 Machine learning framework

Although we used TensorFlow for implementing the models, there are other frameworks for machine learning, such as PyTorch [32], scikit-learn [33], Torch [9] and Theano [43]. Among these, the most relevant to this project were TensorFlow and PyTorch, largely because they are the most popular, and thus have the largest user bases and most learning resources. Scikit-learn is the most mathematically oriented and lowest level ML Framework in use for Python, but it does not have built-in RNNs, so that would have to be made manually. This would likely take a lot of time, and because it would be programmed with Python, it would likely be slower than using PyTorch or TensorFlow. Torch is the framework that PyTorch is based on, with both being developed by Meta Platforms, previously named Facebook. Theano is another ML framework for Python and can also be used as backend for Keras. However, TensorFlow is the default backend for Keras, and much more used.

PyTorch is newer than TensorFlow and does not yet have a community as large, but it is increasing in popularity. Compared to Keras, PyTorch requires coding on a lower level, making implementation more difficult and time consuming. In this project, it would not have been feasible to implement and test the models in the required timeframe using PyTorch. However, as PyTorch is at a lower level than Keras, it also performs better. It might be worth implementing the same models using PyTorch to see if they are able to predict faster.

5.3.2 Framework for jamming application

There are many possibilities for taking live MIDI input and playing music as output, but most music software is developed using C++. However, there is a rising interest in music programming with Python, because of its much more gentle learning curve. Using C++, it is possible to use the JUCE [41] framework to make an application and a Virtual Studio Technology (VST) plugin that can be used in Digital Audio Workstations (DAWs) to record and create music [31]. JUCE is a C++ framework for creating music applications. It has built-in functionality for using external sound devices and VST plugin support, and it has a large user community with many learning resources. Implementing the application in C++ would require programming on a lower level and making the Keras models work in C++. Initially we decided to use the JUCE framework, but we were not able to get the Keras models working in C++. This was partially due to a lack of documentation for the TensorFlow C++ API, and few resources regarding the use of TensorFlow models with C++. After spending a considerable amount of time attempting this, we ultimately decided to implement the application using Python after all.

The main challenges with creating the application using Python were implementing a MIDI interface and creating audio signals based on the model outputs. We considered using the package called Pygame [38] because it contains a MIDI interface and functionality for creating GUI applications, but ultimately decided to use a module called PYO [1].

PYO [1] is a Python package for Digital Signal Processing (DSP) written in C. Due to the nature of Python being an interpreted language, it is very slow when compared to lower-level languages such as C. PYO contains a Python interface, but the actual operations are already compiled with C, making it fast enough to process audio signals efficiently. It contains a built-in MIDI interface and a primitive GUI, making it a good fit for this application. The main drawback with PYO is that the instrument sounds must be designed using the provided functions, which might be very difficult to someone inexperienced with synthesizers and effect chains.

We were able to implement the application with PYO relatively quickly. The program loop itself is run using a class called *Pattern*, which executes a callback function at a constant time interval. The interval is set to the duration of a 16th note in the tempo the user is playing and calls the models to generate music. The chord model is called once every 16 timesteps to generate the next chord, and the polyphonic model is called every timestep to generate the next timestep of polyphonic music. This method was tested at a very slow tempo and seemed to work relatively well. The model showed clear signs of adapting to what the user was playing on a MIDI keyboard, but it was not possible to evaluate the long-term structure of the live accompaniment. When ran at higher speeds, the program drastically slowed down causing lag and stuttering. By testing the performance of all parts of the process, the bottleneck was found to be the model prediction, which was too slow, taking up to 1 second to predict a single timestep.

This occurs because Keras is a high-level API intended for rapid development and is very slow with such a large network. This is one of the limitations of using Keras to implement ML models. It is not really made for being able to run predictions live. However, if a different ML framework were used for implementing the model, it would not have been possible to complete the implementation in the timeframe of this project. Instead of evaluating the live model, we can evaluate the polyphonic accompaniment generated to static melodies, providing a good view of what issues the model might have regarding musical coherency and structure.

The speed performance is definitely something that could be fixed in a later iteration of the project, and we will recommend exploring alternatives to Keras as well as testing the application using different hardware for future work.

Chapter 6

Experiments and results

In this chapter the performance of the proposed model is analysed both objectively through measured metrics, and subjectively through musical analysis.

6.1 Objective evaluation

It can be useful to measure objective evaluation metrics to evaluate whether the model is able to generalize. While the generated output is not supposed to be an exact copy of already existing music, objective measurements can tell a lot about how well the model is learning from the data. Some of the most used metrics for evaluating model performance are accuracy, precision, and recall. The three are closely linked together, and describe how many predictions were correct, and what portion of predictions were true positives and true negatives.

Using these measures, we can get a general overview of how well the model is able to discover patterns in the data. It will not say anything about whether the music generated by the model sounds good to human ears, but how similar it is to the test data. In order to evaluate the model objectively, the tests are performed on the test dataset, which has not been used at all prior to the final testing.

Additionally, it is useful to provide some sort of baseline to compare our model to. For this purpose, we created a more straight-forward LSTM network that takes only a melody as input, and outputs polyphonic accompaniment, skipping the entire step of chord extraction and embedding.

During training, the compound model got a validation accuracy of about 8%, and the baseline model got about 7%. This is very low, however, as the output size of both models is 204, this is much better than using randomized weights. In most machine learning problems, it is expected to get a far higher accuracy than 8%, but this is quite good for these models because of the large output sizes. Additionally, this accuracy metric is selected automatically by Keras, and does not really reflect how accurate the model actually is. It does, however, allow us to see when the model starts overfitting.

6.1.1 Results

The models were tested using the dedicated test dataset after the model was completed. The testing was performed by running the model for each timestep, performing a binary threshold selection, and comparing the result with the target output. This was done for both the compound model and the baseline model. Using this method, the script recorded the number of *true positives*, *true negatives*, *false positives*, and *false negatives*. These values were used to create a confusion matrix and calculate the accuracy, precision, and recall, which all say different things about the predictions. The following tables contain the results of the objective testing.

	True (prediction)	False (prediction)
True (ground truth)	19 759 965	105 488 968
False (ground truth)	40 824 209	3 003 195 786

(a) Confusion matrix for compound model

	True (prediction)	False (prediction)
True (ground truth)	22 282 393	136 097 047
False (ground truth)	38 301 781	2 972 587 707

(b) Confusion matrix for baseline model

	acc.	prec.	rec.
compound	95.4%	15.8%	32.6%
baseline	94.5%	14.1%	36.8%

(c) Results of objective evaluation.

Table 6.1: Objective evaluation results.

Because the accuracy was calculated differently than during training, we got a much higher accuracy than the validation accuracy. Using this method the model gets a very high accuracy, however, this is likely because at most of the timesteps, nearly all the notes are not activated, which is also true for the target outputs. This way, the model gets a huge number of true negatives, which are not as interesting as the true positives, of which there are relatively few. The results from the objective tests show that the compound model and the baseline model perform nearly identically. This is somewhat surprising, but because music is so diverse and complex to model, this does not say anything about how the music actually sounds to the human ear. Based on these results, we cannot say for certain that the models perform differently at all in the objective test, but the tests could be run several times to see whether the differences are statistically significant. However, with the current setup this would take several weeks.

Figure 6.2 shows the distribution of the notes generated by both the compound model and the baseline model. The most commonly occurring notes are severely over-represented. This effect is more severe in the compound model than the baseline model.

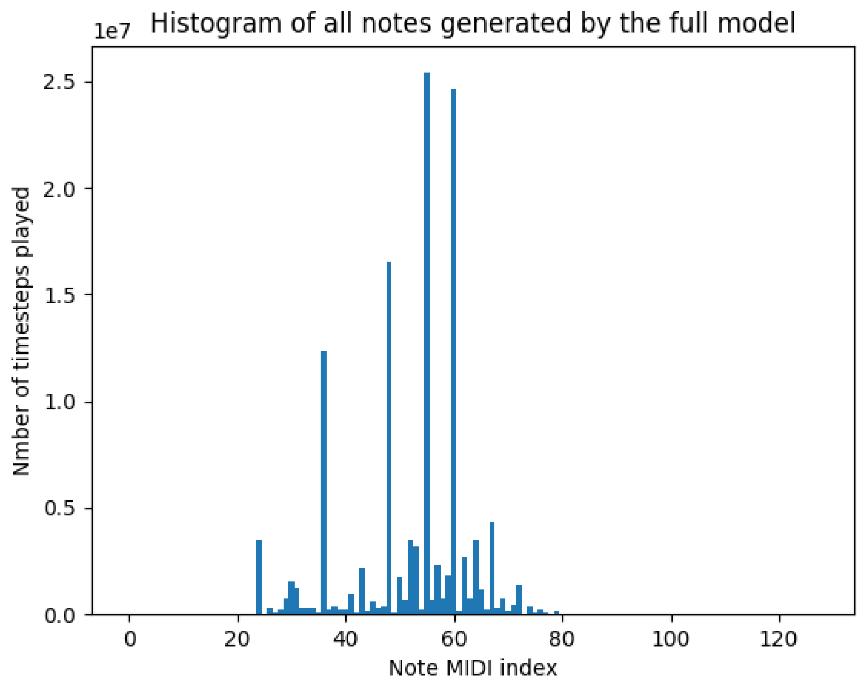


Figure 6.1: Distribution of the notes generated by the compound model. A few notes are severely over-represented in the generated music, more so than in the original data (figure 5.6).

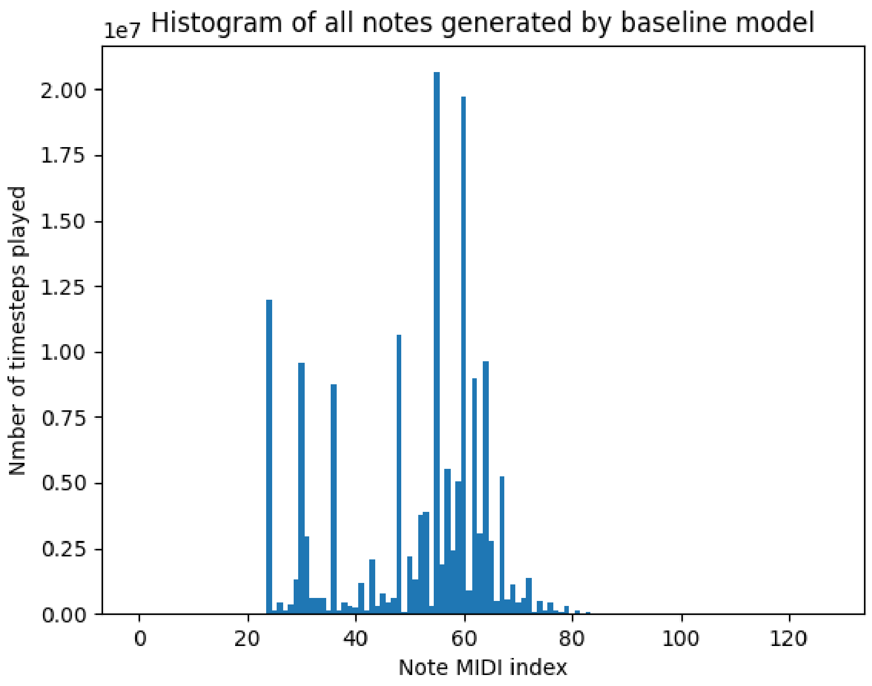


Figure 6.2: Distribution of the notes generated by the baseline model. The most common notes are overrepresented, but less so than in the compound model.

6.1.2 Reproducibility

To be able to reproduce the results of the experiments in this thesis, all experiments are run with the same seed and the same versions of the packages containing randomness. Due to the nature of the CUDA interface used by TensorFlow, it might not be possible to reproduce the results exactly, but steps have been taken to ensure that the results are as reproducible as possible.

6.2 Musical analysis

One method of subjectively evaluating the model's musical performance is through musical analysis. Originally, we had intended to perform a user study, letting musicians and non-musicians play with the application and give feedback through a survey, but as the interactive application does not currently work at full speed, this was not possible. Therefore, in order to get some subjective evaluation of the compound model's performance, we have performed some musical analysis on three examples generated by the model. These examples consist of generated polyphonic accompaniment to static melodies. The accompaniment was generated as if the melodies were played live, but the melodies were predetermined, and could not adapt to what the model was outputting. The samples were hand-picked after listening to several examples of accompaniment generated by the model. We have tried to select melodies of different styles and tonalities, in order to see how the model performs on different playing styles. Additionally, we generated accompaniment to all three selected melodies with the baseline model in order to compare it subjectively with the compound model.

The music we analysed is available on Google Drive through this link: https://drive.google.com/drive/folders/1cq6r-5Q9YFQb8jlWyza0bp9oU MbWC0rh?usp=sharing

The MP3 files were exported using the open source music notation software MuseScore [30], and the MIDI instrument sounds are from that software. The input melody is played by an organ sound, and the instruments generated by the models are played by their respective instrument sounds. As these are the only MIDI instruments we currently have available, we were not able to make the instruments sound more realistic. We did not include the sheet music, as it would require a great amount of manual editing to make it look presentable, and the reader is not expected to be able to read sheet music. If the reader wants to view the raw sheet music, the MIDI files included in the Google Drive folder can be opened with MuseScore or a similar program.

In these analyses we mainly focused on the functional harmonic and rhythmic aspects of the music, and we only used the output of the polyphonic network. Although the chord network generates a chord progression, this is not necessarily reflected in the outputted music, so it makes more sense to see what chords are actually playing by looking at the notes that are played rather than the chord generated by the chord network. We analysed the music completely unrefined, just as outputted by the model, by listening to the music and by looking at the sheet music generated from MIDI by MuseScore. We looked at the melodies from the model's point of view, ignoring the original melodies' actual scales and time signatures. As the model does not predict dynamic features like note velocity, onset, or length, this was not considered in the musical analysis. The music generated is relatively static and straight-forward, and the dynamic variances are limited to how many notes the instruments play simultaneously.

In general, the piano and guitar were the most successful parts of the accompaniment. In the examples analysed here, the bass track mostly played only C and nothing else, sometimes in rhythms fitting the melody, and the drum track only had a few hits on different drums.

Overall, we listened to generated music for about 50 different input melodies before selecting three of the most interesting examples. The accompaniment generated by the model was generally monotonous, often staying on the C chord and playing the notes in that chord. This is also supported by figure 6.2, which shows that a few of the notes are severely overrepresented in the output accompaniment. The following musical analysis focuses on the more interesting passages generated by the model and is not meant to be an accurate general representation of the accompaniment generated by the model.

6.2.1 Sample 1

The first example is a melody that seems to use the natural minor scale. The rhythm in the melody track does not fit the $\frac{4}{4}$ time signature, because the original song is in $\frac{3}{4}$ time. The main issue with this is that the rhythmic properties of the melody might not make much sense. However, the harmonies generated by the model for this track were very interesting. Listening to the generated music itself, our first thought is that it sounds somewhat chaotic and dissonant. However, it does not at all sound randomized, and it is clear that the generated accompaniment tries to follow the melody. The model struggles to understand that the melody is in minor, and often plays unfitting major chords. This is likely because most of the music the model was trained on is in major, and it has not properly learned the differences between the scales. It is not really a large issue in this song, as most of the places where the major chord occurs, it does not really crash with the melody, but fits in a way that is a bit nontraditional for western music.

In the first eight bars, it is clear that the model struggles to find the right tones, and mostly stays on a C major chord, which does not fit well with the melody in C minor. However, in the 14th bar, at about 0:26 in the MP3, the model changes to a more advanced progression as the melody lands on a long Eb. The chord sequence played by the model looks like Am, G, Fm, G, Em and Cm. Apart from the Em chord, which does not fit in a functional harmonics perspective, this chord progression works very well and utilizes the leading tones of the 6th, subdominant, and dominant chords, and then back to the tonic. After this, the accompaniment alternates between Cm and C for a while, before there is a long pause in the melody, in which the accompaniment plays a rhythm in C. When the melody comes back in, the piano surprisingly plays the melody as well, before harmonizing in thirds. After this, the accompaniment settles on a C major chord, and does not play anything else. It plays some different rhythms, and the piano experiments a bit with adding some more harmonic tones, but it gets boring after this point.

This example is interesting because the model was able to generate at least one very interesting chord progression, and the piano was actually able to follow the melodic line really well for a while. However, because the rhythm in the input melody is in a different time signature, it is difficult to say anything about the rhythm of the accompaniment.

6.2.2 Sample 2

The next melody is in C major, and the original song is in $\frac{4}{4}$, so the rhythms are correct. The notes in this melody are all in the lower register, which is interesting because those notes were less represented in the dataset. Like in the first example, the model seems to struggle initially, and mostly plays the notes of a C chord. Eventually, after bar 9, about 18 seconds into the MP3, it starts to experiment with the F chord, alternating between those two chords following the melody. Then, in bar 17, at about 0:35 in the MP3, it plays what looks like Dm, Em, and F, and repeats this pattern along with the melody, and goes back to C as the melody goes up. After this, the accompaniment continues to alter between C, F and G for a few bars, fitting the melody very well. Then, however, the accompaniment seemingly descends into chaos, playing more and more notes and clustered chords that do not fit the melody. It is still possible to hear that it mostly plays the same chords, but it adds a lot of notes that sound dissonant.

After the music has been chaotic for a while, in bar 67, at 2:10 in the MP3, it suddenly modulates to C natural minor, playing Abm and Gm, which fits well for the melody here, and adds some extra tension to the music. When the melody goes back to C major, the model struggles to follow, but eventually lands back in the original scale. After this the music sounds chaotic and too busy for the rest of the song, because too many notes are being played at once. The model seems to alternate between C major and C minor, which in our opinion sounds good, but this is unusual in western music.

6.2.3 Sample 3

In this example, the guitar plays a significant rhythmical role, and the piano was able to create a harmony with the lead melody. Interestingly, this melody is actually in the neighbouring scale F major, which musically is very similar to C major. F and C major are next to each other in the circle of fifths, and share all notes except B, which is flat in F major. Apparently, the method used to extract scales has gotten the scale in this song wrong and modulated the song to the neighbouring scale instead. The reason we use this example is that it is interesting to see what the model does when the neighbouring scale is used, and whether it has been able to extract the relationships of neighbouring scales. Because the melody is in F major, the dominant chord C is used very frequently in this song, which fits well with the melody.

In the beginning of the song, the piano quickly begins to play along with the melody, as well as playing some accompanying notes. The guitar also joins, playing rhythmically over a C chord. In bar 9, at 0:16 in the MP3, there is a pause in the melody, and the piano stops playing. Here, the guitar plays a very interesting rhythm, making space for the incoming melody. The melody starts playing, and the piano immediately plays a very fitting harmony, along with playing a Gm chord and over to what can be interpreted as an F major. The guitar continues to play rhythmically over the same chords. After this point, it seems like there is originally a time signature change, so the melody gets skewed. The rhythm is less prominent after this point, but still present and able to fit the rhythm of the melody. After this point, the accompaniment mostly alternates between the C and F chords and gets relatively boring over time. Unlike the previous examples, the accompaniment did not get noisy and chaotic, but continued with the same amount of activity.

6.2.4 Baseline generated music

To see whether the compound model was better than a basic LSTM network for generating polyphonic accompaniment, the baseline model was used to generate music for the same samples that we already analysed. Listening to the samples, we quickly noticed that the performance of the baseline model is much worse than that of the compound model. The generated music generally does not fit the input melodies at all, other than a few places where the piano seemingly tries to play the same notes as the melody. The bass is only present in one of the songs, and the drums do not play anything at all. The piano and guitar play some chords and small melodies, but nothing that really fits the input melody.

In sample 1, the piano initially tries to play along with the melody, and does so relatively well, but after the first eight bars, it starts to play a lot of notes that do not fit the scale at all. For the rest of the song, the piano plays chaotically, mostly notes and chords that do not fit at all, and the rhythms do not seem to follow the melody in any way. The guitar plays a few notes during the song, but mostly stays quiet. The baseline model was not able to detect that the song was in the minor key or adapt to the rhythms of the melody in any way.

The second sample sounds a bit better, but it is still of far worse quality than the music generated by the compound model. For the first eight bars, the accompaniment played by the piano actually sounds quite pleasant, alternating between the chords C, F and G, which fit the melody really well. However, when the 9th bar arrives, the piano accompaniment promptly discards all musical coherence, and becomes unpleasant and chaotically dissonant. This continues for the rest of the piece, with some short moments where the notes coincidentally sound well with the melody.

The third sample was not really any better than the previous two. However, because the melody was in the neighbouring key of F major, the model was not confident enough to generate anything more than a few notes. The piano mostly tries to copy the input melody but plays some other notes that do not fit the scale at all. The guitar plays a few notes occasionally, but nearly only the same notes that are in the melody.

Chapter 7

Discussion

This chapter contains discussions about the results presented in the previous chapter and some of the issues regarding the practical use of machine learning techniques, as well as performance issues encountered in this project.

7.1 Experiment results

The experiments show that the compound model is capable of generating interesting and fitting passages accompanying melodies, however, the most common notes and chords are overrepresented, making most of the generated music somewhat monotonous. The most interesting samples, which were analysed in the previous chapter, showed clear signs that the model was able to make meaningful connections in the training data, which it used to generate relatively well-sounding music for piano and guitar.

The objective results with the selected metrics did not yield different results for the compound model and the baseline model. However, because the differences were very evident in the subjective analysis, this indicates that the selected metrics do not accurately represent the differences in the models. In future iterations of the project, it might be necessary to find other objective metrics to measure the model outputs to evaluate the performance of the compound model versus the baseline model.

In the subjective musical analysis, the baseline model performed much worse than the compound model. It is very clear that the model using chord embeddings and two different networks is able to extract features from the music better than just a basic LSTM network, which is as expected. The addition of considering the underlying chords without explicitly deciding what notes are associated with each chord helps the model generate much more meaningful and coherent music. Although the music generated by the compound model does not sound like it was human-made, and many of the generated notes do not fit the music very well, the experiment overall was very successful.

The bass and drum tracks generated by the model were not as successful as the piano and guitar, with only a few notes being played by these instruments in all the generated samples. This is likely because the bass and drums are mostly monophonic, as opposed to the piano and guitar, which plays many notes at once. Because of this, there is little data for each output value of these instruments. The bass and drums might have gotten much better results if they were generated by a model trained separately from the piano and guitar. This is something that could be tested in future work.

Listening to the first sample, which was in C natural minor, the model seems not to have been able to fully connect the differences in scales, so it plays many notes from the major scale even though the melody is clearly in minor. This is likely because the majority of the songs in the dataset are in major, while only a few songs are in minor. If the number of songs in major was reduced or there were more songs in minor, it might have been able to learn the differences better.

The second sample contained sections with many notes that sounded dissonant and chaotic. This might be reduced by limiting the number of notes that can be played simultaneously by each instrument, or maybe having a dynamic threshold that moves according to the overall confidence of the model.

There are a lot of adjustments that could be made to the model, such as fine-tuning learning parameters and the threshold value, and as suggested by Brunner et. al. [5], it would be possible to add more layers to the process, for example a network guiding the chord generation, helping with long-term structure on a larger scale. Similarly, it could also be possible to add another network on the lowest level, which could for example predict features like velocity, onset or note length, making the music more dynamic and interesting.

One fault that was observed in both the objective and subjective experiments, is that a few notes are severely overrepresented in the accompaniment generated by both models. Compared to the notes in the original dataset (figure 5.6), the generated notes (figure 6.2) are much less diverse; just a few notes (C and G) occur much more than any other notes. This most likely occurs because the classes (notes) are imbalanced in the original dataset, so the model, which only tries to minimize the loss function, finds that the loss is lowered when it predicts the most common class more often. There are ways to mitigate this, but this was not prioritized in this project. The overrepresentation of the most common notes occurs in both the compound model and the baseline model, but it looks somewhat more severe in the compound model. This likely occurs because the compound model also contains the chord predictions, and looking at figure 5.5, we see that the chord data is also severely imbalanced.

7.2 Performance

The main goal of this project was to create the basis for an application that generates live polyphonic accompaniment to a musician, and the solution was implemented in a way that would allow the application to work in practice. Due to the time constraints of the project, the main focus was to implement and train the model to generate accompaniment to a static melody, which was achieved. The model was implemented in such a way that it can generate accompaniment to a live player, however, due to performance issues it was only possible to test this at a very low tempo. Currently, using the compound model to generate music takes about 0.5 to 1 second per timestep, which is too slow for live music. It should be able to generate 16th notes with a tempo of at least 120 bpm, meaning each timestep would need to take less than 0.0625 seconds. Using a more powerful computer with access to CUDA graphics cards would probably work, but as this was not available during the project, it was not possible to test the live application at higher tempos.

This is mainly a limitation with Keras, as it is a high-level API, and higher levels of abstraction often comes with a performance penalty. Keras does a lot of the complicated transformations, conversions, and parameter selections automatically, which is great for reducing implementation time, but overall, usually results in a sub-optimal model. Additionally, according to our own observations, the official Keras documentation does little to guide users to actually use the models after they are trained, which makes it difficult to employ ML models to a production environment. It seems to us that TensorFlow and Keras are more geared towards creating and testing models for research purposes rather than creating practical models to be used in real life systems. Because machine learning is such a hot topic of research, it makes sense that this side of the API gets more attention. We hope that in future updates, Keras and TensorFlow focus more on the deployment of models, as machine learning becomes a more practical and production-oriented field.

7.3 Practical use of machine learning

As this thesis is focused much more on the practical implementation of deep learning techniques rather than the theory and mathematics behind them, it is natural to discuss the experience of implementing and using deep learning for practical purposes as a software developer. Although machine learning techniques have been explored and researched for more than 50 years, it has mainly been theoretical and experimental. The actual usage of machine learning techniques has not until quite recently been feasible in real projects due to the amount of domain knowledge and time required to implement these techniques, and the processing power required to efficiently apply deep learning in real situations. For a software developer without specialization in machine learning algorithms, it would be impossible to implement a complex machine learning model in a real project without the use of an ML framework. During the last 20 years, increasing interest in machine learning and better access to computational resources has seen the development of several deep learning frameworks designed to be easy to use for non-experts in the field. They still require some knowledge of how machine learning works and what types of models fit what types of problems, but there is no need to have expert knowledge of how deep learning works on a mathematical level.

This project and most other similar projects have used Keras, which runs on top of TensorFlow. Keras allows developers to create and test prototypes that would otherwise have taken very long to implement. It is usually used with Python, a very high-level programming language, and uses a class-based API to make implementing, adapting, and specializing a large number of different techniques feasible for real projects. In this project, having limited prior experience with machine learning, there was a steep learning curve and much to take into consideration, which made the development of the models take a lot of time. If we had chosen any other deep learning framework, we would not have been able to implement and test the models within the timeframe of this project. In our opinion, this makes up for the limitation in performance speed, because if we used a faster framework, there would likely not be any models to test yet. Compared to PyTorch, there is much less the developer needs to consider, and a lot of the complicated time-consuming elements are handled automatically by Keras. However, using TensorFlow without Keras would likely be much more complicated than using PyTorch.

One way to mitigate the performance issues caused by Keras could be to create a different model using a technique known as Knowledge Distillation, [17]. This technique might work well in this scenario, but it would likely have to be two models, one for each of the networks. A simpler model could not learn the complex relations between different chords and the relations between the chords and notes without including the chord step. As seen by the baseline model, excluding the chords in the network makes the music generated by the model much less meaningful. The chord embedding is able to extract important concepts from the music and utilize that to generate better music.

7.4 Machine-generated music that matters

Among literature describing methods for machine generated music, there are few that have a clear goal for what they want to achieve with machine generated music. It seems like the majority of ML models made to generate music have been made only to see whether it is possible, and not whether it can be used for anything. Only a few authors such as Benetatos et al. [2], have a clear reason for why they are creating the model. Benetatos et al. [2] state that their system "allows a human musician to improvise a duet counterpoint with a computer agent in real time", which is very similar to the goal of this project. They also argue that improvisation is a useful tool for musicians, and that classical musicians like Bach practiced improvisation regularly. However, counterpoint duets are a somewhat simpler form of improvisation, in which two melodies with equal prominence are adapting to each other. In this project, the goal was to generate musical accompaniment to one lead player.

A case study that tackles some of the issues with current ML music research was written by Strum et al. [42], which refers to Wagstaff's article about "ML that matters" [44], showing that this is not only an issue in machine learning for music, but also for other areas where ML is applied. Wagstaff [44] points at common issues in machine learning research, such as lack of follow-through, and the hyper-focus on abstract evaluation metrics that say nothing about the real-world impact or usefulness of the models. These issues detach much of the research from the real-world issues they attempt to solve, and the results are rarely communicated back to the actual problem domain. This article is 10 years old and has been cited several hundred times, and a lot of machine learning papers published since then have focused more on the practical impact of the research. However, this does not seem to be the case in the domain of machine generated music.

Strum et al. [42] used two principles derived from Wagstaff [44] to practically measure the impact and quality of several models for generating music, by holding a concert with music generated by these models and getting feedback from both the musicians performing and co-composing the music, and the audience. The feedback was used to evaluate in what ways the models can be used efficiently and propose improvements to the models and their use. Their results are very interesting, and this approach seems like a good way to evaluate the efficiency of ML-based music generators.

Initially, we had intended to perform a user study in which musicians of different skill levels could test the application live and play interactively with it. Because of performance issues and time constraints, this was not possible in this project, but it could be done in the future. Based on the experiment by Strum et al. [42], we would also suggest having musicians play with the application regularly over time and give feedback about how it impacts their improvisational skills. This could bring very valuable information about how the solution works in practice, and what improvements are required for it to be a useful tool for musicians.

7.5 Machine-generated art

The philosophical question of whether a machine can create art is very interesting, and in our opinion worth discussing in order to understand the purpose of artistic expression and impression (not to be confused with the artistic movements called "expressionism" and "impressionism"). By artistic expression we mean the message that the artist is conveying through their art, and by impression, we mean the message that the receiver interprets. In most art forms, these do not have to be the same, and in many cases, they are not meant to be the same either. The listener's life experience might be completely different from that of the artist, and so the music is interpreted in a completely different way. This does not mean that the listener is hearing it wrong, but that the same music gives them a different impression. Neither does it mean that the artist expressed themselves inaccurately, the message was just interpreted differently than the artist intended.

While this is not a thesis about the philosophy about art and music, we want to draw some attention to the topic of machine-generated art. The term "music" has already been defined in the context of this thesis, and according to this definition, music does not necessarily have to be art. Some might perceive this as an issue, but it is true for music created by humans as well. Again, this all depends on definitions, and "art" is not an easy term to define. There have been attempts at structuring and partially answering this conversation. In Mark Coeckelbergh's article about this topic [8], he splits the question of whether machines can create art into three different parts, looking at what is meant by "creating", what is meant by "art", and what is meant by machine-creation in the context of art, especially whether some or all music created with the help of digital tools fall into this category.

These are all complex questions that can be interpreted many different ways, and Coeckelbergh [8] gives examples and arguments for and against many different viewpoints, before concluding that the question of whether machines can create art itself is misleading, because our experiences and culture might play a significant role in how we interpret the question, and it assumes that there is some sort of disconnect between machine and human, which is not necessarily true. Saying that humans, who we know can create art, are not machines, implies that our consciousness exists on a higher plane, and is not just a bio-chemical phenomenon. Although we may be able to answer this question in thousands of years from now, this is currently purely a question of belief and speculation and is viewed very differently in different cultures and religions.

The issue with assigning the concept of art a human-independent definition is that art is generally considered an entirely human concept and is mainly used to describe methods of human expression. Art forms are complex forms of communication, often used to express feelings and concepts that are too complex to describe with words alone. Art is also highly subjective, and what some might define as art might not be art to someone else. Going back to the ideas of expression and impression, these are equally important in art. A painting hated by everyone except the artist is still art, just as an accidental paint spill on the sidewalk might be considered art by some pedestrian. This is obviously a very broad and open definition of art, and it is not really useful for the purpose of telling whether something is art or not, because the answer will always be "yes" or "maybe". However, it might help in answering the question of whether a machine can *create* art.

If we consider accidental or unintentional art, this is art that was not *created* with the intention of being art, but one or several persons saw it and associated it with a specific set of feelings or experiences, and called it art. Someone might even take a picture and post it in an online gallery or print and hang it on their wall. The person that took the image considers this an art piece, but there is no expression associated with it, as it was completely unintentional. The only thing defining it as art is the impression it made on the person that found it. In such a case, one might argue that this person is the artist, as they are the one that found it and gave it artistic meaning. The image of a paint spill at the wall in their apartment is an art piece created by them. However, it is not the image that is considered art, it is the spill itself. If the paint spill is the art piece, it was created by a non-artist, and became art only after it was created. In this line of thinking, one might say that a machine cannot intentionally *create* art, but a piece created by a machine can be considered art nonetheless.

We also want to discuss the notion that not all music has to be art. This is more apparent in other art forms such as painting, architecture, and carpentry. Painting is used to create complex visual representations of emotions and thoughts, but it is also used to protect buildings from the forces of nature. The purpose of a painted wall is often purely practical. Similarly, a building or a table can serve as purely practical objects, but architecture and carpentry can also be artistic, sometimes even sacrificing the usefulness of the object for the purpose of artistic expression. This might also apply to music, but in a somewhat different way. Music is not a tangible object with a practical purpose like buildings and furniture, but it still has purposes other than pure artistry.

The ways in which we listen to music is often of a more practical nature than recreational, as many people for example listen to music while writing or working to help them focus and not get distracted by external sounds or irrelevant thoughts. Music is also used to coordinate manual labour that requires cooperation and synchronicity, as well as raise morale among workers. For example in USA, there have been recorded many such "work songs" created by slave workers in the 19th and early 20th century, which were used to coordinate work and relieve boredom while working [12]. While these songs have been recorded and are considered artistic today, their original purpose was much more practical than artistic. Another example is the sea shanties used by sailors around the same time to coordinate complex manual tasks on board large sea vessels [46]. These types of music have had a substantial impact on musical traditions in the past century. Music made by slave workers formed the basis of the entire blues genre and its branches, and sea shanties have been used in television, video games, and social media.

Our views on art and music change throughout history, and facilitating the expansion of our ability to communicate complex ideas through art is important. Artificial intelligence and other digital tools might allow us to unlock new forms of communication and cooperation in the future. Our ability to practice, create and share music has drastically improved with the continued use of digital tools, and the variety of music available to us is ever increasing. AI-based tools may become very important to musicians and other artists in the future. Like all other major changes in the history of art, this is a natural and inevitable development, that in the future, will most likely be looked at as an improvement for all artistic expression.

Chapter 8

Conclusion

8.1 Conclusions

The goal of this thesis was to answer the question "*Is it possible to create a solution for generating live, interactive musical accompaniment using machine learning techniques, that can be used by musicians to practice improvisation and interplay?*". The project was focused on experimentation and practical implementation, with the objective of implementing and testing a working application, which was partially achieved.

The implemented solution is based on a two-network model designed by Brunner et al. [5], which was adapted to create polyphonic multi-track accompaniment to a melody rather than generate music from scratch. This compound model uses two LSTM networks; the chord network, which trains an embedding layer, a 1-dimensional convolution, and an LSTM layer to generate a chord progression to the melody, and the polyphonic network, which uses the embedding trained by the chord network, as well as a larger LSTM layer to generate polyphonic accompaniment for four instruments. This model was found to generate a few interesting musical passages to static input melodies and showed signs of adapting to live input when tested interactively at a very low tempo. However, much of the music generated by the model became static and monotonous, which likely occurs because a few of the classes are over-represented in the training data.

The compound model was compared with a much simpler baseline LSTM model, which despite performing similarly in objective measurements, generated much less functional and pleasant music compared to the compound model, according to our own musical analysis.

The results of this thesis indicate that the use of LSTMs for interactive music generation is feasible and forms the basis for further development of an ML-based approach to generating live interactive music accompaniment. Such an application could be a valuable resource to musicians wanting to practice musical interplay and improvisational skills in a way that is non-repetitive and intriguing.

8.2 Future work

This section contains ideas and suggestions for further work based on the work presented in this thesis.

8.2.1 Speed performance

As the scope of this thesis had to be limited to a realistic amount of work, we were not able to fully implement and test the application interactively. Other than implementing a suitable user interface, the last challenge to get the program to generate musical accompaniment interactively is the model's speed performance. Keras is a high-level API, which makes it slower than other methods. As it uses TensorFlow backend, it is still much faster than native Python, but there is a lot of unnecessary overhead, and Keras is not really built for using models in Python applications. In the future, the models could be implemented using a different, lower-level framework, which will most likely perform better.

8.2.2 Knowledge distillation

One way of improving model performance is to use a technique known as Knowledge Distillation (KD). This involves transferring knowledge from a larger deep learning model to a smaller model more suitable for deployment. As explained by Hinton et al. [17], a large model is trained to predict well on the training dataset, while the goal of the training process is to generalize to new data. It is not possible to train a model to generalize directly, because that information is usually unavailable. However, it is possible to train a smaller model to generalize similarly to a larger model, which will give better results than training the small model on the training data. Using this strategy it might be possible to create a simpler, distilled model that generates musical accompaniment.

8.2.3 Qualitative studies

If the performance issues were tackled, it would be interesting to see qualitative studies on the use of the solution for the intended purpose of practicing improvisation and musical interplay. Computer-assisted practice might be one of the next great steps in digital musical tools, and this could generate feedback and insights that could be very valuable in developing such tools.

8.2.4 Network structures

It could also be useful to experiment with different network structures, by for example adding an extra LSTM layer in the networks and spending more time tuning the model parameters. This could have a large impact on the music generated by the model. As the proposed solution is largely based on the JamBot model [5] and indicates that the two-model structure with a chord generator and a polyphonic generator performs better than the baseline LSTM for generating polyphonic accompaniment, varying the structure by adding or removing network layers could have positive effects. However, this would also likely make the model slower, so the speed performance issues must be resolved first.

8.2.5 Imbalanced data

The main issue of the generated music is that the most commonly occurring notes are severely overrepresented. This likely occurs because the classes are imbalanced in the training data, which is a normal issue in machine learning. There are a few common techniques that are used to tackle this issue, such as reducing the amount of data with the most common classes, but this could also make the music less functional, as the reason that these classes are overrepresented is that they have a very high importance in the music.

It could be possible to make some sort of probability selection for the polyphonic output based on the distribution of notes in the original dataset, as well as limit the number of notes that can be played simultaneously by each instrument. This might make the model less monotonous while still being able to generate functional and coherent music. The addition of a third model looking at the larger structure of the music, as suggested by Brunner et al. [5], might also help the chord model generate more meaningful and varied chord progressions, leading to more diverse music.

Bibliography

[1] Olivier Belanger. 'Pyo, the Python DSP Toolbox'. In: *Proceedings of the 24th ACM International Conference on Multimedia*. MM '16. Amsterdam, The Netherlands: Association for Computing Machinery, 2016, pp. 1214–1217. ISBN: 9781450336031. DOI: 10.1145/2964284.2973804. URL: https://doi.org/10.1145/2964284.2973804.

[2] Christodoulos Benetatos, Joseph VanderStel and Zhiyao Duan. 'BachDuet: A Deep Learning System for Human-Machine Counterpoint Improvisation'. In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Zenodo, June 2020, pp. 635–640. DOI: 10.5281/zenodo.4813234.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent and Christian Janvin. 'A Neural Probabilistic Language Model'. In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[4] Jean-Pierre Briot, Gaetan Hadjeres and Francois-David Pachet. *Deep Learning Techniques for Music Generation*. 1st. Springer Publishing Company, Incorporated, 2020. ISBN: 978-3-319-70163-9. DOI: 10.1007/978-3-319-70163-9.

[5] Gino Brunner, Yuyi Wang, Roger Wattenhofer and Jonas Wiesendanger. *JamBot: Music Theory Aware Chord Based Generation of Polyphonic Music with LSTMs*. 2017. arXiv: 1711.07682 [cs.SD].

[6] John Burgoyne, Jonathan Wild and Ichiro Fujinaga. 'An Expert Ground Truth Set for Audio Chord Recognition and Music Analysis'. In: *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011* (Jan. 2011), pp. 633–638.

[7] François Chollet et al. *Keras*. https://keras.io. 2015.

[8] Mark Coeckelbergh. 'Can Machines Create Art?' In: *Philosophy & Technology* 30.3 (Sept. 2017), pp. 285–303. ISSN: 2210-5441. DOI: 10.1007/s13347-016-0231-5. URL: https://doi.org/10.1007/s13347-016-0231-5.

[9] R. Collobert, K. Kavukcuoglu and C. Farabet. 'Torch7: A Matlab-like Environment for Machine Learning'. In: *BigLearn, NIPS Workshop*. 2011.

[10] Florian Colombo and Wulfram Gerstner. *BachProp: Learning to Compose Music in Multiple Styles*. 2018. arXiv: 1802.05162 [cs.SD].

[11] D. E. Comer et al. 'Computing as a Discipline'. In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[12] The Library of Congress. *Traditional work songs*. URL: https://www.loc.gov/collections/songs-of-america/articles-and-essays/musical-styles/traditional-and-ethnic/traditional-work-songs/.

[13] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang and Yi-Hsuan Yang. *MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment*. 2017. DOI: 10.48550/ARXIV.1709.06298. URL: https://arxiv.org/abs/1709.06298.

[14] Andrei Faitas, Synne Engdahl Baumann, Torgrim Rudland Naess, Jim Torresen and Charles Patrick Martin. 'Generating Convincing Harmony Parts with Simple Long Short-Term Memory Networks'. In: *Proceedings of the International Conference on New Interfaces for Musical Expression* (Porto Alegre, Brazil). Zenodo, June 2019, pp. 325–330. DOI: 10.5281/zenodo.3672980. URL: https://doi.org/10.5281/zenodo.3672980.

[15] C. Garoufis, A. Zlatintsi and P. Maragos. 'An LSTM-Based Dynamic Chord Progression Generation System for Interactive Music Performance'. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, pp. 4502–4506. DOI: 10.1109/ICASSP40776.2020.9053992.

[16] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[17] Geoffrey Hinton, Oriol Vinyals and Jeffrey Dean. 'Distilling the Knowledge in a Neural Network'. In: *NIPS Deep Learning and Representation Learning Workshop*. 2015. URL: http://arxiv.org/abs/1503.02531.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-term Memory'. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[19] J. D. Hunter. 'Matplotlib: A 2D graphics environment'. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[20] Nan Jiang, Sheng Jin, Zhiyao Duan and Changshui Zhang. 'RL-Duet: Online Music Accompaniment Generation Using Deep Reinforcement Learning'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.01 (Apr. 2020), pp. 710–718. DOI: 10.1609/aaai.v34i01.5413. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5413.

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[22] Jens Kober, J. Andrew Bagnell and Jan Peters. 'Reinforcement learning in robotics: A survey'. In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. DOI: 10.1177/0278364913495721. eprint: https://doi.org/10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721.

[23] Steven G Laitz. *The complete musician: an integrated approach to theory, analysis, and listening*. 4th ed. New York, NY: Oxford University Press, Feb. 2016.

[24] Eui Chul Lee and Min Woo Park. 'Music chord recommendation of self composed melodic lines for making instrumental sound'. In: *Multimedia Tools and Applications* 76.16 (Aug. 2017), pp. 17255–17271. ISSN: 1573-7721. DOI: 10.1007/s11042-016-3984-z. URL: https://doi.org/10.1007/s11042-016-3984-z.

[25] Hyungui Lim, Seungyeon Rhyu and Kyogu Lee. *Chord Generation from Symbolic Melody Using BLSTM Networks*. 2017. DOI: 10.48550/ARXIV.1712.01011. URL: https://arxiv.org/abs/1712.01011.

[26] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. 2nd. Chapman & Hall/CRC, 2014. ISBN: 1466583282.

[27] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[28] Mark McGrain. *Music Notation*. en. Berklee guide. Milwaukee, WI: Hal Leonard Corporation, Aug. 1991.

[29] Merriam-Webster. 'music'. In: *Merriam-Webster.com dictionary*. URL: https://www.merriam-webster.com/dictionary/music.

[30] *MuseScore*. https://github.com/musescore/MuseScore.

[31] *Our technologies*. URL: https://www.steinberg.net/technology/.

[32] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[33] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[34] Colin Raffel. 'Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching'. PhD Thesis. Colombia University, 2016. DOI: 10.7916/D8N58MHV.

[35] Colin Raffel and Daniel P. W. Ellis. 'Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi'. In: *15th International Conference on Music Information Retrieval Late Breaking and Demo Papers* (2014).

[36] F. Rosenblatt. *The perceptron - A perceiving and recognizing automaton*. Tech. rep. 85-460-1. Ithaca, New York: Cornell Aeronautical Laboratory, Jan. 1957. DOI: 10.1037/h0042519.

[37] David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams. 'Learning representations by back-propagating errors'. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[38] Pete Shinners. *PyGame*. http://pygame.org/.

[39] David Silver et al. 'A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play'. In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[40] Dave Smith and Chet Wood. 'The 'USI', or Universal Synthesizer Interface'. In: *Journal of the Audio Engineering Society* (Oct. 1981).

[41] Raw Material Software. *JUCE source code*. https://github.com/juce-framework/JUCE.

[42] Bob L. Sturm et al. 'Machine learning research that matters for music creation: A case study'. In: *Journal of New Music Research* 48.1 (2019), pp. 36–55. DOI: 10.1080/09298215.2018.1515233.

[43] Theano Development Team. 'Theano: A Python framework for fast computation of mathematical expressions'. In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[44] Kiri Wagstaff. 'Machine learning that matters'. In: *Proceedings of the Twenty-Ninth International Conference on Machine Learning (ICML)*. ArXiv, June 2012, pp. 529–536. DOI: 10.48550/arXiv.1206.4656.

[45] Wikipedia contributors. *Circle of fifths — Wikipedia, The Free Encyclopedia*. [Online; accessed 05-May-2022]. 2022. URL: https://en.wikipedia.org/wiki/Circle_of_fifths.

[46] Stephen Winick. *A Deep Dive Into Sea Shanties*. Jan. 2021. URL: https://blogs.loc.gov/folklife/2021/01/a-deep-dive-into-sea-shanties/.

[47] Aston Zhang, Zachary C. Lipton, Mu Li and Alexander J. Smola. *Dive into Deep Learning*. https://d2l.ai. 2020.