I hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of Dr Marc Roper.

Applying machine learning techniques to the classification of classical orchestral music

R. David Dunphy

Supervisor: Dr Marc Roper
Second Assessor: Dr Paul Murray

28th March 2018

Abstract

Music information retrieval is a growing subfield of machine learning, within which music genre classification is one of the most fundamental tasks. Due to the subjective nature of genres, music classification is a non-trivial task that has been solved only imperfectly to date.

This project investigates two neural network approaches to this problem with a view to classifying classical orchestral music into the four genres of Baroque, Classical, Romantic, and Modern, which are informally considered the major subdivisions of the past four centuries of music history.

A dataset comprised of 40 h of audio samples was assembled. This dataset employs a hierarchical classification, with the four main categories subdivided into three composers each.

Using convolutional neural networks on spectrograms of this data, accuracies of around 92% were obtained for a binary classification. However, this approach was too memory-intensive to be extended to the full corpus.

The alternative method was the use of a simple recurrent neural network trained on chroma features, which circumvented this difficulty. Under this paradigm, accuracies of 55% were achieved on a four-way classification of the full corpus. This algorithm was also trained to identify composers, yielding a 31% accuracy for a twelve-way classification.

Particular attention was paid to the effects of varying parameters of input data as well as hyper-parameters of the neural networks. It was found that transposition of features was an effective technique for reducing overfitting by artificially increasing the size of the dataset.

Acknowledgements

My thanks go to Dr Marc Roper for his support and advice throughout the year. Thank you also to Lorna, Rhona, and Matt for their efforts to hunt down typos.

Contents

List of Figures v
List of Tables vi
List of Code Listings vii

1 Introduction **1**
1.1 Project Specification 1
1.2 Report Outline 2

2 Background **3**
2.1 Music History 3
2.1.1 Compositional periods and their boundaries 3
2.2 Human Music Classification 7
2.2.1 Musical features used for classification 7
2.2.2 Human limitations in music perception 9
2.3 Supervised Learning 10
2.3.1 Convolutional neural networks 12
2.3.2 Recurrent neural networks 13
2.3.3 Gradient descent and backpropagation 14
2.4 Digital Signal Processing 15
2.4.1 Fourier transforms and spectrograms 15
2.4.2 Chromagrams and HPCP 16
2.5 Related Work 18

3 Methodology **20**
3.1 Corpus Assembly 20
3.1.1 Breakdown of audio source files 21
3.1.2 Final corpus selection 23
3.2 Preprocessing 26
3.2.1 Segment length 26
3.2.2 Sample rate 27
3.2.3 Transposition 27
3.3 Models and Features 27

3.3.1 CNN with spectrogram 28
3.3.2 LSTM with chroma 28

4 Experimental Design and Implementation 30
4.1 Corpus Assembly 30
4.1.1 Segmentation 31
4.1.2 Labelling and train/test split 32
4.2 Feature Extraction 34
4.2.1 Spectrogram 34
4.2.2 Chromagram and HPCP 35
4.3 Classification 40
4.3.1 Hardware and use of Amazon EC2 40
4.3.2 CNN with spectrogram 41
4.3.3 LSTM with HPCP 42

5 Results 47
5.1 CNN with spectrograms 47
5.1.1 MS dataset 47
5.1.2 Effect of down-sampling 48
5.2 LSTM with HPCP 49
5.2.1 60-second samples 49
5.2.2 15-second samples 50
5.2.3 Transposition 52
5.2.4 Composer classification 52

6 Conclusion 57
6.1 Future Work 58

Bibliography 60

A Overview of repository contents 65

B LSTM Training Code 66

List of Figures

2.1 Timeline of compositional periods and representative composers . 5
2.2 Schematic diagram of a simple CNN 13
2.3 Schematic diagram of an unrolled RNN 14
2.4 Plot of the Hann function 15
2.5 An audio sample represented in time domain, frequency domain, and as a spectrogram 16
2.6 Chromagram of a C major scale 17

3.1 Breakdown of corpus with and without concertos 22
3.2 360-bin HPCP of the start of Beethoven's Symphony No. 5 . . . 29

4.1 Spectrogram of the start of Beethoven's Symphony No. 5 35
4.2 Bregman chromagram of the start of Beethoven's Symphony No. 5 36
4.3 Bregman chromagram of a chromatic scale 37
4.4 12-bin HPCP with increasing window sizes 39
4.5 TensorBoard graph of LSTM architecture 44

5.1 CNN accuracy for 44.1 kHz samples 47
5.2 CNN accuracy for 16 kHz samples 48
5.3 Accuracy of LSTM (batch size = 10) on BCRM dataset with 60 s HPCP samples 49
5.4 Accuracy of LSTM (batch size = 100) on BCRM dataset with 15 s HPCP samples 50
5.5 Accuracy of LSTM (batch size = 10) on BCRM dataset with 15 s HPCP samples 51
5.6 Colour scale of LSTM confusion matrix for periods 51
5.7 Accuracy of LSTM on BCRM dataset with 6 transpositions . . . 52
5.8 Accuracy of LSTM classifying BCRM composers 53
5.9 Colour scale of LSTM confusion matrix for composers 54
5.10 Number of errors against metric of distance between comosers . . 55

List of Tables

2.1 Characteristics of compositional periods 6
2.2 Frequencies of musical notes 17

3.1 Number of tracks and playing time by period 23
3.2 Composers represented in the corpus 24
3.3 Composers represented in the BCRM dataset 25

4.1 Number of samples in the BCRM dataset 32

5.1 Confusion matrix for CNN classifying MS dataset 48
5.2 Confusion matrix for LSTM classifying periods 51
5.3 Confusion matrix for LSTM classifying composers 53
5.4 Distance values for pairs of composers 54
5.5 Confusion matrix for classifying periods using an LSTM trained on composers 56

List of Code Listings

4.1 Usage of audio segmentation script 31
4.2 Functions for generating train/test split of BCRM dataset 33
4.3 Spectrogram generation script 34
4.4 Chromagram extraction script 36
4.5 HPCP transposition function 39
4.6 LSTM one-hot encoding function 42
4.7 Data-loading function for LSTM script 43
4.8 Changes to LSTM script to classify composers 45

Chapter 1

Introduction

Music genre classification is an important problem in the field of music information retrieval. Driven partly by advances in machine learning and computation more generally, and partly by the need for classification and identification capabilities in consumer products such as online music streaming services, this area of research is increasingly coming to the attention of the machine learning community. Applications which use music retrieval techniques include recommender systems, automatically generated play-lists and virtual radio stations, and music identification apps.

In the context of music, the term "genre" is used to refer to "categorical labels created by humans to characterize pieces of music" [1]. These labels may refer to styles of music (such as pop, rock, or metal), periods (such as classical or modern), or types of piece (such as a symphony or a string quartet). Since there is often overlap between genres, with both the boundaries between genres and their definitions subject to ambiguity and subjectivity, identifying genres automatically is a non-trivial task.

1.1 Project Specification

This project will focus on the classification of Western classical orchestral music, with classification following a hierarchical categorisation of musical periods and composers. The top level of this categorisation hierarchy corresponds to the four major eras of classical music: the Baroque, Classical, Romantic, and Modern periods. A detailed explanation of what is meant by each of these terms is given in Section 2.1. Within each of these categories are a number of subcategories corresponding to representative composers. The aim of this project is to identify methods suitable for learning this classification, and implement some of these methods with the help of modern machine learning libraries. Ultimately, the goal is to be able to provide the finished application with a sample from an audio

track, and have the application predict the period and composer of the piece of music.

The specific objectives of this project were as follows:

- Conduct a review of existing research into automated music classification
- Assemble a corpus of labelled audio files suitable for training neural networks
- Identify techniques, features, and algorithms to be used for classification
- Write machine learning applications capable of classifying the period of a piece of music
- Compare the performance of these applications
- Extend the applications to identify composers in addition to periods

1.2 Report Outline

The aim of this report is to document and justify the decisions made during the course of the project, and to present all relevant findings.

Chapter 2 gives an overview of the background in the three disciplines combined within this project—musicology, machine learning, and digital signal processing. In addition, Section 2.5 outlines some of the prior work conducted around music genre classification, as well as other research relevant to the current project.

Chapter 3 describes the methodology used for training the classification algorithms, as well as explaining the high-level decisions that were made with respect to corpus selection, feature extraction, and the choice of algorithms. Following on from this, Chapter 4 goes into more detail and explains which design choices were made and how the applications were implemented.

The results of running these applications are presented in Chapter 5, along with the interpretation of those results. A summary of the findings of this paper and an overview of future areas of research can be found in Chapter 6.

Finally, the appendices contain information about the source code used in the project. Appendix A details the contents of the project's Git repository, which contains all source code discussed in this paper and can be found online. Appendix B lists the source code for the LSTM training script, which forms a core part of the implementation.

Chapter 2

Background

2.1 Music History

The music which is the focus of this project, generally referred to as "classical" music, is also commonly named Western art music, in order to distinguish it from music of the narrower Classical period (set apart by the upper case "C") of the late eighteenth century. This term refers to works composed in a written musical tradition originating in Western Europe, in contrast to vernacular music such as jazz, folk, and popular music, which for the most part are rooted in oral and improvisational traditions. Art music has its origins in medieval Europe with the earliest forms of music notation, which were exclusively used for sacred vocal works. Out of a number of practical considerations that will be explained in Section 3.1, it was decided to confine the project to orchestral works. The earliest appearances of large instrumental ensembles, which we now think of as orchestras, are in Monteverdi's operas in the early 1600s, but written compositions for orchestras without vocalists did not become commonplace until later in the Baroque period. From this point onwards, orchestral music grew in popularity, and the orchestra expanded to involve more players until its heyday in the late nineteenth century. The period under consideration for classification in this project spans approximately from 1700 to the present day.

2.1.1 Compositional periods and their boundaries

In order to classify music by compositional periods, these must first be defined so as to allow manual labelling of the dataset. This presents some difficulties, as there is no universal consensus among experts in the field of Musicology with respect to such a categorisation. There is significant dispute around not only the precise dates that should be assigned to genres, but even which periods should be considered distinct genres at all. This should come as no surprise, since

these genres are the product of the stylistic diversity of a large number of artists working in a field where creativity and individuality are prized. Many composers defy a simple categorisation, as they did not always write in the same style. A particularly clear example of this is the Russian composer Igor Stravinsky, who went through three distinct phases in his career (his Russian, neoclassical, and serial periods) [2], with music from each of these phases differing so much that they could easily be mistaken for the work of different people. In the works of most composers, stylistic development is less pronounced and abrupt than this; however, it is common for many composers' styles to evolve as their skills mature and they are influenced by the developments of their contemporaries.

Another consideration is that, regardless of where the boundaries are drawn between different periods, there are always some composers whose lives overlap with more than one period. Some of the composers to whom this applies are ahead of their time and contribute towards the birth of the new period, while others are more conservative, and continue writing music in the older style while others push the boundaries (Richard Strauss, who continued writing music in a late Romantic style until the late 1940s, is a good example of this). As a result, a work's year of composition is often not a good indicator of its compositional style. In order to avoid ambiguity, only composers that fall within a single period are considered in this paper.

Clearly, any categorical classification of the sort required for a supervised learning project will by necessity be an artificial construct, as the development of music over time resembles a continuous spectrum more than a discrete set of styles. However, clear trends can be found in many aspects of musical development, and most musicologists agree on an approximate subdivision of music history into different periods, separated by events or brief periods during which the musical world underwent relatively rapid change. (Not coincidentally, these periods of transition often coincide with times of wider societal upheaval.) The four periods that will be examined in this paper are the Baroque, the Classical, the Romantic, and the Modern periods. Figure 2.1 shows an approximate timeline of these eras, with the lifetimes of key composers marked for reference.

Baroque

The Baroque period runs from around 1600 to 1750 [3]. Due to the requirement for orchestral works, the compositions used in this project come primarily from the late Baroque period (i.e. from around 1700 onwards), as orchestras in the modern sense of the word were rarely used before this time.

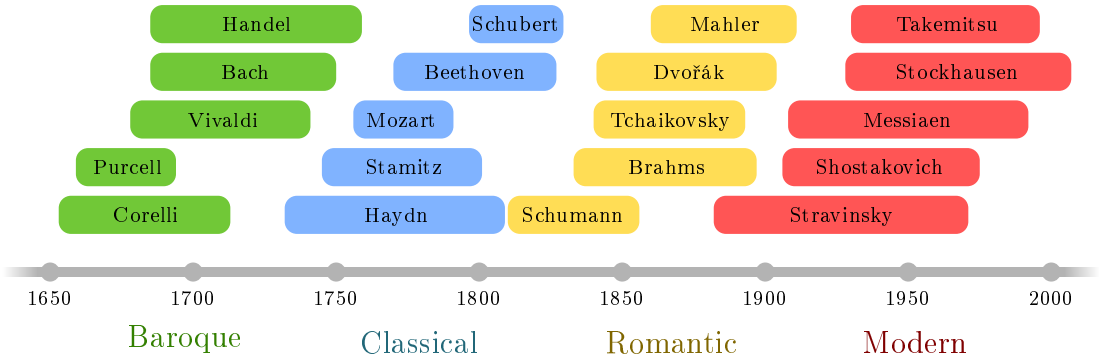


Figure 2.1: Timeline of compositional periods and representative composers

Classical

The Classical period is generally considered to run from around 1750 to 1830 [4], although there is considerable overlap between this period and the Romantic. The late works of Beethoven, who died in 1827, are frequently subject to debate, with some authors arguing that they form the start of the Romantic movement.

Romantic

The Romantic period extends from the end of the Classical era to around 1910 [5]. During the Romantic period, musical styles diverged more than in earlier periods with distinct national styles emerging, and there are significant differences between the early, middle, and late Romantic periods. Due to the overlap between the late Classical and early Romantic periods, only composers from the middle and late Romantic periods are included in the corpus.

Modern

"Modern" is the most controversial of the four labels, as musical styles in the twentieth century diverged radically. In this paper, the term is used to cover a variety of twentieth century art music genres, including impressionism, neoclassicism, modernism, socialist realism, and postmodernism [6, p. 43]. The composers representative of this period were chosen in order to maximise the cross-section of these different compositional styles included in the corpus, but the diversity of music written in the last century means that there are inevitably some omissions. A more detailed account of the composers included and the sub-genres they represent will be given in Section 3.1.2.

Characteristics

Table 2.1 gives an overview of some of the basic characteristics of these periods, and lists examples of representative works that are available to watch on YouTube.

Table 2.1: Characteristics of compositional periods

Period	Orchestra	Rhythm	Harmony	Example
Baroque (1600–1750)	Small orchestra: mainly harpsichord, strings, woodwind, and historical instruments such as the theorbo	No changes in tempo or metre, often polyphonic (i.e. with simultaneous rhythmically independent voices)	Tonal chords, one key throughout	Bach: Brandenburg Concerto No. 2 (1721) [7]
Classical (1750–1820)	Larger orchestra, now often with a conductor: generally strings, woodwind, some brass, and timpani	No changes in tempo, some use of more complex rhythms such as syncopation	Tonal with generally simple harmonies, occasional modulation to closely related keys	Haydn: Cello Concerto No. 1 (1765) [8]
Romantic (1820–1910)	Often very large orchestra (60–100 players): more brass and percussion, up to three harps	Generally no changes in metre, but very flexible tempos	More complex harmonies with distantly related keys and frequent use of chromatics	Tchaikovsky: Romeo and Juliet overture (1880) [9]
Modern (1910–present)	Large orchestra: often large percussion section, occasionally electronic instruments	Often complex and irregular rhythms with changes in metre and tempo	Widespread use of dissonance, often atonal and not melodic	Stravinsky: Rite of Spring (1913) [10]

2.2 Human Music Classification

The ability to recognise the period of a classical composition is considered by many people to be a skill confined to trained musicians. However, while some knowledge of the terminology is required in order to apply the correct label to a style, research by Bella and Peretz suggests that non-musicians with little experience of classical music may nevertheless easily recognise the difference in style between music of different periods, and can intuitively group compositions by the historical time period in which they were written [11].

The ability to classify music by composer does require some experience of their other works, and may not be picked up as easily. However, many composers are considered to have distinctive personal styles which can be immediately recognised by people who have this level of familiarity with the music. Composers whose styles are considered especially idiosyncratic and recognisable include Beethoven, Bruckner, Mahler, and Shostakovich.

It should be noted that although the ultimate goal of automated music classification projects is to achieve at least near-human levels of accuracy in identifying works, there is no consistent way of measuring the results of an algorithm against a human benchmark. This is partly because there is little research to systematically quantify human performance at this task, but mainly because even any attempt to do so is fraught with difficulties. The results would depend substantially not only on the level of expertise of the sample pool, but also on the specific selection of pieces on which they are tested. Both the inclusion of well-known pieces that can be identified without needing to recognise the genre, and the use of pieces which lie on the boundaries between periods and can therefore legitimately be classified in more than one way, could significantly skew the results. Instead of using human performance as a quantitative benchmark for machine learning results, this report will take a more qualitative look at the types of features used by humans in music recognition, and consider to what extent those features correspond to errors in automated classification.

2.2.1 Musical features used for classification

As with many other high-level problems at which humans intuitively excel (such as image recognition), the means by which people do so are frequently unclear. Musicians identifying the composer of a work often use abstract or subjective concepts to compare pieces of music, and may end up basing their prediction on the fact that a piece simply "feels" like a certain composer's style. These difficulties are compounded by a large number of outliers, where composers write music that is atypical of their usual compositional style, either in the spirit of

experimentation, or at the request of a patron. Some composers even specialise in emulating the styles of others, such as the violinist Fritz Kreisler, who ascribed a number of his own compositions to earlier composers like Vivaldi and Pugnani.

In spite of these hurdles, a number of features can be identified that allow associations to be made between works by individual composers, and there are many features that differentiate the music of different periods. These features can be organised into a hierarchy of complexity, with high-level features built on top of lower level features. The four fundamental features at the lowest level are pitch, rhythm, intensity, and timbre. Pitch refers to the frequency or combination of frequencies that are audible at a given point in time; rhythm is the sequence of durations between the onsets of notes; intensity is the relative volume of a particular pitch, as measured by the signal amplitude; and timbre refers to the quality of a note often referred to as "tone colour", which is the result of the relative intensities of the harmonic overtones of a pitch.

Often simple features such as timbre are sufficient to immediately identify the approximate genre of a piece of music (for instance, the distinctive sound of an electric guitar is sufficient to instantly place a piece of music in the latter half of the twentieth century or later). However, for a more fine-grained classification, an understanding of higher level features is required. These range from melodic "mannerisms" that are unique to certain composers, to the structure of entire movements, which can be more than half an hour in duration.

Pitch

Pitch in music is generally expressed in terms of pitch classes and octaves. Features derived from these include dissonance and harmony. Complex harmonies can often be found in music of the Romantic period, while high levels of dissonance are typical of many Modern composers.

Rhythm

Rhythm gives rise to metre (the recurring pattern of beats in a piece of music) and tempo (the rate at which these beats occur). Syncopation, where the rhythm subverts expectations by temporarily deviating from the metre, is an example of a higher level rhythmic feature. Frequent changes in metre are often associated with Modern music, while large tempo variations within the same metre were more common in the late Romantic period. Prior to this, both tempo and metre generally remained constant for the duration of a movement. Researchers at the University of Montreal concluded that rhythmic features are among the most useful when identifying periods [11].

Intensity

Intensity is most plainly associated with dynamics, or the volume at which musicians are playing, but also plays a role in articulation (the intensity of onsets and the amount of the separation between notes). Baroque music generally has few variations in volume, with dynamics restricted to a small number of discrete levels. In the Classical period, innovations in the manufacture of bows for string instruments allowed for a greater range of dynamics, as well as gradual changes in volume.

Timbre

Finally, timbre not only gives an insight into the instrumentation (the combination and number of instruments that make up the orchestra), but also offers insight into the manner in which instruments are being played (e.g. whether or not mutes are being used). Instrumentation is an important factor in music classification, not only because some instruments are only used in certain musical periods, but also because specific combinations of instruments are favoured by different composers.

2.2.2 Human limitations in music perception

In spite of the human brain's powerful capabilities of pattern recognition, which allow us to subconsciously identify many features that are difficult to detect computationally, there are a number of limitations experienced by humans in the perception of music. Some of these, such as our inability to hear pitches above a certain frequency, are due to the physical limitations of our hearing, while other limitations are cognitive in nature. The range of human hearing has been extensively studied, in particular in respect to the deterioration of the upper range of human hearing with increasing age. For the purposes of this paper, it is sufficient to note that the range of human hearing is around 20 Hz to 20 kHz in children, while the upper limit typically decreases to around 15 kHz in adults above the age of 30 [12]. While the deterioration of our hearing over age can have a significant deleterious impact, it does not normally inhibit the ability to recognise pieces of music or compositional styles. This suggests that most of the features used in music recognition and classification occur within the latter frequency range.

An example of a cognitive limitation on music perception is the limit of our ability to detect distinct temporal events. An overview of the extent and the implications of this constraint is given by Justin London [13]. The fundamental nature of this problem is that rhythmic stimuli are perceived as pitch if they are

repeated at a frequency within the range of human hearing, the lower bound of which is at around 50 Hz. As rhythmic impulses speed up to approach this limit, the ability to perceive them as distinct temporal events is lost. As a result, there is an upper limit on the tempo at which notes can be played and still be heard sequentially. Above this tempo, notes begin to "blur" and are perceived more as auditory textures than rhythms. London calculates that the fastest tempo perceivable as metre is around 180 bpm (beats per minute), and the fastest notes that can be heard as distinct events at this tempo are semiquavers (i.e. four notes to a beat). This yields an approximate upper rate of onsets at around

$$f_{\max} = \frac{180 \cdot 4}{60\,\mathrm{s}} = 12\,\mathrm{Hz}\,, \tag{2.1}$$

corresponding to a window length of 83 ms. An example of a performance at around this tempo is Anne-Sophie Mutter's 2014 recording of the *Presto* from Vivaldi's *Summer* [14].

Limitations such as these have important implications for any automated approach to music classification. Using various signal processing techniques, the input to a machine learning application can be restricted to those features used by humans when classifying music. While it is not necessary to restrict inputs in such a way, it appears reasonable to assume, at least in principle, that such restrictions should not be an obstacle to near-human levels of classification. This seems especially likely when considering that, as previously observed, many features that are typical of specific styles or composers are high-level and take place over extended periods of time. The advantage of imposing such a restriction is a significant reduction in the quantity of data passed to the application, which may significantly reduce the time and computational power required for training. The technical means by which such processing steps may be undertaken will be discussed in Section 3.2.

2.3 Supervised Learning

A supervised learning algorithm is one that learns from examples provided by means of labelled data, and generalises from patterns in this data in order to predict labels for previously unseen data. This is in contrast to unsupervised learning, where an algorithm attempts to find patterns in unlabelled data. The most common supervised learning tasks are classification and regression, both of which can be applied in the analysis of music. Classification tasks involve identifying a category from a finite and discrete set of possible values. A wide range of classification problems in music have been addressed in the past,

including genre classification in popular music, mood classification, and key recognition.

Regression is the analogous task for continuous output values; that is, tasks where the aim is to predict an approximate label on a continuous spectrum. Initially regression was considered as an alternative approach in this project, with the possibility of trying to predict the year of composition proposed as a solution to arbitrary nature of the categorical approach. However, the contemporaneous existence of different styles means that regression has its own problems, and it was decided that predicting an approximate period would be an easier starting point.

A large number of models exist for performing classification tasks, and new techniques are continually being developed, with rapid advances especially being made in the field of deep learning. However, many concepts are common to all of these models. One of these is overfitting. Overfitting occurs when parameters are defined that are not justified by the data, because the data contains patterns that do not generalise beyond the dataset [15]. This is particularly often the case when the dataset is too small.

One way to ensure overfitting does not affect results is by splitting data into training, validation, and test sets [16, p. 118]. In order to accurately measure the performance of an algorithm, datasets are generally split into three subsets, with the largest set used to train the model and the remaining two used in order to monitor its performance. The validation set is used to determine how the model performs on unseen data, and in general the hyper-parameters (i.e. any parameters that change the architecture of the model) are chosen based on the accuracy of this set. Any overfitting is usually revealed by the difference in accuracy between these first two sets as well as an increase in error over time in the validation set. The final test set is used to calculate the accuracy of the model for the chosen parameters, to ensure that the act of selecting the highest validation accuracy has not resulted in the model being overfitted to the validation set. The ideal ratio of the split between these sets varies depending on the amount of data available, the type of algorithm used, and the requirements of the specific task. A common baseline is to allocate 80% of data to the training set, and 10% each to the remaining sets.

Underfitting is the reverse problem, and occurs when there are parameters that match patterns in the data but are not included in the model. Underfitting can result in very low accuracies in the training data, and is generally an indication that the model is too simple for the task in hand.

2.3.1 Convolutional neural networks

Neural networks are a class of algorithm loosely inspired by the structure of the human brain, in which data is stored in neurons configured as an input layer, an output layer, and any number of hidden layers. Each neuron is a variable holding an activation value a, which represents an input value in the input layer, or a prediction in the output layer. In each layer other than the input layer, the activation is calculated from values of the previous layer by means of weights w and biases b. If the neurons of a layer are calculated from the activations of all neurons in the previous layer, this layer is referred to as fully connected, and the activation for the ith neuron in the jth layer is calculated as

$$a_{i,j} = \sum_{n=1}^{N} w_{n,i,j} a_{n,j-1} + b_{i,j} \,, \tag{2.2}$$

where N is the number of neurons in layer $j-1$ [17]. Neural networks are trained by adjusting the weights and biases in such a way that the activations of the output layer match the desired labels as closely as possible.

In general, neural networks require more computational power than conventional algorithms such as support vector machines (SVMs), as well as a large amount of data to learn from. However, unlike more traditional algorithms, they do not require hard-coded *a priori* knowledge of features, instead learning to identify features automatically, which makes them more adaptable to new problems and reduces the amount of code-writing required to perform a task.

A convolutional neural networks (CNN) is a feedforward neural network (i.e. one that contains no cycles) which contains convolutional layers. CNNs were first developed in the field of image processing, where the number of pixels in large images made the use of fully connected layers impractical, as the number of weights in each layer is proportional to the square of the number of pixels. Convolutional layers overcome this problem by replacing the weights with a single convolutional filter, called a kernel, which is passed across the entire image to generate the inputs to the next layer. CNNs may also use pooling layers to reduce the dimensionality of the input, and frequently have a small number of fully connected layers between the final convolutional layer and the output. Figure 2.2 shows a diagram of a simple CNN with one of each of these three types of layers.

Despite their close association with image recognition, CNNs have successfully been applied to many other areas. They are particularly useful in cases where data samples are multi-dimensional, fixed in size, and contain many inputs. As such, they might not seem like a natural fit for audio processing, as audio consists of

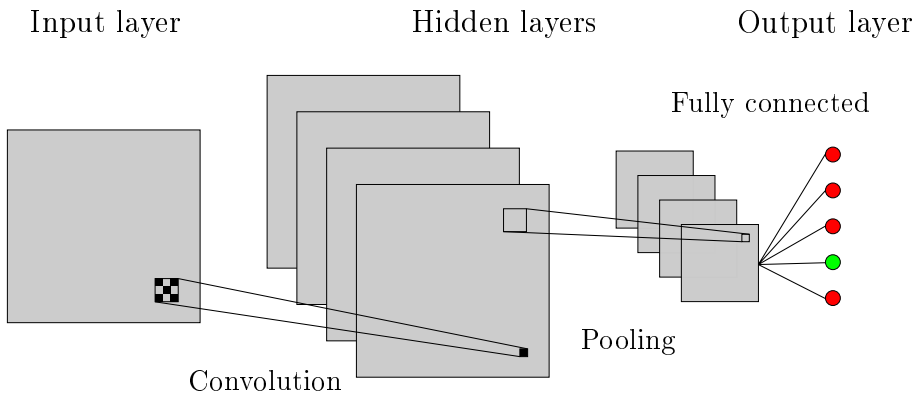


Figure 2.2: Schematic diagram of a simple CNN

a one-dimensional array of sequential inputs which can vary in length. However, a variety of techniques from digital signal processing (DSP) allow audio data to be represented as two-dimensional arrays, with dimensions representing time and frequency, making a CNN a more suitable choice.

One notable feature of CNNs is shift-invariance; identical clusters of inputs will result in the same stimulus after passing through the kernel regardless of where in the input layer they are located. This is useful in image processing, as the way an object is classified generally should not depend on its position within the image. A similar reasoning applies to audio data; a composition does not fundamentally change if it is shifted in time or frequency, stylistic classification should not depend on such changes.

2.3.2 Recurrent neural networks

Recurrent neural networks (RNNs) are neural networks that operate iteratively on sequential data and pass information from the output of one iteration to the input of the next. This means that the output of an RNN depends not only on its input, but also on its previous state, effectively giving it memory [16, p. 367]. Figure 2.3 shows how the passing of outputs to inputs is equivalent to an arbitrarily long row of RNN cells with a shared context. This configuration makes RNNs more flexible than feedforward neural networks, as they are not constrained to a fixed input size or a fixed number of calculations. As a result, they are especially successful at solving problems that involve temporal sequences that vary in length, such as in natural language processing, but they have been applied to an extremely wide range of tasks, including non-sequential problems such as image processing [18].

The RNN's ability to work with sequences makes it a natural fit for music-related tasks. However, they have a number of limitations that need to be

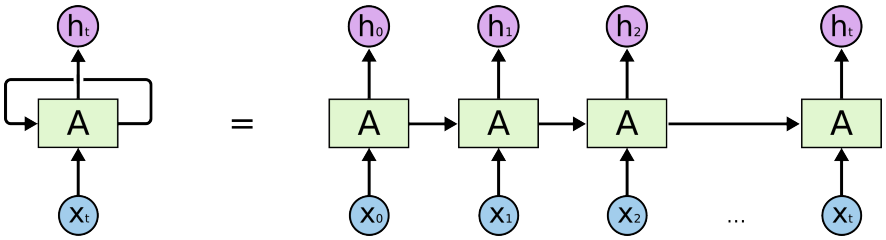


Figure 2.3: Schematic diagram of an unrolled RNN [19]

considered. Their primary drawback is that, as they operate sequentially and depend on previous iterations, it is not normally possible to parallelise their computation. This makes them slow to train on parallel hardware, making it difficult to take full advantage of modern GPUs.

Secondly, unlike CNNs, RNNs are not shift-invariant. For music classification, this means that they are unable to recognise the relationship between two audio samples that are identical apart from a transposition, unless the inputs are expressed as relative pitches (i.e. intervals) rather than absolute frequencies.

Finally, when operating on long sequences, basic RNNs are susceptible to the vanishing gradient problem. This is because at each iteration, the context is modified by combining it with the input using a single tanh gate. This makes it difficult for the RNN to recognise dependencies on information that is far removed in the sequence from the current input [20].

LSTMs (long short-term memory networks) are a variant of RNNs that do not suffer from the vanishing gradient problem. They differ from standard RNNs in that they have additional input, output, and forget gates. The input and forget gates determine which information in the hidden context is overwritten with new information before being passed on, and the weights of these gates can be trained to not overwrite important data. This makes them well suited for dealing with tasks that involve time series in which relevant information may be separated by indefinitely long sequences of less important information.

2.3.3 Gradient descent and backpropagation

The weights and biases that determine the activations of a neural network are found using stochastic gradient descent (SGD), a method in which a gradient descent function is approximated by incremental stochastic calculations [16, p. 149]. A small batch of samples drawn uniformly from the training set is used to gain an estimate for the gradient of the loss function, and this gradient is used to adjust the weights based on the learning rate, which is typically a very small increment. Because this process is repeated many times, any errors in the estimation of a single step have very little impact.

In feedforward neural networks, the gradient is calculated using the backpropagation algorithm [16, p. 200]. In backpropagation the error observed at the output layer is distributed backward through the network to gain an estimate of the error for each of the weights. The error on the output layer is calculated using a loss function such as cross entropy.

2.4 Digital Signal Processing

Several digital signal processing (DSP) techniques are used in this project in order to extract features from the audio data. A number of other features were considered but ultimately not used, including mel-frequency cepstrum coefficients (MFCCs) and onset detection.

2.4.1 Fourier transforms and spectrograms

The Fourier transform is a function which breaks a signal down into its constituent sinusoidal frequencies. Since frequency is a measure of oscillations per unit of time, it is meaningless to talk about the frequency at a specific instant in time. Therefore, in order to express frequency information as a function of time, the Fourier transform of successive windows over the signal must be calculated. The window function most commonly used for this purpose is the Hann function, as it is relatively insusceptible to aliasing. The Hann window is described by

$$w(n) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \qquad \forall n \in [0, N-1] \ , \tag{2.3}$$

where n is the sample index and N is the number of samples [21]. Figure 2.4 shows the plot of this function.

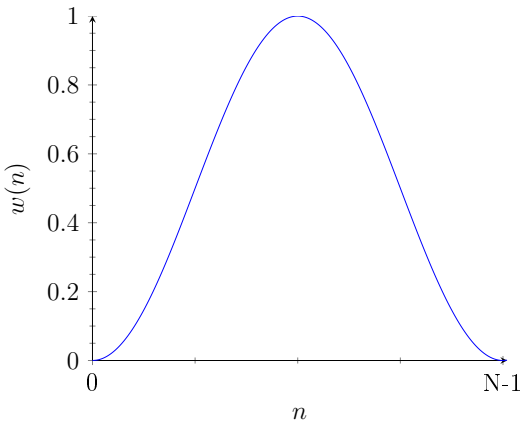


Figure 2.4: Plot of the Hann function

Applying the Fourier transform to successive windows of a signal results in a sequence of frequency spectra which can be plotted against time. The resulting function of amplitude and frequency over time is referred to as a spectrogram, and is generally plotted as a heat map. Figure 2.5 shows an audio signal in the time domain, its Fourier transform, and the resulting spectrogram.

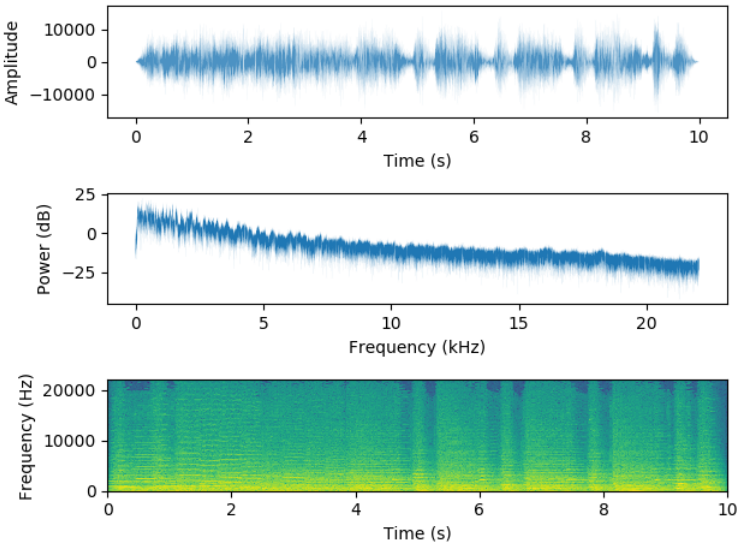


Figure 2.5: An audio sample represented in time domain, frequency domain, and as a spectrogram

2.4.2 Chromagrams and HPCP

Spectrograms are useful for identifying spectral information such as timbre and tuning, but they are inefficient for the identification of harmonies and pitches, since the spectrum covers a frequency range broader than the range of fundamental frequencies of notes used in music. Furthermore, since each octave has double the frequency range of the next lower one, most of the spectrogram is given over to the highest octaves, and the most relevant information for pitch identification is contained exclusively in the lowest portion of the spectrogram.

In order to represent pitches in a more concise way, the spectrogram can be converted into a chromagram. This is done by dividing the spectra into twelve bins, corresponding with the twelve pitch classes, or chroma, of the chromatic scale [22, p. 115]. The frequencies of these twelve notes are calculated as

$$f_n = f_0 \left(\sqrt[12]{2}\right)^n , \tag{2.4}$$

where n is the number of semitones above some arbitrary base pitch f_0, usually chosen as A_4 (440 Hz). Frequencies are binned to the nearest semitone, so that

the bin for each note is the frequency range between a quarter tone below the fundamental frequency to a quarter tone above. Table 2.2 shows the frequencies of the pitches in an octave starting from A_4. Beyond this octave, the cycle of pitch classes repeats, and for the purposes of the chromagram all octaves of the same pitch class are combined into the same bin.

Table 2.2: Frequencies of musical notes, starting from $A_4 = 440\,\text{Hz}$

Note name	**Frequency [Hz]**	**Frequency bin [Hz]**
A_4	440	427.47–452.89
$B\flat_4$	466.16	452.89–479.82
B_4	493.88	479.82–508.35
C_5	523.25	508.35–538.58
$C\sharp_5$	554.37	538.58–570.61
D_5	587.33	570.61–604.54
$E\flat_5$	622.25	604.54–640.48
E_5	659.25	640.48–678.57
F_5	698.46	678.57–718.93
$F\sharp_5$	739.99	718.93–761.67
G_5	783.99	761.67–806.96
$G\sharp_5$	830.61	806.96–854.95
A_5	880	854.95–905.79

Like spectrograms, chromagrams are most commonly represented as heat maps. An example is shown in Figure 2.6.

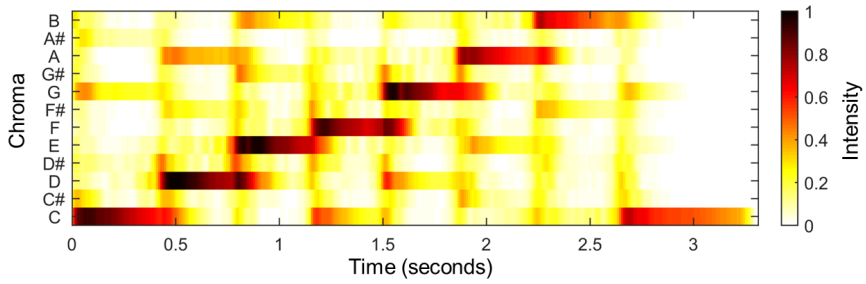


Figure 2.6: Chromagram of a C major scale played on a piano [23]

Harmonic pitch class profiles (HPCP) are a variant on the chromagram feature in which only local maxima on the frequency spectrum are considered, the audio

signal is filtered to remove high frequencies, and the final output is normalised so that the values do not depend significantly on intensity. When combined with a number of bins greater than twelve, these modifications result in a high-resolution map with very little noise on which pitches can be clearly distinguished (c.f. Figure 3.2).

2.5 Related Work

Research projects in music information retrieval have been conducted into a host of tasks, including automated formal analysis [24], music transcription (i.e. converting audio into written music notation) [25], and even music generation [26]. Music classification is one of the most widely researched tasks in this field, as can be seen by Sturm's 2014 survey of classification approaches [27].

The approaches used in automatic music classification broadly fall into two categories: those that use conventional algorithms such as support vector machines (SVMs) that require large numbers of features to be extracted in advance; and those that focus on deep learning algorithms such as neural networks, which learn to automatically extract high-level features. These later approaches not only have the advantage of requiring less effort to program; they are also more general and can easily be applied to different types of music, as they depend less on *a priori* knowledge of the features present in the music. The trade-off is that they require a great deal of computation to train.

Examples of successful efforts using non-neural network algorithms include Tzanetakis and Cook's use in 2002 of statistical pattern recognition classifiers to identify genres based on timbral, rhythmic, and pitch features [1] (68% accuracy over 10 genres) and the identification of melodic fragments that are useful for classification by Lin *et al.* in 2004 [28]. A particularly good example of this type of approach is a 2004 paper by McKay and Fujinaga, in which 109 features were extracted from MIDI representations of music using genetic algorithms, resulting in 98% accuracy when classifying between classical, jazz, and pop [29]. Finally, a 2014 paper by Weiß *et al.* uses SVMs on representations of triads, classifying four classical eras with 82.5% accuracy [30].

What all of these approaches have in common is that they use a large number of high-level features that are tailored to the specific music being classified. As a result, while some of the accuracy scores achieved are impressive, these approaches are difficult to generalise from. A broad overview of feature extraction techniques useful for classification, with particular focus on DSP techniques, is presented by Weihs *et al.* [31].

By contrast, neural network-based approaches are generally based on a small number of low-level features that require relatively little effort to extract and are often transferable to related problems. An early example of this type of approach is found in Hörnel and Menzel's use of feedforward neural networks for classifying harmonic progressions [32]. More recently, increased computational power has allowed neural networks to be used on audio data. A recent example of this is the evaluation of CNNs used with spectrograms by Costa *et al.* [33]. RNNs and LSTMs appear to be used less frequently for music classification. One example of a paper that does use LSTMs pairs them with an interval-based feature called Circle of Thirds [34].

A different approach is used in a 2014 paper by researchers from the Centre for Digital Music at Queen Mary University of London (QMUL): here, CRNNs, a combination of CNNs for local feature extraction and RNNs for high-level learning, are applied (Choi *et al.* [35]). This method is discussed further in Section 3.3.

A caveat to bear in mind when researching these approaches is that, while it is tempting to compare results of different papers numerically, this can be misleading. Even where the number and type of categories are identical between papers and the datasets are of comparable sizes, the difficulties in pigeonholing composers discussed in Section 2.1.1 mean that results can depend significantly on the specific works included in the corpus.

Chapter 3

Methodology

Most machine learning projects are inherently experimental in nature, and this project is no exception. The fundamental procedure used is to assemble a labelled corpus of audio files, perform some preprocessing steps to convert the audio into a standard format and extract features that are useful for classification, and use the resulting samples to train neural network algorithms.

These algorithms have a number of hyper-parameters that control various aspects of their behaviour. As the optimal values for these hyper-parameters are difficult to determine theoretically, approximate values will be determined empirically by choosing arbitrary starting values and iteratively modifying them based on the results.

Results will be evaluated based on the accuracy (i.e. the percentage of predictions that are correct) for the validation set. In addition, confusion matrices (a tabular representation of predictions) will be used to identify the types of errors made in classification. Together, this information will be used to inform adjustments to hyper-parameters of the model as well as parameters for feature extraction. The final results will be evaluated with the accuracy of predictions for the test set.

3.1 Corpus Assembly

For many machine learning problems, standard corpora are available as online resources. However, this is not the case for audio recordings of classical music, mainly because the majority of these recordings are under the copyright protection of record labels. For that reason, the corpus was assembled from audio source files taken from the author's personal music collection, primarily from audio CDs. The somewhat eclectic nature of this music collection is one reason for many of the constraints placed on the tracks included. Including certain genres of music for some periods and not others may skew results by allowing

the algorithm to train on the genre rather than the period; for example, if a large number of tracks for solo piano from the Romantic period are included, the algorithm may learn to recognise the sound of a piano as Romantic while ignoring musical features that are actually indicative of the Romantic style. In order to prevent such learning biases, the breakdown of genres in each period should be similar. Furthermore, to rule out biases resulting from certain periods being overrepresented in the corpus, the overall quantity of music from each period should be similar. Conforming to these constraints while maximising the size of the corpus was the main challenge during its assembly.

In this project, the term corpus will be used to refer to the full body of unprocessed audio source files from which samples are drawn, while the subsets of the corpus used for training an algorithm are referred to as datasets.

3.1.1 Breakdown of audio source files

The majority of the music collection consists of orchestral music, with lesser quantities of vocal music, chamber music (music written for smaller ensembles), and solo instrumental works. While examining the collection it soon became apparent that the majority of the vocal music present would be classed as either Baroque or Romantic, while the majority of the chamber works were Classical and Romantic. By contrast, orchestral works were more evenly spread across all four periods. For this reason only orchestral music was included in the corpus, comprising mainly symphonies, ballet suites, orchestral suites, tone poems (a Romantic genre similar to a symphony but looser in structure), and *concerti grossi* (an orchestral concerto of the Baroque era). Opera was excluded from the corpus on the grounds that this is primarily a vocal genre rather than an orchestral one, with the exception of operatic overtures in which only the orchestra plays.

Inclusion of solo concertos

Concertos, in which solo instrumentalists are accompanied by an orchestra, were initially excluded from the corpus due to concerns that passages in which only the soloist plays might be subject to some of the same biases outlined above. This effect was considered to be unlikely to make a significant impact, as in most concertos the orchestra plays for the majority of the work, and most machine learning algorithms are robust towards small numbers of outliers. The most common types of concertos (violin, piano, and cello) can be found in all four periods. (Despite its name, the *concerto grosso* is not a concerto in the modern sense of the word.)

Excluding concertos, the corpus contained 1368 tracks, with a total playing time of 134 hours. However, there was a significant disparity between the different periods, with around twice as many tracks from the Romantic and Modern periods as from the Baroque and Classical. This discrepancy is exacerbated when considering the total playing time, as works from the Baroque period are usually significantly shorter than those from later periods. As the concerto was a particularly popular genre during the Baroque period, it was ultimately decided that these should be included. In addition to increasing the number of tracks by many composers already in the corpus, this allowed the inclusion of works by the Italian composer Antonio Vivaldi, whose orchestral output is comprised almost exclusively of concertos. Meanwhile, composers accounting for less than 90 minutes of audio source files were removed from the corpus, as there were a number of Romantic composers with very small numbers of works included that would probably be insufficient for a neural network to learn from. Figure 3.1 demonstrates the effect of these changes on the make-up of the corpus.

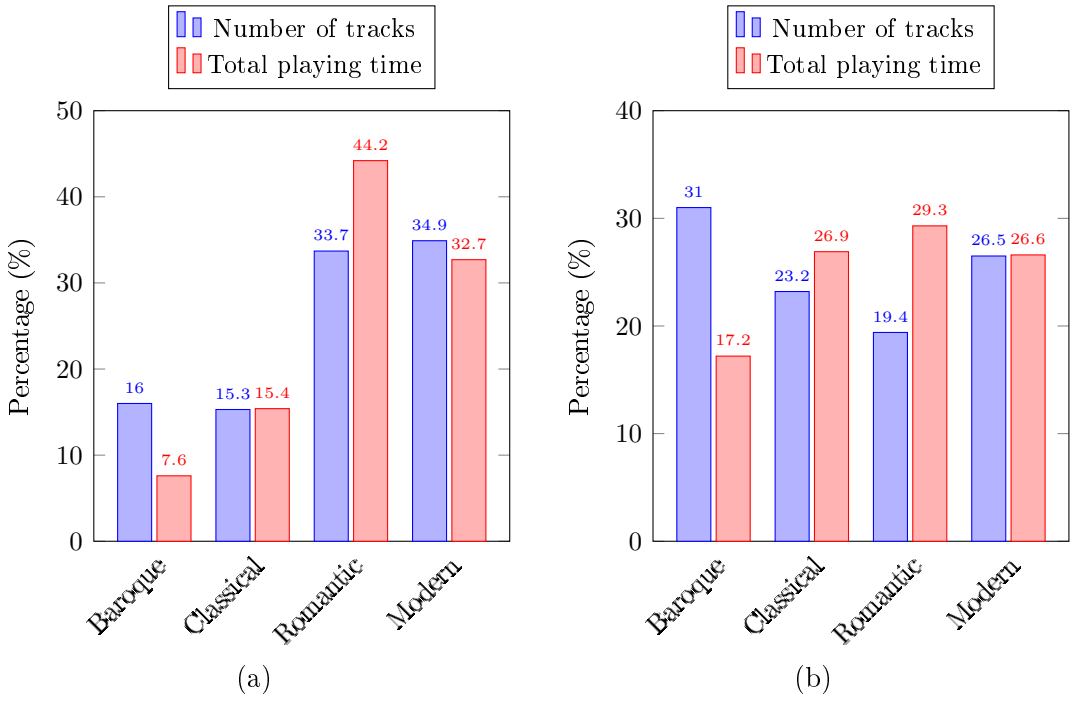


Figure 3.1: Breakdown of the corpus by compositional period before (a) and after (b) including concertos

Number of tracks vs. playing time

The average duration of tracks varies between different periods. Table 3.1 demonstrates that, on average, tracks from the Romantic era are almost three times the length of those from the Baroque period. This is the result of a trend towards larger and more structurally complex works during the nineteenth

century, with some composers such as Mahler writing symphonies on an epic scale, lasting well over an hour. (Symphonies are normally written in four or five movements, so they are split up into a corresponding number of tracks.) In contrast, a typical symphony by the Classical composer Haydn lasts around 20 minutes. Symphonies at this time made extensive use of "sonata form", a detailed structural model that specifies the relationship between different themes and keys in a piece [36]. Prior to this, music was written in simpler binary and ternary structures which were harder to scale up, resulting in movements that are typically only a few minutes long.

This poses a difficulty as it is initially unclear whether the number of tracks or the playing time should be kept constant across periods. This answer depends on the chosen machine learning approach, as the number of samples available to the algorithm should be approximately equal between different periods; an algorithm that takes in entire tracks and analyses features of a whole movement requires the same number of tracks from each period, while an algorithm that takes in fixed-length segments of tracks requires an equal amount of playing time. The approaches ultimately used in this project fall into the latter category, although several full-track approaches were considered.

Table 3.1: Number of tracks and playing time by period

Period	**Number of tracks**	**Playing time** [h : min : s]	**Avg. playing time per track** [min : s]
Baroque	432	23:54:51	3:19
Classical	323	37:18:03	6:56
Romantic	270	40:38:08	9:02
Modern	369	36:49:04	5:59
Total	1394	138:40:07	5:58

3.1.2 Final corpus selection

Table 3.2 lists all composers represented in the final corpus, along with their approximate periods and the number and total duration of tracks representing them.

Table 3.2: Composers represented in the corpus

Composer	**Approx. period**	**Tracks**	**Playing time** [h : min : s]
Mozart, W.A. (1756–1791)	Classical	186	19:05:27
Shostakovich, D. (1906–1975)	Modern	64	12:01:24
Bach, J.S. (1685–1750)	Baroque	135	10:51:16
Beethoven, L.v. (1770–1827)	Classical (late)	68	10:47:09
Dvořák, A. (1841–1904)	Romantic	63	10:08:38
Prokofiev, S. (1891–1953)	Modern	68	8:10:50
Haydn, J. (1732–1809)	Classical	69	7:25:27
Stravinsky, I. (1882–1971)	Modern	138	6:50:37
Brahms, J. (1833–1897)	Romantic	52	6:48:03
Tchaikovsky, P.I. (1840–1893)	Romantic	50	6:33:39
Elgar, E. (1857–1934)	Romantic (late)	54	6:27:11
Mahler, G. (1860–1911)	Romantic (late)	23	6:00:29
Messiaen, M. (1908–1992)	Modern	41	4:44:50
Sibelius, J. (1865–1957)	Romantic (late)	28	4:40:08
Corelli, A. (1653–1713)	Baroque	130	4:26:14
Vivaldi, A. (1678–1741)	Baroque	88	4:23:30
Ives, C. (1874–1954)	Modern	22	2:35:15
Stockhausen, K. (1928–2007)	Modern	36	2:26:09
Telemann, G.P. (1681–1767)	Baroque	41	2:25:37
Handel, G.F. (1685–1759)	Baroque	38	1:48:14

MS dataset

In order to test machine learning methods on a smaller dataset ahead of training on the full corpus, which is inevitably more time- and memory-consuming, a subset of the corpus consisting only of works by Mozart and Shostakovich was created (the "MS" dataset). The choice of these two composers resulted in a dataset of around 24 h of audio source files. In addition, as Mozart and Shostakovich are far removed from one another historically and have very different compositional styles, this presents a relatively easy test case. This corpus will therefore be used as a rough benchmark in order to test the feasibility of methods,

as any algorithm unable to perform well on this dataset will not fare any better on the full corpus.

BCRM dataset

One of the requirements for the main dataset was that the breakdown of each period should be as consistent as possible, so that each period is represented by the same number of composers, and each composer by the same number of tracks. In addition to preventing biases during training, this has the advantage that the same corpus can later be used for composer identification. This restriction means that there is a trade-off between the number of composers and the number of tracks by each composer; as more composers are included, the number of tracks by all composers is limited to the number available by those newly added. Eventually a set of three composers from each period were chosen so that over 260 min are available from each. The selected composers are shown in Table 3.3. For composers from the Baroque and Classical periods, available playing time was the primary consideration in making this selection; however, for Romantic and Modern music, where there was more choice available, another concern was to obtain a sufficiently representative cross-section of the styles found in these periods. In the case of Romantic music, the middle Romantic era is represented by Brahms and Tchaikovsky, while the late Romantic period is represented by Mahler (early Romantic music was omitted due to its overlap with Beethoven's later works). The Modern era is represented by Stravinsky, who wrote music in a variety of styles, including neoclassical and serialist; Shostakovich, who wrote some modernist music, but mainly composed in a socialist realist style; and Messiaen, whose music is harder to classify, but shows signs of both impressionist and modernist influences.

Table 3.3: Composers represented in the BCRM dataset

Baroque	**Classical**	**Romantic**	**Modern**
Corelli	Haydn	Brahms	Stravinsky
Vivaldi	Mozart	Tchaikovsky	Shostakovich
Bach	Beethoven	Mahler	Messiaen

3.2 Preprocessing

"Preprocessing" refers to any data processing step performed on the corpus in order to get the samples into the format required by the machine learning algorithms. This includes segmentation of the audio files, down-sampling, and transposition, as well as feature extraction in cases where this is not performed by the training script. However, feature extraction will be discussed in conjunction with classification methods in Section 3.3, as the choice of features is closely connected to the choice of algorithm.

3.2.1 Segment length

Most neural network architectures are unable to handle inputs of variable size, as the input size is hard-coded in the dimensionality of the input layer. This means that inputs that vary in size need to be converted to a standard size during preprocessing. For audio files, this means that all samples in the dataset should be the same duration and have the same sample rate. While RNNs are able to parse sequences of variable length, in order to ensure consistency of results, it was decided to use the same samples on both architectures.

The optimal sample length was one of the unknown quantities in this project, and an acceptable value had to be determined experimentally. In general, it was believed that longer samples would benefit the LSTM more than the CNN, as LSTMs are able to take past events into consideration, whereas the CNN is unable to make the temporal connections to analyse high-level temporal structures. For this reason the initial sample length was chosen to be 60 s for the LSTM, with an initial segment length of 10 s for the CNN.

One consideration when segmenting files was whether or not to allow overlapping segments. By including an overlap, a larger quantity of samples can be generated; however, it was felt that this would create too many incidental similarities between different segments, which would be especially problematic for the shift-invariant CNN.

The final consideration for segmentation was how to handle the cut-off between segments. As there was a risk of boundary effects caused by the onsets of notes falling close to the cut-off, it was decided that each segment should have a fade-in and a fade-out. Smoothing out any boundary effects as far as possible minimises the impact that the precise start time of each audio segment could have on training.

3.2.2 Sample rate

The sample rate of the audio segments was another consideration where there was a trade-off between performance and information. A reduction of the sample rate could significantly reduce data sizes, improving performance; but this would be at the cost of the loss of high-frequency information. It was initially unclear whether the upper frequencies of the spectrum were helpful for classification, so this was another parameter that was to be determined experimentally. Based on the findings in Section 2.2.2, it was believed that sample rates could be significantly reduced without having an adverse effect on results. This assumption was based on the reasoning that, if the upper frequency of the hearing range is 15 kHz for many adults with otherwise unimpaired hearing, it would seem reasonable that audio data above this frequency range can be discarded without losing most features used by humans in classifying music.

As in the case of the sample length, the sample rate must be consistent between segments, as this value is hard-coded into the neural network architecture. Experiments were run with both 44.1 kHz and 16 kHz. The results are compared in Section 5.1.

3.2.3 Transposition

A final preprocessing step that was considered was transposition. Transposition is the shifting of music in the frequency space by a musical interval (i.e. a number of chroma). In the case of a CNN, this should not be helpful; however, as LSTMs do not have the same shift-invariant property, they could potentially benefit from additional samples generated by transposition. The risk with this approach was that adding transpositions would prevent the LSTM from learning key preferences associated with certain periods; for instance, Baroque and Classical composers were more likely to compose in keys with few sharps and flats than Romantic or Modern composers. Whether this loss of information would be offset by the additional training data was an open question that was to be determined by experimentation.

3.3 Models and Features

Several algorithms were considered for the training process, including some non-neural network approaches such as random decision forests. The difficulty with these algorithms was that they require high-level features to be extracted in advance, which requires a considerable time investment. The main advantage that neural networks have over these more conventional algorithms is that, given

a low-level feature or even raw data, they can learn and select high-level features automatically.

One approach that was particularly intriguing was the combination of CNN and RNN described by researchers from QMUL [35]. These CRNNs allow local feature extraction to be performed by CNNs, while RNNs learn the temporal relationships of these features. This enables the inputs to the RNN to be represented in a more compact form. One possibility that was considered was the use of a CRNN with HPCP as the input feature, in place of the mel-spectrograms used at QMUL.

The two approaches finally chosen for experimentation were CNNs and LSTMs, with the combination of the two left to future research.

3.3.1 CNN with spectrogram

Using a CNN in conjunction with spectrograms for audio classification is already a well established practice in the area of speech recognition. The advantage of this approach is that CNNs are particularly well-suited to analysing two-dimensional arrays of data when the input dimensions are known in advance.

The basic process is to use time-frequency analysis to generate a spectrogram, and train a CNN on the generated image. While it was believed that this approach was unlikely to learn very high-level musical features, it was considered likely that this would be the most effective method for learning timbral features.

Using a CNN necessitated segmentation into fixed-size samples, but this does not mean that the trained network cannot be used to classify full tracks. Classification of full tracks was done by the simple means of segmenting the track into overlapping samples of the size required by the CNN and averaging the output probabilities across the series of samples.

3.3.2 LSTM with chroma

The idea of using LSTMs was particularly tempting due to their flexibility and ability to learn long-term feature dependencies. The choice of feature to use with this algorithm was less obvious.

The initial aim was to use a sequence of pitch class vectors, similar to a simplified MIDI format. The difficulty found with this approach was that libraries for performing this conversion are extremely unreliable. Even the simpler task of onset detection, where the extracted feature is a series of timestamps at which the onsets of notes occur, is a hard enough problem that existing libraries which perform relatively well on recordings of instruments with very clear onsets such as a piano are unusable with orchestral music. Algorithms

based on detecting increases in spectral energy were found to miss many onsets in orchestral recordings, as well as delivering a high number of false positives. More recent approaches to this problem use CNNs to achieve a better performance [37]; however, implementing this approach is beyond the scope of the current project.

The next best option was considered to be a chroma-based feature, where the same pitch information is conveyed in a less compact form. The choice here was between a standard chromagram and the more recent HPCP variant.

Chromagram vs. HPCP

The HPCP algorithm differs from a standard chromagram in that it uses a variable number of bins (120 by default), instead of just 12 in a standard chromagram, and performs some additional pre- and postprocessing steps [38], [39]. In particular, the signal is passed through a band-pass filter to reduce the influence of high and low frequencies, and peak detection allows only local maxima on the frequency spectrum to be considered, significantly improving the Signal-to-Noise Ratio (SNR) of the data. In addition, the output is normalised, with all levels scaled so that the most prominent pitch class at each point in time is given a value of 1.0, while the class with the lowest intensity is assigned a value of 0. This allows a sequence of pitches to be detected easily, regardless of dynamic levels. This can be seen in Figure 3.2, which shows the HPCP of the first 30 s of Beethoven's Fifth Symphony.

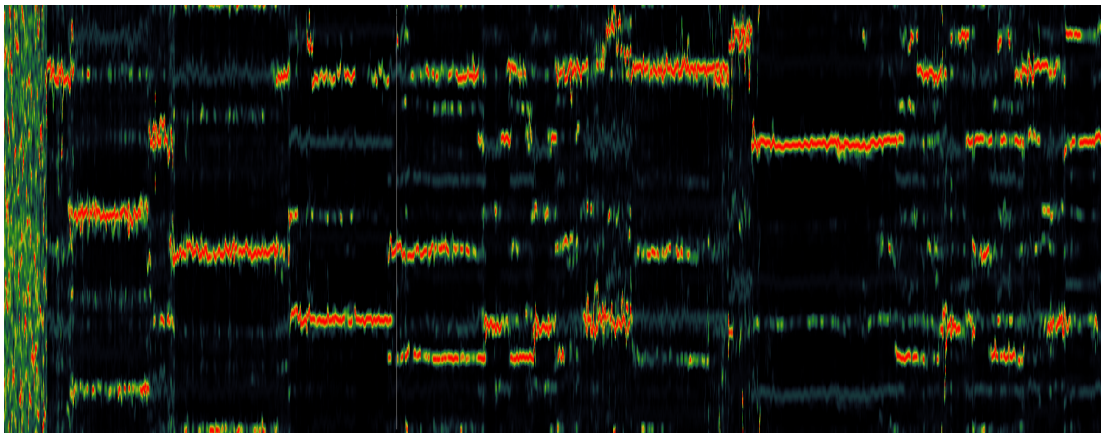


Figure 3.2: 360-bin HPCP of the start of Beethoven's Symphony No. 5

The practical advantages of using HPCP over a more basic chroma feature are explored in Section 4.2.2. HPCP has previously been used for other machine learning tasks such as audio matching [40].

Chapter 4

Experimental Design and Implementation

The first implementation decision was the choice of programming language, with MATLAB, R, and Python all considered for this task. Out of the three, Python has some of the most advanced and widely-used deep learning frameworks, and the standard Python libraries NumPy and SciPy enable fast C-based execution of mathematical calculations. Prior experience of the language, together with the wide variety of library choices and the broad availability of online resources in the form of detailed documentation and tutorials, made this the preferred choice for the project.

4.1 Corpus Assembly

The full corpus and its index, as well as smaller datasets from the corpus and indexes of previous versions were stored in the `corpus` directory of the project. The initial gathering and sorting of audio source files had to be done manually, by selecting tracks from the larger music collection and copying them into an `audio_sources` subdirectory. This process required a significant time investment due to the large number of files that had to be processed, but presented no other technical challenges. The source files were sorted into directories of the form `audio_sources/[period]/[composer]`, which allowed automated generation of the datasets.

The majority of source files were encoded as CD audio, with a sample rate of 44.1 kHz, a 16-bit resolution, and 2 channels.

4.1.1 Segmentation

A segmentation script (`segment_audio.py`) was written to take audio files and cut them into segments of a given length. The script can be executed from the command line with the following usage:

```
usage: segment_audio.py [-h] [--recursive] [--separator SEPARATOR]
            [--segment_length SEGMENT_LENGTH] [--overlap OVERLAP]
            [--fade_length FADE_LENGTH]
            [--sample_rate SAMPLE_RATE] [--stereo]
            [--csv_path CSV_PATH]
            src out
```

Listing 4.1: Usage of audio segmentation script

This splits all audio files in the `src` directory into segments of the length specified by `--segment_length` (or 60 s by default), and saves the result to the `out` directory. The resulting files are named from a UUID indicating the source file, followed by the value indicated by `--separator`, followed by the index of the segment within the source file. The `--separator` option will play a role in Section 4.3.2. Renaming the files using a unique identifier was necessary as some of the source files have generic names (e.g. `Track_01.wav`) which may conflict with files of the same name from other subdirectories. The `--recursive` option includes all audio files saved in subdirectories, and the `--overlap` option specifies the overlap between successive segments (0 s by default). This option was included so that a trained graph that takes a fixed-length input can be used to classify arbitrarily long tracks by classifying successive overlapping windows of that track.

In addition to segmenting the files, a fade-in and fade-out are applied to the start and end of each segment. The duration of the fade can be specified with the `--fade_length` option, which by default is set to 250 ms. The script also accepts a `--sample_rate` argument in Hz, with a default value of 44.1 kHz, which specifies the sample rate of the generated segments. This allows down-sampling to be performed in the same script run. The `--stereo` option was included so that the outputs could be stored as stereo audio wherever two source channels are available. This option was ultimately not used in the project, as all samples were converted to single-channel in order to minimise file sizes, but the option was retained for future use.

Finally, the `--csv_path` option specifies an output path for the CSV file containing the generated index. The index is required in order to identify the source file that a segment was generated from, and is therefore needed in order to generate the labels. The columns of the index file indicate the output files' UUID, the source file path, and the number of samples generated from that source.

The script has two public functions, `segment_file()` and `segment_files()`. This allows the file to be imported as a module, so that the functionality can be called on by other scripts. The first of these returns a list of segments for a single input file, while `segment_files()` stores segments generated from an input directory into a specified output directory. Both public functions as well as the `__main__` function of the script perform validation on all inputs in order to ensure that the script is not caught in an infinite loop (e.g. because the overlap between samples is equal to the sample length). Finally, if the script is run from the command line, progress bars indicating the number of files left to process are printed to `stdout`. This feedback was desirable as a run of the script on the full corpus had a typical duration of around 20 min.

`segment_audio.py` uses the open-source Python library Pydub [41] for segmenting files as well as applying fades and setting the sample rate. This dependency can be installed by running `pip install pydub`.

4.1.2 Labelling and train/test split

The script `assemble_bcrm.py` assembles the BCRM dataset from the audio sources. Segmentation and labelling are done in one step, as the script calls `audio_segment.segment_file()`. The script draws a fixed number of samples from each of the directories representing composers in the BCRM corpus. The approximate maximum number of samples that could be extracted given the quantity of music available from each composer is shown in Table 4.1.

Table 4.1: Number of samples in the BCRM dataset

Sample length [s]	**Samples per composer**	**Total samples**
60	200	2400
30	400	4800
15	800	9600
10	1200	14400

Initially, the samples were generated by segmenting source files sequentially until the specified number of samples was reached. This approach was later revised; instead, all source files are segmented into a temporary directory, and the specified number of samples are selected randomly from the temporary sample set. This process is significantly slower, as for composers with large numbers of source files (e.g. Mozart), many more files need to be segmented, but has the advantage that samples are drawn with equal probability from all of the source material.

Having samples drawn from a larger number of source files is advantageous as it minimises any biases introduced by allowing the neural network to learn from pieces rather than composition styles.

Labelling in the BCRM dataset is done by file name, rather than as a directory structure, as this makes it easier to identify labels during training. The files are given names of the form `[p]_[com]_[index].wav`, where `p` is the first letter of the period, `com` is the first three letters of the composer's surname, and index is a three-digit integer (e.g. a sample by Tchaikovsky might have the file name `r_tch_001.wav`). This allows labels for either the period or the composer to be extracted as needed by looking at specific character positions within the file name. A CSV file is generated with an index that maps samples to their source files and time stamp within the source file.

The final stage of corpus assembly is to split the samples into sets for training, validation, and testing. This is done using the script `train_test_split.py`. The primary public function of this script is `split_bcrm()`, which together with the utility function `_form_bcrm_dict()` splits the contents of the BCRM directory into subdirectories named `train`, `valid`, and `test`, while preserving the proportion of samples by each composer in each set (i.e. each directory contains the same number of samples by each composer). These functions are shown in Listing 4.2.

```
def _split(dir_, files, train, test, validation):
    # Splits the given list of files in dir_ into subdirectories
    # named "train", "valid", and "test" and returns a tuple
    # representing the number of files added to each set.


def _form_bcrm_dict(files):
    # Returns a dictionary where each entry corresponds to a composer,
    # and the values are lists of sample files by those composers.
    file_dict = {}
    for f in files:
        key = f[:5]  # first five characters, e.g. "c_moz"
        if key in file_dict:
            file_dict[key].append(f)
        else:
            file_dict[key] = [f]
    return file_dict


def split_bcrm(dir_, train, validation, test):
```

```
    """Split the contents of BCRM directory into train and test sets.

    Preserves the ratio of samples by the same composers.
    Args:
        dir_ (string): The BCRM root directory
        train (int), validation (int), test (int): The ratio of these
            parameters specifies the percentage of samples to go in
            each dataset (e.g. inputs of 3, 1, and 1 result in sets
            of 60%, 20%, and 20% of samples, respectively)
    """
    files = os.listdir(dir_)
    print("Splitting {} files...".format(len(files)))
    file_dict = _form_bcrm_dict(files)
    results = (0, 0, 0)
    for _, comp_files in file_dict.items():
        r = _split(dir_, comp_files, train, test, validation)
        results = tuple(sum(x) for x in zip(results, r))
    print("train={}, valid={}, test={}".format(*results))
```

Listing 4.2: Functions for generating train/test split of BCRM dataset

A similar function, named `split()`, performs the same task without preserving the ratio of samples by composers, and was used for splitting the MS dataset, as samples by each composer in that dataset were stored in separate directories.

Train/test split was executed once for each dataset, so that the same sets were used between different runs.

4.2 Feature Extraction

4.2.1 Spectrogram

The spectrogram of an audio file can be generated using Matplotlib's `pyplot.specgram()` function. This function takes in an array of signal amplitudes, as well as keyword parameters for the sample rate (`Fs`) and the number of samples used for each FFT iteration (`NFFT`). A function `show_spectrogram()`, shown in Listing 4.3, was written that takes a stereo wave file as an input, performs mono conversion, and displays its spectrogram. Figure 4.1 shows an example of a spectrogram generated in this way.

```
import numpy as np
from matplotlib import pyplot
from scipy.io import wavfile
```

```
4
5  def run(file):
6      fr, audio = wavfile.read(file)
7      mono = [0.5 * x[0] + 0.5 * x[1] for x in audio]
8      pyplot.specgram(np.asarray(mono), Fs=fr, NFFT=1024)
9      pyplot.xlabel('Time (s)')
10     pyplot.ylabel('Frequency (Hz)')
11     pyplot.show()
```

Listing 4.3: Spectrogram generation script

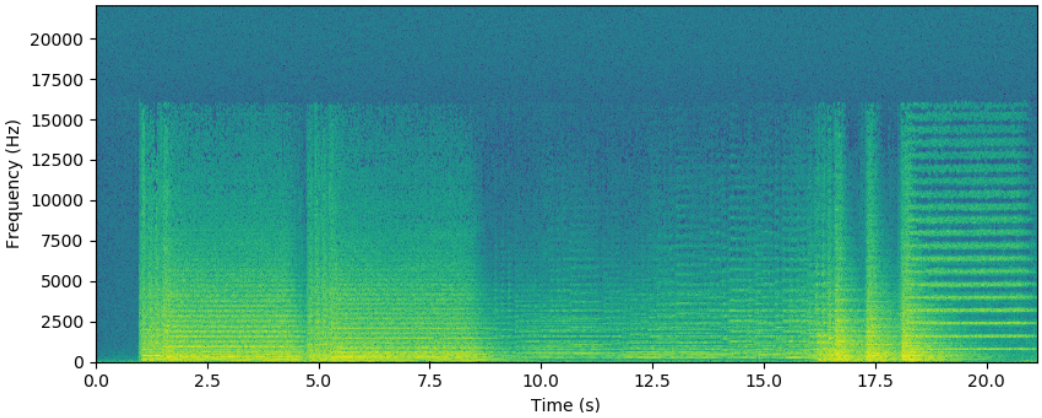


Figure 4.1: Spectrogram of the start of Beethoven's Symphony No. 5

For the purposes of the CNN, this script was not required, as the TensorFlow script that was used performed the spectrogram conversion at run-time.

4.2.2 Chromagram and HPCP

While creating a chromagram using the SciPy's FFT implementation would not be especially technically challenging, writing a script to do so in a manner efficient enough to convert large quantities of audio in a reasonable time period is not within the scope of this project. Therefore, various tools were explored for extracting the chroma feature, the most useful of which are detailed here.

Bregman Audio-Visual Information Toolbox

The Bregman Audio-Visual Information Toolbox is a Python library which offers a variety of audio processing tools [42]. The class `bregman.suite.Chromagram` creates a chromagram of an input audio file. Listing 4.4 shows how the class can be used to plot the chromagram of a 20 s sample from the start of Beethoven's Fifth Symphony.

```
#!/usr/bin/env python2

from bregman.suite import Chromagram, wavread


def show_chromagram(audio_file):
    audio, fs, _ = wavread(audio_file)
    chroma = Chromagram(audio, sample_rate=fs, nfft=16384, wfft=8192,
                        nhop=2205)
    chroma.feature_plot(dbscale=False, normalize=True)


if __name__ == '__main__':
    show_chromagram("beethoven.wav")
```

Listing 4.4: Chromagram extraction script

Running this script results in the output shown in Figure 4.2. There are a few issues with this output as implemented in Bregman. Firstly, the chroma appear to be offset from the correct values; this is apparent as the sample is in C minor, but the predominant pitch classes indicated in the chromagram are G♯ and E (they should be G and E♭). Secondly, during the quieter passage after the opening chords (from around 8 s to 16 s), the intensity is so low that no chroma can be detected; however, when listening to the sample the notes can clearly be heard, albeit at a lower dynamic level.

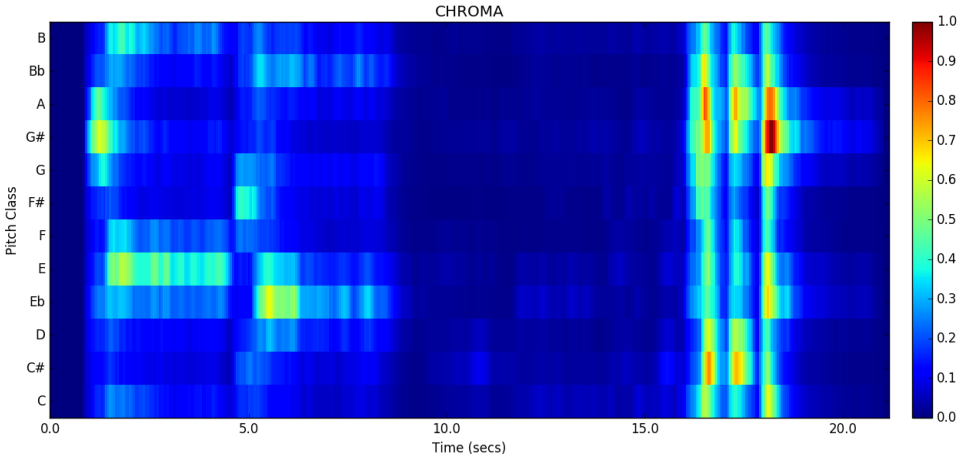


Figure 4.2: Bregman chromagram of the start of Beethoven's Symphony No. 5

Running the tool on a MIDI-generated audio file of a chromatic scale results in the chromagram shown in Figure 4.3, which emphasises the first of these issues. Not only is the predominant pitch class offset by a semitone (this is a scale beginning on C), but the pitch classes either side of this also experience significant

levels of intensity. Since this is a computer generated sound file, this error cannot be attributed to inaccuracies in the performance. Harmonic overtones can also not account for this, as the most pronounced overtone should be a fifth above the fundamental frequency (G for the first note). The semitone offset would almost certainly have little or no impact on machine learning algorithms, as these algorithms need not work in an absolute reference frame. However, the overspill onto neighbouring pitches is very likely to affect the network's ability to recognise harmonies, particularly in more complex musical samples where multiple pitches are played at once.

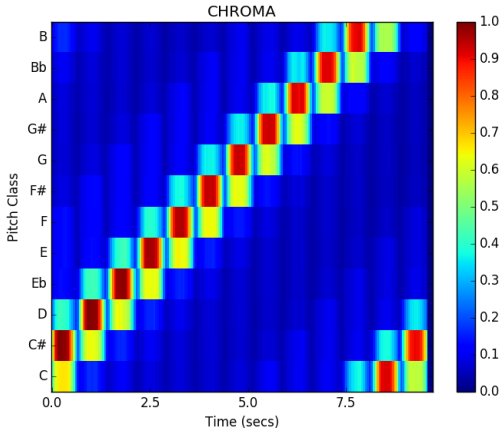


Figure 4.3: Bregman chromagram of a chromatic scale

The difficulty in identifying pitches at low intensities is also likely to be problematic for a neural network, even if the pitch can be computationally differentiated from background levels. This is because most neural network architectures are sensitive to input scaling, so networks will be unable to recognise the relationship between a melody and the same melody played at a different dynamic level. For this reason it is common practice in many neural network applications to normalise the inputs to a known range [43].

While the offset problem appears to be a feature of this implementation and could be resolved by finding another implementation, the intensity issue is fundamental to the chromagram feature. While these values could be normalised to the range 0.0–1.0 after feature extraction, this would result in noisy data as the signal was not filtered prior to feature extraction, and both ambient noise and harmonic overtones of the music are more pronounced at higher frequencies. The Bregman library offers a number of parameters for fine-tuning the calculation of chroma (including notably a setting for switching between linear and logarithmic intensity scales), but none of these addressed the fundamental problems experienced with the library.

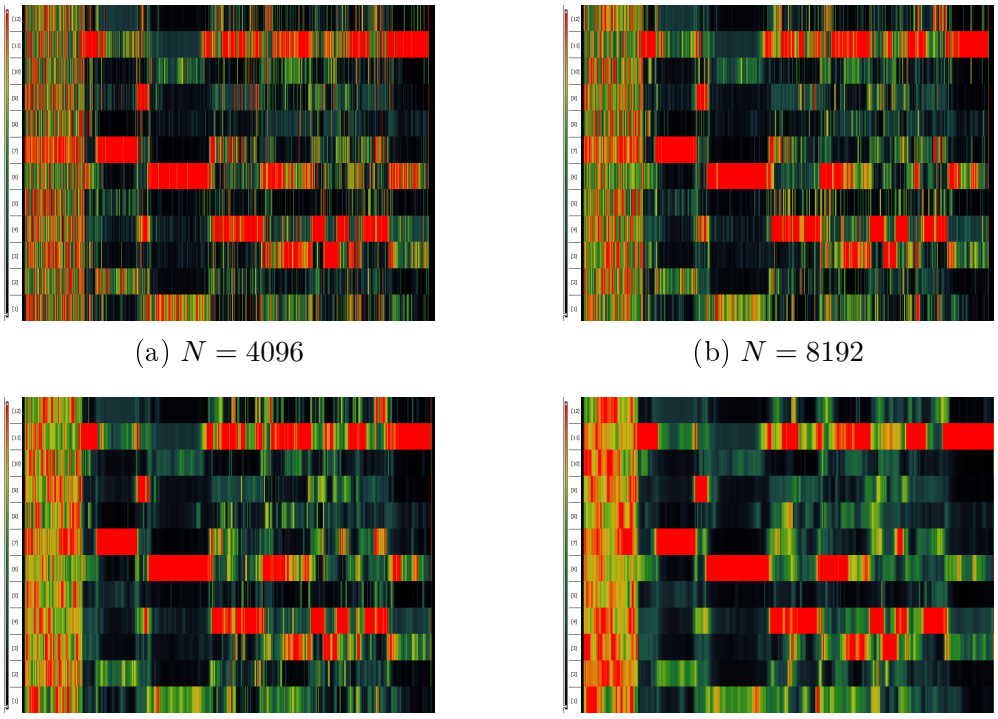
HPCP vamp plugin

Harmonic pitch class profiles (HPCP) present an alternative to the chroma feature. The HPCP feature is generated by a Vamp plugin [44]. This plugin can be used in conjunction with audio analysis applications compatible with the Vamp plugin system developed by the Centre for Digital Music at QMUL [45], such as Sonic Annotator and Sonic Visualiser. A graphical representation of HPCP can be generated using Sonic Visualiser [46]. Sonic Visualiser was used to to experimentally determine suitable parameters for the plugin. These parameters were then used in the command line application Sonic Annotator [47] in order to batch process the entire corpus. The following parameters were set on for HPCP extraction:

- **Number of bins:** This determines the size of the output vectors. The default value is 120, but for the LSTM this was reduced to the minimum of 12 bins (one per pitch class). This choice was made primarily out of memory considerations.
- **Highest and lowest frequencies:** The upper and lower bounds of the band-pass filter. These were left unchanged from the default values of 50 Hz to 5000 Hz.
- **Reference tuning frequency:** The frequency used for the first bin. The default value of 440 Hz (A_4) was used, as this is the most common tuning frequency across the entire corpus (although 415 Hz is often used in recordings of Baroque music).
- **Spectral whitening:** Whether or not to include a spectral whitening step prior to spectral analysis. Spectral whitening is a class of filtering process that reduces frequency attenuation by normalising signals in the frequency domain. This option was turned on in order to deliver clearer onsets.

In addition, the type, size, and increment of window used for spectral analysis could be specified. The Hann window was used to avoid boundary effects that could occur using a square window. Figure 4.4 shows the results of changing the window size. Increasing the window size has the advantage of reducing noise along the temporal axis, but it was found that increasing it to more than 16384 samples (at a sample rate of 44.1 kHz) resulted in significant temporal overlap between notes at fast tempos. A window size of 8192 samples (Figure 4.4b) was chosen for use with the LSTM, corresponding to a duration of

$$T_{Hann} = \frac{N}{f_s} = \frac{8192}{44\,100\,\text{Hz}} = 186\,\text{ms}\,. \tag{4.1}$$



(a) $N = 4096$ (b) $N = 8192$

(c) $N = 16384$ (d) $N = 32768$

Figure 4.4: 12-bin HPCP with increasing window sizes

A similar trade-off was made with respect to the window increment. A larger window increment reduces the amount of data in each file, but also reduces the temporal resolution, introducing inaccuracies in timing that may be crucial for detecting high-level rhythmic features. A window increment of 512 samples, corresponding to a temporal increment of 12 ms, was chosen as a sensible balance.

A Vamp skeleton file corresponding to these settings was created and used to batch process the BCRM dataset. The outputs, created by Sonic Annotator, were initially saved as CSV files, with the first column holding a time stamp and rows representing representing HPCP vectors. A simple Python script was written to remove the time stamp column, which is superfluous for the purposes of classification, and save the contents of the CSV files as Pickle objects, allowing faster loading by the training script.

HPCP transposition

Another Python script, `transpose_hpcp.py`, was used to create transposed copies of the HPCP files. This was done by performing circular rotations on each of the output vectors, as demonstrated in Listing 4.5. The script takes an integer representing the number of transpositions as an input, with values in the range of 1 to 11 (not including the null transposition). This script was run before splitting the outputs into training, validation, and test sets.

```
1 def transpose(src_dir, out_dir, file):
```

```
    data = _load(src_dir, file)
    # Save unchanged for null transposition
    _save(data, out_dir, file, 0)
    for i in range(n_transpositions):
        for r, row in enumerate(data):
            # Rotate list by one
            data[r] = [row[-1]] + row[:-1]
        _save(data, out_dir, file, i+1)
```

Listing 4.5: HPCP transposition function

4.3 Classification

Out of a large number of neural network frameworks that are currently freely available, Theano, TensorFlow, and PyTorch were considered for use in this project. All three are well established frameworks with similar functionality, but active development on Theano was discontinued in November 2017 [48]. Out of the two newer frameworks, PyTorch (Facebook's Python implementation of Torch) has perhaps the more intuitive API. However, Google's TensorFlow has experienced more widespread adoption, and the resulting abundance of libraries, tutorials, and support pages made this the preferred choice for this project, particularly in light of the brief amount of time available for learning the API [49].

4.3.1 Hardware and use of Amazon EC2

Training was initially performed on the author's home computer, which uses an Intel Core i7-4710HQ CPU and 8 GB RAM. The lack of a dedicated GPU meant that training was slow and impacted on system performance.

In order to speed up training times and allow multiple runs to be executed simultaneously, the use of Amazon's EC2 cloud computing service was investigated [50]. It was found that the free-tier instances would not be of any practical use to the project due to memory and storage constraints; however, larger instances can be obtained as Spot instances, which have a fraction of the cost of a normal instance (the exact price fluctuates based on demand), but are not guaranteed to run uninterrupted. This was not considered a problem for a machine learning project as the process can be interrupted at any point and will continue once the instance becomes available again.

One major drawback of the service was that instances which included GPU capabilities were prohibitively expensive. This meant that a CPU-only t2.large instance (with 8 GB memory) was used, which did not run noticeably faster

than the Intel CPU, but did free up resources for other purposes. Depending on training parameters, which dataset was being used, and fluctuations in the performance of EC2, a training run on the Spot instance required between 3 and 6 days to complete. This constituted the main bottleneck of the project and resulted in a limited number of runs being executed.

The EC2 instance was launched with Ubuntu and the Deep Learning AMI (Amazon Machine Image), which comes with TensorFlow pre-installed. Once samples had been uploaded, an image of the machine was saved to allow future instances to be created without requiring samples to be re-uploaded. The EC2 instance was accessed using SSH, with training scripts running in a tmux session. Once the corpus and code base had been uploaded, the storage was exported to a new image, allowing later instances to be created without the need to upload data repeatedly.

4.3.2 CNN with spectrogram

A script used for classifying voice commands in using a CNN in conjunction with spectrograms is included as one of the standard example scripts in the TensorFlow repository [51]. The training script for this network is located at `tensorflow/examples/speech_commands/train.py`. This script was used as a starting point for music classification. The script can be run using the following command:

```
python train.py --data_url= --data_dir="corpus/ms" \
    --sample_rate=44100 --clip_duration_ms=10000 \
    --wanted_words="mozart,shost"
```

The `--data_dir` parameter is set to `None` to prevent the script from downloading the Speech Commands dataset. The `--clip_duration_ms` parameter indicates the sample length in ms, and was chosen to be 10 s, as CNNs learn best from large numbers of samples, and performance is unlikely to improve given longer samples. Finally, `--wanted_words` indicates a comma-separated list of directory names representing labels. The sub-directories of the MS dataset were used for this purpose. The script initially ran for 18000 epochs, but it quickly became clear that an epoch on the MS dataset was slower by an order of magnitude, so the script was modified to run for 2000 epochs. The script initially used a learning rate of 0.001. After 1500 steps, this was reduced to 0.0001 in order to fine-tune the weights. The script automatically splits samples into training and validation sets, and prints the validation accuracy along with a confusion matrix every 400 epochs.

Although the initial results of this approach were promising (c.f. Section 5.1.1), the large quantities of data implied by the use of raw 44.1 kHz audio meant that training was extremely slow, and the script was terminated after around 1000 to 1500 steps due to insufficient memory. In order to counter this problem, the MS dataset was down-sampled to 16 kHz. Section 5.1.2 offers a comparison of results at different sample rates. Due to the difficulties caused by the high memory demands of the script, it was decided to proceed to the LSTM portion of the project without running the CNN on the significantly larger BCRM dataset.

4.3.3 LSTM with HPCP

A script named `hpcp_lstm_train.py` was written to train a simple RNN on the corpus. The full LSTM script can be found in Appendix B. The network consists of a single LSTM cell with 24 hidden units and an input dimension of 12 (corresponding to the 12 HPCP bins). The output of the network is a vector of probabilities corresponding to the four output labels. The loss function computed the cross entropy of these values with the one-hot encoded labels, which were generated using the function shown in Listing 4.6.

```
20 def _one_hot(sample, labels, comp):
21     if comp:
22         return [1 if sample[2:5] == x else 0 for x in labels]
23     return [1 if sample[0] == x else 0 for x in labels]
```

Listing 4.6: LSTM one-hot encoding function

The Adam optimizer [52] was used for gradient descent, as this obviated the need to manually determine learning rates. The optimizer was used with default settings, which in TensorFlow are `learning_rate=0.001`, `beta1=0.9`, `beta2=0.999`, and `epsilon=1e-08`. Some difficulty was experienced with the optimizer, as the cross entropy would abruptly increase on some epochs, accompanied by a sudden drop in accuracy. The exact cause of these anomalies could not be determined, although a too small value for epsilon was suspected. Although the correct parameters for this configuration were not determined, the anomalies did not prevent correlations between changes to the data and accuracy from being observed.

In order to limit the amount of memory used by the script, a constant `TRAIN_SAMPLES` was defined, which specified the number of training samples to use in each epoch. These samples were randomly selected from the training samples using the `_load_data()` function shown in Listing 4.7. This way, the network was trained on the entire training set over the course of multiple epochs, without the total volume of data stored by the script in memory during each epoch

exceeding 3.5 GB. This was a simple but effective means of circumventing the memory issues experienced with the CNN script without significantly impacting performance or necessitating the added complication of using the Dataset and Estimator APIs.

```
26 def _load_data(path, labels, comp, limit=None):
27     samples = os.listdir(path)
28     np.random.shuffle(samples)
29     d = []
30     o = []
31     if not limit:
32         limit = len(samples)
33     for i in range(limit):
34         sample = samples[i]
35         file = os.path.join(path, sample)
36         d.append(pickle.load(open(file, 'rb')))
37         o.append(_one_hot(sample, labels, comp))
38     return d, o
```

Listing 4.7: Data-loading function for LSTM script

Figure 4.5 shows the graph of the LSTM architecture generated by TensorBoard. This simple LSTM architecture was intended as a starting point for experimentation; however, the number of variants in the input data and parameters meant that alternative architectures were not explored. This was because the architecture and the input format should not be changed at the same time; if this were done it would be impossible to correctly attribute any improvements in performance to one factor. For this reason a large number of training runs are required in order to find the optimal combination of architecture and input features, which was not feasible in the time available. Therefore, emphasis was instead placed on identifying good parameters for the input data.

Since calculating the validation accuracy required a fraction of the time taken by training, validation was performed after each epoch. Every 50 epochs, a checkpoint was created and a confusion matrix was generated and printed to the terminal. A final test accuracy and confusion matrix on the test set were printed after 1000 epochs.

LSTM with 60-second samples

Unlike the speech command script, this script accepted a directory containing separate train, validation and test directories, so that the split was preserved between runs. After initial results on the MS dataset showed promise, the BCRM dataset was used. The duration of a full run of the training script on this corpus

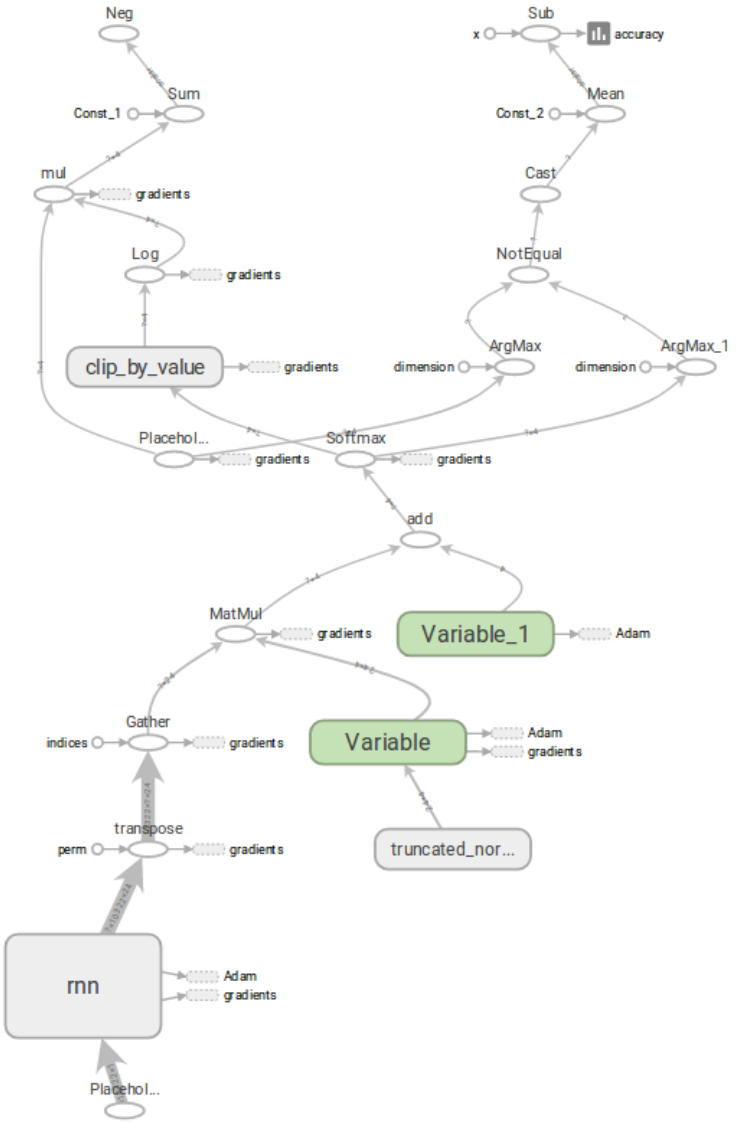


Figure 4.5: TensorBoard graph of LSTM architecture

was around five to six days. The `SEQUENCE_LENGTH` constant was set to 10322, the number of vectors in the 60 s HPCP samples. Due to clear signs of overfitting in the results (Section 5.2.1), it was established that a larger dataset was necessary.

LSTM with 15-second samples and transposition

The first step in increasing the number of samples was to create a new BCRM dataset from the original audio source files with samples of duration 15 s. This resulted in a total of 9600 samples with a `SEQUENCE_LENGTH` of 2570, and `TRAIN_SAMPLES` was increased to 2000, which significantly reduced overfitting.

Finally, an additional five transpositions were generated from this dataset, resulting in a final total of 57600 samples, of which 3% each were allocated to validation and testing.

Reduction in batch size

The batch size was one final parameter that was adjusted to improve results. Initially this was set to 100, resulting in 5 batches per epoch for the 60 s samples. It was found that reducing the batch size to 10, with 50 batches per epoch, had a significant positive effect on results.

Composer identification

In order to gain more insight into the types of error being made by the network than could be gleaned from the basic BCRM confusion matrix, a variant of the script (`hpcp_lstm_train_composer.py`) was created with precisely the same training process that classifies the twelve composers in the dataset instead. This involved only a trivial change to the code, which is shown in Listing 4.8.

```
11 HPCP_PATH = "corpus/bcrm_hpcp_12bin"  # Source directory
12 CHKPT_INTERVAL = 50  # How often to create checkpoints
13 INPUT_DIMENSION = 12  # Number of HPCP bins
14 SEQUENCE_LENGTH = 2570  # 10322 for 60 second samples
15 BATCH_SIZE = 10
16 LABELS = ['cor', 'viv', 'bac', 'hay', 'moz', 'bee',
17           'bra', 'tch', 'mah', 'str', 'sho', 'mes']
18 N_EPOCHS = 1000
19 TRAIN_SAMPLES = 2000  # 500 for 60 second samples
20 N_HIDDEN = 24
21
22
23 def _one_hot(sample):
24     return [1 if sample[2:5] == x else 0 for x in LABELS]
```

Listing 4.8: Changes to LSTM script to classify composers

Classifier script

Finally a classifier script (`hpcp_lstm_classify.py`) was created which takes in a single 15 s HPCP sample and prints a probabilistic prediction of both the period and the composer to the terminal. This script served to demonstrate predictions being made by the fully trained network. Due to the reliance on the Vamp plugin to generate HPCP samples, the script is unable to take in raw audio files, but as

the file names of the HPCP samples match those of the source audio files they originated from, playing the correct audio for a labelled track is trivial.

Chapter 5

Results

5.1 CNN with spectrograms

5.1.1 MS dataset

Figure 5.1 shows the results of a training run of the CNN/spectrogram method on the MS dataset (c.f. Section 3.1.2), with validation accuracies calculated every 400 steps, as described in Section 4.3.2. As can be seen, the training accuracy exceeds 90% accuracy, with the curve of the validation accuracy closely tracking that of the training accuracy. This is promising, as it shows that the model is a good fit for the data, with no over- or underfitting taking place.

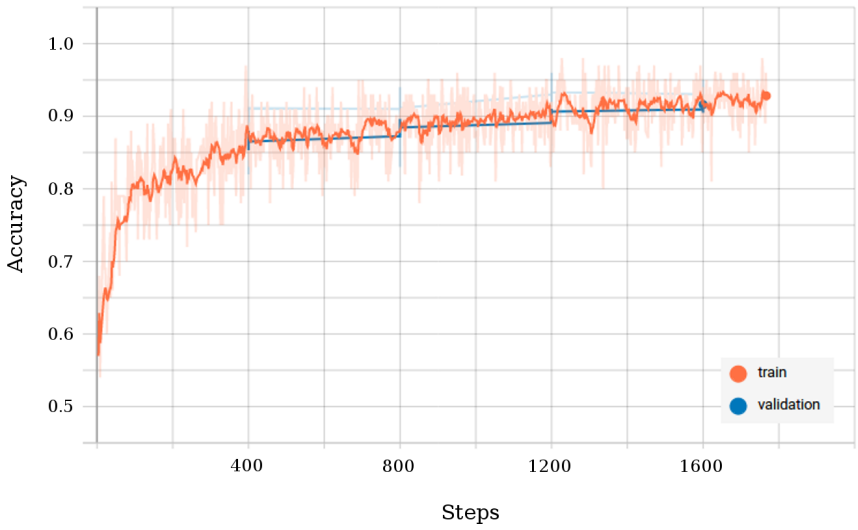


Figure 5.1: CNN accuracy for 44.1 kHz samples

Unfortunately, despite several attempts to solve the issue, memory problems resulted in the script being terminated after around 1500–1800 steps so that the training run could not be completed. The final validation accuracy for the run shown in Figure 5.1 was 91.6%. Table 5.1 shows the confusion matrix for the

final validation step of this run. The confusion matrix maps each prediction onto its ground-truth label, allowing inferences to be made about the types of error affecting accuracy levels. In this case, the matrix shows that tracks by Mozart are more likely to be identified as Shostakovich than the reverse.

Table 5.1: Confusion matrix for CNN classifying MS dataset

		Prediction		
	$(n = 960)$	M	S	
Actual value	M	**362**	52	414
	S	27	**418**	454
		389	470	

5.1.2 Effect of down-sampling

In order to reduce the memory footprint of the script, the sample rate of the MS segments was changed to 16 kHz. This was met with partial success in that it enabled training to proceed for longer, although the script still ran out of memory eventually, resulting in the accuracy plot shown in Figure 5.2. While the curve for the training accuracy is very similar to that shown in Figure 5.1, the curve representing the validation accuracy is far more erratic, with a final accuracy of around 80%.

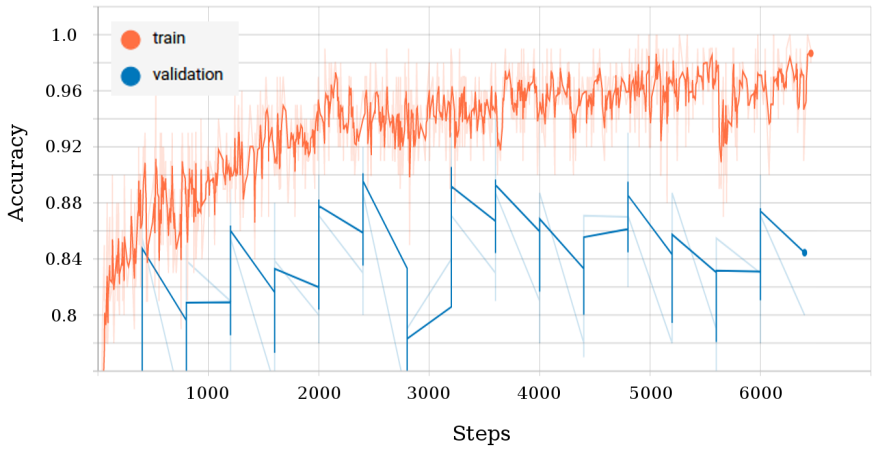


Figure 5.2: CNN accuracy for 16 kHz samples

This clearly demonstrates that more overfitting occurs in the run with 16 kHz samples than at 44.1 kHz. While the accuracy score of 80% may seem high, it is worth bearing in mind that this is the easiest test case, with a binary classifier. Results can be expected to drop dramatically when using the BCRM dataset, due

both to the increased number of categories and the significantly smaller number of samples per composer.

The difference in accuracy between the original files and the down-sampled dataset is an indicator that the former run used information in the upper frequency spectrum that was not available to the latter. As the sample rate after down-sampling was 16 kHz, all frequency information above 8 kHz is lost. This threshold is high enough that no harmonic or melodic information is lost, suggesting that this method results in a timbre-driven approach that relies on high-frequency features.

5.2 LSTM with HPCP

5.2.1 60-second samples

The LSTM from Section 4.3.3 was initially run on 60 s samples (with a batch size of 10 and 500 training samples used per epoch), resulting in the training and validation accuracy shown in Figure 5.3. After 1000 steps, the training accuracy had reached around 77%, while the validation accuracy, which had initially risen slightly, dropped down to around 25%. For a four-way classifier in which all categories are equally prevalent, this is equivalent to the null error rate of 75% (i.e. the error rate that would be obtained by always making the same prediction). The final test accuracy was even lower, at 23%. These results are strongly suggestive of overfitting, given that the training accuracy rose steadily.

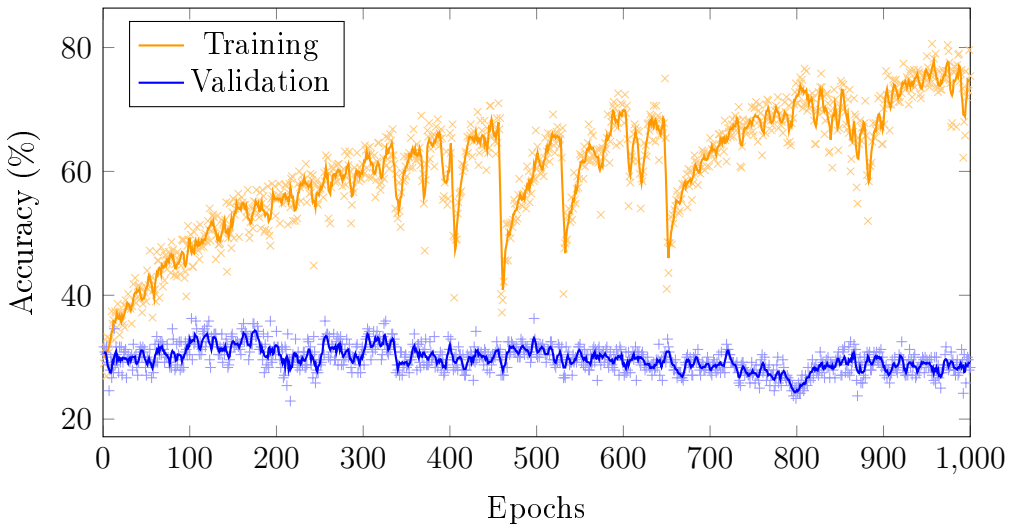


Figure 5.3: Accuracy of LSTM (batch size = 10) on BCRM dataset with 60 s HPCP samples

5.2.2 15-second samples

In order to combat the overfitting of the previous section, the sample length was reduced to 15 s in order to increase the number of samples available for training. Figure 5.4 shows the results of this training process with a batch size of 100 and 2000 training samples used per epoch.

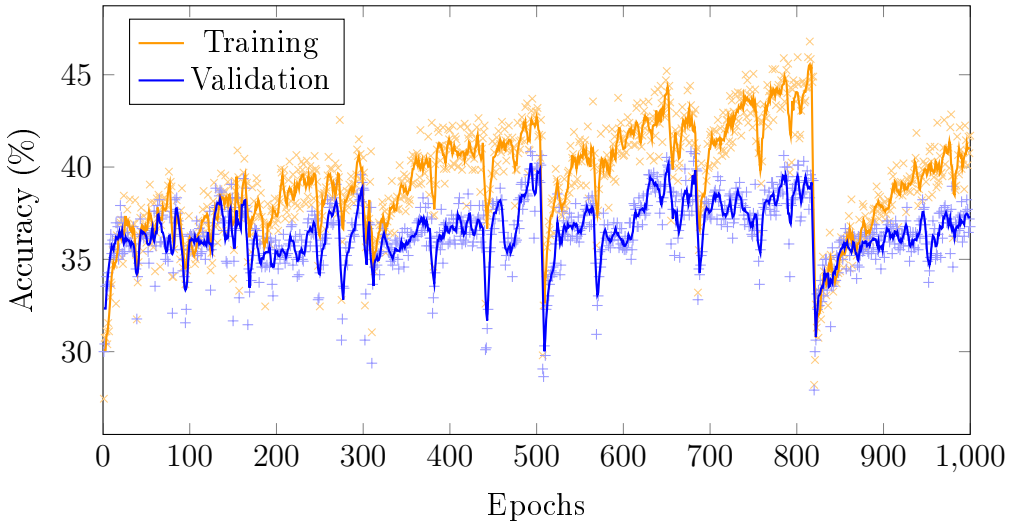


Figure 5.4: Accuracy of LSTM (batch size = 100) on BCRM dataset with 15 s HPCP samples

The validation accuracy reached around 37% with this data, and the graph clearly shows that the validation accuracy follows the training accuracy, suggesting that the overfitting problem had been solved by the change in sample length. However, the training accuracy never reached above 45%. After some experimentation it was found that reducing the batch size significantly improved performance. This is because a reduction in batch size increases the number of batches calculated per epoch. Any associated reduction in the precision of the gradient descent calculation is handled well by the optimizer.

Figure 5.5 shows the results with the lower batch size. While some overfitting is still in evidence, the validation accuracy of 50% is a significant improvement over initial efforts.

Table 5.2 shows the confusion matrix for the final test accuracy of this run. This matrix shows that for each category, the correct label is also the most common prediction.

As it can be difficult to spot patterns in tables of numbers, a colour scale of the confusion matrix was created. This diagram replaces the values with coloured fields, where the colour indicates the value. The colours are normalised such that the highest value is indicated in green and the lowest in red, with intermediate values forming a spectrum. Figure 5.6 shows the colour scale for the matrix in

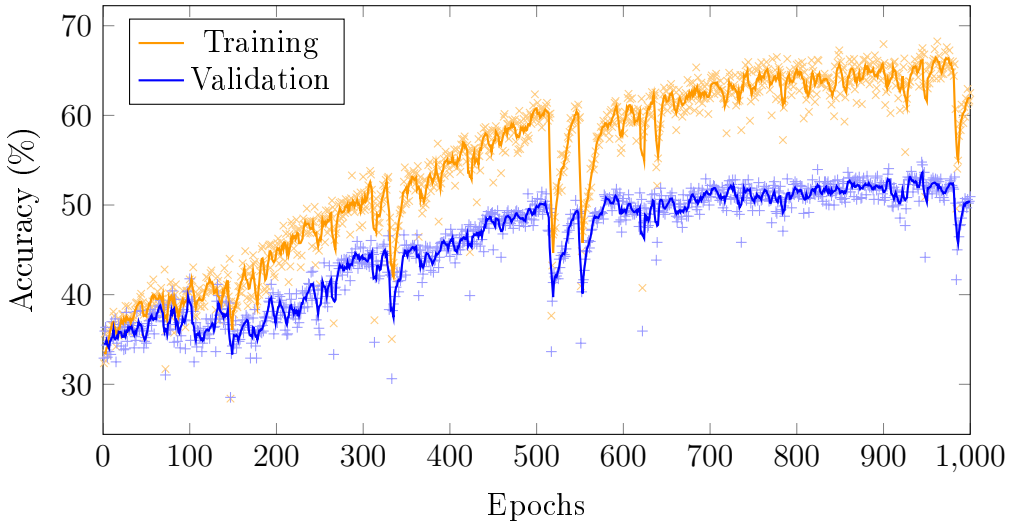


Figure 5.5: Accuracy of LSTM (batch size = 10) on BCRM dataset with 15 s HPCP samples

Table 5.2: Confusion matrix for LSTM classifying periods

		Prediction			
	$(n = 960)$	B	C	R	M
Actual value	B	**148**	58	27	7
	C	44	**125**	48	23
	R	25	62	**88**	65
	M	10	32	59	**139**
		227	277	222	234

Table 5.2, along with an ideal colour scale matrix for reference (i.e. the colour scale of a confusion matrix in which only correct predictions were made).

(a) (b)

Figure 5.6: Ideal (a) and actual (b) confusion matrix on a colour scale

The clear presence of the green diagonal is a sign that most samples are being classified correctly; however, it is the yellow bands on either side of this diagonal

which are most interesting, as they show that where errors occur, it is more common for periods to be mistaken for adjacent eras than for ones far removed in history.

5.2.3 Transposition

In an attempt to further improve the accuracy of the model, five transpositions of the HPCP samples were created, which together with the originals resulted in a dataset six times the size. Figure 5.7 shows the training and validation accuracy of this run. While the accuracy did not experience huge improvements, with validation accuracy at around 53% after 1000 steps, the inclusion of transpositions resulted in a significant reduction of overfitting, showing that this strategy is an effective means of compensating for small datasets. The final test accuracy of 54.9% was the highest accuracy recorded on the full BCRM dataset.

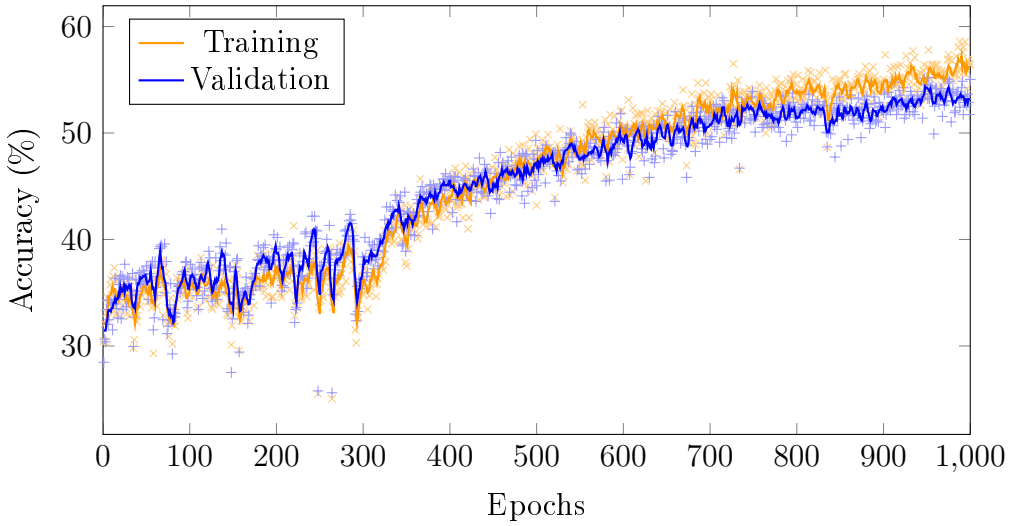


Figure 5.7: Accuracy of LSTM on BCRM dataset with 6 transpositions

5.2.4 Composer classification

As a final experiment, the same script was trained to classify composers instead of compositional periods. Figure 5.8 shows the training and validation accuracy. Some overfitting is in evidence here (transpositions had not been performed prior to this run), but the final test accuracy of 30.6% is significantly better than the null error rate, which for a twelve-way classifier is 91.7% (equivalent to an accuracy of 8.3%).

Table 5.3 shows the confusion matrix for the composer classification run. A particularly noticeable feature of this matrix is the diagonal symmetry, with particularly strong results amongst Baroque and Modern composers, while the

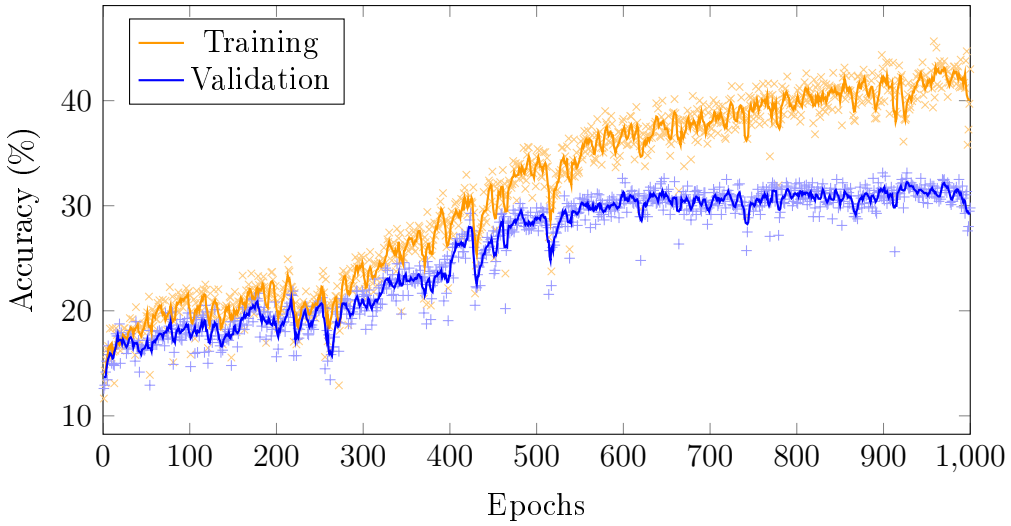


Figure 5.8: Accuracy of LSTM classifying BCRM composers

number of correct predictions in the Classical and middle Romantic composers is significantly lower. Corelli and Messiaen stand out in particular, with 63% of samples by each classified correctly.

Table 5.3: Confusion matrix for LSTM classifying composers

		Prediction											
	($n = 960$)	Cor	Viv	Bac	Hay	Moz	Bee	Bra	Tch	Mah	Str	Sho	Mes
Actual value	Cor	**50**	8	1	11	5	2	2	1	0	0	0	0
	Viv	4	**31**	6	10	9	2	4	2	4	0	8	0
	Bac	7	17	**17**	8	1	4	6	9	3	4	3	1
	Hay	8	16	5	**24**	10	4	6	2	3	0	2	0
	Moz	8	8	4	12	**35**	4	1	1	1	2	4	0
	Bee	4	8	4	11	10	**10**	12	6	3	5	5	2
	Bra	3	8	6	3	3	4	**9**	9	18	11	4	2
	Tch	3	2	9	5	2	7	8	**17**	4	8	7	8
	Mah	0	5	5	4	4	4	6	11	**11**	13	11	6
	Str	0	7	1	1	4	4	3	10	8	**19**	16	7
	Sho	0	4	1	4	1	2	4	9	6	17	**21**	11
	Mes	0	2	1	0	5	2	0	3	4	2	11	**50**
		87	116	60	93	89	49	61	80	65	81	92	87

As before, a colour scale of the confusion matrix, shown in Figure 5.9 makes some patterns easier to perceive visually. The most striking feature of this diagram is the clear green diagonal running across most of the categories, corresponding to the correct predictions made. Another observation is that the frequency of errors appears to decrease the further from this diagonal the

predictions get, suggesting that the LSTM has learnt differences between periods more strongly than between composers of the same period.

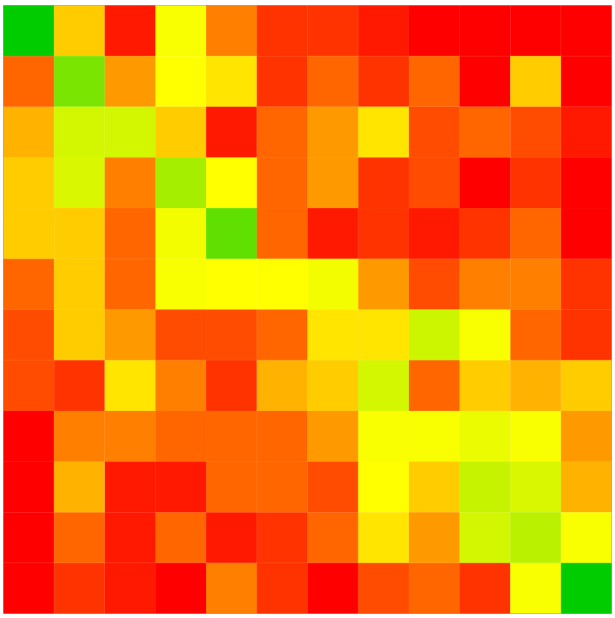


Figure 5.9: Colour scale of LSTM confusion matrix for composers

This intuition can be confirmed by analysing how the number of errors made varies with the historical "distance" between composers, where the notion of distance is simulated by calculating the the difference between the column of a prediction in the confusion matrix and the column corresponding to the correct prediction, as shown in Table 5.4.

Table 5.4: Distance values for pairs of composers

	Cor	Viv	Bac	Hay	Moz	Bee	Bra	Tch	Mah	Str	Sho	Mes
Cor	0	1	2	3	4	5	6	7	8	9	10	11
Viv	1	0	1	2	3	4	5	6	7	8	9	10
Bac	2	1	0	1	2	3	4	5	6	7	8	9
Hay	3	2	1	0	1	2	3	4	5	6	7	8
Moz	4	3	2	1	0	1	2	3	4	5	6	7
Bee	5	4	3	2	1	0	1	2	3	4	5	6
Bra	6	5	4	3	2	1	0	1	2	3	4	5
Tch	7	6	5	4	3	2	1	0	1	2	3	4
Mah	8	7	6	5	4	3	2	1	0	1	2	3
Str	9	8	7	6	5	4	3	2	1	0	1	2
Sho	10	9	8	7	6	5	4	3	2	1	0	1
Mes	11	10	9	8	7	6	5	4	3	2	1	0

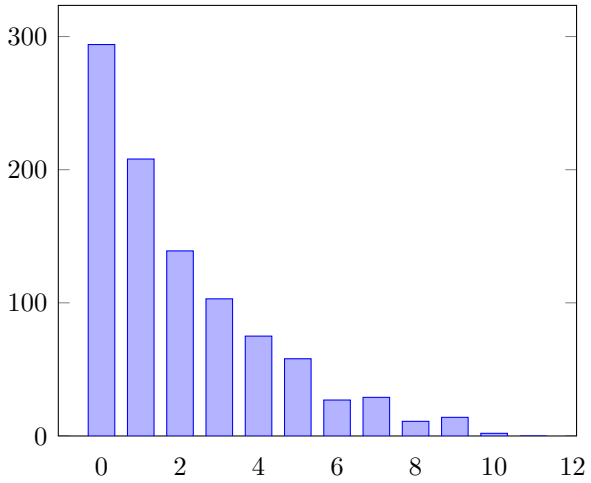


Figure 5.10: Number of errors against metric of distance between comosers

After summing all the values along the diagonals that correspond to the same distances, the number of these errors can be plotted as shown in Figure 5.10. The number of errors falls sharply as the distance metric increases, confirming the intuition gained from the colour scale that "near miss" errors are significantly more common than broad misclassifications. Around 67% of predictions are either correct or within distance 2 of the correct prediction, an error equivalent to mistaking Beethoven for Haydn or Stravinsky for Tchaikovsky.

A number of other patterns are apparent in the data. For instance, pairs of composers can be found that the network struggles to tell apart more than others. Such pairs include Haydn and Mozart, Beethoven and Brahms, and Stravinsky and Shostakovich. It is noteworthy that all of these pairings are adjacent in the confusion matrix, and Haydn and Mozart in particular can be difficult even for trained musicians to tell apart.

The high levels of confusion between Stravinsky and Shostakovich is telling, as in many respects these two composers have quite different styles. Particularly in terms of rhythmic patterns, Stravinsky's music tends to be far more complex and irregular than that of Shostakovich. In terms of complexity of harmony, however, there is considerable overlap between these composers. While it is difficult to draw conclusions about the high-level features being learnt by the LSTM, this relationship makes it seem likely that harmony plays a greater role than rhythm.

Finally, the errors made by the LSTM while classifying composers were grouped by period, as shown in the derived confusion matrix in Table 5.5. This table shows remarkable similarities to Table 5.2, and indeed the test accuracy calculated from these values shows that the LSTM trained on composers effectively predicts the period (or rather, predicts a composer from the correct

period) with an accuracy of around 52%. This is comparable to the results obtained by training the same network specifically to detect periods.

Table 5.5: Confusion matrix for classifying periods using an LSTM trained on composers

		Prediction			
($n = 960$)		B	C	R	M
Actual value	B	**141**	52	31	16
	C	65	**120**	35	20
	R	41	36	**93**	70
	M	16	23	47	**154**
		227	277	222	234

This realisation has two important implications: firstly, it demonstrates that musical styles of composers within each period are indeed closely related to features that are typical of those periods, and that therefore the hierarchical subdivision of the dataset into periods and composers is sensible; and secondly, it means that in future work there is no need to train separate instances of the network on periods and composers. Instead, an era prediction can be generated from the composer prediction together with a look-up table that maps composers to their respective periods.

Chapter 6

Conclusion

Two fundamentally different methods of using neural networks for the classification of orchestral music have been investigated in this paper, with both the network architecture and the features used differing considerably. In both cases results indicative of successful learning were obtained, although neither approach came close to matching results achieved in some of the earlier research investigated in Section 2.5. The reasons for this were different for the two methods.

In the case of the CNN used in conjunction with the spectrograms, initial results of over 90% for a binary classification were promising, and better results could almost certainly have been obtained with improved hardware, as it was the demands of the large quantities of data that caused the application to fail. Given more time, a solution to this problem could undoubtedly be found, although this would not address the underlying issue that the method is highly reliant on extremely large datasets. As is evident from the literature, while CNNs are a popular choice for music classification due to the good results that can be obtained by them, the large datasets and extensive training required are a fundamental shortcoming of this approach.

In contrast to this, no memory problems were experienced where LSTMs were applied to chroma features. There are several reasons for this, ranging from the more compact data representation in the form of the HPCP feature to the smaller architecture of the LSTM cell. The main limitation of the LSTM approach was that, due to the difficulties in parallelising this type of architecture, training times were even more extended than those of the CNN.

The latter approach also suffered from poorer results than the CNN, with test accuracies reaching around 55% in a four-way classification. This was due to significant underfitting, which was the result of the use of too simple a model. This issue could be addressed either by building a more complex model using a stack of LSTM cells or by increasing the number of hidden layers in each cell.

Unfortunately, time constraints did not permit these improvements from being fully explored.

The effects of several other variables on the performance of this system were however determined, including the impact of changing the sample length and the mini-batch size hyper-parameter. The possibility of using transpositions to compensate for smaller datasets was also explored, with promising results; the inclusion of transposed samples resulted in a significant reduction of overfitting for the RNN. This technique can be used to improve results in cases where the dataset is insufficiently large.

Finally, a more fine-grained classification by composers, with twelve categories used, revealed interesting patterns in the types of error made by the network. In particular, it seems clear that the network can distinguish more clearly between styles that are distant in time, and that there are some composers, such as Messiaen, that the algorithm is consistently better at classifying than others. The precise reason for these propensities cannot be known, but it seems likely that composers with particularly idiosyncratic harmonic writing styles are easier to identify, as the clusterings of errors tend to centre around composers with similar levels of harmonic dissonance.

One notable difference between the two approaches examined is that the LSTM uses a timbre-invariant feature, whereas the use of CNNs together with spectrograms appears to result in a timbre-driven approach. The fact that these disparate feature sets can result in similar classifications is a sign of the variety of approaches that can be successfully applied to this problem, which is one of the primary reasons that this remains an ongoing area of research.

6.1 Future Work

It should be noted that while the results presented in Section 5.2 do not come close to some results found in the literature, these results were obtained with the simplest possible LSTM architecture, consisting of only a single cell. One area where further work is required is in finding the optimal architecture for this type of problem, where the number and configuration of LSTM cells is sufficiently complex to learn from the low-level sequential audio data without being so large that substantial overfitting is inevitable.

Furthermore, a hybrid approach between the two presented in this paper has the potential to outperform either algorithm by itself. A CNN, while not very efficient at identifying compositional styles directly from the spectrogram, is the obvious tool for detecting onsets of notes in an HPCP sample. Using this approach, a sample could be converted into a sequence of simultaneous pitch

classes with durations, substantially compressing the most valuable information that can then be learnt by an LSTM. This is similar to the approach investigated by Choi *et al.*, but using a different set of features [35]. The difficulty in this approach lies in training the CNN; this would require a large number of samples to be labelled manually, a time-consuming task that is beyond the scope of this project.

Bibliography

[1] G. Tzanetakis and P. Cook, 'Musical genre classification of audio signals', *IEEE Transactions on Speech and Audio Processing*, vol. 10, no. 5, pp. 293–302, 2002.

[2] S. Walsh, *Stravinsky, Igor*, in *The New Grove Dictionary of Music and Musicians*, 2nd ed., London, UK: Oxford University Press, 2001.

[3] C. Palisca, *Baroque*, in *The New Grove Dictionary of Music and Musicians*, 2nd ed., London, UK: Oxford University Press, 2001.

[4] D. Heartz and B. Brown, *Classical*, in *The New Grove Dictionary of Music and Musicians*, 2nd ed., London, UK: Oxford University Press, 2001.

[5] J. Samson, *Romanticism*, in *The New Grove Dictionary of Music and Musicians*, 2nd ed., London, UK: Oxford University Press, 2001.

[6] E. Salzman, *Twentieth-century Music, An Introduction*, 4th ed. London, UK: Prentice Hall, 2002, ISBN: 9780130959416.

[7] Freiburger Barockorchester. (2000). Bach: Brandenburg Concerto no. 2 in F major, YouTube, [Online]. Available: `https://www.youtube.com/watch?v=3HSRIDtwsfM` (visited on 18/03/2018).

[8] M. Maisky and Wiener Symphoniker. (2014). Haydn: Cello Concerto no. 1 in C major, YouTube, [Online]. Available: `https://www.youtube.com/watch?v=mooB5Q-0FIE` (visited on 18/03/2018).

[9] V. Gergiev and London Symphony Orchestra. (2007). Tchaikovsky: Romeo and Juliet overture, YouTube, [Online]. Available: `https://www.youtube.com/watch?v=Cxj8vSS2ELU` (visited on 18/03/2018).

[10] S. Rattle and London Symphony Orchestra. (2017). Stravinsky: The Rite of Spring, YouTube, [Online]. Available: `https://www.youtube.com/watch?v=EkwqPJZe8ms` (visited on 19/03/2018).

[11] S. Bella and I. Peretz, 'Differentiation of classical music requires little learning but rhythm', *Cognition*, vol. 96, no. 2, pp. B65–B78, Jun. 2005.

[12] S. Takeda, I. Morioka, K. Miyashita *et al.*, 'Age variation in the upper limit of hearing', *European J. Applied Physiology*, vol. 65, no. 5, pp. 403–408, Nov. 1992.

[13] J. London, 'Cognitive constraints on metric systems: Some observations and hypotheses', *Music Perception*, vol. 19, no. 4, pp. 529–550, Jun. 2002.

[14] A. Mutter and Mutter Virtuosi. (2014). Vivaldi: The Four Seasons, Summer, 3. Presto, YouTube, [Online]. Available: `https://www.youtube.com/watch?v=124NoPUBDvA` (visited on 11/03/2018).

[15] B. S. Everitt and A. Skrondal, *Overfitted models*, in *The Cambridge Dictionary of Statistics*, 4th ed., Cambridge, UK: Cambridge University Press, 2010.

[16] I. Goodfellow, B. Yoshua and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, ISBN: 9780262035613.

[17] S. E. Dreyfus, 'Artificial neural networks, back propagation, and the Kelley-Bryson gradient procedure', *J. Guidance, Control, and Dynamics*, vol. 13, no. 5, pp. 926–928, 1990.

[18] A. Karpathy. (2015). The unreasonable effectiveness of recurrent neural networks, [Online]. Available: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[19] C. Olah. (2015). Understanding LSTM networks, [Online]. Available: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[20] Y. Bengio, P. Simard and P. Frasconi, 'Learning long-term dependencies with gradient descent is difficult', *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[21] F. J. Harris, 'On the use of windows for harmonic analysis with the discrete Fourier transform', in *Proc. IEEE*, vol. 66, Newark, DE, USA, Jan. 1978, pp. 51–83.

[22] M. Müller, *Fundamentals of Music Processing, Audio, Analysis, Algorithms, Applications*. New York City, NY, USA: Springer, 2015, ISBN: 9783319219448.

[23] M. Müller. (2016). Chroma feature C major scale, Wikimedia, [Online]. Available: `https://commons.wikimedia.org/wiki/File:ChromaFeatureCmajorScaleScoreAudioColor.png`.

[24] O. Kühl and K. K. Jensen, 'Retrieving and recreating musical form', in *Computer Music Modeling and Retrieval*, Copenhagen, Denmark, Aug. 2007, pp. 263–275.

[25] M. P. Ryynanen and A. Klapuri, 'Polyphonic music transcription using note event modelling', in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, NY, USA, Oct. 2005, pp. 319–322.

[26] L. Yang, S. Chou and Y. Yang, 'Midinet: A convolutional generative adversarial network for symbolic-domain music generation', in *Int. Soc. Music Information Retrieval Conf.*, Suzhou, China, Oct. 2017, pp. 324–331.

[27] B. L. Sturm, 'A survey of evaluation in music genre recognition', *Lecture Notes in Computer Science*, vol. 8382, pp. 29–66, 2014.

[28] C. Lin, N. Liu, H. Yi and A. Chen, 'Music classification using significant repeating patterns', *Lecture Notes in Computer Science*, vol. 2937, pp. 506–518, 2004.

[29] C. McKay and I. Fujinaga, 'Automatic genre classification using large high-level musical feature sets', in *Proc. Int. Conf. Music Information Retrieval*, Barcelona, Spain, Oct. 2004, pp. 525–530.

[30] C. Weiß, M. Mauch and S. Dixon, 'Timbre-invariant audio features for style analysis of classical music', in *Proc. Int. Computer Music Conf.*, Athens, Greece, Sep. 2014, pp. 1461–1468.

[31] C. Weihs, U. Ligges, F. Mörchen *et al.*, 'Classification in music research', *Advances in Data Analysis and Classification*, vol. 1, no. 3, pp. 255–291, Nov. 2007.

[32] 'Learning musical structure and style with neural networks', *Computer Music J.*, vol. 22, no. 4, pp. 44–62, 1998.

[33] Y. Costa, L. Oliveira and C. Silla, 'An evaluation of convolutional neural networks for music classification using spctrograms', *Applied Soft Computing*, vol. 52, pp. 28–38, 2017.

[34] J. A. Franklin and L. K. K. K., 'Recurrent neural networks for musical pitch memory and classification', *Int. J. Artificial Intelligence Tools*, vol. 14, pp. 329–342, 2005.

[35] K. Choi, G. Fazekas, M. Sandler and K. Cho, 'Convolutional recurrent neural networks for music classification', *CoRR*, vol. abs/1609.04243, 2016. [Online]. Available: `http://arxiv.org/abs/1609.04243` (visited on 21/03/2018).

[36] C. Rosen, *Sonata Forms*, revised. London, UK: Norton, 1988, ISBN: 9780393302196.

[37] J. Schluter and S. Bock, 'Improved musical onset detection with convolutional neural networks', in *Acoustics, Speech and Signal Processing, IEEE Int. Conf.*, Florence, Italy, Jul. 2014, pp. 6979–6983.

[38] E. Gomez and J. Bonada, 'Tonality visualization of polyphonic audio', in *Proc. Int. Computer Music Conf.*, Barcelona, Spain, Sep. 2005, pp. 77–86.

[39] E. Gomez, 'Tonal description of music audio signals', PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, 2006.

[40] M. Müller, F. Kurth and M. Clausen, 'Audio matching via chroma-based statistical features', in *Proc. Int. Conf. Music Information Retrieval*, London, UK, Sep. 2005, pp. 288–295.

[41] J. Robert, *Pydub (version 0.20.0)*, `https://github.com/jiaaro/pydub`, 2017. (visited on 20/03/2018).

[42] M. Casey, *Bregman Audio-Visual Information Toolbox (version 1.0)*, `http://digitalmusics.dartmouth.edu/~mcasey/bregman/`, 2010–2012. (visited on 18/03/2018).

[43] D. Kim, 'Normalization methods for input and output vectors in backpropagation neural networks', *Int. J. Computer Mathematics*, vol. 71, no. 2, pp. 161–171, 1999.

[44] J. Bonada and E. Gomez, *HPCP—Harmonic Pitch Class Profile vamp plug-in (version 1.0)*, `https://www.upf.edu/web/mtg/hpcp`, 2012. (visited on 18/03/2018).

[45] Centre for Digital Music, Queen Mary, University of London, *The Vamp audio analysis plugin system*, `https://www.vamp-plugins.org/`, 2007–2017. (visited on 22/03/2018).

[46] C. Cannam, C. Landone and M. Sandler, 'Sonic Visualiser: An open source application for viewing, analysing, and annotating music audio files', in *Proc. ACM Multimedia Int. Conf.*, Florence, Italy, Oct. 2010, pp. 1467–1468.

[47] C. Cannam, M. Jewell, C. Rhodes *et al.*, 'Linked data and you: Bringing music research software into the semantic web', *J. New Music Research*, vol. 39, no. 4, pp. 313–325, 2010.

[48] P. Lamblin. (2017). MILA and the future of Theano, [Online]. Available: `https://groups.google.com/forum/#!topic/theano-users/7Poq8BZutbY` (visited on 10/03/2018).

[49] M. Abadi, A. Agarwal, P. Barham *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, `https://www.tensorflow.org/`, 2015. (visited on 18/03/2018).

[50] Amazon Web Services, Inc., *Amazon EC2*, 2018. [Online]. Available: `https://aws.amazon.com/ec2/` (visited on 21/03/2018).

[51] TensorFlow. (2017). Simple audio recognition, [Online]. Available: `https://www.tensorflow.org/versions/master/tutorials/audio_recognition` (visited on 10/03/2018).

[52] D. Kingma and J. Ba, 'Adam: A method for stochastic optimization', *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: `http://arxiv.org/abs/1412.6980` (visited on 19/03/2018).

Appendix A

Overview of repository contents

A Git repository for this project is hosted on GitHub and can be accessed at:

`https://github.com/rddunphy/MusicClassification`

The contents of the repository are:

- `corpus`—details of the tracks included in the corpus and the datasets, as well as some analytical information about the breakdown of these contents.
- `docs`—project documentation, including this report.
- `python`—original code created for this project. All scripts are written in Python 3.6 unless otherwise specified in the file.
- `results`—graphs and tabular data summarising the results of training runs.
- `README.md`—the README file contains information about dependencies, as well as URLs of all additional resources that were used.

As the corpus is very large, the audio source files are not included in the repository (with the exception of two short test samples which were used for demonstrating DSP techniques). The datasets are available on request.

Appendix B

LSTM Training Code

This TensorFlow script trains a simple LSTM on 15 s 12-bin HPCP files (HPCP data is first converted to pickle objects in order to speed up load times). Training and validation accuracy are stored both as TensorBoard logs and in a CSV file in order to allow easy creation of graphs at a later date. The script prints a confusion matrix and creates a checkpoint of the graph every 50 epochs. (C.f. Section 4.3.3 for more details.)

Variants of this script with minor hard-coded changes were created in order to vary the length of the HPCP samples or classify by composer instead of by composition period. The full code base—including utility functions and scripts that were not used in the final approach—can be found online (c.f. Appendix A).

```
#!/usr/bin/env python3

import argparse
import csv
import os
import pickle
from datetime import datetime

import numpy as np
import tensorflow as tf

# Periods: Baroque, Classical, Romantic, Modern
PERIOD_LABELS = ['b', 'c', 'r', 'm']
# Composers: Corelli, Vivaldi, Bach, Haydn, Mozart, Beethoven,
# Barhms, Tchaikovsky, Mahler, Stravinsky, Shostakovich, Messiaen
COMP_LABELS = ['cor', 'viv', 'bac', 'hay', 'moz', 'bee',
               'bra', 'tch', 'mah', 'str', 'sho', 'mes']
```

```
18
19
20  def _one_hot(sample, labels, comp):
21      if comp:
22          return [1 if sample[2:5] == x else 0 for x in labels]
23      return [1 if sample[0] == x else 0 for x in labels]
24
25
26  def _load_data(path, labels, comp, limit=None):
27      samples = os.listdir(path)
28      np.random.shuffle(samples)
29      d = []
30      o = []
31      if not limit:
32          limit = len(samples)
33      for i in range(limit):
34          sample = samples[i]
35          file = os.path.join(path, sample)
36          d.append(pickle.load(open(file, 'rb')))
37          o.append(_one_hot(sample, labels, comp))
38      return d, o
39
40
41  def run(corpus_path, n_hidden, n_epochs, chkpt_interval, train_samples,
42          batch_size, sample_length, n_bins, comp):
43      n_batches = int(train_samples/batch_size)
44      labels = COMP_LABELS if comp else PERIOD_LABELS
45
46      run_id = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
47      data_path = os.path.join("tf", "lstm_data_" + run_id)
48      log_path = os.path.join(data_path, "log")
49      acc_csv_path = os.path.join(data_path, "acc.csv")
50
51      if not os.path.isdir(data_path):
52          os.mkdir(data_path)
53
54      with open(acc_csv_path, 'w') as csv_file:
55          writer = csv.writer(csv_file, delimiter=',', quotechar='"',
56                              quoting=csv.QUOTE_MINIMAL)
57          writer.writerow(["Epoch", "Training accuracy",
58                           "Validation accuracy"])
```

```
59
60      valid_path = os.path.join(corpus_path, "valid")
61      valid_input, valid_output = _load_data(valid_path, labels, comp)
62
63      data = tf.placeholder(
64          tf.float32, [None, sample_length, n_bins])
65      target = tf.placeholder(tf.float32, [None, len(labels)])
66
67      cell = tf.nn.rnn_cell.LSTMCell(n_hidden, state_is_tuple=True)
68
69      val, _ = tf.nn.dynamic_rnn(cell, data, dtype=tf.float32)
70      val = tf.transpose(val, [1, 0, 2])
71      last = tf.gather(val, int(val.get_shape()[0]) - 1)
72
73      W = tf.Variable(tf.truncated_normal(
74          [n_hidden, int(target.get_shape()[1])]))
75      b = tf.Variable(tf.constant(0.1, shape=[target.get_shape()[1]]))
76
77      prediction = tf.nn.softmax(tf.matmul(last, W) + b)
78
79      clipped = tf.clip_by_value(prediction, 1e-10, 1.0)
80      cross_entropy = -tf.reduce_sum(target * tf.log(clipped))
81
82      optimizer = tf.train.AdamOptimizer()
83      minimize = optimizer.minimize(cross_entropy)
84
85      mistakes = tf.not_equal(tf.argmax(target, 1),
86                              tf.argmax(prediction, 1))
87      error = tf.reduce_mean(tf.cast(mistakes, tf.float32))
88      accuracy = tf.subtract(1.0, error)
89
90      conf_matrix = tf.confusion_matrix(
91          tf.argmax(target, 1), tf.argmax(prediction, 1))
92
93      tf.summary.scalar('accuracy', accuracy)
94
95      sess = tf.Session()
96
97      merged = tf.summary.merge_all()
98      train_log_path = os.path.join(log_path, "train")
99      train_writer = tf.summary.FileWriter(train_log_path, sess.graph)
```

```
100     test_log_path = os.path.join(log_path, "validation")
101     test_writer = tf.summary.FileWriter(test_log_path)
102 
103     saver = tf.train.Saver()
104 
105     tf.global_variables_initializer().run(session=sess)
106 
107     train_path = os.path.join(corpus_path, "train")
108     for e in range(n_epochs):
109         train_input, train_output = _load_data(
110             train_path, labels, comp, limit=train_samples)
111         ptr = 0
112         for _ in range(n_batches):
113             in_ = train_input[ptr:ptr+batch_size]
114             out = train_output[ptr:ptr+batch_size]
115             ptr += batch_size
116             sess.run(minimize, {data: in_, target: out})
117 
118         train_sum, train_acc = sess.run(
119             [merged, accuracy],
120             {data: train_input, target: train_output})
121         train_writer.add_summary(train_sum, e)
122 
123         # Calculate and write validation accuracy
124         test_sum, test_acc = sess.run(
125             [merged, accuracy],
126             {data: valid_input, target: valid_output})
127         test_writer.add_summary(test_sum, e)
128         print(("Epoch {:3d}: training accuracy {:3.1f}%, "
129                "validation accuracy {:3.1f}%").format(
130                    e + 1, 100 * train_acc, 100 * test_acc))
131 
132         # Save accuracy to CSV
133         with open(acc_csv_path, 'a') as csv_file:
134             writer = csv.writer(csv_file, delimiter=',', quotechar='"',
135                                 quoting=csv.QUOTE_MINIMAL)
136             csv_row = [e+1, 100 * train_acc, 100 * test_acc]
137             writer.writerow(csv_row)
138 
139         if (e+1) % chkpt_interval == 0 or e == n_epochs - 1:
140             # Create checkpoint and print confusion matrix
```

```
141             chkpt_dir = "chkpt_{}".format(e+1)
142             save_path = os.path.join(data_path, chkpt_dir)
143             saver.save(sess, save_path)
144             print(sess.run(conf_matrix,
145                            {data: valid_input, target: valid_output}))
146
147     # Print final test accuracy and confusion matrix
148     test_input, test_output = _load_data(
149         corpus_path + "/test", labels, comp)
150     test_acc = sess.run(accuracy,
151                         {data: test_input, target: test_output})
152     print("Final test accuracy {:3.1f}%".format(100 * test_acc))
153     print(sess.run(conf_matrix,
154                    {data: test_input, target: test_output}))
155
156     sess.close()
157
158
159 if __name__ == '__main__':
160     parser = argparse.ArgumentParser(
161         description="Train an LSTM to classify HPCP samples.")
162     parser.add_argument(
163         '--corpus_path',
164         type=str,
165         default="corpus/bcrm_hpcp_12bin",
166         help="HPCP sample directory path")
167     parser.add_argument(
168         '--n_hidden',
169         type=int,
170         default=24,
171         help="Number of hidden layers")
172     parser.add_argument(
173         '--n_epochs',
174         type=int,
175         default=1000,
176         help="Number of epochs in training run")
177     parser.add_argument(
178         '--chkpt_interval',
179         type=int,
180         default=50,
181         help="Number of epochs between checkpoints")
```

```
182     parser.add_argument(
183         '--train_samples',
184         type=int,
185         default=2000,
186         help="Number of samples used in each training step")
187     parser.add_argument(
188         '--batch_size',
189         type=int,
190         default=10,
191         help="Number of samples in each minibatch used for SGD")
192     parser.add_argument(
193         '--sample_length',
194         type=int,
195         default="2570",
196         help="Number of HPCP vectors per sample (use 10322 for 60s)")
197     parser.add_argument(
198         '--n_bins',
199         type=int,
200         default=12,
201         help="Number of HPCP bins")
202     parser.add_argument(
203         '--comp',
204         action='store_true',
205         help="Classify by composer instead of by period")
206     args = parser.parse_args()
207     run(args.corpus_path, args.n_hidden, args.n_epochs,
208         args.chkpt_interval, args.train_samples, args.batch_size,
209         args.sample_length, args.n_bins, args.comp)
```