Utrecht University

MASTER'S THESIS

An Approach for Automated Pattern Discovery in Symbolic Music with Long Short-Term Memory Neural Networks

Author:
Erik SCERRI

Supervisors:
Dr. Anja VOLK
Iris Yuping REN

Second Examiner:
Dr. Peter van KRANENBURG

Thesis Number:
ICA-6158811

A thesis submitted in partial fulfillment of the requirements for the degree of M.Sc. Game & Media Technology

in the

Department of Information & Computing Science
Graduate School of Natural Sciences

UNIVERSITEIT UTRECHT

Abstract

Graduate School of Natural Sciences

Department of Information & Computing Science

M.Sc. Game & Media Technology

An Approach for Automated Pattern Discovery in Symbolic Music with Long Short-Term Memory Neural Networks

by Erik SCERRI

A typical musical piece often has noticeable recurring segments of music, often referred to as musical patterns, that provide important insights into the structure of the track. There is no established set of rules that define a musical pattern, and the process of human pattern annotation is long, tedious, and highly subjective. Algorithmic versions of such annotation techniques using a structured approach do exist, but perform markedly worse in tasks such as classification when compared to their human counterparts.

We discuss a number of existing pattern discovery algorithms and propose an experimentation framework that utilises long-term memory recurrent neural networks (LSTMs) to attempt to teach a model about the characteristics of individual pattern classes in order to allow that system to identify the same pattern classes in an unstructured environment. The framework consists of three stages, whereby we progressively increase the complexity of musical training data to track and analyse the performance of the proposed system. We also show a preliminary process of parameter tuning that provides a more optimal parameter combination for ensuing training and testing.

Our results show that it is possible for an LSTM-based neural network to establish a definition for pattern classes given enough training time and training data. We have also learned that it is possible to train an LSTM neural network on patterns combined synthetically in sequence that are taken from a realistic dataset and have the learned model be able to identify those patterns in their original scenarios. However, we conclude that the proposed system is still a very basic approach that cannot yet compete alongside current state-of-the-art pattern discovery algorithms, and that the results we provide can serve as a promising baseline for future work on this topic.

Contents

Abstract **i**

1 Introduction **1**
1.1 Overview of MIR . . . 1
1.2 Repeated Musical Patterns . . . 3
1.3 Discovering Patterns in Music . . . 4
1.4 Machine Learning . . . 6
1.5 Research Goal . . . 7
1.6 Thesis Structure . . . 8

2 Background Information **9**
2.1 Music Terminology . . . 9
2.1.1 Notes & Notation . . . 9
2.1.2 Intervals & Scales . . . 10
2.1.3 Note Duration . . . 10
2.1.4 Musical Texture . . . 11
2.2 Musical Representation . . . 12
2.3 Neural Networks . . . 14
2.3.1 Network Learning & Optimisers . . . 15
Stochastic Gradient Descent . . . 16
Adaptive Moment Estimation . . . 17
2.3.2 Long-Term Dependencies & LSTMs . . . 18

3 Pattern Discovery in MIR **22**
3.1 Search Domain & Matching Style . . . 22
3.2 Musical Texture & Representation . . . 23
3.3 Geometric vs. String-based Approaches . . . 23
3.4 Pattern Discovery Algorithms . . . 25
3.4.1 MGDP Algorithm . . . 25
3.4.2 MotivesExtractor . . . 27
3.4.3 SymCHM . . . 28

4 Implementation **31**
4.1 Experiment Framework . . . 31
4.1.1 Training . . . 32
4.1.2 Testing . . . 32
4.1.3 Parameter Tuning . . . 34
4.2 Experiment Data . . . 35
4.2.1 Simple Data (Stage 0) . . . 35
4.2.2 Generated Data (Stage 1) . . . 36
4.2.3 MTC-ANN . . . 37
4.2.4 Data Format . . . 38
4.2.5 Internal Representation . . . 38
4.3 Overview . . . 40

5 Results **43**
5.1 Baseline Metrics . . . 44
5.2 Pattern Discovery on Simple Data . . . 44
5.2.1 Overview of Results . . . 45
5.2.2 Parameter Tuning: Adam Optimiser . . . 46
Hidden Size . . . 46
Number of Epochs . . . 48
Number of Cross-Validation Folds . . . 51
5.2.3 Parameter Tuning: SGD Optimiser . . . 54
Hidden Size . . . 54
Number of Epochs . . . 56
Number of Cross-Validation Folds . . . 58
5.2.4 SGD vs Adam . . . 60
5.3 Pattern Discovery on Generated Data . . . 61
5.3.1 Overview of Results . . . 62
5.3.2 Output Samples . . . 63
5.3.3 Batch Length: 8 . . . 67
5.3.4 Batch Length: 16 . . . 68
5.3.5 Batch Length: 24 . . . 69
5.3.6 Summary . . . 71
5.4 Pattern Discovery on Synthetic MTC-ANN . . . 71
5.4.1 Overview of Results . . . 72
5.4.2 Output Samples . . . 73
5.4.3 Batch Length: 8 . . . 76
5.4.4 Batch Length: 16 . . . 78
5.4.5 Batch Length: 24 . . . 78
5.4.6 Summary . . . 80
5.5 Pattern Discovery on MTC-ANN . . . 80
5.5.1 Overview of Results . . . 81
5.5.2 Batch Length: 8 . . . 82
5.5.3 Batch Length: 16 . . . 83
5.5.4 Batch Length: 24 . . . 84

6 Conclusion **85**
6.1 Research Summary . . . 85
6.1.1 Influence of Synthetic Ground Truths and Musical Complexity . . . 85
6.1.2 Using Training on Artificial Data to Classify Real Data . . . 87
6.1.3 Comparison with Pattern Discovery Algorithms . . . 87
6.2 Future Work . . . 88
6.3 Final Remarks . . . 90

Acknowledgements **91**

A Overview of Tests **92**

Bibliography **94**

Chapter 1
Introduction

Music is, and has always been, a vital part of human culture. It covers a vast array of styles and genres, ranging from simple solo music to complex orchestral pieces. With technology having advanced as rapidly as it has, it is no surprise that music distribution in the digital form has followed suit, and this in turn has sparked an interest in technologies that can be applied to musical content in order to organise, structure, and manage this kind of data. The study of such technologies is placed under the research field of *Music Information Retrieval*.

In this thesis, we will study and produce automated processes for the extraction of repeated sections of musical information from a piece by utilising the *repetitive nature* of music. Repetition is a crucial part of many art forms, and is used to good effect in previous studies in the field (Collins, 2017; Ren et al., 2018b; Conklin, 2009; Conklin, 2010; Karydis et al., 2006). We propose a method to use machine learning techniques in order to extract these repeated sections from a piece by learning from human efforts into the same task. More specifically, we aim to study the difficulties associated with increasing complexity in music, and what effect varying levels of complexity have on the learning process and on the quality of the discovered patterns. In the study, we will use two main datasets, one artificially generated and one of real musical data, in an effort to construct as robust a system as possible. The datasets will be artificially augmented to further increase our control on the amount and the complexity of data.

In section 1.1, an overview is given of Music Information Retrieval, and some of the challenges and applications faced in the field. In sections 1.2 and 1.3, more detail is given about musical patterns and repeated sections, as well as an overview of some of the challenges faced during pattern detection. Then, in section 1.4, the field of Machine Learning in the context of music and computer science is briefly explored. These four sections are then summarised into a definition of the research problems, the approach proposed to tackle this problem, and the aims and objectives for the project (section 1.5). Finally, an outline of further chapters in this thesis is given in section 1.6.

1.1 Overview of MIR

In modern computing, finding and accessing relevant information about any topic has become a relatively simple procedure. Search engines such as *Google* or *Bing* make searching for text-based data trivial, returning hundreds of results and usually pointing directly to the most relevant requested information in the ever-expanding cloud of data that makes up the internet. The continuous improvement of online search systems has driven them to quality levels at which relevant output is almost always guaranteed (Manning et al., 2008). In computer science, the process of tracing and recovering specific information in stored data is called *Information Retrieval*. Colloquially it is referred to as "searching", which is more ambiguous but has come to

be near synonymous with the process of retrieving data. *Information Retrieval* is defined more concretely as: *"finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)"* (Manning et al., 2008). While this definition leans more towards text-based searching, Information Retrieval is not limited to searching by text. Efforts into searching through other forms of data such as images or videos have also made significant progress. More relevant to this thesis is Information Retrieval through the use of musical data, a field aptly known as *Music Information Retrieval*.

Music Information Retrieval (MIR) is an interdisciplinary sector of science that deals with the study of extracting, utilising, and manipulating information from musical data (Downie et al., 2009). It's a particularly active research area, with yearly conferences organised by the *International Society for Music Information Retrieval (ISMIR)*[1] during which open research problems are available for participation. An open exchange platform for the evaluation of MIR challenges is also available, the *Music Information Retrieval Evaluation eXchange (MIREX)*[2]. In the context of computer science, MIR deals with various research projects; from automatic score transcription, chord detection/estimation, or music identification (e.g. Shazam), to genre classification, music generation, and many other tasks to which the expanding fields of scientific computing can be applied (Manning et al., 2008). Previous to the study of MIR, queries about musical information did not use actual *musical knowledge* (for example, using lyrics to find a song instead of a musical snippet). This meant that any efforts into the field were severely limited in nature, both by the lack of relevant data where music could be easily represented as simple text, and by the difficulty of doing so.

One of the major challenges of MIR is the fact that music is, like most non-verbal art forms, quite complex (Downie et al., 2009; Schoenberg, 1999; Cook, 1994). Looking at a single instrument playing a single note at a time, there are already numerous complicated relationships between the notes in a specific melodic sequence, such as: what key the melody is in, which notes belong to that key and which are merely chromatic ornaments[3], when does the key change, when does the melody end and another begin, and so on. This complexity is only increased when additional polyphony is introduced. Multiple instruments can swap melodies back and forth, harmonise with each other, introduce completely new themes in the background, or link to each other in any number of subtle ways.

As a result of this complexity, it is often a crucial part of any MIR problem to begin by extracting the correct[4] information from the correct section of music. Schedl et al. (2014) define MIR as a broad effort to improve the *"extraction and inference of meaningful features from music, indexing of music using these features, and the development of different search and retrieval schemes"*, while Meadows and Cronin (2003) suggest that MIR is more concerned with the context of the relationship and interaction between smaller parts and characteristics of the musical piece rather than as a whole, but maintains that finding the relevant smaller parts is still crucial. Doing this through a content-based approach, which is to say through musical features (rhythm, melody, harmony, and so on) rather than textual meta-data, requires a further understanding of the rules that govern musical composition and analysis (Schoenberg, 1999; Cook, 1994).

[1] http://www.ismir.net/

[2] http://www.music-ir.org/mirex/

[3] Any added notes that do not belong to the current chord, but ultimately lead to or from a note that does (chord resolution).

[4] In most studies, *correctness* is synonomous with relevance to the study's expected results.

1.2 Repeated Musical Patterns

A common approach to structuring music data is to exploit its inherent repetitiveness (Collins, 2017; Ren et al., 2018b; Conklin, 2009; Conklin, 2010; Karydis et al., 2006). The repetitive nature of music is a fundamental concept that has seen a significant amount of research (Lerdahl and Jackendoff, 1985; Downie et al., 2009; Janssen et al., 2013), and detecting repeated sections of music is an important part of many MIR tasks. Repeated patterns provide structure and coherence to the piece and are present in most (if not all) forms of music (Mirka, 2010), leading to the idea that finding ways to accurately separate music into the most relevant pattern components would give more accurate insights into the common characteristics of music (for example; for melody classification by genre, artist, etc. (Conklin, 2009; Karydis et al., 2006)), as well as allow for easier manipulation and understanding of the piece and musical data in general (for example; for compression or prediction (Boot et al., 2016)).

Repetition is a common aspect of several art forms. In poetry and writing it is used to secure emphasis, and in visual art (paintings, photography, film) it is used to identify objects and similarities between different pieces (for example, in film, sometimes a particular colour scheme is associated with a character and repeated whenever the character is on screen to induce a connection between that appearance and previous appearances of the character). Beyond this, repetition is also a vital part of everyday life. When reading text, we are simply comparing previously learned patterns (phrases and sentences) in order to process the patterns we are currently reading (we can extrapolate the meaning of a brand new sentence by using our experience with previous, potentially similar, sentences). Personal experience, which we consider vital knowledge in dealing with scenarios, is only established through repeated involvement in that scenario, often through repeating similar tasks using previously experienced repetitions. In music, repetition can be anything from short repeated motifs[5] (such as a string *ostinato*[6]) to long overarching phrases that repeat multiple times (such as a chorus in pop music). A good definition for a *pattern* is given in MIREX tasks for the Discovery of Repeated Themes & Sections, stating that *"a pattern is defined as a set of ontime-pitch pairs that occurs at least twice (i.e., is repeated at least once) in a piece of music. The second, third, etc. occurrences of the pattern will likely be shifted in time and perhaps also transposed, relative to the first occurrence."*

The importance of repetition is considered a key part of both the listening experience and the research aspect of music. Through repetition, the musical piece becomes coherent and structured, which by extension makes it more memorable and less overwhelming for a listener. Much like language is easier to understand when you have experienced its vocabulary previously, music is easier to process when linking patterns being currently heard with patterns heard previously. Hearing repeated motifs or melodic patterns associated with a specific memory can trigger emotional responses in a listener, and memorable patterns in music (for example, a certain melody, bass line, or drum beat) are often the main way to identify the full piece without listening to it entirely. In fact, patterns are considered one of the basic rules of composition (Dannenberg, 1993), and a good metric for the internal structure and coherence of the musical piece in question (Mirka, 2010). Given how crucial such repetition is to the

[5] According to The New Grove Dictionary of Music and Musicians (Sadie and Tyrrell, 2002), a *"motif may be of any size, and is most commonly regarded as the shortest subdivision of a theme or phrase that still maintains its identity as an idea."*

[6] An *ostinato* is a short phrase or melody that persistently repeats in the same musical voice, typically without changing pitch.

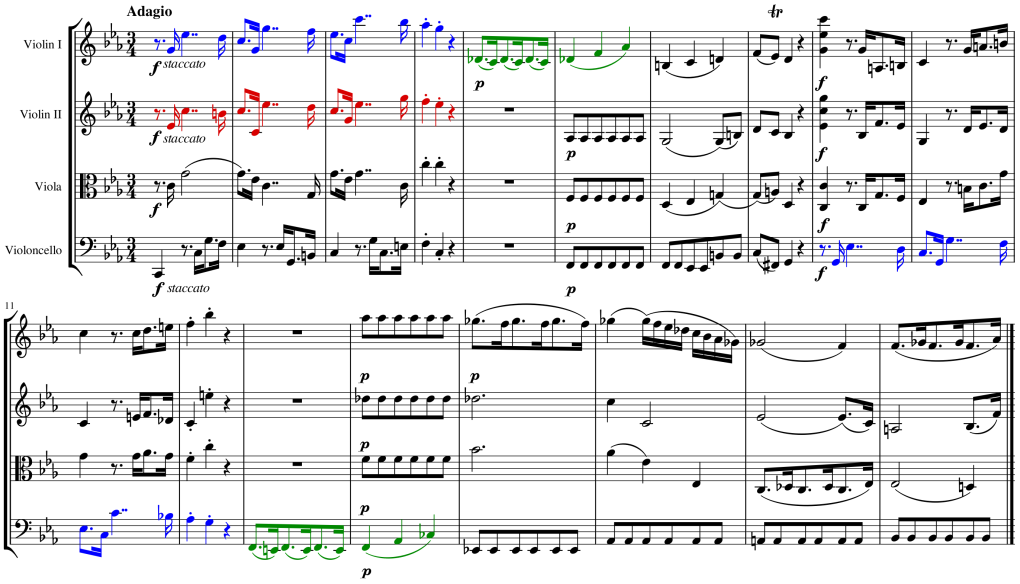


FIGURE 1.1: An example of some patterns from Mozart's Adagio and Fugue in C Minor. The pattern in blue is identically repeated later in a different clef. The pattern in red is identical to the pattern in blue, but only rhythmically. The pattern in green could be considered an extension to the patterns in blue, but is not present in the pattern in red.

listener and to the means of understanding and extracting knowledge from music (for entertainment or intellectual purposes), it is sensible to value the importance of musical repetition in research highly.

Repeated patterns have been used in various different studies. Meredith (2015) used musical patterns for compression, using *normalised compression distance* (Li et al., 2004) as a metric for calculating the degree of compression from similarities between musical patterns. Conklin (2009) used them as musical feature sets for folk song classification, and similarly Li et al. (2004) used an automatic pattern discovery algorithm to generate output used for finding relevant patterns to use as features for classification. Chou et al. (1996) used repeated patterns to try and find common themes in large musical databases, and Karydis et al. (2007) used patterns as the basis for content-based retrieval of music from similarly large databases.

1.3 Discovering Patterns in Music

The discovery and annotation of patterns in music is a well-researched topic. A well-organised summary of many pattern discovery techniques is given by Janssen et al. (2013), who discuss the state of knowledge of musical pattern discovery, and the relevant challenges faced by the research field. In music theory and musicology, the traditional method for discovering patterns in music is to have experts manually annotate repeated motifs in the piece and then use these patterns for the research project (this sometimes also includes manually transcribing annotated patterns into a digital format, which is a long process in itself). Naturally, this is both tedious and also highly subjective. Different annotators, despite having the same level of musical knowledge, will tend to hear a range of different repetitions in a musical piece, leading to difficulty in establishing a common ground truth against which to evaluate research.

An alternative to manual annotation is to discover repeated patterns automatically through the use of some discovery algorithm (Janssen et al., 2013). Given musical input, either as symbolic or recorded data, the aim of such algorithms is to output a set of discovered patterns from the piece. While significantly faster than the manual approach, this comes with several problems. Firstly, musical patterns can be determined through the use of any number of features, such as rhythm, harmony, melody, chord progression, and so on (Schedl et al., 2014). There is no clear indication which musical feature is most relevant for the discovery of patterns, even in manual annotation, and the importance of a musical feature can vary from one piece to another. This ambiguity is two-fold, as it can stem both from the piece itself (for example, a dance piece might place more emphasis on rhythmic qualities, whereas an orchestral piece might place more emphasis on melodic and harmonic features), as well as from the perspective of the listener (different musical backgrounds might prioritise different musical features). Using multiple features in unison continues to complicate the development of the algorithm itself. Furthermore, algorithms capable of extracting patterns will often return a massive amount of these patterns in comparison to human-annotated pieces. Janssen et al. (2013) state *"A frequently described problem in musical pattern discovery is the great amount of algorithmically discovered pattern as compared to the patterns that would be considered relevant by a human analyst."* The main reason for this is that it is difficult for a computer algorithm to understand and correctly evaluate the relevance of a pattern without additional computation and filtering.

In order to reduce the output size of the pattern discovery algorithm to a feasible quantity, discovered patterns should be rejected or accepted based on their importance (Janssen et al., 2013). In the MIREX task for pattern discovery, importance of a pattern is defined as how noticeable it is to a listener. In this sense, an algorithm that can discover a set of patterns will find it difficult to determine which subset is "important" because importance is often a matter of perception (different listeners might consider different parts more or less noticeable). Some studies (Swierstra et al., 2018) suggest that listeners agree more on highly relevant patterns as opposed to those deemed less relevant, but whether this extends over a significantly larger sample size of listeners is unclear. The quality of the ground truth against which the algorithm measures importance is almost entirely dependent on the manual annotation used to create the ground truth. If this annotation is not consistent, the supposed ground truth might not be an accurate barometer for comparison. Such a ground truth relies on agreement between annotators, who might have different musical backgrounds and perceptions of music[7]. Finding a ground truth, if one does exist that can be applied to every possible style of pattern (simple, complex, and everything in between) while also factoring in all other forms of ambiguity in pattern discovery, is difficult (Ren et al., 2018a; Reidma and Akker, 2008).

[7]Someone trained in music might be able to find more complex patterns and give more importance to those, but simple patterns are no less important.

1.4 Machine Learning

The core concept behind Machine Learning is to learn how to perform important tasks by generalising from provided examples. This is often used as an automatic alternative to manually constructing algorithmic structures, which can be a monumental task when dealing with overly complex problems or when working on massive amounts of data. A relatively recent concept, machine learning has spread throughout most fields in computer science over the past decade, used in everything from web searching and indexing, fraud detection systems, and data mining, to stock trading, complex protein folding, and a multitude of computer vision tasks.

In MIR, machine learning has seen some use in the context of genre classification (Kim et al., 2016) and music generation (such as projects carried out by Google Magenta). A notable example is Google's *Now Playing* sound search, integrated into the Google voice assistant. At the core of this system is the *Neural Network Fingerprinter*, which uses *convolutional neural networks* to generate unique fingerprints from a few seconds of audio, which can then be used to find and retrieve the relevant musical piece which is closest to that fingerprint from a database (Tsaptsinos, 2017). The network is trained to minimise the distance to examples of that same audio segment, while simultaneously increasing distance to all other audio segments.

A popular branch of machine learning is the use of Artificial Neural Networks (ANN), which provide a framework for layers of different machine learning algorithms to collectively work on information that might be too complex for a single algorithm. The approach is based loosely on biological animal brains and the concept of learned experience. The idea, in principal, is that human brains are able to learn how to perform specific tasks by considering previous examples and extrapolating a generalised model from those examples. Neural networks aim to replicate this concept in a controlled training environment. Neural networks have seen widespread use in computer vision tasks, notably used in image recognition and retrieval, natural language processing, object detection, and so on. They are particularly adept at detecting repeated features across data pieces and learning a general model that can be applied to detecting future features from doing so (Bishop, 1995; Basu et al., 2010). Multiple types of neural networks exist and have seen use, such as: Recurrent Neural Networks that save the output of a layer and feed it back to the input in order to better predict the outcome of the layer (Karpathy, 2015),and Convolutional Neural Networks that convolve layers in batch-wise filters to better detect large-scale features (Krizhevsky et al., 2017; Simonyan and Zisserman, 2014).

Recurrent Neural Networks (RNNs) are a particular branch of ANNs that seem well suited for the issue at hand. The main principal of RNNs is that short-term memory is an important part of human thinking (for example, when speaking you do not reset your brain at every word, but remember what you just said to continue off that). In other words, thoughts have persistence, which is lacking in traditional neural networks. RNNs address this issue by building networks with internal loops that allow information to persist. Long Short-Term Memory networks (LSTMs) are a special subset of RNNs that take this further by being capable of learning long-term dependencies, which is to say that information used a while earlier (and not immediately previously) can be used to influence current data as well. RNNs, and more specifically LSTMs, have been applied succesfully to a variety of problems: speech recognition, language modeling, translation, and computer vision (Karpathy, 2015).

1.5 Research Goal

To summarise, there are therefore a few apparent problems. Firstly, the relevance (or importance) of musical patterns is subjective, which makes it difficult to evaluate the quality of automatically-discovered musical patterns. Subjectivity in this context means that it is not always entirely clear or defined when a sequence of notes makes up a pattern, and different annotators can determine different patterns within the same piece of musical data. As a result, we cannot easily say which patterns from which annotators are more important than others, as different patterns might perform better or worse by different metrics or tasks. Studying and understanding this subjectivity is difficult without significantly larger sample sets of human-annotated data, or else by introducing less complex artificially-generated data with more consistent ground truths and attempting to establish a correlation with increasing complexity. Secondly, systematic evaluation of pattern discovery algorithms is difficult, given the lack of a single ground truth between algorithms. In the MIREX task for Discovery of Repeated Themes & Sections, state-of-the-art algorithms perform reasonably well by the metrics of the task, but cannot yet replicate human-annotated patterns. Furthermore, this performance is inconsistent across a range of musical pieces, leading to the question of whether a pattern discovery algorithm should evaluate pattern importance solely by correspondence to a ground truth instead of *"an objectively evaluable task that the pattern helps us to perform more successfully."* (Meredith, 2018).

As mentioned in the introduction, the aim of this research project is to take a data-driven approach to creating a pattern discovery algorithm by using machine learning trained on synthetically-created and manually-annotated data. To accomplish this, we attempt to reduce the complexity of the problem and slowly re-introduce aspects of that complexity with every stage. As such, the experiment will progress in three stages:

1. Firstly, the algorithm will be trained on pieces generated synthetically from basic musical structures. This is done by taking pre-defined simplistic patterns (such as scales, interval repetitions, and so on) combined with random note sequences, and concatenating everything in random orders to create a "musical piece" with exact segmentation points (where a pattern starts and ends). While such patterns are not representative of actual musical data, the advantage here is that there is less ambiguity in the pattern boundaries, which ensures that the ground truth is at the very least easier to determine and train with.

2. In the second stage, a similar artificial joining approach will be employed, but this time by randomly concatenating patterns taken from real musical pieces. This ensures that some pattern boundaries are still clear, but introduces additional complexity that is not present in simplistic patterns like scales or interval sequences. The concatenation process introduces the problem of new patterns potentially being created between adjoined segments, but the defined boundaries will still be present and usable for training. This means that we can at least know when the model finds or does not find these boundaries and evaluate that, while also looking at new interconnected patterns discovered and discuss the features they exhibit. Additional experimentation in this stage can be explored by training models on a range of polyphony (training on a single instrument such as a piano vs. more complex polyphony such as a string quartet) and comparing the performance of the system on increasing complexity.

3. In the third and final stage, training will be done on non-artificial musical pieces that have been previously manually-annotated by musicologists or by average listeners. The idea here is to create a model that can learn from complex sources and attempt to establish a common pattern discovery technique. Ultimately the goal for this stage is to observe the behaviour of the learning process on unprocessed musical data at full complexity.

Throughout these three stages, we try to control and analyse pattern complexity in musical data by training models on music pieces with both artificial and natural pattern boundaries and comparing the performance of both. We tackle the problems mentioned above regarding the ambiguity of pattern ground truths by defining our own ground truth for the first two stages, allowing us to focus instead on the process of separating a piece into patterns and studying the effectiveness of neural networks at generalising definitions for pattern classes and boundaries. We then use what is learned to train on realistic test data and compare performance. Analysing differences in these results should give some insight into the pattern discovery process and the benefits (or lack thereof) of utilising data-driven pattern discovery techniques.

We can summarise the research problem into three research questions:

Research Question 1

Is an artificially-generated fixed ground truth sufficient for training pattern discovery (stages 1 and 2), and does performance drop when this fixed ground truth is not available (stage 3)?

Research Question 2

Can a model trained on artificial data (stage 2) correctly identify patterns in realistic music to a level comparable with human-annotated counterparts?

Research Question 3

Can the proposed system perform comparably with other pattern discovery algorithms that utilise differing approaches?

1.6 Thesis Structure

The remainder of this thesis is structured as follows: in Chapter 2 we give an overview of the basic terminology and theory that is necessary to understand the concepts we use in the remainder of the study. In Chapter 3, we provide a thorough discussion of pattern discovery in MIR, as well as an overview of some pattern discovery algorithms. In Chapter 4, we go over the implementation details for the project, as well as discuss the types of data we use at each stage. An overview of the results from training and testing we perform is provided in Chapter 5, along with discussion of those results in light of our research questions and goals. Lastly, we will present our conclusion in Chapter 6.

Chapter 2
Background Information

This chapter will explore a few basic concepts that are important to this thesis in an effort to make the reader at least somewhat familiar with the basic concepts behind the research used throughout the study. Notably, it will cover the basics of musical terminology, some concepts regarding the representation of music and the different approaches to doing so, and an overview of neural networks (particularly RNNs). The information covered here will provide a basis for the remainder of the thesis, and is particularly important in the discussion of previous studies into the field that is covered in the following chapter.

2.1 Music Terminology

2.1.1 Notes & Notation

The musical *note* is the most fundamental building block of a musical piece. It is defined as a designation of the *duration* and *pitch* of a sound, and can also refer to the symbolic representation of that sound in musical notation (♪, ♩). The *duration* of a note is simply how long it lasts, in either time (seconds) or notation length (bars). The *pitch* corresponds to the frequency of the produced *tone*.

Two tones with their fundamental frequencies at a ratio equal to any integer power of two (that is, half, double, quadruple, and so on) are said to belong under a *pitch class*, and are defined to be a whole number of *octaves* apart. An *octave* indicates the span of frequency between a tone and another tone that has double its frequency. Since notes in the same pitch class are perceived as very similar, it is important to be able to distinguish between each note's tone pitch and the octave it is in. There are two formal systems in use to define tone and octave: *Helmholtz Pitch Notation* (Helmholtz, 1913) and *Scientific Pitch Notation* (Young, 1939). In this thesis, we will use the latter. This notation specifies pitch using a musical note name (the letters $A - G$) corresponding to the pitch class, and a number identifying the octave of that pitch. For example, the standard tuning pitch for most Western music is at $440Hz$, named A_4, an octave higher than A_3 which corresponds to a frequency of $220Hz$.

In Western music, the frequency range defined by an octave is split into 12 equal divisions, called *semitones*. On a piano keyboard, these 12 semitones are represented by the 12 black and white keys between two notes an octave apart. The white keys correspond to the 7 pitch classes defined by the letters $A - G$. The black keys are pitches a semitone higher or lower than their neighbouring white key. We notate such pitches using the two *accidentals*: ♯ and ♭. The *sharp* sign ♯ raises the pitch by a semitone, while the *flat* sign ♭ lowers it by a semitone. For example, $G♯_6$ is the note with a tone pitch one semitone higher than G_6, identical in pitch to $A♭_7$ which is one semitone lower than A_7.

2.1.2 Intervals & Scales

In musical theory, an *interval* is defined as the offset in pitch between two notes. A semitone, (defined in the previous section) is considered the smallest interval in Western music, and all successive intervals are counted as a number of semitones. For example, there is an interval of 5 semitones between C_4 and F_4. In Western music, we give each interval between 0 and 12 semitones a name, for ease of use. In the previous example, the interval would be a *Perfect Fourth*, the name corresponding to a difference of 5 semitones.

Following from this, notes arranged in an ordered sequence of pitches following a given interval pattern are called a *scale*. Often, melody and harmony in a given section of music are built around the notes of a single scale. A scale typically spans a single octave (simply repeating the pattern into the next octave if necessary), and is classified by the starting pitch and the type of interval pattern the scale follows. For example, a *Chromatic scale* has an interval pattern of 12 successive semitone intervals (see Figure 2.1), whereas a *Major scale* is classified by an interval pattern of T–T–S–T–T–T–S, where T stands for a whole tone (2 semitones), and S stands for a semitone (see Figure 2.2).

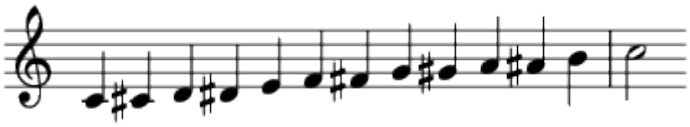


FIGURE 2.1: The chromatic scale starting at Middle C, ascending.

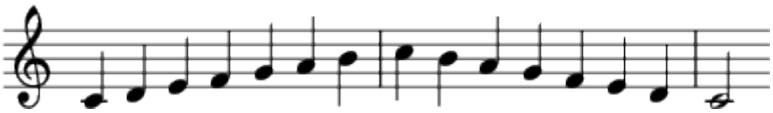


FIGURE 2.2: The C Major scale starting at Middle C, ascending and descending.

2.1.3 Note Duration

The second important property of a musical note is *duration*. The duration of a note is defined in terms of *beats*. A *beat* is the basic unit of time in a musical piece, combining strong and weak beats to define a rhythm. The value of each beat is defined in Western music by the *time signature* which can be found at the beginning of the staff. *Time Signatures* are a notation convention which specify how many beats are contained in each bar measure and the note length value assigned to each of those beats. For example, a time signature of $\frac{3}{4}$ states that there are 3 beats in a bar, and that each beat is valued at a quarter note (also known as a *crotchet*). Notes can then have a duration of any number of beats, including fractions.

A note's duration can also defined by the *onset* and *offset* of that note. The *onset*, measured in beats from the beginning of a piece (onset of 0), defines the starting point of a note, while the *offset* defines the endpoint. The difference between offset and onset therefore gives the duration in beats of a note. For example, an eighth note (or *quaver*) at the start of a $\frac{4}{4}$ piece would have an onset of 0 and an offset of 0.5, giving it a duration of 0.5 beats.

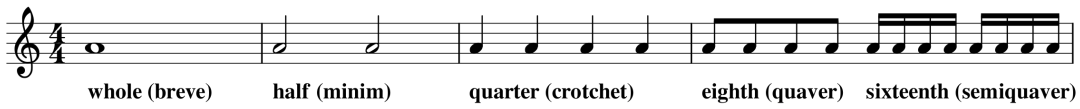


FIGURE 2.3: Different note durations showing their score notation, and two variants of their names. The time signature is $\frac{4}{4}$ with a fixed four beats per measure. Four quarter notes are needed to fill each measure. A single whole note is equivalent to the full measure, as are two half notes, 8 eighth notes, and so on.

2.1.4 Musical Texture

The texture of a musical piece is described as how melody, harmony, and tempo are combined into a single composition. Typical characteristics of musical texture include density or thickness (number of instruments or voices playing simultaneously), range or width (distances between lowest and highest pitches), and sometimes characteristics related to rhythm. There are several common types of musical texture often used in the fields of musical analysis: *monophonic, biphonic, polyphonic, and homophonic*. A brief overview of the more relevant musical textures is given below, based on the descriptions provided by Benward and Saker (2015).

Monophonic texture is the simplest level of musical texture, consisting of a single melodic line with no accompaniment. The defining feature is that only one note is played at any single time in a monophonic piece.

Biphonic texture consists of two distinct lines playing simultaneously, where one is a sustained drone of constant pitch, and the other is a more elaborate melodic sequence. Usage of *Pedal tones*[1] is one example of biphonic texture.

Polyphonic texture is a piece with two or more melodic sequences playing simultaneously, with a large degree of independence between each other. This texture is particularly characteristic of Renaissance and Baroque music, where numerous instruments are playing independently to create a single harmony. True polyphony defines each instrument as equally important and independent, while playing simultaneously to other instruments.

Homophonic texture is the most common style in modern Western music, consisting of a single prominent melody, along with polyphonic accompaniment that forms a background to the main voice. In some ways, it can be considered an extension of polyphonic music (all instruments are playing independently, but do not have equal importance). This style of music became popular during the Classical period and has remained widely used till the modern day. In fact, most popular music can be considered homophonic, since the singing voice is given predominance over the background music. A notable exception is some Jazz music, where *"the simultaneous improvisations of some jazz musicians creates a true polyphony"* (Benward and Saker, 2015).

[1] A sustained note, typically in the lower clefs, during which dissonant harmonies are sounded in the other parts.

2.2 Musical Representation

As with many art forms, music is defined by the presence of relationships between its constituent parts. In painting and photography for example, this would be defined as the relationship between colours, lighting, the subjects of the image, and so on. Music is portrayed instead by a series of mathematical relationships (Jones, 2002), such as rhythm (a mathematical relationship in terms of note duration) and harmony (a mathematical relationship in terms of note pitch). Non-mathematical elements also play a part, with elements such as emotion and style providing an extra complexity to the mathematical elements that typically represent musical information. Since the latter elements are specific to the performance of a musical piece, and evolve/change with any new composition, it is difficult to define a single exhaustive representation of music. Musicians will tend to deal with a varying level of abstraction in musical performance, which makes it open to interpretation and therefore difficult to represent properly.

To simplify matters somewhat, musical representation tends to focus on the elements that can be a exactly represented. At the most basic level, this is simply some kind of representation of a sequence of notes, each with pitch and duration, each assigned to an instrument, and arranged over time over the duration of the piece. More complex instructions, such as musical expressions that are present in most musical scores, can also be added, but are difficult to quantify. For example, in Figure 1.1, *Adagio*, meaning an instruction to perform the piece slowly and ponderously, is a musical expression provided for interpretation by the performer, but does not provide a concrete instruction into how *much* the performance should slow down.

In MIR, we typically focus on the digital representation of music. This stems from the common need to access and analyse musical data in bulk. Since computers cannot store ambiguity or interpretation, digital representations require a formal approach, demanding that every detail of the representation is precisely specified. Digital representation of music can be split into two types: *symbolic representation* and *audio representation*.

Audio representations are the most commonly distributed forms of digital music. Whether compressed (.mp3, .aac, and so on), or uncompressed/lossless (.wav, .flac, and so on), the main representation method remains the same: representing a waveform as a sequence of amplitudes over time (Pohlmann, 2000). The advantage of this format is that it is available for most music available digitally, meaning that the available pool of data is massive. However, an audio representation makes it difficult to separate voices within polyphony [2], and can have various distortions, noise, or compression artifacts that affect performance. That being said, the listener's perception of music relies almost entirely on hearing the piece (ideally as realistically as possible), and certain intricacies that are only detectable by listening to the piece (such as particular combined harmonies or sounds over multiple instruments) cannot be as easily detected by reading through symbolic data. Conversely however, certain subtle contributions by single instruments can be missed in an audio representation, since not every note will be equally important in an audio rendering of a piece.

[2] Polyphony is music that consists of two or more simultaneous lines of independent melody, as opposed to a musical texture with just one voice, monophony.

FIGURE 2.4: An image of the audio representation of Mozart's Eine Kleine Nachtmusik, for piano.

Symbolic representations in comparison present more in-depth theoretical information, because they forego the storage of an actual audio signal in favour of representing the notes and musical structures that should be played by each instrument to create an audio signal. This includes any and all notes that form the piece, no matter how small of a contribution they may make in the audio representation. Traditional sheet music is an example of symbolic music representation in a non-digital space. In a digital space the idea is to carry over as much information from an equivalent sheet music, creating an explicit encoding of notes, annotations (such as expressions, tempo notation, etc.), and any other relevant information or musical events. The main advantage of symbolic representation is that every instrument and its contribution is clearly readable and separated from other voices. As a result, identifying the different voices in polyphonic music is far easier in a symbolic representation, but certain subtle interactions between those same voices (for example contrapuntal derivations where the counterpoint melody is visually very different from the original) are conversely more difficult to notice unless the symbolic representation is simultaneously played back as an audio signal. Furthermore, symbolic formats can be less straightforward to interpret for an average musically illiterate listener as they often require a level of musical knowledge to be properly read.

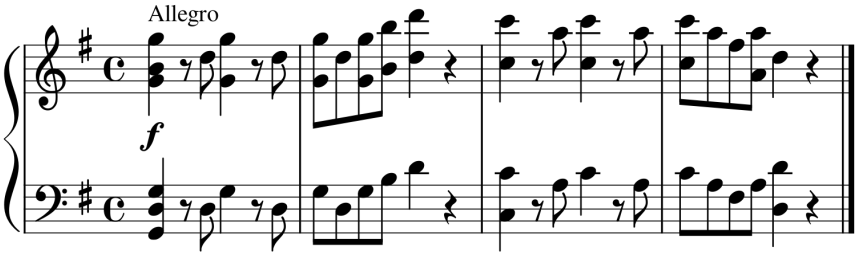


FIGURE 2.5: An image of the same part of Mozart's Eine Kleine Nachtmusik, in symbolic representation.

Considering the two techniques above, the main challenge seems to be in the difference between notation and perception of music. Audio representation puts more emphasis on the latter, providing the listener with a performance of the musical piece as pure sound data, but making it difficult for the distinction of individual elements to be processed without sufficient training or experience. Symbolic representation does the opposite, by providing exact specific data on every small part of the piece, but introducing a layer of seperation that makes it harder for a listener to interpret the data as sound. Since this thesis focuses on discrete notation and the seperation of note sequences into patterns, we will be using symbolic representations as input.

The most commonly used example of symbolic representation is the Musical Instrument Digital Interface (MIDI) format. MIDI acts as a technical standard that can be easily ported between digital interfaces, electronical instruments, computers, and other related audio devices. It consists of sequences of *events* that contain information about each note being played (pitch, velocity, expression, duration, and so on), which makes it relatively simple to parse and process digitally. The pitch notation in MIDI uses a *note number*, starting with 0 at a frequency of $8.18Hz$. Corresponding to the Scientific Pitch Notation, MIDI note number 21 is A_0, increasing by 1 for every rising semitone. Another example of digital representation is the `**kern` data format, used in this thesis in data gathered from the classical dataset at `http://kern.humdrum.org`, which represents information as data tokens of three types: notes, rests, and barlines. For a more comprehensive overview of digital symbolic representation of music, consult Selfridge-Field (1997).

2.3 Neural Networks

The approach we propose in this thesis is built on the concepts of neural networks and supervised learning through said networks. Neural networks are a complex subject and diverge into numerous sub-types with different approaches, strengths, and weakness. That being said, there are several concepts which remain consistent throughout. In this section, we aim to give an overview and explanation of these terms and concepts in order to hopefully avoid confusion later on, before moving on to the specific sub-type we propose using in this study and explaining it in more detail.

Neural networks are a branch of machine learning that attempts to mimic the learning process in the human brain. The human brain is, put simply, an extremely large interconnected network of neurons, which are used for information processing and for modelling the world around us based on previous experience. A neuron is a basic input/output processor, collecting all inputs from other neurons, summing them, and firing an output if the resulting value is greater than a given threshold. The fired symbol is then sent off to other connected neurons for the same process to repeat, until a final output is achieved. Artificial neural networks are modelled after the same principle. At the most basic level, a neural network is made up of a collection of *neurons* and *connections*, developing through the use of a *learning rule*.

Neurons are the most basic component of the network. Much like in the human brain, they receive individually-weighted inputs from other neurons, summate those inputs, and then determine output based on an *activation function*. The *activation function* maps output to a desired range, given the input/s to the neuron. There are various types of activation functions, simplest of which is the *step function* which outputs a 1 if the input is higher than a given threshold, or a 0 if not. Other kinds include tanh activation, Sigmoid activation, Rectified Linear Unit (ReLU) activation, and so on (Nwankpa et al., 2018).

Individual neurons are adjoined to each other using *connections*, whose only purpose is to transfer the output of a neuron to another neuron in order to serve as input to the new neuron. Each connection is assigned a weight and bias value, and when summation occurs inside the neuron inputs are scaled based on the weight and bias of the input connection. The shifting of these internal parameters on every iteration is the main point of learning using neural networks. Adjusting of parameters is done via

backpropagation, and is carried out based on a *learning rule* or *optimiser*, which modifies the weights of these connections in such a way that it minimises an *error function*. The goal is for a given input to ultimately be able of producing a favoured output, by modifying the weights and thresholds of the overall network to a generalised model that can successfully predict information about new data based on its abstracted information from training.

2.3.1 Network Learning & Optimisers

Backpropagation is the central mechanism by which neural networks learn (explained in detail by Bishop (1995)). Propagation implies the spreading transmission of something through a medium. In this case, the medium is the weights of the neural network, and the transmission is information related to the error produced by the neural network when it attempts a guess on input training data. In simple terms, a learning iteration in a neural network first *forward* propagates the input data signal forward through the network to "make a guess", calculates the error associated with the guess, then *backward* propagates information about the error through the network, altering the learnable parameters to attempt to reduce that error. We monitor the error value for each iteration through the use of an *error function*, which is any function that maps the output values of one or more variables into a single real number that represents the error value of the output. An error in this case is the difference between the network's output and the expected output for the training example. The difficult part of the weight adjustment comes in determining the direction in which weights must be adjusted to reduce the error of the network. This is generally done through the network's *optimiser*.

Optimisers, or *optimisation algorithms*, are the rules which guide backpropagation towards a minimised error function. If one visualises the error value against the network parameters as a multi-dimensional surface, the goal of the optimiser is to find the valleys in that representation, meaning the parameter values at which the error is lowest. In this regard, optimisers follow an approach that is called *gradient descent*. The name is fairly self-explanatory. A *gradient* in this case is any slope whose angle can be measured, representing a relationship between two or more variables. In a two-dimensional scenario, a *gradient* is defined as: *"instantaneous rate of change of y with respect to x"*. In the case of a function with multiple variables, the gradient is calculated using partial derivatives. A *gradient descent* is therefore the process of sequentially following these gradients towards a minimum value, using the slope to guide direction of optimisation. To find such a local minimum, the parameters must update proportionally to the negative of the gradient of the function at the current point in the multi-dimensional graph. If instead a local maximum is required (a process known as *gradient ascent*), then the gradient should be used.

The size of each step towards the given direction is controlled by the *learning rate*. *Learning Rate*, often denoted by the symbol η, is the rate at which new information updates from previous information during network learning. Choosing a learning rate is a difficult process. Smaller values converge more accurately, but are much slower to do so; and vice versa, larger values can hinder exact convergence and fluctuate around the minimum instead of landing exactly on it.

The traditional form of gradient descent, using the exact methodology described above, is called *Batch Gradient Descent*. In gradient descent, a *batch* is the total number of examples used to calculate the gradient in a single iteration. For this form of gradient descent the batch is taken as the entire dataset. This is beneficial in reaching minimum values with very little noise and randomness, but is computationally expensive as the datasets get larger, and borders on completely infeasible when the dataset is past a particular size. For example, if there are a million samples in the dataset, Batch Gradient Descent will use each of the one million to complete one update, and repeats this entire process for every single iteration until a minimum is reached. The size of the update is controlled by the learning rate, and is guaranteed to converge towards a minimum value. If the error surface is convex, this will be the global minimum, but as is often the case error surfaces can be non-convex, and this form of gradient descent will in this case get stuck in a local minimum instead. Since progress is sequential, once it reaches a minimum where the gradient slopes upward in the entire neighbouring area, the algorithm is incapable of leaving that minimum in a sequential fashion, and therefore believes it has achieved an optimal result.

Stochastic Gradient Descent

The inherent problems of traditional gradient descent are rectified to some extent through the use of an optimisation algorithm that builds on top of standard gradient descent, called *Stochastic Gradient Descent (SGD)*. The term *stochastic* is used when a system or algorithm relies on some form of random sampling or probability. In SGD, a randomly selected sample from the whole dataset is selected for each iteration. The batch in this case changes from the entire dataset to this single sampling. In this way, the gradient of the error function is calculated on a small sample set at each iteration instead of the entire sum of gradients from all samples in the dataset. Since the algorithm has significantly less data to work with finding an optimal direction, the path taken towards the minimum value is usually far noisier than typical Batch Gradient Descent, but since each iteration is also significantly faster the minimum is still reached within a far shorter training time despite a higher total number of training iterations.

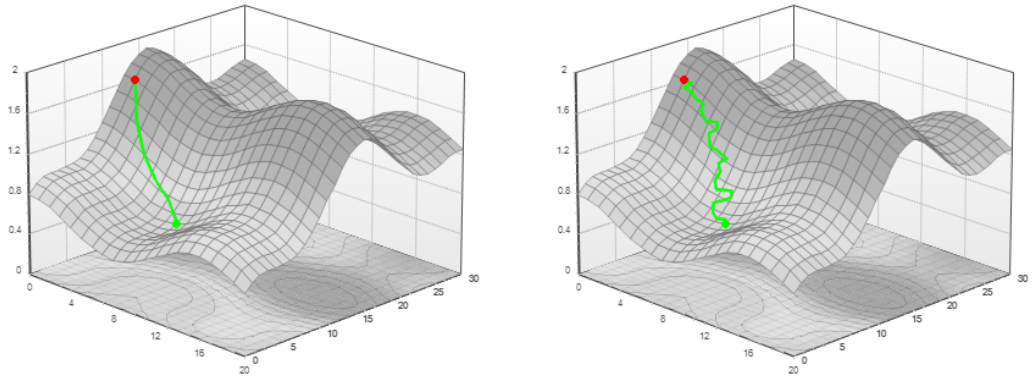


FIGURE 2.6: A visual rendition of how a theoretical path taken towards a minimum differs between Batch Gradient Descent (left) and Stochastic Gradient Descent (right).

The seemingly random jumping in SGD is also beneficial for avoiding local convergence, as it helps the algorithm discover different minimums and increases the chance of converging towards a global one instead of local. On the other hand, the disadvantage is that due to the frequent updates and large fluctuations, the convergence can take more time to reach an exact minimum as it will often overshoot during a fluctuation. To solve this, variations of SGD algorithms can slowly change the learning rate over time, thereby adapting to the rate of fluctuation so that the optimiser can begin to converge more steadily towards the nearest minimum.

Due to its tendency to fluctuate drastically between iterations, SGD has trouble navigating areas in the error surface where the slope curves far more steeply in one dimension over another, called *ravines*. In these cases, SGD tends to oscillate around the slopes of the ravine while making very little progress towards the local optimum. To work against this problem, a concept of *momentum* is introduced. In a physical scenario, a ball running down a slope builds up speed as it progresses downhill, becoming faster until it reaches a terminal velocity. In the context of network learning, this concept is applied to accelerate SGD in the relevant direction of the required minimum point, and thereby dampens variance during the journey. This is accomplished by routinely adding a fraction of the update value from the previous time step to the current update value, causing the learning rate to increase over time rather than remain fixed.

Adaptive Moment Estimation

Adaptive Moment Estimation (Adam) (Kingma and Ba, 2015) is an algorithm that builds on top of SGD and the concept of momentum by computing adaptive learning rates for each parameter. Adam uses squared gradients to scale the learning rate and keeps an exponentially decaying average of past gradients. With momentum, the virtual ball running down a slope continues to build up speed and can keep that momentum to roll past a smaller valley (local minimum) in the slope. Adam instead behaves like a heavy ball with increasing friction, causing it to slow down when it encounters flat minimum valleys in the error surface. The goal is to ensure that too much momentum does not cause the descent to overshoot the desired result.

The decaying averages of past and past-squared gradients m_t and v_t respectively are calculated as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where g_t is the gradient, and β_1 and β_2 are the decay rates. The values of m_t and v_t are initialised as vectors of 0's, and are observed to possess a bias towards zero during the earlier time steps of the descent. A biased estimator is a common problem with adaptive learning rate algorithms, and is typically resolved through the use of bias correction techniques for initial estimates. In the Adam algorithm, this bias correction is done as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The authors propose default starting values of 0.9 for β_1 and 0.999 for β_2 , which are empirically proven to work well in practice (Kingma and Ba, 2015), comparing favourably to other adaptive learning algorithms.

2.3.2 Long-Term Dependencies & LSTMs

One of the major disadvantages of traditional neural networks is that they lack persistence. When humans think, they will continue their thinking process based on previous context rather than restarting from scratch every second. Traditionally, neural networks cannot accomplish this and are unable to reason about previous events in an input to reason about later events in that same input. Recurrent Neural Networks (RNN) attempt to address this issue by introducing a system of loops within the network, allowing information to persist. Essentially RNNs are much like multiple copies of a normal neural network, each passing output to a successor to create a chain-like architecture.

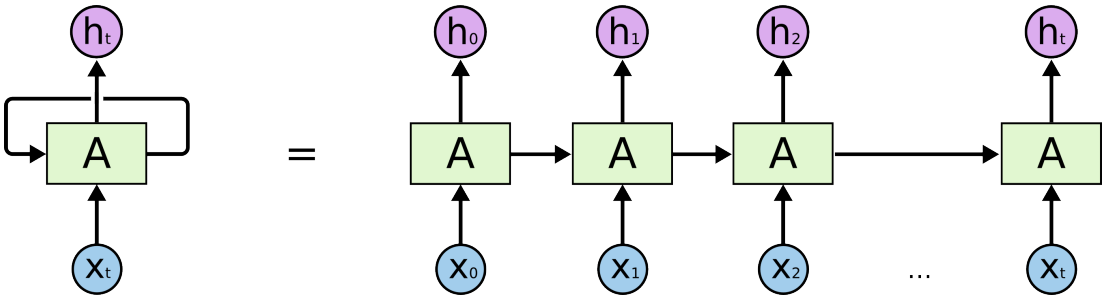


FIGURE 2.7: An example of an unrolled RNN's basic architecture, where A is a layer of the network, x_n is the input, and h_n is the output of that layer. Diagram by Olah (2015).

Over recent years, RNNs have been used to great success in a variety of problems, such as: language modeling, translation, speech recognition, and image captioning (Karpathy, 2015). This success is mainly due to a particular type of RNN, *Long Short Term Memory (LSTM)* networks.

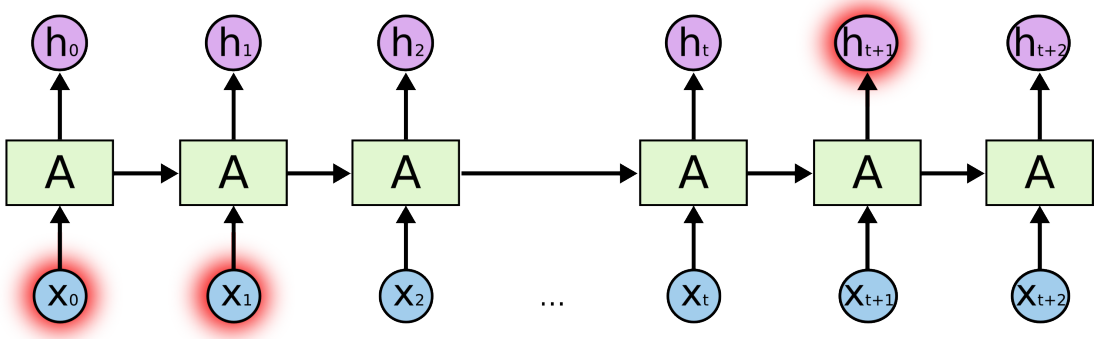


FIGURE 2.8: An example of an RNN where the output h_{t+1} requires inputs from x_0 and x_1 to compute properly, both of which are far earlier inputs that a normal RNN cannot process. Diagram by Olah (2015).

As stated previously, the main appeal of RNNs is the concept of persistence, and the idea that it could be possible to connect previous information to the present task.

However, in reality, RNNs are somewhat limited in how much they can accomplish. If the gap between relevant information and the place where it is relevant is relatively small, RNNs *can* use the past information, but the larger that gap grows the less likely it is that RNNs can actually connect the two pieces of context. For example, when trying to predict the last word in a short phrase, such as "the sun is in the *sky*", very little long-term context is needed, since it is fairly obvious that the last word is "sky". If that text grows longer, such as "I lived my entire life in England, and since I went to school and did a lot of reading, I can speak fluent *English*", it becomes substantially harder to predict the final word. Recent information would suggest that the next word is a language, but to narrow down *which* language, context from much further back is necessary, chiefly the context of "living in England". The text in question can continue to grow, and it is therefore entirely possible for the gap between context and prediction to become very large. This problem, called *long-term dependency*, and the associated difficulty is explored in more detail by Bengio et al. (1994).

LSTMs, introduced by Hochreiter and Schmidhuber (1997), are a special kind of RNN capable of handling long-term dependency. They work very similarly to RNNs, using a chain of repeating modules of neural networks, but whereas in RNNs the repeating modules are relatively simple (usually a single layer), LSTMs use a more complex structure with four interacting layers. The core of an LSTM module is the *cell state*. As shown in Figure 2.9 this is a pass-through line which runs straight down the entire chain, with only a few minor linear interactions making it easy for information moving along it to remain unchanged if necessary. The LSTM does have the ability to add or remove information from the cell state, a process which is regulated using *gates*.

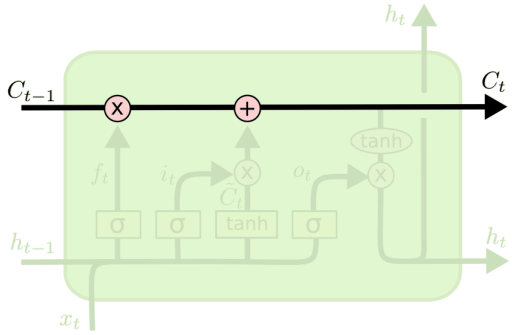
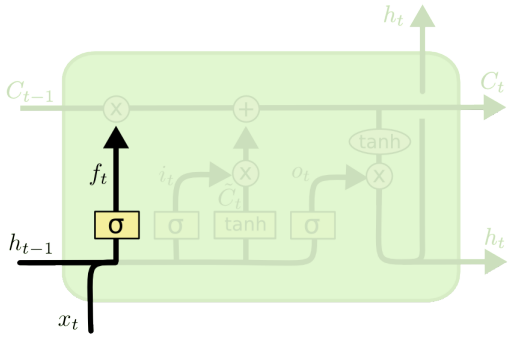


FIGURE 2.9: A diagram showing an LSTM cell where C_{t-1} is the input cell state, and C_t is the output cell state. Diagram by Olah (2015).

Gates are a small internal structure designed as a way to regulate information flow, typically composed of a sigmoid neural net layer and a point-wise multiplication function. The sigmoid function is a mathematical function characteristically having an "S"-shaped curve, known as a sigmoid curve. Mathematically the sigmoid function corresponds to: $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$. The sigmoid layer outputs a value between 0 and 1, which corresponds to the percentage of how much information should be let through. An LSTM has three such gates, designed to control the information in the cell state.

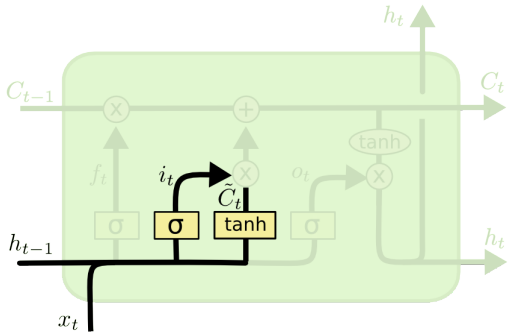
The first gate, called the *forget gate layer*, decides what information from the cell state is to be "forgotten". As shown in Figure 2.10, this gate looks at the inputs of h_{t+1} and x_t and outputs a number between 0 and 1 for each value in the cell-state C_{t-1}. A 1 implies completely keeping the input, while a 0 implies complete replacement. Values in between scale in a weighted ratio between the two.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

FIGURE 2.10: A diagram highlighting the first gate in an LSTM cell (left), and the functions it implements (right). Diagram by Olah (2015).

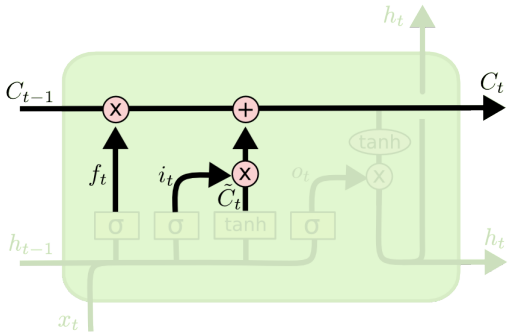
The second gate, called the *input gate layer*, separately determines which values are to be updated in the cell state. This is highlighted in Figure 2.11.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

FIGURE 2.11: A diagram highlighting the second gate in an LSTM cell (left), and the functions it implements (right). Diagram by Olah (2015).

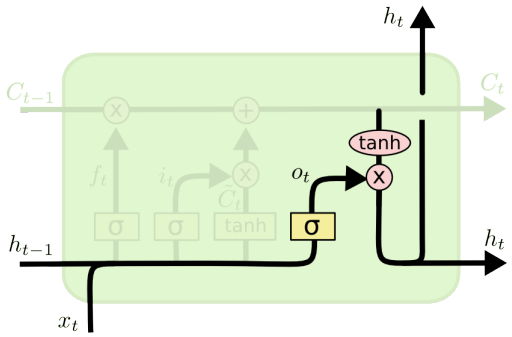
Combined with a tanh layer that generates candidate values, this creates a weighted update vector for the cell state, which can be multiplied with the old state to update it. The old state is first multiplied by the output f_t of the first gate to forget whichever parts of it should be forgotten, then it is added to i_t (the values to be updated as generated by the second gate) multiplied by the vector of potential candidates.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

FIGURE 2.12: A diagram highlighting the update in an LSTM cell (left), and the functions it implements (right). Diagram by Olah (2015).

The third and final gate determines which parts of the updated cell state should be outputted. This is a filtered version of the cell state, determined by another sigmoid layer followed by a tanh layer to push the values to be in the range between -1 and 1, which is then multiplied by the resulting cell state from the previous two gates.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

FIGURE 2.13: A diagram highlighting the third gate in an LSTM cell (left), and the functions it implements (right). Diagram by Olah (2015).

Consider an example language model that is trying to predict the next word in a text based on previous ones. In this model, part of the cell state would hold the gender of the present subject, so that any prediction made uses the correct pronoun for that subject. When a new subject is detected, it is important to update the gender held within the cell state. The first gate will determine which part of the cell state should be forgotten, that is the part of the cell state holding the old subject. The second gate will then add the new subject to the cell state to replace the one being forgotten. The third gate will attempt to predict the next input. Since a subject just occurred in the text, it would for example output information relating to a verb or action, such as singularity or plurality of the subject for proper conjugation.

Variants on this base form of LSTM do exist, with differing internal structures tuned to specific tasks. In their paper, Greff et al. (2017) provide a good overview and comparison of several popular variants. Alternatively, Jozefowicz et al. (2015) carry out a more comprehensive empirical study of RNN architectures in general, also comparing LSTM architectures to other types in an effort to determine optimal algorithms for different tasks.

Chapter 3
Pattern Discovery in MIR

In the context of MIR, the aim of pattern discovery algorithms is to separate a musical piece into its constituent repeated patterns through computational means. In the task for pattern discovery in the MIREX competition (Collins, 2017), pattern discovery algorithms are those that *"take a single piece of music as input, and output a list of patterns repeated within that piece"*. As mentioned previously (Section 1.2), patterns in this sense can range from small segments such as motifs, to large building blocks and structures such as a chorus or stanza. This section will take a look in some detail at the current field of musical pattern discovery and the different approaches in use.

3.1 Search Domain & Matching Style

Before looking at the approach for particular algorithms, it is important to consider the discovery domains in which the algorithm is operating. Pattern discovery can take place in two particular domains: within the same musical piece, or between different pieces of music within a corpus. Conklin and Bergeron (2008) label these as *intra-opus* and *inter-opus* respectively. In the former, the focus is on gaining an insight into the single piece with focus on its structure and composition. This is mostly useful for manipulating the piece in isolation (such as for compression or prediction). In the latter, the focus is more on how the piece relates to other pieces that may or may not be similar to it. This is, for example, useful for classification of musical pieces by melody (Conklin, 2009) or other features. The distinction between the two discovery domains immediately creates a separation between styles of pattern discovery algorithms, although this separation can sometimes be blurred[1].

A further separation can be made in the style of similarity matching implemented by the pattern discovery algorithm. When looking for pattern matches for a specific segment, one approach is to find exact matches within the search domain. Exactness in this context typically ignores transposition (Janssen et al., 2013), meaning that a pattern repetition a few tones lower or higher is taken as an exact match. In contrast, approximate or inexact pattern discovery algorithms attempt to find patterns that have undergone some form of variation, which can occur rhythmically, melodically, or through the transformation of music (inversion[2], retrograde[3], augmentation[4], ornamentation, and so on). Finding approximate matches instead of exact matches introduces an additional layer of complexity to the task, as there are many variations

[1] Technically speaking, one could concatenate a set of musical pieces together into a single piece to convert between the two domains.

[2] A process where a sequence of notes is inverted by flipping it "upside-down", reversing the melody's contour.

[3] A musical line which is the horizontal reverse of an another musical line

[4] A series of notes is augmented if the lengths of the notes is extended; augmentation being the opposite of diminution, where notes are shortened.

that can occur to factor into the algorithm. An implementation of approximate matching was used by Rolland (1999) who made use of a distance threshold to determine the closeness of a match.

3.2 Musical Texture & Representation

In any pattern discovery algorithm, it is important to consider the level of complexity in the musical data being used, as well as how that musical data is represented. Musical texture, as discussed in Section 2.1.4, is a crucial point of consideration for such algorithms. At the simplest level, a *monophonic* musical piece can be parsed quite easily into the formats required by the algorithm. Intra-opus pattern discovery in this case is a matter of comparison within the single sequence of notes or tokens, while inter-opus pattern discovery is a one-to-one comparison between single sequences.

A degree of complexity is introduced with *polyphonic* music, as in this case multiple voices are playing simultaneously which means sequential approaches to pattern discovery are more difficult to process. In symbolic representation, multiple voices are separated into respective channels, providing a degree of separation that makes it easier to process simultaneous events. The benefits of this are that the musical piece can be processed as multiple separate sequences quite easily. In audio representation, there are various levels of separation at which pattern discovery can occur (Klapuri, 2010). At one end of the spectrum, a polyphonic audio signal can be considered a single coherent whole with no voice separation. On the other end, the signal is processed such that parts of the audio signal are tuned out or separated into their own parts (such as keeping drums or vocals from a musical piece as a separate audio signal and performing pattern discovery on the remainder). Human listeners, particularly if trained in musical comprehension, can often switch between both ends of this spectrum when listening to an audio signal, but it is more difficult for a digital system to do so.

3.3 Geometric vs. String-based Approaches

The next step is to look at the way data is organised inside the pattern discovery algorithm. According to the overview presented by Janssen et al. (2013), there are two main ways of doing so: string-based or geometric methods.

In string-based methods, also called sequential structures, (Janssen et al., 2013), music is represented as a sequence of tokens called a string. The style of tokens can vary, representing notes as simply a sequence of MIDI pitch numbers (for example, *[44;48;42]*) or in more complex tokens such as pitch/duration tuples (for example, *[(F,2.0);(A,1.5);(D,0.5)]*). Algorithms based on this method then search for identical or near-identical sections of tokens in sequence in order to identify patterns. This approach is based on techniques used in computational biology for the comparison of gene sequences, an overview of which is given by Gusfield (1997). Some examples of string-based algorithms are the MGDP algorithm presented by Conklin (2010) which discovers exact matches in a corpus based on a predefined set of sequential viewpoints, and the MotivesExtractor-algorithm developed by Nieto and Farbood (2014) which finds both exact and approximate matches in a symbolic or audio piece through the use of a self-similarity matrix.

One advantage of string-based structures is their similarity with textual strings. This correspondence means that normal text mining algorithms can be easily applied to this form of structure. Of particular note to this project, LSTMs that are used for word prediction or text classification can be adapted to work on similar strings of note sequences. The disadvantage of string-based structures is that due to their linear nature the amount of information possible for representation is limited. In its simplest form, a string-based structure can only represent a string of note pitches, where each segment of information takes up exactly one spot in the string. While this may be sufficient for text or DNA strings, in music many more factors are of interest, not least of which is note duration. Representing such additional information would require multiple strings for each piece, with each string holding one part of the information, or else would require large tuples where not all retrieved information is relevant to the task. Similarly, polyphony cannot be easily represented without multiple string structures, one for each instrument or voice participating in the piece.

In order to deal more appropriately with polyphony and the complexity of musical pieces, string-based approaches will sometimes organise string sequences into an indexing structure, such as a graph or hierarchical tree that represent the repetitions better and make pattern discovery more efficient. Numerous examples of indexing structures in string-based approaches exist and are explained in more detail in the same overview by Janssen et al. (2013). Alternatively, Conklin and Witten (1995) use a system of *viewpoints*. A *viewpoint* creates a link between a specific identifier value assigned to a note in a sequence, and all the corresponding musical features for that note. It can be seen as a pointer that maps a particular event in the string structure (i.e. a note or a rest) to the required musical feature (such as pitch or note duration). Following this approach, a note sequence can therefore be represented as a sequence of viewpoints instead.

Viewpoints can be distinguished into two types: *primitive* and *derived* viewpoints. *Primitive* viewpoints are taken from musical features that are defined directly by the symbolic representation of the musical piece, such as note pitch, note duration, and start/end times. *Derived* viewpoints are taken from musical features that can be calculated using one more more primitive viewpoints. For example, the pitch interval is calculated using the difference between two successive note pitches, therefore the interval viewpoint is *derived* from the pitch viewpoint.

In geometric methods, such as the one presented by Meredith et al. (2002), a sequence of notes is represented as a shape in n-dimensional space. Each note becomes a multidimensional vector in that same space. This is similar to the viewpoint method of Conklin and Witten (1995) as described above, where each viewpoint corresponds to a dimension in n-dimensional space. Comparison is then carried out by matching shapes together to analyse similarity, identifying patterns as identical (or near-identical) shapes in the geometry. The main benefit of this approach is a more elegant handling of polyphonic music. Other examples of geometric approaches include: Collins et al. (2010) who made improvements to the geometry used by Meredith et al. (2002), and Szeto and Wong (2006) who use a graph-based approach rather than an n-dimensional shape.

3.4 Pattern Discovery Algorithms

In this section, we will look at some examples of Pattern Discovery algorithms that we can compare our approach and results to. Table 3.1, adapted from the same visualisation by Boot et al. (2016), shows a direct comparison between the three algorithms and how they correspond to the distinctions mentioned in the previous sections.

	Domain		**Matching**		**Texture**		**Format**		**Approach**	
Algorithm	intra-	inter-	exact	inex.	mono	poly	audio	symb.	string	geom.
MGDP		x	x		x			x	x	
MotivesEx.	x		x	x	x	x	x	x	x	
SymCHM	x	x	x	x	x	x		x	x	

TABLE 3.1: Properties of the algorithms described in this section, based on a similar summarisation by Boot et al. (2016).

The aim of this section is to get a better understanding of approaches that have been tried before and what results they achieve. By doing so, we can draw on ideas on how to structure experiments, input/output, and results. Discussing previous algorithms also gives a baseline for performance comparison that is useful during later analysis, and is generally useful as research to further understand Pattern Discovery in MIR before beginning implementation.

The MGDP and MotivesExtractor algorithms in Sections 3.4.1 and 3.4.2 are popular examples of pattern discovery algorithms using symbolic and audio data respectively. From these two algorithms we draw a better understanding of pattern discovery techniques and how they are applied. The SymCHM algorithm is chosen because it is a rare example of a pattern discovery algorithm in MIR that utilises a machine learning approach. While the approach is not based on LSTMs like this project, there is still merit in researching the techniques used by the algorithm to get an understanding of the difficulties associated with such an approach.

3.4.1 MGDP Algorithm

The *Maximally General Distinctive Pattern (MGDP)* algorithm (Conklin, 2010) is a pattern discovery algorithm designed to discover pattern occurrences in monophonic musical data, based on a sequential viewpoint approach as described in the previous section. In this algorithm, only exact repetitions are discovered. Musical pieces are represented as a sequence of predetermined viewpoints that are processed sequentially by the algorithm. To cull output size to an acceptable level, patterns that are non-characteristic of the *corpus*[5] are ignored. In order to determine if a pattern is characteristic, the approach is to compare it with a number of songs that are *not* in the corpus and check if the pattern still occurs. This set of songs is termed an *anti-corpus*. For example, to remove non-distinctive patterns in pop music each pattern can be compared to a dataset of classical music used as the anti-corpus. Patterns that occur frequently in both corpora are removed as they are not distinctive for the corpus of pop music, but general musical patterns. Since the dataset used by the algorithm is class-based, the anti-corpus is taken to be the classes that are not currently being processed.

[5] A corpus is a large, structured set of data.

The MGDP algorithm defines a number of criteria to determine pattern importance. Fulfilling these criteria is necessary for a pattern to be returned by the process. The conditions for the pattern are as follows:

- ***Supportive:*** Patterns that occur less frequently in the corpus are considered less relevant than patterns that occur in a large portion of the data. The *support* of a pattern is therefore taken as the *probability* that the pattern appears in the corpus. A threshold value is defined as a pre-determined cutoff point, and any pattern with a probability less than this *support threshold* is removed. The equation to calculate support of a pattern P is as follows:

$$p(P|\oplus) = \frac{c^{\oplus}(P)}{n^{\oplus}}$$

  where $c^{\oplus}(P)$ is the number of musical pieces in the corpus that contain the pattern, and $n^{\oplus}$ is the total number of musical pieces. The ratio $p(P|\oplus)$ of songs that contain P in the corpus must therefore be greater than the aforementioned threshold value.

- ***Distinctive:*** Pattern distinctiveness is calculated as a comparison of the support of the pattern in both the corpus and the anti-corpus. This means that a distinctive pattern must have a high probability of showing up in the corpus, and a low probability of showing up in the anti-corpus. The probability of a pattern showing up in the anti-corpus (the support of the pattern in that anti-corpus) is the same equation as above except $\ominus$ is used instead of $\oplus$ to represent the the anti-corpus. The ratio of the two probabilities is therefore as follows:

$$I(P) = \frac{p(P|\oplus)}{p(P|\ominus)} = \frac{c^{\oplus}(P)}{p(P|\ominus) \cdot n^{\oplus}}$$

  Similarly to the support threshold, this value must be above a *distinctiveness threshold* to be accepted by the algorithm as sufficiently distinctive.

- ***Maximally General:*** If a pattern P is a sub-pattern of another pattern Q, then P is more *general* than Q if all occurrences of the latter are also occurrences of the former. For example, the pattern A_5, B_5, C_5 is more general than the pattern A_5, B_5, C_5, D_5, as it can also include other occurrences where A_5, B_5, C_5 are not followed by D_5. The algorithm returns only those patterns that are classified as *maximally* general. In other words, no returned pattern must be a more general version of any other returned pattern.

To fulfill these criteria efficiently, the search is structured into a sequential tree of patterns. Child nodes are extensions of parent nodes, and are therefore less general. The tree is traversed depth-first, such that if a node is found to succeed on all the above criteria, the sub-tree of that node does not need to be traversed.

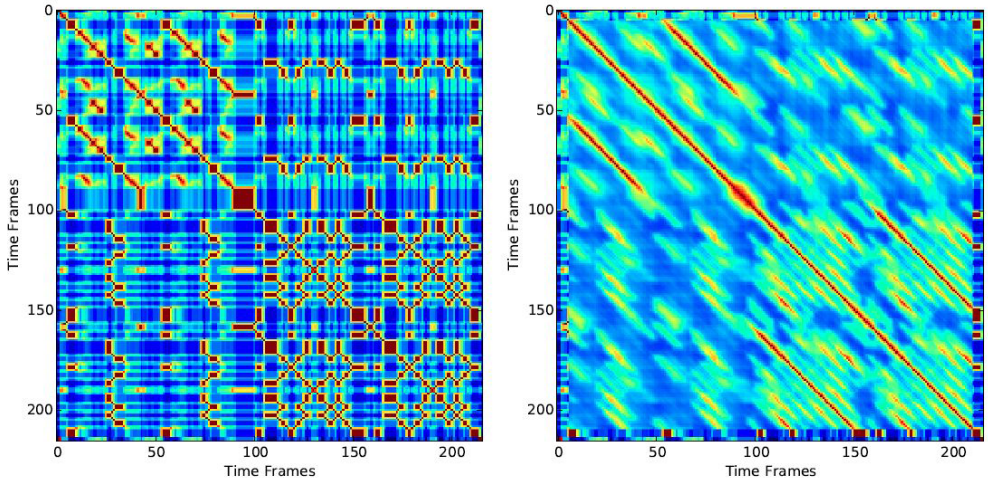
The MGDP discovery method was used successfully on two studies, based on data provided in the original paper. The first use was to discover melodic patterns specific to folk songs from different geographic regions, as represented in the Essen folk song database (Schaffrath, 1997). The algorithm was capable of finding highly distinctive patterns in each corpus when the remaining regions were used as an anti-corpus. The patterns were found to be very general, covering a large percentage of pieces in the regional dataset. The second use was applied to musical chord sequences (instead of note sequences), chords being often considered as a level of abstraction that reduces

the complexity of polyphonic music. The dataset used was divided into three genres, and each genre was tested with the algorithm while using the remaining two as an anti-corpus. The identified patterns were once again very general and distinctive, specific to the genre subset they came from rather than appearing across all musical pieces in the dataset.

3.4.2 MotivesExtractor

MotivesExtractor was originally developed by Nieto and Farbood (2014) for discovering patterns in polyphonic audio recordings but was extended to symbolic pattern discovery later on to participate in the 2014 MIREX task for the *Discovery of Repeated Themes and Sections*. To compare with the purposes of this project, the latter extension will be discussed here.

The approach used by this method is to compare audio pieces to themselves in order to discover repeated sections. If a piece is split into a number of time frames, then each pair of time frames can be analysed through a self-similarity matrix. In this way, patterns then become diagonal clusters of high similarity scores in the matrix. The main downside of this method is that transposition is difficult to detect in that the same note sequence but transposed is still considered different by the similarity matrix. To resolve this, a transposition-invariant matrix (Muller and Clausen, 2007) is used to detect repeated segments even when they are transposed during different repetitions. To calculate this matrix, all 12 possible transpositions are calculated then combined into a single self-similarity matrix. The result is a matrix such as the one in Figure 3.1 where red lines represent areas of high similarity score. To better observe the diagonals in the matrix, a filter is applied, as seen in the same figure.



(A) A transposition-invariant matrix for a symbolic musical piece.

(B) The same self-similarity matrix with a diagonal filter applied.

FIGURE 3.1: An example of a self-similarity matrix from Nieto and Farbood (2014). Colours in the matrix indicate similarity between time frames, where red pixels show high similarity and blue pixels show low similarity.

Once the matrix is calculated, a greedy tracing algorithm is used to search for the diagonal paths that represent sequences of high similarity. Since a self-similarity matrix is symmetric along the main diagonal, only paths above the main diagonal are searched. Each path is given a similarity score based on the total value of the path in the matrix. The sharper the line of the path, the higher the similarity score, and therefore the more prominent the similarity between sequences of time frames.

In the original paper, the algorithm is evaluated using the JKU Patterns Development Dataset and the metrics proposed in MIREX (Collins, 2017). The algorithm achieved state-of-the-art results when extracting patterns from audio data, and performed marginally worse on symbolic pieces. The algorithm was also found to be significantly faster than comparable audio-based pattern discovery methods, but somewhat slower than symbolic algorithms of the same type (Nieto and Farbood, 2014).

3.4.3 SymCHM

The *Symbolic Compositional Hierarchical Model (SymCHM)* developed by Pesek et al. (2017b) is an algorithm derived from CHM (Pesek et al., 2014; Pesek et al., 2017a), a model inspired by the learned Hierarchy of Parts (Fidler et al., 2009) approach to object categorisation. This algorithm creates a representation of a symbolic musical piece in a hierarchical structure, with individual notes forming the lowest layer and complex patterns forming the higher layers. The musical piece is therefore split up into a hierarchy of simpler blocks, called *parts*, where a part can be anything from a note event to a complex note sequence. Higher layers are then formed as *compositions* (sequential combinations) of parts from lower layers. This means that a *part* in higher layers is made up of *subparts* taken from lower layers. *Subparts* are still parts in their own layers.

The SymCHM algorithm consists of several compositional layers derived from an input layer. The input layer, L_0, is formed from the note sequence of the input file. Compositional layers $\{L_1, ..., L_N\}$ each contain a set of parts $\{P_1^n, ..., P_{M_n}^n\}$ where the compositonal layer L_n is formed of combinations of parts from the previous layer L_{n-1}. The parts of the previous layer become subparts for each part in the current layer. An example of this hierarchy is given in Figure 3.2. From bottom to top, the diagram shows the input layer and the compositional layers. The input layer corresponds to a symbolic music representation (a sequence of pitches). Parts on higher layers are formed as compositions of lower-layer parts (depicted as connections between parts). A part may be contained in several compositions, for example $P_{M_1}^1$ is a part of compositions P_2^2 and P_3^2 in the same figure.

One of the main differences between CHM and other hierarchical methods is that compositions are encoded in a relative rather than absolute manner. This means that for each part, the list of subparts that make up that part calculate their relative distance (or *offset*) from the first subpart of that list. This subpart is called the *central part*. In SymCHM, the offset is calculated by difference in semitones in the pitch axis. In other words, a composition encodes the distance in semitones between different patterns, which are represented by different subparts. The main advantage of this approach is that the model can learn from position-independent input which reduces the necessary size of the training data. This is in contrast to other deep learning approaches, which tend to accept input data in an absolute manner and therefore require larger amounts of input data to train a properly generalised model.

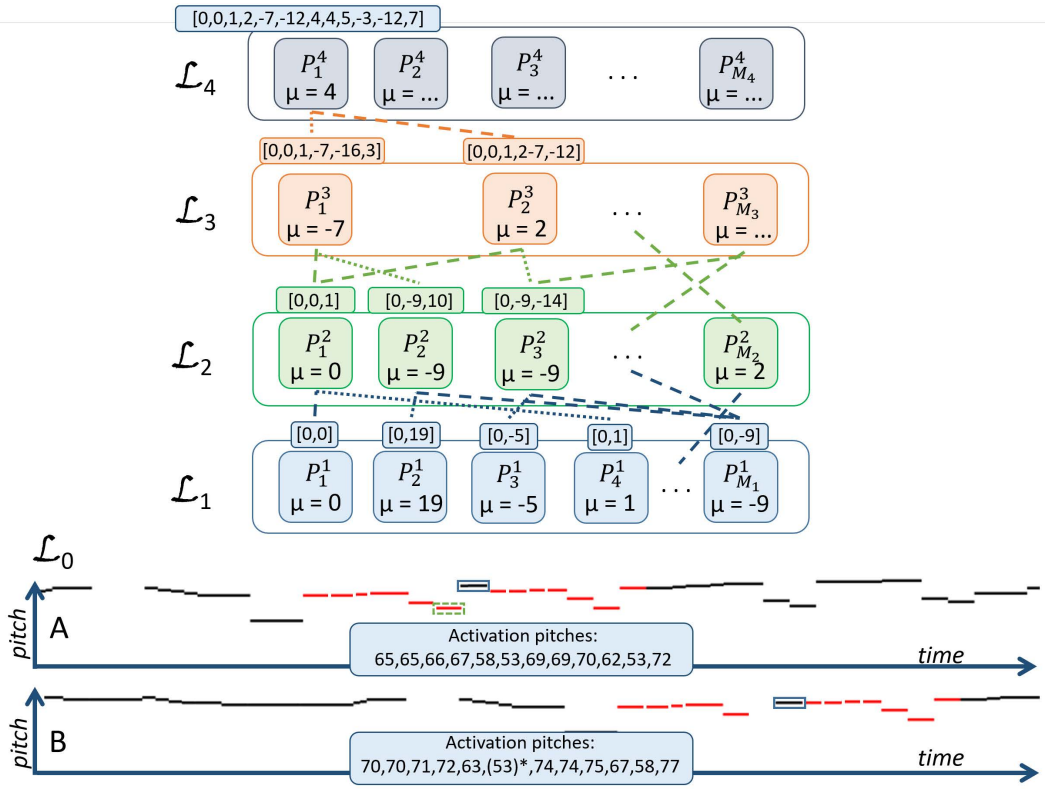


FIGURE 3.2: A diagram of the symbolic compositional hierarchical model, taken from the SymCHM paper (Pesek et al., 2017b).

To construct the hierarchy of parts, the model is generated layer by layer through the use of unsupervised learning. The input is a single or multiple musical pieces, and the learning process is an optimisation problem where each layer's possible set of part compositions is searched iteratively for a minimal subset that covers a maximal amount of parts in the training data. The process thereby consists of two main steps: (1) finding the set of possible compositions, and (2) selecting those compositions that use up the maximum amount of events from the training set. Once a model has been trained, it can locate the learned patterns in new input data through the process of *inference*. *Inference* calculates the activation of parts on this new input data, starting from the bottom-up and working layer by layer. The activation of a part in this case is when a specific occurrence of that part is found in the input. An activation is made of three parts: location, onset time, and magnitude. Location and onset time convert the relative representation of the part into an absolute position within the input, mapping the pattern onto the specific set of pitches that form it in the input sequence. Magnitude represents the strength of correspondence in the part. Since patterns can be repeated, a part can activate at multiple different locations simultaneously, representing multiple occurrences of that pattern in the input data.

The process of inference in SymCHM is designed to be either exact or approximate. In the latter case, a further two mechanisms must be employed to enable the finding of parts that have changes, insertions, or deletions compared to the original pattern. This helps to increase the robustness of the algorithm to changing input data. The mechanisms are called *hallucination* and *inhibition*.

Hallucination relaxes the conservative nature of the inference process. In the basic approach, a part activation is produced only if all subparts activate with a magnitude that is greater than zero at locations which approximately correspond to the structure encoded by the part. Hallucination bypasses this by enabling parts to activate even when the structure is incomplete or modified in the input data. For example, modifications can include missing notes, added notes, pitch changes, or changes in note order. By doing so, hallucination enables the model to find pattern variations by filling in the missing information through knowledge acquired during the learning process. When using hallucination, the model will generate activations of parts that most fittingly cover the input, rather than exactly match. This is implemented by changing the conditions under which parts may activate. Instead of activating only when all subparts are activated, the part now activates when the percentage of events it represents in the input representation exceed a hallucination threshold. The value of this threshold influences the number of patterns discovered. When lowered, the amount of activations increases since parts can activate on increasingly more incomplete matches.

Inhibition in the SymCHM model is a process of hypothesis refinement that aims to reduce the amount of redundant activations. A part activation is inhibited when one or more other parts cover a large percentage of the same events in the input representation but with a stronger magnitude. The inhibition mechanism helps to reduce the number of activations and output patterns, but can also be useful for producing alternative explanations of an input. If the activations of a strong pattern which inhibit other competing patterns are removed from the model, the next best hypothesis is selected during inference, thus providing an alternative explanation with different pattern occurrences in the model's output.

SymCHM was evaluated as part of the MIREX evaluation campaign (Collins, 2017) on the JKU PDD dataset. Results showed that the algorithm could obtain favourable performance and could be used for finding patterns in symbolic music in an unsupervised manner without hard-coding the basic rules of musical theory. They surmise that the definition of a "pattern" remains elusive, but that the proposed model could employ unsupervised learning to better approximate how listeners recognise patterns, and that it could do so better than rule-based alternatives. They also emphasise that the model produces multiple hypotheses on several layers, which can be used as reference points in a deeper semi-automatic music analysis, further increasing the algorithm's usefulness to the MIR community.

Chapter 4
Implementation

4.1 Experiment Framework

As mentioned previously in Section 1.5, the training and testing process of the project will take place in three distinct stages. The first stage will be trained on pieces created synthetically. This is done by taking pre-defined simplistic patterns (such as scales, interval repetitions, and so on) combined with random notes and rests at random places, and concatenating everything in random orders to create a "musical piece" with exact boundary points (where a pattern starts and ends). In the second stage, a similar artificial joining approach will be employed, but this time by randomly concatenating patterns taken from real musical pieces. This ensures that some pattern boundaries are still clear, but introduces additional complexity that is not present in simplistic patterns like scales or interval sequences. The third and final stage, will be trained on non-artificial musical pieces that have been previously manually-annotated by musicologists or by average listeners.

The main goal of the three stages is to analyse performance on increasingly more complex data. Complexity in this context comes from two sources: more diverse pattern types and note sequences, and more ambiguous pattern boundaries. This section will go through an overview of the system pipeline, and describe each part in more detail to determine the parameters that will be tested for each stage's training phase. The neural network setup will be coded in Python, built off the *Google Tensorflow*[1] framework, more specifically using *Keras*[2] to quickly build network models for faster testing. The *music21*[3] toolkit will be used to handle MIDI parsing and generation in Python.

The pipeline for each stage is more or less identical, as shown in Figure 4.1. The process begins with a dataset tailored for that particular stage of training. Datasets will be discussed in more detail in a later section in this chapter. The *Training* phase of the pipeline is the main focus point of the project. In this phase, loaded information from the dataset is passed in vector format to the Neural Network, which is arranged in a sequence of layers known as a *model* or *topology*. This layer topology is constructed using *Keras*, which in turn uses *Tensorflow* as its back-end but provides a simpler interface for interacting with the complex framework.

FIGURE 4.1: The basic sequence of events followed for each stage.

[1] https://www.tensorflow.org/
[2] https://keras.io/
[3] https://web.mit.edu/music21/

4.1.1 Training

During the training phase, each sequence of notes from each data file is split into a number of smaller sections, typically overlapping, called *batches*. Each batch is passed through the network layers, which includes two LSTM layers in the system topology. In each layer, weighted neurons transform the input and pass it to the next layer. The neurons are weighted randomly at first, but will adjust with every iteration of the training process through the use of back-propagation. One pass through the network ultimately results in a predicted output. This output is matched against the expected result, which in this case would be the predefined pattern boundaries arranged synthetically (for Stages 1 and 2) or annotated manually (for Stage 3), and then back-propagated to adjust the weights of the network model in an effort to reduce the difference between prediction and expected. This process is repeated for each batch in a file. A single iteration through every batch in the entire input note sequence is called an *epoch*. The number of epochs per input sequence typically determines how closely the network weights are fitted to that exact sequence of notes, and should be adjusted to avoid *over-fitting*[4].

At the end of every epoch, the current state of the network is tested against a *validation set*. This is typically a small subset of the training data that is used exclusively to test performance at every iteration of the training phase. As the datasets used in this project are relatively small, removing parts of the data for validation can result in an opposite problem, *under-fitting*[5]. By reducing training data, there is a distinct possibility of losing unique pattern sequences that can influence the fit of the network. To counter this while still validating results, we use *n-Fold Cross Validation*. In this method, the training data is divided into n subsets, called *folds*. One fold is set aside for validation, while the remainder are used for training. The training is then repeated n times, with each fold being used once for validation. The error estimation is then averaged over all iterations to get a sum total effectiveness of the model. In this way, every bit of data is used as validation for exactly once, reducing the possibility of over- or under-fitting.

A separate portion of data at every stage is kept aside for final testing. The goal of this *Testing* phase is to evaluate performance of the network on a sequence of data that has not been explicitly trained upon, thereby further avoiding the chances of over-fitting.

4.1.2 Testing

When a training run is complete, the state of the neural network is saved as an `mdf5` file (called a *model file*) so that it can be reloaded when necessary. This file stores a snapshot of the exact network layers, weight matrices, and any other information pertinent to the training process. Due to cross-validation, each training run will create a number of model files equal to the number of folds for that training run. Each fold is theoretically considered a separate training run (until later averaging) and must therefore create its own snapshot of the neural network state due to the possibility of different training outcomes. In order to test the trained LSTM models for each fold, the separate input set of testing files is passed into the loaded network model for that fold. The output

[4]Over-fitting in neural networks is the creation of a weighted network that corresponds too closely or exactly to a particular set of data, and may therefore fail to predict future data reliably.

[5]Under-fitting in neural networks is the creation of a weighted network that is too general and does not sufficiently fit the data or capture the properties of that data

is a prediction by that version of the network of what patterns each note sequence is split into and can be compared against the actual expected output to calculate *precision*, *recall*, and *f-score* metrics.

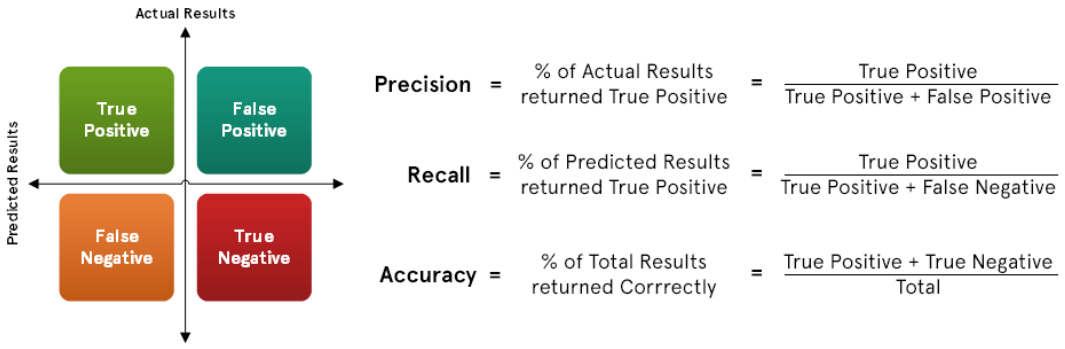


FIGURE 4.2: A diagram showing the equations for precision, recall, and standard accuracy and how they are derived from the four types of results.

Precision and *Recall* both represent an aspect of the accuracy of the model. *Precision* is the percentage of returned results that are relevant to the task, whereas *Recall* is the percentage of total relevant results that are correctly classified by the algorithm. Both metrics are important, so a metric that takes the harmonic mean of both, *F1-Score*, is typically used to grade performance by a single value. The equation to calculate F1-Score is given below:

$$F1 - Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

During the prediction process, a batch of data is taken from the input test file and passed to the trained network. This returns an equivalent batch of predicted output, represented in a matrix of categorical likelihood. This representation is a multi-dimensional vector where each possible output is given a prediction value in the form of a float number between 0 and 1, such that the highest value is the actual predicted output. For example, if there are 10 possible outputs for an 8 note sequence, the output is a 8x10 vector where for each of the 8 rows, the 10 columns are all given a float value between 0 and 1 (essentially a percentage confidence that the class that column represents is the predicted output). The column with the highest value is therefore considered the predicted output. Once the index of that predicted output is identified, it can be translated into the actual predicted value (for example, the name or number of the pattern class) based on the dictionary of possible outputs.

By looking at the predicted value compared to the expected value, we can identify true positives, false positives, true negatives, and false negatives. From these four values, it is then a simple matter to calculate the necessary metrics based on the equations shown above in Figure 4.2. In a non-binary classification problem such as this project, precision and recall must be calculated for each class. To do this, a confusion matrix is generated during testing. This matrix can then be used at the end of testing to calculate all the necessary metrics.

4.1.3 Parameter Tuning

In machine learning, one of the main problems during the learning process is finding the optimal parameters and constraints on the system. Given a fixed machine learning model, the system can utilise different weights, constraints, learning rates, or starting values in order to perform differently during the training process. Any measures that control the learning process through their values are called *hyperparameters*. Generally, the process of optimising these parameters is guided by some form of metric. In the case of our system, we use metrics from the implemented cross-validation, looking at the categorical accuracy on the validation subsets to measure the improvement (or lack thereof) over each epoch, as well as the set of three performance metrics listed in the previous section. The parameters that we investigate are listed below:

- `batch_length` Information is passed into the LSTM network in chunks of information, called *batches*. These chunks in this project are small sub-strings from the note sequence, and the length of the sub-string can vary. Typically speaking the length is kept as a multiple of the time signature in order to keep each batch as a whole number of bars and avoid complication, but theoretically the batch length can be any value within the limits of the sequence length.
- `hidden_size` The hidden size of an LSTM network is the number of internal learned parameters, and is a direct representation of the learning capacity of a neural network. The number of hidden units reflects the storage capacity of the LSTM layer, which in turn affects the size of the connection weights matrix as described in Section 2.3. Generally, more hidden units tends to correspond to a more perfectly memorised training set, which can result in over-fitting. A lower number can give a more flexible network, but run the risk of not being able to learn a suitably generalised model.
- `num_epochs` As described in the previous section, an epoch is a single iteration through every batch in the entire input note sequence. A higher number of epochs can cause the model of learning more about that exact sequence of notes, but can also become over-fitted to that sequence at the expense of lower performance on other inputs. Vice versa, a lower number of epochs can remain too close to random performance as it does not have enough iterations to learn enough about the note sequence.
- `nfold_splits` As described in the previous section, in the cross-validation method, the training data is divided into n subsets, called *folds*. One fold is set aside for validation, while the remainder are used for training. The training is then repeated n times, with each fold being used once for validation. The ideal number of folds is dependent on the size of the training set. The aim is to balance the size of validation sets and the length of training with a number of folds that properly represents the variation of the underlying distribution of data. A higher number of folds can create a scenario where the validation subset is far too small and the number of tests is far too high, but on the other hand a smaller number of folds can lead to not enough of a spread of the data to properly assess the general performance of the model on the "independent" validation sets.

To exhaustively analyse the performance of the model when tuning these parameters, a *grid search* approach will be used, also known as a *parameter sweep*. This method involves an iterative search through different values of each parameter, tracking the performance of the model with each change to eventually figure out an optimal (or near-optimal) combination of parameter values. The list of parameter values being tested during the evaluation of this system is given below:

Parameters		
Name	**Description**	**Values**
`batch_length`	Number of notes in input batch	{8, 16, 24}
`hidden_size`	Number of hidden units in LSTM	{64, 128, 512}
`num_epochs`	Number of iterations per input	{5, 10, 25}
`nfold_splits`	Number of subsets for cross-validation	{5, 10, 15}
`optimiser`	Optimisation algorithm used	{adam, sgd}

4.2 Experiment Data

Each stage in the project requires input data to perform training. For the first stage, a pool of synthetically-generated patterns from which to randomly adjoin into the training and testing data is required. These patterns are monophonic and simplistic, created from variations of scales and interval repetitions. The length of each pattern varies, ensuring that pattern boundaries are not at fixed gaps in the sequence. Once a pool of patterns is generated, they can be randomly adjoined together to create a number of simple musical pieces that serve as training input. Random notes and rests of varying length can be added into the mix to introduce a further element of randomness and attempt to avoid overfitting. More information about this part of the dataset is provided in Section 4.2.2.

For the second and third stages, more realistic musical data is required. This is collected in the form of symbolic MIDI files split into pre-determined patterns or motifs, usually through manual annotation. The dataset MTC-ANN is used for this purpose, as files are conveniently organised into monophonic MIDI files with annotated pattern boundaries. For the second stage, additional processing is required to split the MIDI data into a pool of predefined patterns for synthetic joining (similar to the process in the first stage). More information about this part of the dataset is provided in Section 4.2.3.

4.2.1 Simple Data (Stage 0)

Before proper training of the network in Stages 1, 2, and 3, it is necessary to run some preliminary tests to ensure the network is actually performing as expected and returning some form of result, as well as to isolate the best values for certain parameters that can affect the length of training time more dramatically (such as the number of epochs or the number of n-fold splits). To do this in a controlled manner, an even simpler set of randomly generated data is created. This takes the form of short note sequences with uniform length, where each sequence is a single pitch out of three possible pitches, as shown in Figure 4.3.

A series of preliminary parameter tuning tests using the parameter sweep method described in Section 4.1.3 is carried out on this simplified dataset to get a better understanding of how parameters can affect the model and inform the sequence of tests carried out in later stages. This is coined as "Stage 0" of the process.

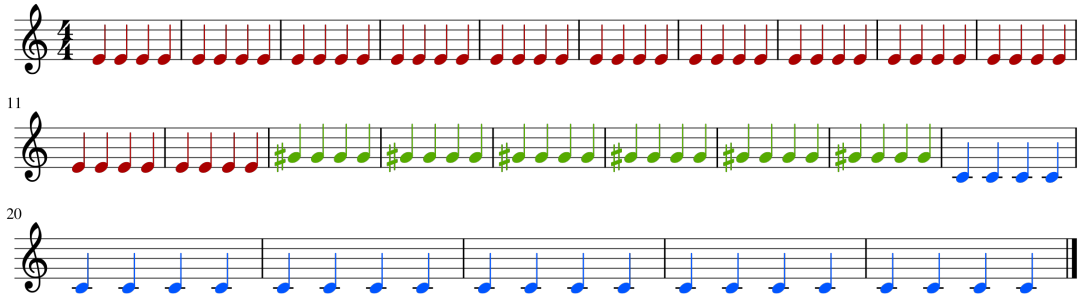


FIGURE 4.3: MIDI representation of an example file from Stage 0 with notes coloured by class: `pat1` (blue), `pat2` (red), and `pat3` (green).

4.2.2 Generated Data (Stage 1)

In order to establish an understanding of the neural network's performance and how it deals with musical data of lesser complexity, it is necessary to first test it on simpler music that is generated artificially. This means that the pattern types are not realistic in that they are very uniform in structure and do not necessarily represent real musical data, but should in theory be easier for a neural network to distinguish between. In this project, Stage 1 fulfills this part of the task. To prepare the data for this stage, 100 MIDI files were generated using random sequences of three pattern types: `ran`, `pat1`, and `pat2`. The length of all patterns is forced to be a whole number of bars (i.e. a multiple of 4 in this case). The `ran` pattern type is a generated sequence of random quarter notes. The `pat1` pattern type is an ascending scale of notes, starting and ending at fixed pitches. The `pat2` pattern type is a sequence of intervals between two fixed pitches, lasting for a fixed number of whole bars. In this way, the three pattern types are fundamentally different, which should hopefully provide enough of a distinct pattern boundary for the neural network to learn during training.

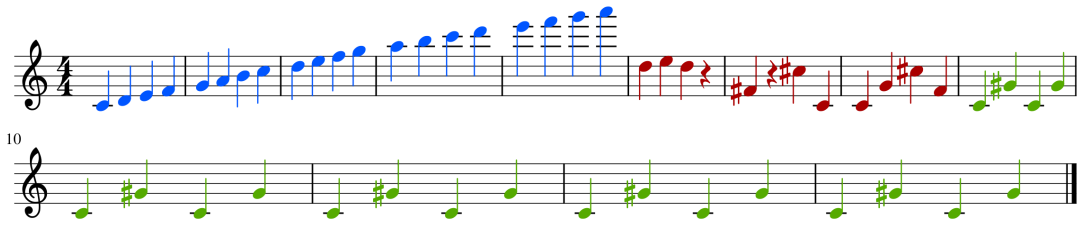


FIGURE 4.4: MIDI representation of an example file from Stage 1 with notes coloured by class: `ran` (red), `pat1` (blue), and `pat2` (green).

Since 100 files is a relatively small number for training purposes, the dataset is expanded through the use of some standard data augmentation techniques. We use augmentation instead of simply generating more files because the latter would involve introducing more variations of the `ran` pattern type, which means that the network could potentially skew towards it more as it associates more note sequences with that class of patterns. By using augmentation, we limit the internal variation of

pattern types but still introduce more occurrences of each pattern type in sequence with other pattern types. In this way, the network is able to learn how pattern types differ without struggling too much to generalise a definition for each pattern type. Augmentation also provides the benefit that the original 100 files can be used as a test set, since they are similar and made of the same patterns, but different in order and length. Patterns extracted from the original 100 files are shuffled together into 200 new MIDI pieces to create a more varied dataset and further reduce the chance of overfitting. Further manipulation, such as pitch-shifting or inversion of patterns, can also be applied, but were not used in this case to avoid introducing additional complexity to this stage of the project.

4.2.3 MTC-ANN

The Meertens Tune Collection (Kranenburg et al., 2016), or MTC, is a compilation of datasets covering 4830 Dutch folk songs. Songs are labelled into a *tune family*, defined as: *"a group of melodies that descent from a single source but can vary by variation, imitation, and assimilation"* (Bayard, 1950). MTC-ANN is a smaller subset of the tune collection, consisting of 360 songs, each with annotated pattern boundaries for motifs within the note sequence. A motif is a small pattern of notes in the piece that is repeated in an exact or near-exact manner across multiple files. For the purposes of this project, motifs inside the MTC-ANN dataset are considered synonymous with patterns during network training, in that the boundaries of the motifs are taken as the pattern boundaries that the network is attempting to learn from.

In the MTC-ANN dataset, motifs are sorted into motif classes where motifs that are similar to each other are grouped into their own class. As the dataset is derived from real music, certain notes in the sequence do not belong to any motifs. In real music, it is rare for patterns to cover the entirety of the note sequence, and therefore notes that are not listed as belonging to a pattern are given a class of `None` instead. The list of motifs in the entire dataset is listed in a `.csv` file along with the start and end times of each motif. Each file is a transcription of song recordings and is therefore saved as a monophonic piece, which is ideal for avoiding the complexity polyphonic music introduces. The annotation of pattern boundaries in this dataset is done manually by musicology experts (Volk and Kranenburg, 2012), meaning that some ambiguity is unavoidable. This is countered to an extent by the synthetic joining process in Stage 2, where patterns from different files are joined together, making boundaries more prevalent and hopefully less ambiguous.

The synthetic joining process works through the use of a pattern dictionary. Each motif from every song of the MTC-ANN dataset is loaded into a dictionary of patterns. A random selection of patterns is then taken from that dictionary and joined together into one piece. The main issue with this approach is that the MTC-ANN dataset has a large number of pattern classes (94 in total), which leads to the network having very few examples of each class to learn from. To place this into perspective, the 5 most common motif classes in the dataset have roughly 50 occurrences in a dataset of 360 MIDI files. The majority of other motif classes have 20 or less occurrences in the entire dataset. When combined together into an input sequence, no class ends up having enough repeated occurrences for the network to sufficiently extrapolate a general definition of that class without a significantly increased number of epoch repetitions.

To reduce this problem, only a smaller sample set of classes is taken for the synthetic-joining process used in Stage 2. During the training and testing process of this project, the most common 5 motif classes are used. Each of these classes provides approximately 50 motif occurrences, which are randomly repeatedly sampled and joined into a dataset of 100 files where only occurrences of these 5 classes exist. The main goal here is to reduce ambiguity by having defined patterns joined together without an extraneous number of infrequent classes or notes that do not form part of a pattern. For this stage, the notes that do not belong to a motif class (assigned the `None` class) are ignored and only fixed motif classes are considered. The `None` class is re-introduced in Stage 3 where the goal is to train on completely unchanged real musical data and analyse performance.

4.2.4 Data Format

In all the datasets above, songs are stored in two files. For each piece, a MIDI file holds all musical information, namely the note pitch sequence and the duration of each note. A correspondingly numbered `.csv` file then stores the pattern boundary information. This is stored in a key-value format, where the key states the pattern type (for example, in the generated dataset for Stage 1, this would be either `Ran`, `pat1`, or `pat2`), and the value holds the duration (in quarter notes) of that pattern. The duration can be stored with fractions in the eventuality of pattern boundaries that do not line up with the quarter note separation. An example of this representation and the corresponding MIDI file is given in Figure 4.5.

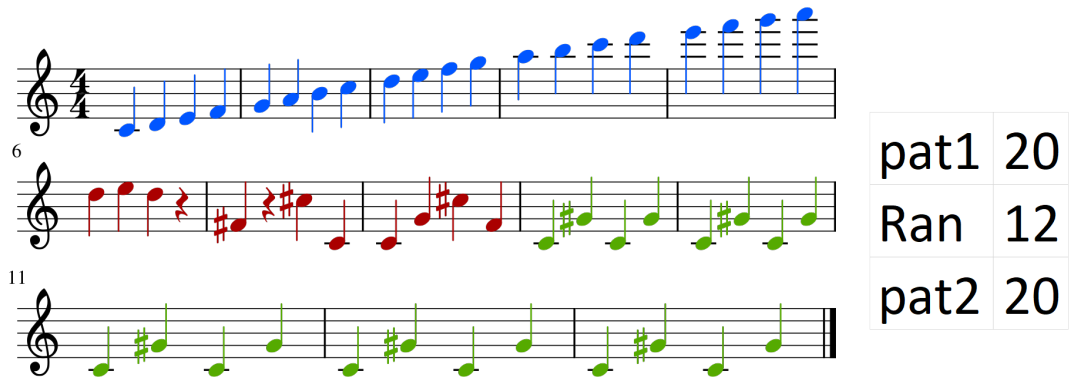


FIGURE 4.5: MIDI and CSV comparison of an example file from Stage 1 with notes coloured by class: `ran` (red), `pat1` (blue), and `pat2` (green).

4.2.5 Internal Representation

When loading the two separate parts of each file, there are alternative ways of representing the information during execution. The system uses a string-based approach, representing information as two parallel arrays of data, one for the pitch/duration sequence, and one for the pattern boundary information. The length of the arrays corresponds to the total sequence duration, and the number of array slots each note occupies corresponds to the single note's duration. In its simplest format, this means that each slot of the array represents one quarter note. In order to handle the possibility of smaller note values, such as eighth notes or triplets, a `slots_per_quarter` (or

spq) variable is introduced. The value of this variable represents how many slots in the array make up a full quarter note. For example, the default value is 1, meaning that 1 slot is equal to 1 quarter note. A ppq value of 12 means that a quarter note is 12 slots long, an eighth note is 6, and a triplet is 4. Both data arrays scale according to this ppq variable to maintain consistency of data.

The actual information regarding note pitch and pattern boundaries stored within each of these arrays depends on the choice made regarding the *mode* of representation for that particular training or testing run. Each of the two parts of data has two different representations that are used for this system, listed below:

- **Raw Pitch vs Pitch Intervals** The MIDI files in the dataset store data as a sequence of MIDI pitch numbers. Loading this information directly results in a *raw pitch* representation where the note sequence is an array of numbers between 0 and 128. The benefits of this representation are that it does not require any extra processing or manipulation of data, while simultaneously still representing all the necessary information about the note pitch for that note sequence. Alternatively, instead of storing the raw pitches, the difference between each consecutive pitch in the sequence can be stored to get a sequence of *pitch intervals*. More information about pitch intervals is given in Section 2.1.2. The advantages of this representation are that it is *transposition invariant*, which means if an identical pattern is shifted up or down in pitch, it is still detected as an identical note sequence, which is not the case for the raw pitch option.
- **Pattern Boundaries vs Pattern Class** After loading the `.csv` files, each slot in the boundary array corresponds to the slot of the same index in the note sequence array. This means that we can use this array to represent where each note belongs in the pattern separation of the piece. There are two methods for doing this utilised in this project. Firstly, in the *pattern boundary* format, each slot is given an integer value that corresponds to `pattern_start`, `pattern_end`, or `pattern_continue`. The location of pattern starts or ends can therefore represent the boundaries of every pattern in the piece, with no limitation of pattern size or number of pattern types. The disadvantage is that there is no distinction between different pattern types, and therefore the network is learning what makes up a pattern boundary, rather than what makes up a pattern as a whole. Alternatively, in the *pattern class* format, each slot is given an integer value that corresponds to a pre-determined list of pattern types. For example, in the generated data described in Section 4.2.2, there are three pattern classes: `ran`, `pat1`, and `pat2`. The advantage of this method is that the problem becomes one of classification, where the network can learn that a specific sequence of notes corresponds to one pattern type. The disadvantage is that in a large corpus, or a diverse one (such as the MTC-ANN dataset described in Section 4.2.3), the list of pattern classes can be far too large for the network to learn any meaningful distinction from the limited training data. By increasing the amount of pattern types, one is conversely decreasing the amount of occurrences of each pattern type for a dataset containing a fixed number of pattern occurrences.

4.3 Overview

The tables given below outline the series of training runs that are carried out on each stage of the system. Each run has two stages: training and testing. During training, input data in the format and representation established for that run is passed into a model with the setup of parameters established for that run. Once the model is trained, an independent test set can be passed into the trained model to get performance metrics for the combination of parameters being tested in each specific run. Each training run is executed with a set of parameter values as part of the aforementioned parameter sweep. The values highlighted in bold in these tables are the parameters currently being tested. For the purposes of time, only 3 values are tested for each parameter, though in theory more thorough testing of different parameter combinations and values can be carried out to fine-tune the system's performance more accurately. The values are selected based on the default values (and values close to that default) provided by `keras`, which are determined from common practice for neural network training. Batch length is selected as multiples of 4 due to standard bar length of 4 quarter notes.

	Data Format		**Parameters**				
Test ID	**Pattern**	**Pitch**	`batch length`	`hidden size`	`num epochs`	`nfold splits`	`optimiser`
s0t001a/s0t008a	class	raw	8	**512**	10	10	`adam/sgd`
s0t001b/s0t008b	class	interval	8	**512**	10	10	`adam/sgd`
s0t002a/s0t009a	class	raw	8	**128**	10	10	`adam/sgd`
s0t002b/s0t009b	class	interval	8	**128**	10	10	`adam/sgd`
s0t003a/s0t010a	class	raw	8	**64**	**10**	10	`adam/sgd`
s0t003b/s0t010b	class	interval	8	**64**	**10**	10	`adam/sgd`
s0t004a/s0t011a	class	raw	8	128	**5**	10	`adam/sgd`
s0t004b/s0t011b	class	interval	8	128	**5**	10	`adam/sgd`
s0t005a/s0t012a	class	raw	8	128	**25**	**10**	`adam/sgd`
s0t005b/s0t012b	class	interval	8	128	**25**	**10**	`adam/sgd`
s0t006a/s0t013a	class	raw	8	128	25	**5**	`adam/sgd`
s0t006b/s0t013b	class	interval	8	128	25	**5**	`adam/sgd`
s0t007a/s0t014a	class	raw	8	128	25	**15**	`adam/sgd`
s0t007b/s0t014b	class	interval	8	128	25	**15**	`adam/sgd`

TABLE 4.1: Table of tests for Stage 0, where each set of tests is repeated on each of two optimisers. Highlighted values indicate changes in parameter values for the parameter sweep.

In these tables, the test ID is an indication of the stage (`s0`), the number of the parameter combination for that stage (`001, 002, etc.`), and the pattern/pitch representation combination for that test (`a, b, c, d`). For example, the parameter combination of a batch length of 8, a hidden size of 512, an epoch count of 10, an n-fold count of 10, using the `adam` optimiser is given the number `t001` in this stage (`s0`). That parameter set is used for each combination of pattern format (class or bounds) and pitch format (raw pitch or intervals), labelled using letters `a, b, c, d`. For example, tn this case, the `class/raw` combination is given the letter `a`. For stage 0, tests are carried out on a simple dataset as described in Section 4.2.1. The dataset consists of 100 files, where each file is made of 3 pattern types, each being a simple sequence of fixed length of the note pitch corresponding to its class.

In this stage, the goal is to isolate a more optimal value for the 3 parameters that were observed to affect training time the most: hidden size, number of epochs, and number of n-fold splits. By doing so, the time taken to train in later stages is reduced as the parameter sweep does not need to be repeated. However, due to the simplistic nature of the musical data (a pattern in this stage is simply a sequence of the exact same pitch and duration), it is more difficult to determine the effect of the batch length on performance, as patterns in this stage are all of fixed and uniform length. As such, these parameters were not tested in this stage, but kept aside for their own parameter sweep in later stages. In theory, since later stages have patterns of varying length, the parameter value of the batch length should have a more pronounced effect in those later stages.

For Stage 0, the parameter sweep is carried out twice, once for each optimiser being tested. The two optimisers, `adam` and `sgd`, are the ones described in Section 2.3.1. Other optimisers can be applied to the system easily, but the ability to do so in this project was restricted by time constraints. The results of all these training runs and the associated tests are discussed in the following chapter.

The parameter values that give the best results in the Stage 0 tests are carried forward into the next stage, training on generated data as described in Section 4.2.2. The parameters that were not initially tested, batch length and boundary pattern representations, are used during this stage. As explained previously, the effect of these parameters is theorised to be dependent on the input data. The format of the test ID remains consistent for this stage. The parameter values used in Table 4.2 are those determined by the Stage 0 tests, the results of which are explained in more detail in the next chapter.

	Data Format		**Parameters**				
Test ID	**Pattern**	**Pitch**	`batch length`	`hidden size`	`num epochs`	`nfold splits`	`optimiser`
s1t001a	class	raw	8	128	25	10	`sgd`
s1t001b	class	interval	8	128	25	10	`sgd`
s1t001c	bounds	raw	8	128	25	10	`sgd`
s1t001d	bounds	interval	8	128	25	10	`sgd`
s1t002a	class	raw	16	128	25	10	`sgd`
s1t002b	class	interval	16	128	25	10	`sgd`
s1t002c	bounds	raw	16	128	25	10	`sgd`
s1t002d	bounds	interval	16	128	25	10	`sgd`
s1t003a	class	raw	24	128	25	10	`sgd`
s1t003b	class	interval	24	128	25	10	`sgd`
s1t003c	bounds	raw	24	128	25	10	`sgd`
s1t003d	bounds	interval	24	128	25	10	`sgd`

TABLE 4.2: Table of tests for Stage 1, with parameter values determined by Stage 0. Highlighted values are those that change between tests.

The same set of tests shown in Table 4.2 is repeated for each stage, albeit with different test IDs based on the stage. Due to the differences in pattern length and style between each stage, it made sense to run the full series of batch length and format combinations on each stage. For Stage 2 and 3 tests, the training is carried out on the MTC-ANN dataset described in Section 4.2.3. In stage 2, motifs from this dataset are randomly combined into a set of 100 files on which training is carried out. In stage 3, the raw unchanged files from the original dataset are used instead.

Chapter 5
Results

In this chapter, we will show the test results of the training runs described in the previous chapter. Section 5.2 will show the results of the initial pattern sweep on the simple dataset created for Stage 0 as described in Section 4.2.1. Section 5.3 will show the results of pattern discovery on generated data as described in Section 4.2.2. Section 5.4 will show the results of synthetically joined MTC-ANN motifs, as described in Section 4.2.3, and Section 5.5 will finally look at the performance of the network on unchanged realistic music taken directly from the MTC-ANN dataset.

Results in this chapter will be analysed in two parts, just as the training runs described in the previous chapter have two parts. For the first part (training) we look at the change in *validation categorical accuracy* over each epoch. *Categorical accuracy* is the percentage of actual output that is correct when compared to the expected output. In other words, each output that is correctly assigned to the correct pattern class or pattern boundary increases the categorical accuracy, and vice versa if incorrectly assigned. To avoid over-fitting, we look at categorical accuracy on the validation set of each fold, and average that result over the number of folds (cross-validation). Results for categorical accuracy will be represented as a line graph showing the progress of the categorical accuracy with each epoch.

The second part of results comes from the testing phase of each run. This is in the form of the *performance metrics* described in Section 4.1.2: precision, recall, and F1-score. When training is complete, a test dataset is passed into the learned model in batches, and the model attempts to predict a correct output for each batch based on what it has learned. The predicted output is compared to the expected output to generate confusion matrices, as described in Section 4.1.2. Precision and recall are calculated from the confusion matrices, and F1-score is calculated from the precision and recall. Since this is a non-binary classification problem we get performance metrics for each class, and since we have multiple folds in each run we get performance metrics for each class for each fold. The performance metrics of each class are averaged over each fold to get a single performance value for each metric per class. For example, if there are 10 folds, we will get 10 values for precision for Class A, which can be averaged to get a single precision value for Class A over the 10 folds. The average values for each class are then averaged again to get a single value for the overall performance of the model. In other words, if a model results in an average precision value of 0.4 for Class A (derived by averaging the precision on Class A for each fold in that run), an average precision value of 0.5 for Class B, and an average precision value of 0.7 for Class C, the total average precision of that model is 0.5. The same approach is used to calculate the average recall and F1-score.

5.1 Baseline Metrics

To properly gauge performance of each training run, the performance metric results should be compared against a set of baseline performance metrics. In this project, the baseline is determined through *random guessing*. For this form of prediction, the predicted output of each batch is simply a random guess from the pattern dictionary, rather than an educated guess based on previous training. In other words, if there are three possible output classes, the predicted output will be a random guess out of those three possibilities.

If the occurrences of the available classes are uniformly balanced, a random guessing approach should in theory result in a balanced spread of correct guesses between the number of classes available. In other words, if there are three possible output classes, the expected F1-score of random guessing is ∼33.3% (with precision and recall both averaging to the same value across classes), whereas if there are five possible output classes, the expected F1-score of random guessing is ∼20.0%, and so on. As a general rule of thumb, the expected F1-score is $\frac{1}{NumberofOutputClasses}$.

Note that this is only the case if the occurrences are balanced between the possible output classes. This is notably not the case when using the boundary representation (as described in Section 4.2.5). In this representation, `pattern_continue` is a far more common occurrence than `pattern_start` or `pattern_end`. The reason for this is that, given a pattern of fixed length, only a single slot at the start and end of the sequence is marked as `pattern_start` and `pattern_end` respectively. This means the remainder of the pattern is marked as `pattern_continue`. The longer the average length of patterns, the more the output skews towards `pattern_continue`. In this case, random guessing will have a higher precision rate for `pattern_continue` occurrences, and a very low precision rate for `pattern_start` and `pattern_end`. This discrepancy will cause the F1-score to average lower than ∼33.3% since the spread of output "classes" is not uniform. By looking at the resulting performance metrics of random guessing, we can therefore get a good idea of how uniformly balanced the dataset is between output classes by comparing the resulting F1-score against the expected.

5.2 Pattern Discovery on Simple Data

The preliminary stage of training runs is carried out on simple data, as described in Section 4.2.1. Table 5.1 shows the baseline precision, recall, and F1-score of this stage. With 3 classes, the average value of each is roughly at 0.333, which is the expected value for a uniform spread on 3 possible values (each class is getting roughly one third of the guesses, or ∼33.3%). Individual average metrics per class confirm this, showing a relatively even spread across the three classes.

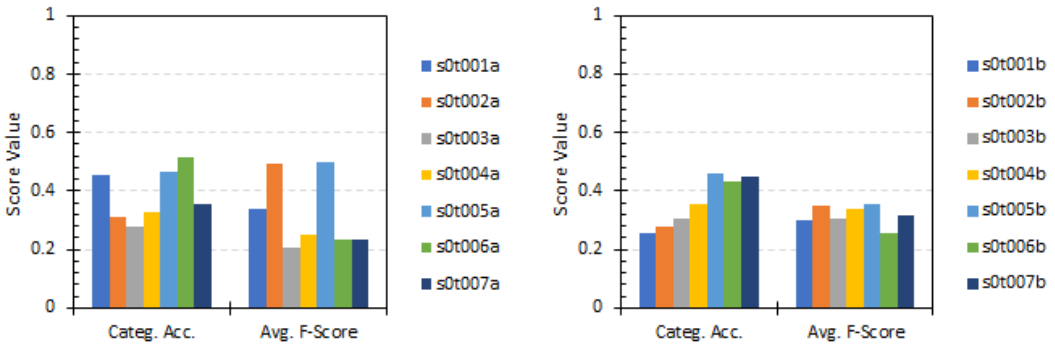
In this stage, only the `Class` representation is used (not the `Boundary` representation), so only one baseline is required. With this baseline established, we can compare the overall performance of each training run as listed in the tables in Section 4.3 with the performance of the baseline. By doing so, we can determine the set of parameters that performs more optimally and track performance with parameter changes.

Dataset	Average Metrics		
	Precision	**Recall**	**F1-score**
Simple Data (Stage 0)	0.332929	0.333210	0.331218
Class `pat1`	0.311676	0.332310	0.321663
Class `pat2`	0.287113	0.336465	0.309836
Class `pat3`	0.399997	0.330856	0.362156

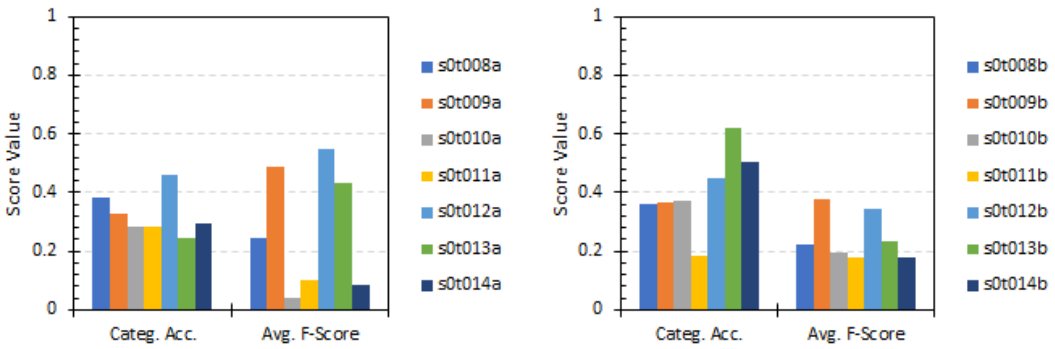
TABLE 5.1: Table of baseline performance metrics using `class` representation in Stage 0

5.2.1 Overview of Results

This section provides a quick overview and summary of the detailed results given in the next few sections. We observe gradual improvement over the series of tests for both optimisers by iteratively selecting the parameter values that perform best at each step. The parameter value with the best performance in terms of training accuracy over time and in terms of the performance metrics discussed in Section 4.1.2 is carried forward into following tests. Figure 5.1a shows the categorical accuracy and the average F1-score for each test during the `adam` optimiser tuning, and Figure 5.1b shows the categorical accuracy and average F1-score for each test during the `sgd` optimiser tuning.



(A) Using the `adam` optimiser.



(B) Using the `sgd` optimiser.

FIGURE 5.1: Graphs showing the categorical accuracy and average F1-score for the Stage 0 series of tests with `class/raw` representation (left) and `class/interval` representation (right).

In both series of tests, looking at a combination of performance over time during training, and stability in performance metrics during evaluation, we arrive to the same optimal parameter combination: a hidden size of 128 over 25 epochs, with a 10 n-fold cross-validation split, as used in tests `s0t005a/b` and `s0t012a/b` for the `adam` optimiser and `sgd` optimiser respectively. With the `adam` optimiser, this combination yields an average F1-score of 0.4966 when using the `class/raw` representation, and an average F1-score of 0.3561 when using the `class/interval` representation. With the `sgd` optimiser, this combination yields an average F1-score of 0.5493 when using the `class/raw` representation, and an average F1-score of 0.3454 when using the `class/interval` representation. These scores are above the baseline metrics given in Table 5.1. A comparison of performance between the two optimisers is given in Section 5.2.4.

5.2.2 Parameter Tuning: Adam Optimiser

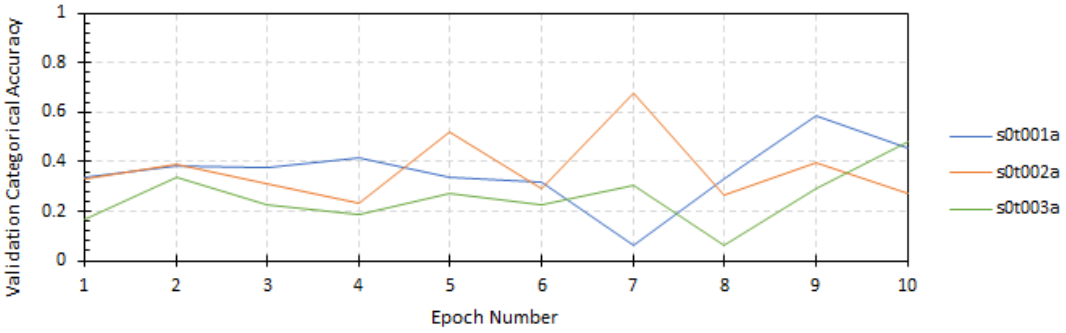
The first series of tests deals with parameter combinations involving the `adam` optimiser. These are tests 001 through 007 in the tables from Section 4.3 (test IDs: `s0t001` to `s0t007`). Each set of parameter values was tested on two pattern representations (marked as `a` and `b` respectively in the test IDs): `class/raw` (Class representation with raw pitch format), and `class/interval` (Class representation with interval pitch format).

Hidden Size

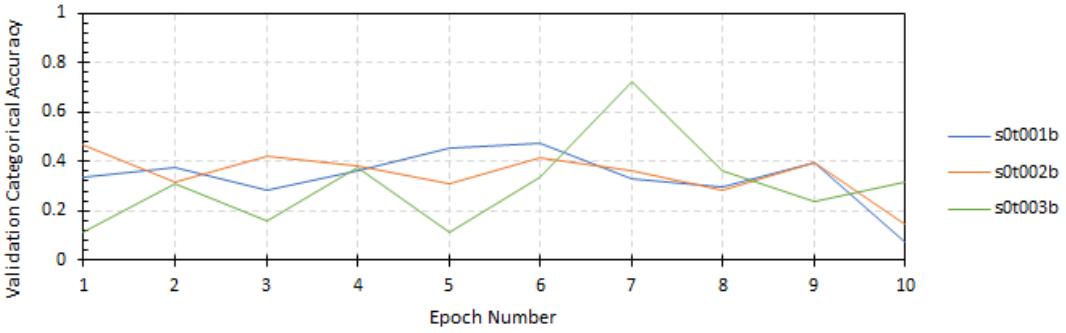
Tests 001 to 003 carry out training using a different amount of hidden units in the LSTM layers. Tests `s0t001a` and `s0t001b` use a hidden size of 512, tests `s0t002a` and `s0t002b` use a hidden size of 128, and tests `s0t003a` and `s0t003b` use a hidden size of 64. All other parameter values are set to their default values, which is to say: batch length of 8, 10 epochs per fold, and 10 n-folds.

Figures 5.2a and 5.2b show a graph of the validation categorical accuracy over 10 epochs for the three hidden size values. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

The immediate observation from these figures is that the categorical accuracy is highly inconsistent using the `Adam` optimiser. Accuracy seems to swing wildly without settling on a particular value or gradually climbing/improving. Over the last 3 epochs of each training run for the `class/raw` representation, the accuracy averages to 0.4573 for 512 hidden units, 0.3113 for 128 units, and 0.2788 for 64 units. For the `class/interval` representation, the last 3 epoch average is 0.2547 for 512 hidden units, 0.2753 for 128 units, and 0.3056 for 64 units. We take an average over the last 3 epochs (instead of just the final value) due to the accuracy inconsistency between epochs. That same inconsistency however leads one to believe that the categorical accuracy in this set of tests is not the best indicator of network performance, as the accuracy could continue to shift by large margins, thereby affecting the average by similar large amounts.



(A) Using `class/raw` representation.



(B) Using `class/interval` representation.

FIGURE 5.2: Graphs of validation categorical accuracy for hidden size of 512 (blue), 128 (orange), and 64 (green).

Looking instead at the performance metrics in Figure 5.3 we see a clearer distinction in performance between the three parameter values, especially when using the `class/raw` representation.

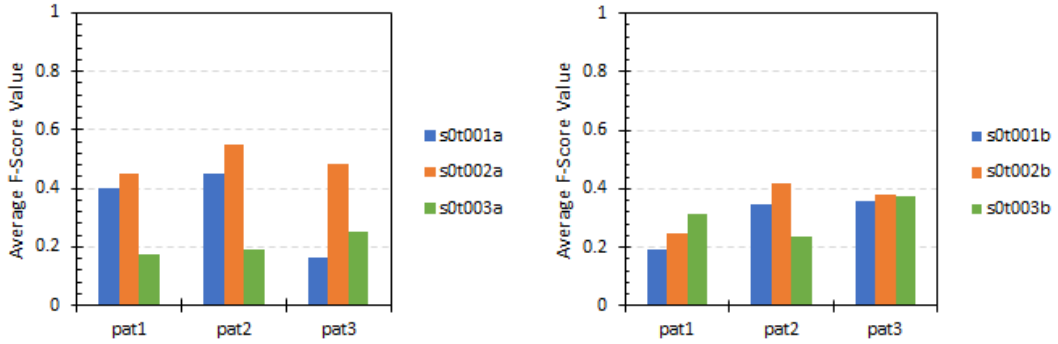


FIGURE 5.3: The average F1-score for each pattern class using `class/raw` (left) and `class/interval` (right) for hidden size of 512 (blue), 128 (orange), and 64 (green).

In the `class/raw` representation, a hidden size of 128 performs better on every class, with an average F1-score of 0.4933 across the three classes. This is in comparison to an average F1-score of 0.3376 for a hidden size of 512, and an average F1-score of 0.2069 for a hidden size of 64. Looking more in-depth at the performance metrics of test `s0t002a` in Table 5.2, we see that the precision and recall are higher than the baseline metrics across the board.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t002a`)	0.5506	0.4795	0.4933
Class `pat1`	0.3671	0.5735	0.4476
Class `pat2`	0.6561	0.4747	0.5508
Class `pat3`	0.6284	0.3901	0.4814

TABLE 5.2: Table of performance metrics for test `s0t002a`.

In the `class/interval` representation, a hidden size of 128 still performs better on average, but with a smaller margin. The average F1-score is 0.3491 across the three classes, in comparison to an average F1-score of 0.2985 for a hidden size of 512, and an average F1-score of 0.3079 for a hidden size of 64. Looking more in-depth at the performance metrics of test `s0t002b` in Table 5.3, we see that the precision is still higher than the baseline metrics for all classes, but the recall is lower than the baseline for Classes `pat1` and `pat3`.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t002b`)	0.3944	0.3355	0.3491
Class `pat1`	0.3326	0.1951	0.2459
Class `pat2`	0.3639	0.4997	0.4211
Class `pat3`	0.4866	0.3118	0.3801

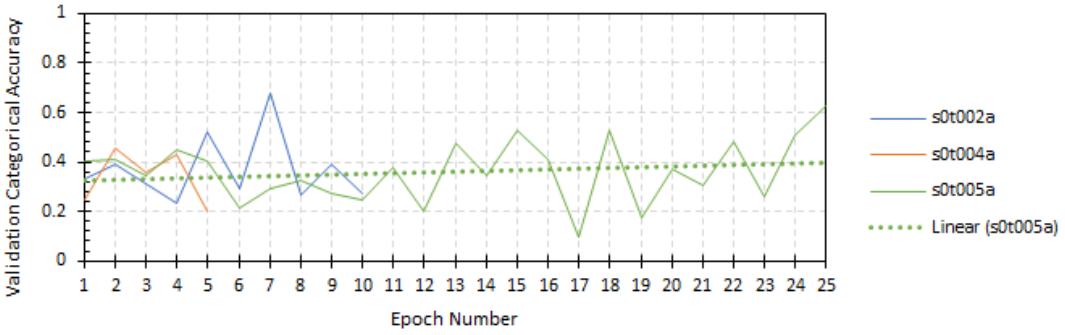
TABLE 5.3: Table of performance metrics for test `s0t002b`.

Based on the better performance metrics that a hidden size of 128 seems to offer, both in the `class/raw` and `class/interval` representation, it seems a good choice to carry forward that parameter value into the rest of the parameter sweep. The average F1-score of this parameter combination is above baseline, which is a good sign that the network is making some progress in learning sufficient information about the pattern classes to make correct predictions during testing. Naturally, this is so far only the case on the simple data generated for this stage.

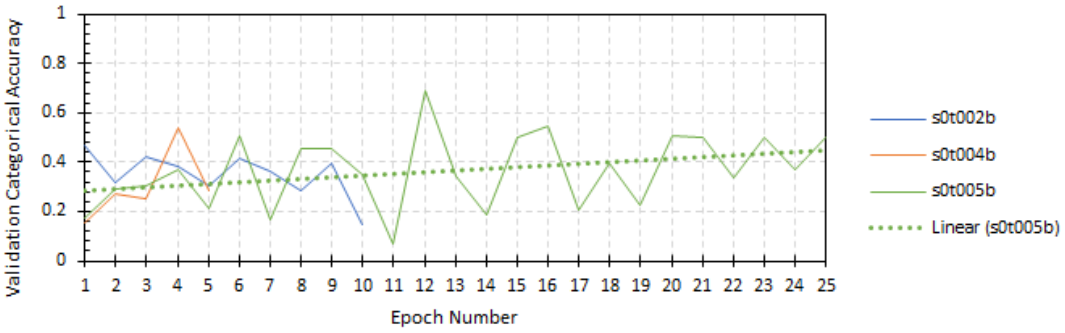
Number of Epochs

Tests 004 and 005 carry out training using a reduced and increased number of training epochs respectively. Since a hidden size of 128 was the most optimal parameter from the previous set of tests, it is carried forward, meaning that tests 002 are re-used for the parameter combination with an epoch number of 10. Tests `s0t004a` and `s0t004b` use a training length of 5 epochs, tests `s0t005a` and `s0t005b` use a training length of 25 epochs, and tests `s0t002a` and `s0t002b` are carried forward from the previous set of tests. All other parameter values are set to their default values, which is to say: batch length of 8, and 10 n-folds.

Figures 5.4a and 5.4b show a graph of the validation categorical accuracy over 5, 10, and 25 epochs for the aforementioned set of tests. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).



(A) Using `class/raw` representation.



(B) Using `class/interval` representation.

FIGURE 5.4: Graphs of validation categorical accuracy for 10 epochs 512 (blue), 5 epochs (orange), and 25 epochs (green).

Once again, the categorical accuracy is observed to be highly inconsistent even over a larger number of epochs, indicating that the `Adam` optimiser is struggling to converge to a single optimal learned model. However, looking at the average trend over time (the dotted line in Figures 5.4a and 5.4b), it is clear that the average categorical accuracy is climbing over time. Over the last 3 epochs of the `s0t005a` training run (`class/raw` representation with 25 epochs), the categorical accuracy averages to 0.4663. For the `s0t005b` training run (`class/interval` representation with 25 epochs), the average over the last 3 epochs is 0.4575. Both cases are an improvement over the average accuracy achieved by tests `s0t002a` and `s0t002b` (training with 10 epochs carried over from previous tests), which achieved an average categorical accuracy of 0.3113 and 0.2753 respectively over the last 3 epochs. This improvement suggests that, despite the inconsistent swings caused by the `Adam` optimiser, improvement is definitely occurring over an increasing number of epochs.

Looking at the performance metrics for these tests in Figure 5.5 we see that training over 25 epochs achieves comparable performance values across the 3 classes to the 10 epoch training, and in fact achieves a marginally higher average F1-score in both the `class/raw` and `class/interval` representations.

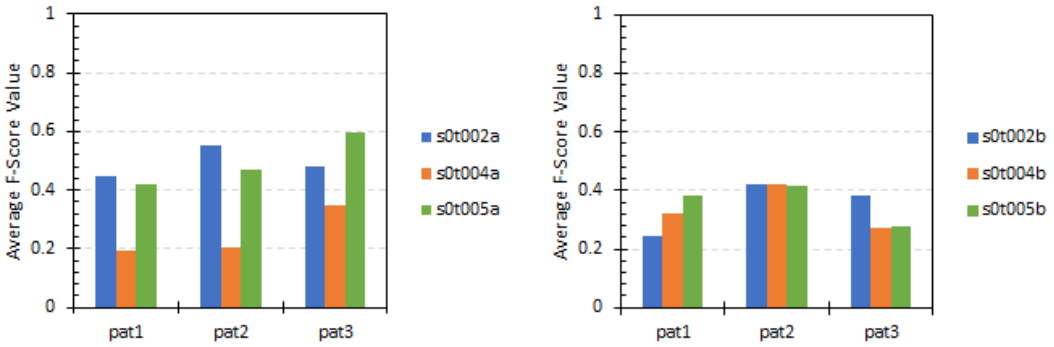


FIGURE 5.5: The average F1-score for each pattern class using `class/raw` (left) and `class/interval` (right) for 10 epochs (blue), 5 epochs (orange), and 25 epochs (green).

In the `class/raw` representation, training over 25 epochs yields an average F1-score of 0.4966 as shown in Table 5.4, in comparison to the average F1-score achieved during training over 10 epochs of 0.4933 as shown previously in Table 5.2. This improvement is largely due to an increase in recall values when compared to a lesser number of epochs, however this improvement is only marginal.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t005a`)	0.4790	0.5455	0.4966
Class `pat1`	0.3241	0.6079	0.4228
Class `pat2`	0.4642	0.4747	0.4694
Class `pat3`	0.6488	0.5541	0.5977

TABLE 5.4: Table of performance metrics for test `s0t005a`.

Similarly, in the `class/interval` representation, training over 25 epochs yields an average F1-score of 0.3561 as shown in Table 5.5, in comparison to the average F1-score achieved during training over 10 epochs of 0.3491 as shown previously in Table 5.3. Once again, the improvement is only marginal and largely due to improvements in recall values across the three classes.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t005b`)	0.3048	0.4320	0.3561
Class `pat1`	0.3067	0.4986	0.3798
Class `pat2`	0.3527	0.4975	0.4128
Class `pat3`	0.2550	0.3333	0.2758

TABLE 5.5: Table of performance metrics for test `s0t005b`.

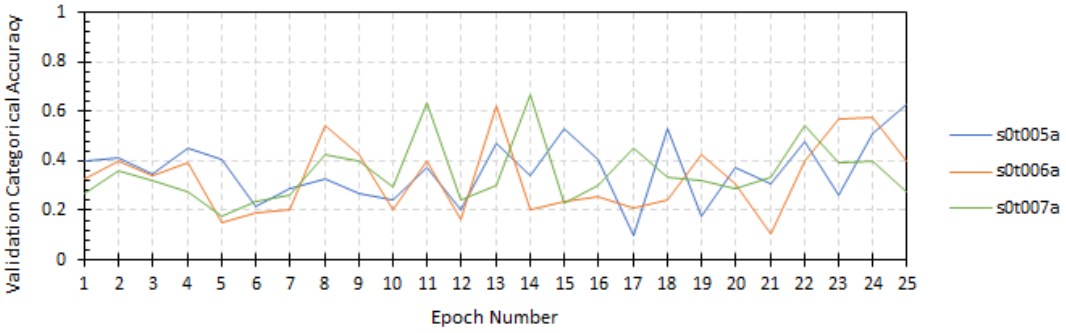
An important note about the improvements in performance metrics is that, while the improvements are only marginal in comparison to previous tests, the actual values are still sufficiently higher than baseline metrics.

Given the increasing improvement in categorical accuracy that training over 25 epochs seems to provide, as well as the marginal improvement in performance metrics, an epoch number of 25 will be carried forward into following tests. The increasing average trend suggests that further larger numbers of epochs would also be beneficial, but

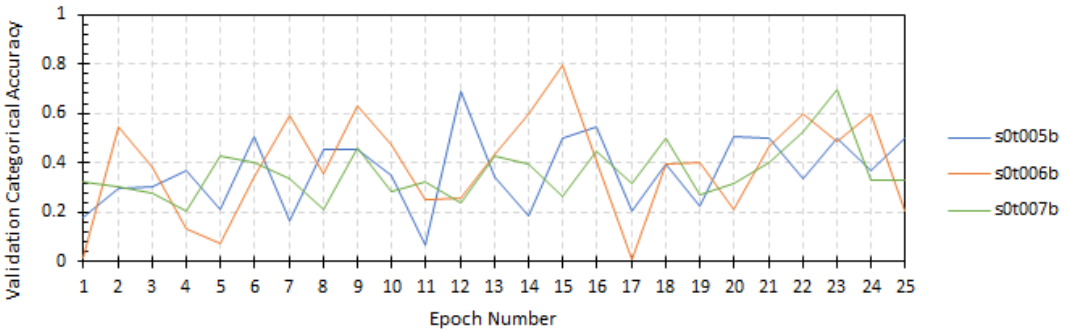
due largely to time constraints we will stick with a feasible value of 25. While higher numbers of epochs offer potentially better performance, they also increase training time exponentially. For example, 25 further epochs (to a total of 50), means that each n-fold must train for 25 epochs extra. With 10 n-folds this means a total of 250 extra epochs, which means a significant increase in training time.

Number of Cross-Validation Folds

Tests 006 and 007 carry out training using a reduced and increased number of n-folds respectively. The number of epochs determined by tests 005 in the previous set of tests is carried forward to this set of tests, meaning that all tests are now carried out using 25 epochs of training. Tests `s0t006a` and `s0t006b` use a cross-validation split of 5 n-folds, tests `s0t007a` and `s0t007b` use a split of 15 n-folds, and tests `s0t005a` and `s0t005b` are carried forward from the previous set of tests, using 10 n-folds. All other parameter values are set to their default values or the values determined by the previous parameter sweep, which is to say: batch length of 8, and a hidden size of 128.



(A) Using `class/raw` representation.

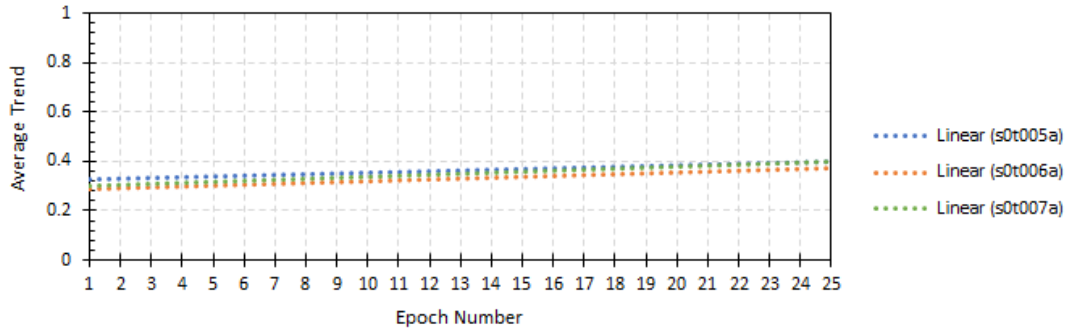


(B) Using `class/interval` representation.

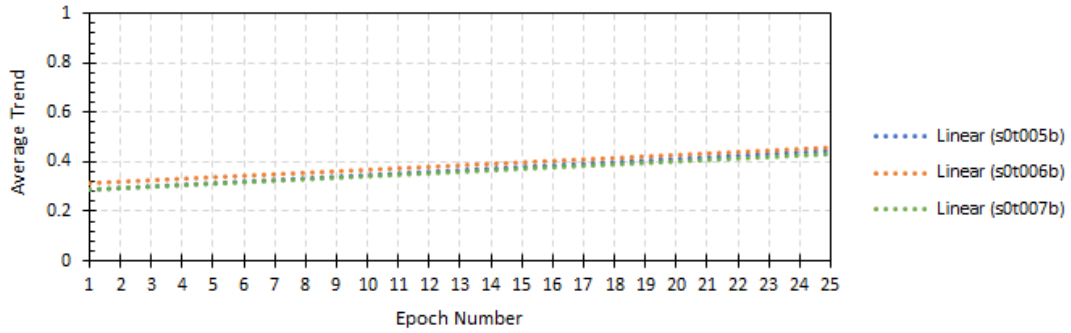
FIGURE 5.6: Graphs of validation categorical accuracy over 25 epochs with 10 n-folds (blue), 5 n-folds (orange), and 15 n-folds (green).

Figures 5.6a and 5.6b show a graph of the validation categorical accuracy over 25 epochs for the aforementioned numbers of n-folds. The value of the categorical accuracy at each epoch is an average over the 5, 10, or 15 n-folds used in the respective test.

The inconsistency of the `Adam` optimiser can make it difficult to get a clear idea of improvement (or lack thereof) in categorical accuracy from these figures. As such, to get a clearer idea of progress without the clutter of the categorical accuracy graph lines, Figures 5.7a and 5.7b show the average trends for this set of tests plotted independently of the main graph.



(A) Using `class/raw` representation.



(B) Using `class/interval` representation.

FIGURE 5.7: Graphs showing the average trend-line of categorical accuracy over 25 epochs with 10 n-folds (blue), 5 n-folds (orange), and 15 n-folds (green).

From these two figures we observe that the average categorical accuracy increases over time almost identically across the set of tests in both representations. This suggests that the number of n-folds has little to no effect on the improvements in categorical accuracy over time, whether positively or negatively. In other words, changing the number of n-folds that the input training set is split into does not seem to change the learned improvement of the system over 25 epochs.

However, looking at the performance metrics for these tests in Figure 5.8 we observe that training with 10 n-folds yields markedly better results than training with 5 or 15 n-folds in both the `class/raw` and `class/interval` representations.

In the `class/raw` representation, training with 10 n-folds yields an average F1-score of 0.4966 as shown previously in Table 5.4. In comparison, the average F1-score with 5 n-folds is 0.2361, and the average F1-score with 15 n-folds is 0.2924. In the `class/interval` representation, training with 10 n-folds yields an average F1-score of 0.3561 as shown previously in Table 5.5. In comparison, the average F1-score with 5 n-folds is 0.2584, and the average F1-score with 15 n-folds is 0.3191.

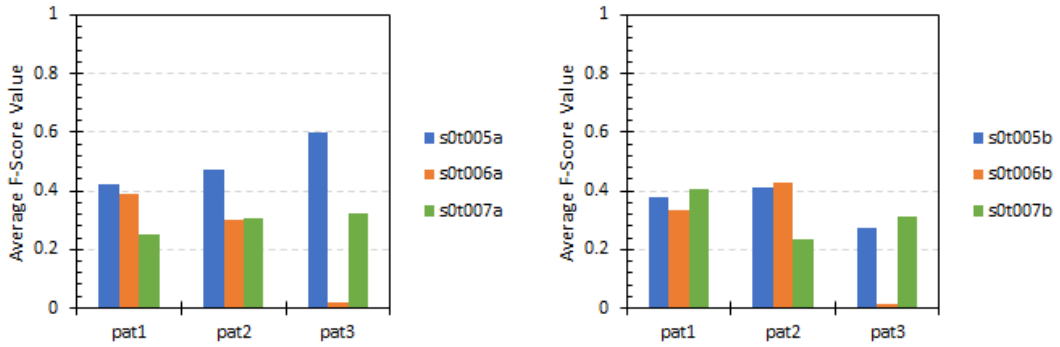


FIGURE 5.8: The average F1-score for each pattern class using `class/raw` (left) and `class/interval` (right) with 10 n-folds (blue), 5 n-folds (orange), and 15 n-folds (green).

We observe that decreasing the number of n-folds leads to skews where 2 out of 3 classes perform above baseline metrics, while the remaining class drops heavily in performance. An example of this is demonstrated in Table 5.6. This effect could be caused by an imbalance between class occurrences in the decreased number of folds. On the other hand, we observe that increasing the number of n-folds decreases the overall performance of the system across all classes. One possible reason for this is that a smaller validation set (which guides the learning of the network model) does not provide enough information for the network model to converge towards, causing performance to suffer.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t006a`)	0.2231	0.2828	0.2361
Class `pat1`	0.3745	0.4000	0.3869
Class `pat2`	0.2313	0.4373	0.3026
Class `pat3`	0.0633	0.0111	0.0189

TABLE 5.6: Table of performance metrics for test `s0t006a`.

The results of these performance metrics suggest that keeping a n-fold split of 10 is better for overall performance when using the tuned set of parameters determined in the overall set of training runs. In theory, the number of n-folds should change based on the size of the dataset in order to find a number of folds that properly represents the variation of the underlying distribution of data. However, the datasets used in the 3 stages of this project all have similar lengths and similarly uniform spreads of class occurrences, and should therefore be affected similarly by the number of n-folds.

In summary, the optimal parameter combination from those tested when using the `Adam` optimiser seems to be the one used in tests `s0t005a` and `s0t005b`: a hidden size of 128 over 25 epochs, with a 10 n-fold cross-validation split. This combination yields an average F1-score of 0.4966 when using the `class/raw` representation, and an average F1-score of 0.3561 when using the `class/interval` representation. Both scores are above the baseline metrics given in Table 5.1.

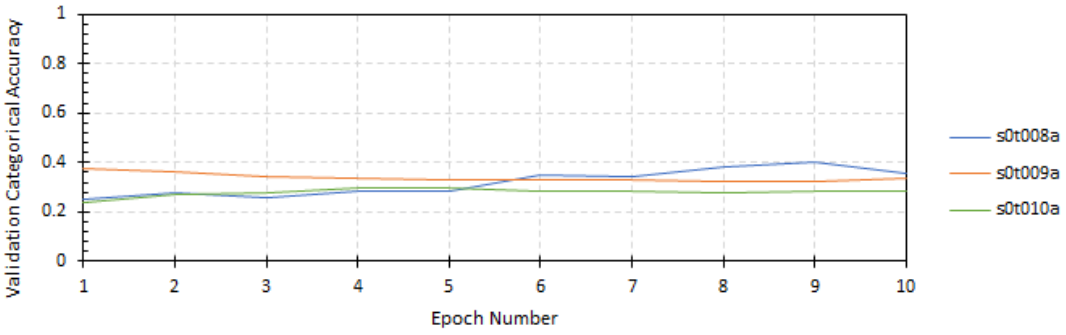
5.2.3 Parameter Tuning: SGD Optimiser

The second series of tests deals with parameter combinations involving the `sgd` optimiser. These are tests 008 through 014 in the tables from Section 4.3 (test IDs: `s0t008` to `s0t014`). Like the previous series of tests, each set of parameter values was tested on two pattern representations (marked as `a` and `b` respectively in the test IDs): `class/raw` (Class representation with raw pitch format), and `class/interval` (Class representation with interval pitch format).

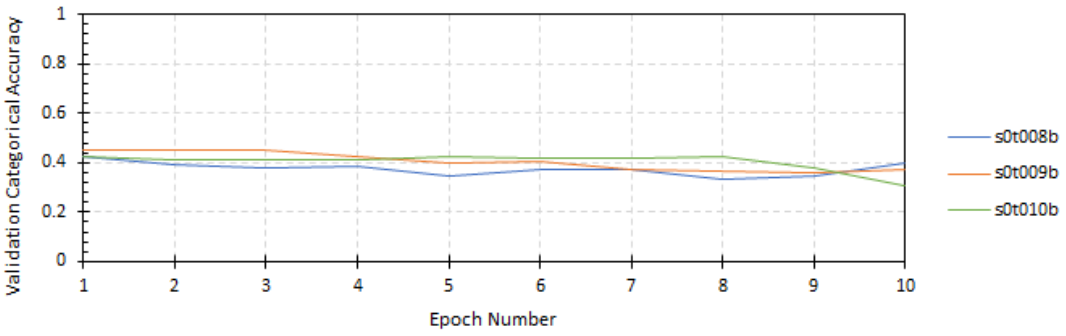
Hidden Size

Tests 009 to 010 carry out training using a different amount of hidden units in the LSTM layers. Tests `s0t008a` and `s0t008b` use a hidden size of 512, tests `s0t009a` and `s0t009b` use a hidden size of 128, and tests `s0t010a` and `s0t010b` use a hidden size of 64. All other parameter values are set to their default values, which is to say: batch length of 8, 10 epochs per fold, and 10 n-folds.

Figures 5.9a and 5.9b show a graph of the validation categorical accuracy over 10 epochs for the three hidden size values. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).



(A) Using `class/raw` representation.



(B) Using `class/interval` representation.

FIGURE 5.9: Graphs of validation categorical accuracy for hidden size of 512 (blue), 128 (orange), and 64 (green).

From these figures we observe that the categorical accuracy is now far more stable per epoch than it was for the `Adam` optimiser. Performance remains relatively consistent over the entire 10 epochs showing only slight improvement over time, but no sudden

dips either. Over the last 3 epochs of each training run for the `class/raw` representation, the accuracy averages to 0.38191 for 512 hidden units, 0.3270 for 128 units, and 0.2824 for 64 units. For the `class/interval` representation, the last 3 epoch average is 0.3592 for 512 hidden units, 0.3663 for 128 units, and 0.3695 for 64 units. Once again, we take an average over the last 3 epochs (instead of just the final value) in order to get an idea of the average trend of the categorical accuracy, rather an absolute value. This is because, despite the relative consistency of the accuracy over time, small variances are always a possibility and can be misleading if taken at face value.

Looking next at the performance metrics in Figure 5.10 we see a clearer distinction in performance between the three parameter values, especially when using the `class/raw` representation.

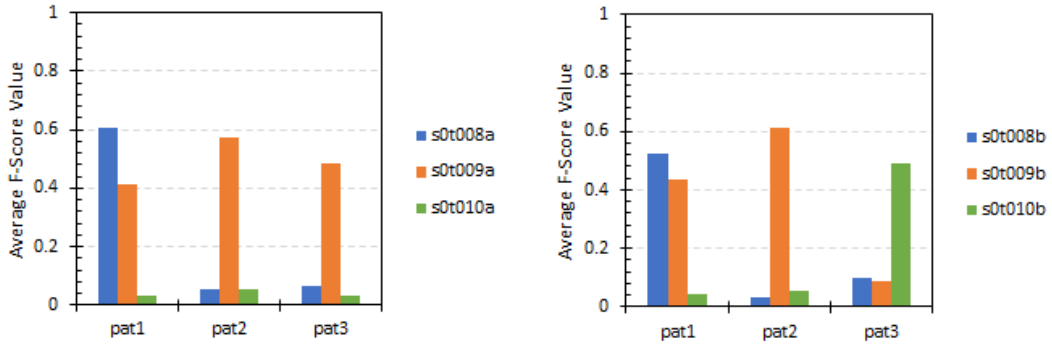


FIGURE 5.10: The average F1-score for each pattern class using `class/raw` (left) and `class/interval` (right) for hidden size of 512 (blue), 128 (orange), and 64 (green).

In the `class/raw` representation, a hidden size of 128 (test `s0t009a`) performs better on average between classes, and more importantly does not seem to skew completely towards one class or the other, as is the case with tests `s0t008a` and `s0t008b`.

The average F1-score for a hidden size of 128 is 0.4898 across the three classes. This is in comparison to an average F1-score of 0.2434 for a hidden size of 512, and an average F1-score of 0.0408 for a hidden size of 64. Looking more in-depth at the performance metrics of test `s0t009a` in Table 5.7, we see that the precision and recall are relatively high across the board, and do not skew too heavily in favour of a single class.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t009a`)	0.4663	0.5263	0.4898
Class `pat1`	0.3454	0.5155	0.4136
Class `pat2`	0.5971	0.5494	0.5723
Class `pat3`	0.4563	0.5140	0.4834

TABLE 5.7: Table of performance metrics for test `s0t009a`.

In the `class/interval` representation, we observe similar results, but in this case the tests ran with a hidden size of 128 seem to lean towards 2 out of 3 classes while neglecting the other. This means that the average F1-score still remains relatively high in the case where the test data is evenly spread between all 3 classes, but if the data were to be skewed towards the neglected class the F1-score would drop.

The average F1-score for a hidden size of 128 is 0.3789 across the three classes (in comparison to an average F1-score of 0.2202 for a hidden size of 512, and an average F1-score of 0.1971 for a hidden size of 64), but we can observe from the performance metrics of test `s0t009b` in Table 5.8, that the precision and recall are only high for classes `pat1` and `pat2`.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t009b`)	0.4034	0.4801	0.3789
Class `pat1`	0.2991	0.8192	0.4382
Class `pat2`	0.8494	0.4789	0.6124
Class `pat3`	0.0618	0.1423	0.0862

TABLE 5.8: Table of performance metrics for test `s0t009b`.

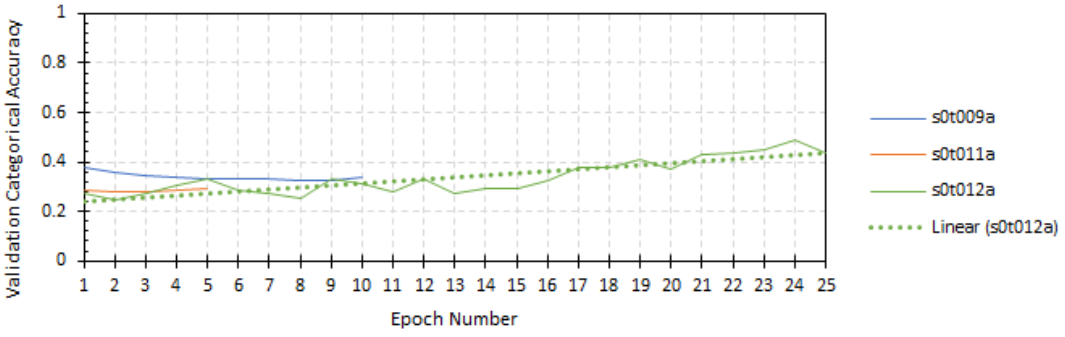
Based on the better average performance metrics that a hidden size of 128 seems to offer, both in the `class/raw` and `class/interval` representation, it seems a good choice to carry forward that parameter value into the rest of the parameter sweep. The average F1-score of this parameter combination is also above baseline metrics, which suggests that the network is making more educated predictions than random guesses.

Number of Epochs

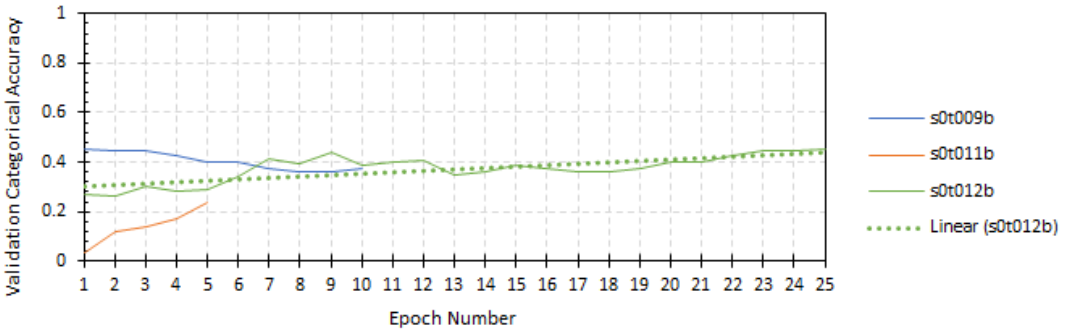
Tests 011 and 012 carry out training using a reduced and increased number of training epochs respectively. Since a hidden size of 128 was the most optimal parameter from the previous set of tests, it is carried forward, meaning that tests 009 are re-used for the parameter combination with an epoch number of 10. Tests `s0t011a` and `s0t011b` use a training length of 5 epochs, tests `s0t012a` and `s0t012b` use a training length of 25 epochs, and tests `s0t009a` and `s0t009b` are carried forward from the previous set of tests. All other parameter values are set to their default values, which is to say: batch length of 8, and 10 n-folds.

Figures 5.11a and 5.11b show a graph of the validation categorical accuracy over 5, 10, and 25 epochs for the aforementioned set of tests. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

Looking at the average trend (the dotted line in Figures 5.11a and 5.11b), we observe that the average categorical accuracy is climbing steadily over time. Over the last 3 epochs of the `s0t012a` training run (`class/raw` representation with 25 epochs), the categorical accuracy averages to 0.4587. For the `s0t012b` training run (`class/interval` representation with 25 epochs), the average over the last 3 epochs is 0.4485. Both cases are an improvement over the average accuracy achieved by tests `s0t009a` and `s0t009b` (training with 10 epochs carried over from previous tests), which achieved an average categorical accuracy of 0.3270 and 0.3663 respectively over the last 3 epochs. Note that the categorical accuracy of these tests is also significantly steadier than that shown by the `Adam` optimiser, with fewer swings in accuracy across epochs.



(A) Using `class/raw` representation.



(B) Using `class/interval` representation.

FIGURE 5.11: Graphs of validation categorical accuracy for 10 epochs 512 (blue), 5 epochs (orange), and 25 epochs (green).

From the the performance metrics for these tests in Figure 5.12 we see that training over 25 epochs achieves comparable performance values across the 3 classes to the 10 epoch training, and in fact achieves a substantially higher average F1-score in both the `class/raw` and `class/interval` representations.

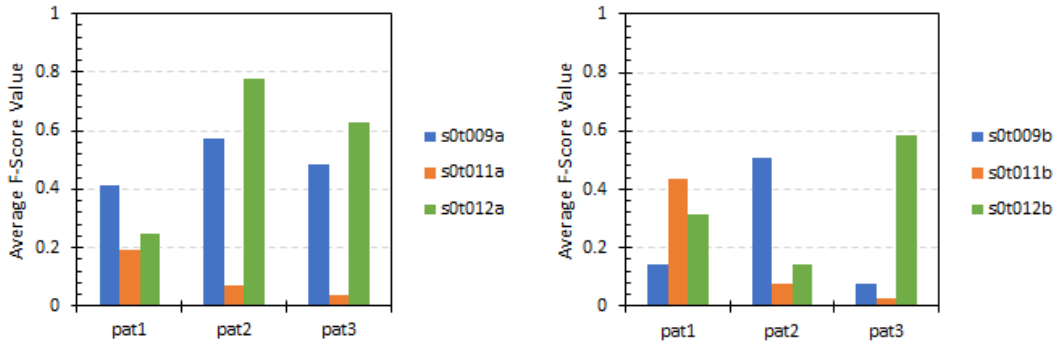


FIGURE 5.12: The average F1-score for each pattern class using `class/raw` (left) and `class/interval` (right) for 10 epochs (blue), 5 epochs (orange), and 25 epochs (green).

In the `class/raw` representation, training over 25 epochs yields an average F1-score of 0.5493 as shown in Table 5.9, in comparison to the average F1-score achieved during training over 10 epochs of 0.4898 as shown previously in Table 5.7.

Dataset	Average Metrics		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t012a`)	0.4790	0.5455	0.4966
Class `pat1`	0.3303	0.1961	0.2461
Class `pat2`	0.8975	0.6814	0.7746
Class `pat3`	0.4686	0.9482	0.6274

TABLE 5.9: Table of performance metrics for test `s0t012a`.

Similarly, in the `class/interval` representation, training over 25 epochs yields an average F1-score of 0.3454 as shown in Table 5.10, in comparison to the average F1-score achieved during training over 10 epochs of 0.2434 as shown previously in Table 5.8.

Dataset	Average Metrics		
	Precision	**Recall**	**F1-score**
Simple Data (`s0t012b`)	0.5779	0.3891	0.3454
Class `pat1`	0.8859	0.1894	0.3120
Class `pat2`	0.4157	0.0857	0.1421
Class `pat3`	0.4319	0.8922	0.5821

TABLE 5.10: Table of performance metrics for test `s0t012b`.

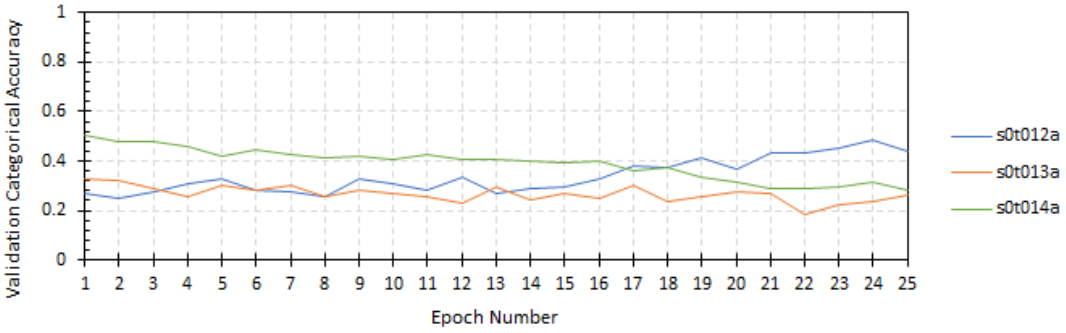
Given the increasing improvement in categorical accuracy that training over 25 epochs seems to provide, as well as the overall improvement in performance metrics, an epoch number of 25 will be carried forward into following tests. As explained previously, larger numbers of epochs would likely prove to be beneficial, as suggested by the increasing average trend, but will not be tested due to time constraints.

Number of Cross-Validation Folds

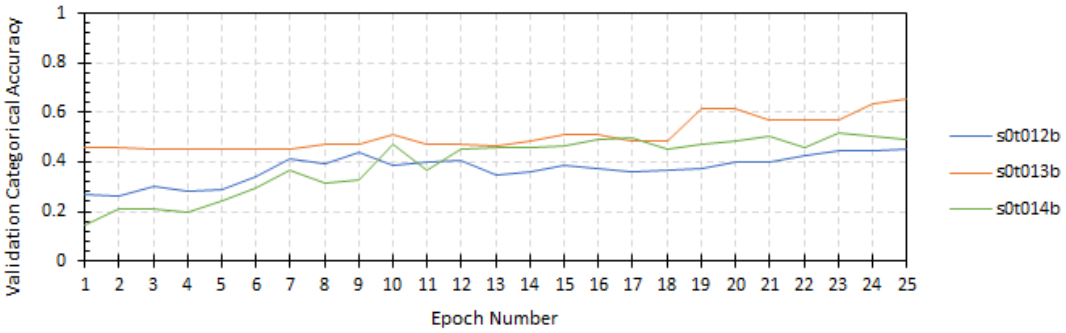
Tests 013 and 014 carry out training using a reduced and increased number of n-folds respectively. The number of epochs determined by tests 012 in the previous set of tests is carried forward to this set of tests, meaning that all tests are now carried out using 25 epochs of training. Tests `s0t013a` and `s0t013b` use a cross-validation split of 5 n-folds, tests `s0t014a` and `s0t014b` use a split of 15 n-folds, and tests `s0t012a` and `s0t012b` are carried forward from the previous set of tests, using 10 n-folds. All other parameter values are set to their default values or the values determined by the previous parameter sweep, which is to say: batch length of 8, and a hidden size of 128.

Figures 5.13a and 5.13b show a graph of the validation categorical accuracy over 25 epochs for the aforementioned numbers of n-folds. The value of the categorical accuracy at each epoch is an average over the 5, 10, or 15 n-folds used in the respective test.

From these figures we observe that changing the number of n-folds has a marked effect on the categorical accuracy of the network. Notably, in the `class/raw` representation, accuracy seems to actually drop over time when changing away from 10 n-folds. Why this is the case is not clear, but could have to do with the problems mentioned in Section 4.1.3 regarding lowering or increasing the number of n-folds by too high a margin.



(A) Using `class/raw` representation.



(B) Using `class/interval` representation.

FIGURE 5.13: Graphs of validation categorical accuracy over 25 epochs with 10 n-folds (blue), 5 n-folds (orange), and 15 n-folds (green).

However, looking at the performance metrics for these tests in Figure 5.14 we observe that training with 10 n-folds yields better results on average than training with 5 or 15 n-folds in both the `class/raw` and `class/interval` representations.

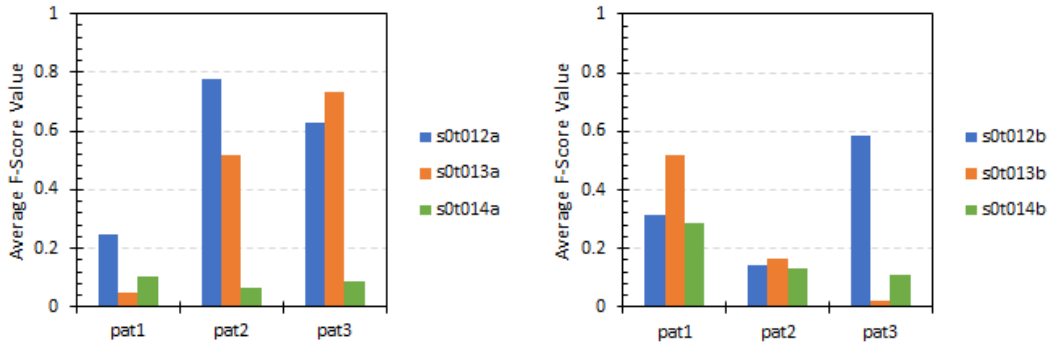


FIGURE 5.14: The average F1-score for each pattern class using `class/raw` (left) and `class/interval` (right) with 10 n-folds (blue), 5 n-folds (orange), and 15 n-folds (green).

In the `class/raw` representation, training with 10 n-folds yields an average F1-score of 0.5493 as shown previously in Table 5.9. In comparison, the average F1-score with 5 n-folds is 0.4346, and the average F1-score with 15 n-folds is 0.0842. In the `class/interval` representation, training with 10 n-folds yields an average F1-score of 0.3454 as shown previously in Table 5.10. In comparison, the average F1-score with 5 n-folds is 0.2347, and the average F1-score with 15 n-folds is 0.1766. The results of these performance

metrics suggest that keeping a n-fold split of 10 is better for overall performance when using the tuned set of parameters determined in previous training runs.

In summary, the optimal parameter combination from those tested when using the `SGD` optimiser seems to be the one used in tests `s0t012a` and `s0t012b`: a hidden size of 128 over 25 epochs, with a 10 n-fold cross-validation split. This combination yields an average F1-score of 0.5493 when using the `class/raw` representation, and an average F1-score of 0.3454 when using the `class/interval` representation. Both scores are above the baseline metrics given in Table 5.1.

5.2.4 SGD vs Adam

Based on the results from the two batches of tests discussed in the previous sections, there are a few things we can conclude regarding which optimiser to pick moving forward into the later stages. Firstly, the optimal parameter combination from the values tested during these two batches of tests is common between the two optimisers, suggesting that this combination of values is well-suited to the task. However, given the same optimal parameter combination, the `SGD` optimiser outperformed the `Adam` optimiser with an average F1-score of 0.5493 against 0.4966 when using the `class/interval` representation. When using the `class/interval` representation, the two optimisers yielded comparable results, with an average F1-score of 0.3454 using `SGD` against 0.3561 using `Adam`.

The most significant difference between the two optimisers is in the stability of the learning process. In the set of tests using the `Adam` optimiser, the categorical accuracy always varied by significant amounts between every successive epoch. This was not the case when using the `SGD` optimiser. Based on this, we determine that the `SGD` optimiser is likely a safer approach for stability during future training on more complex data.

Another observation of note is that tests using the `Adam` optimiser seemed to maintain a relatively balanced performance across the three classes. In other words, the precision and recall (and by extension the F1-score) remained close to each other and did not skew to one or two of the classes at the expense of the rest. The opposite is true for those tests that used the `SGD` optimiser. In this case, performance metrics seemed to rise to very high values on two of the classes, while dropping drastically for the third. This is particularly noticeable in Figure 5.14 for example, where the average F1-score for Class `pat1` was much lower than the average F1-score for Classes `pat2` and `pat3`. Despite this drop in performance for one of the classes, the overall average F1-score across the three classes still remained higher for the the `SGD` optimiser when compared to the `Adam` optimiser, making it the better choice moving forward.

5.3 Pattern Discovery on Generated Data

The first stage of training following the preliminary parameter tuning in the previous stage is carried out on data generated from simple musical structures, as described in Section 4.2.2. Tables 5.11 and 5.12 show the baseline precision, recall, and F1-score of this stage. In this stage of the testing, the pattern boundary representation is introduced, so two baselines must be established (one for each representation). In the pattern boundary representation, the expected output skews towards `pattern_continue`, as it is a far more common occurrence in a given dataset, as explained in Section 5.1.

This stage of training and testing will utilise the parameter combinations determined by the previous stage throughout the tests in Section 5.2. The `SGD` optimiser will be used as per the explanation given at the end of Section 5.2, and the parameter setup consists of a hidden size of 128 over 25 epochs, with a 10 n-fold cross-validation split. Similar to the tests in Stage 0, each training run will be carried out on different combinations of pattern representation and pitch format. With the boundary representation, this brings the total number of possible representations to four: `class/raw`, `class/interval`, `bounds/raw`, and `bounds/interval`. Peformance is compared between the four different representations using the same methods as in Section 5.2 above, as described in the introduction of this chapter.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Generated Data/Class (Stage 1)	0.3326	0.3326	0.3325
Class `ran`	0.3398	0.3346	0.3372
Class `pat2`	0.3383	0.3309	0.3345
Class `pat2`	0.3196	0.3322	0.3258

TABLE 5.11: Table of performance metrics for random guess on Class representation in Stage 1

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Generated Data/Bounds (Stage 1)	0.3334	0.3346	0.2284
pattern_continue	0.8825	0.3345	0.4851
pattern_start	0.0551	0.3325	0.0946
pattern_end	0.0626	0.3369	0.1056

TABLE 5.12: Table of performance metrics for random guess on Bounds representation in Stage 1

Furthermore, varying values for the batch length parameter will also be tested in this stage (and in future stages). The series of tests in this stage is split into three sections. Each section uses a different batch length. The first section will use a batch length of 8, the second a batch length of 16, and the third a batch length of 24. A section is made up of four tests, one for each possible representation described above. Therefore, there are a total of twelve tests in this stage, as listed in Section 4.3.

5.3.1 Overview of Results

This section provides a quick overview and summary of the detailed results given in the next few sections. We observe varying improvement between different representations and batch lengths. This stage mainly aims to answer (in part) the first research question posed in Section 1.5: *"Is an artificially-generated fixed ground truth sufficient for training pattern discovery?"*

Our findings suggest that yes, a model trained on artificially-generated data is capable of pattern discovery (to an extent) above baseline level on similar artificially-generated data. Some examples of this are given in Section 5.3.2. We observe relatively accurate predictions in class-based representations and less accurate predictions in boundary-based representations. The network model seems to perform better in a classification oriented task and manages to abstract enough information for each class to make accurate guesses on similarly structured test data. Naturally, this poses the disadvantage that new pattern classes cannot be added to the model without new training. One observation of note is that classification in class-based representations seems to improve when the batch length being used is close to the average length of patterns. Some further examples and explanation of this are given in Section 5.3.2.

Moreover, we see that classification is improved when using raw pitch formats, rather than interval formats. One possible reason for this is that interval formats are more general, in that small reoccurring interval patterns can occur even in completely different patterns. For example, the `pat2` class in the generated dataset consists of repeating fixed intervals between two exact note pitches. When using the interval format, this is represented as a sequence of identical numbers because the gap between notes remains the same. However, that same number could also occur in the `ran` class between two different notes that happen to have the same pitch gap between them. This leads to scenarios where the model struggles to distinguish between two different pattern classes because their interval sequences are similar. This is not necessarily a bad thing. For example, this can be beneficial if two pattern classes are simply transpositions of each other, or if we are running a trained model on completely unclassified data and simply want to find similar sequences. However, since the model in this series of tests is matching against a fixed ground truth such a scenario is still marked as a wrong classification.

On the other hand, we observe that interval pitch formats show performance improvements on boundary-based representations when using longer batch lengths. The reason for this could have to do with the synthetic data causing noticeable sudden jumps in the note sequence when switching from one pattern to the next. Since boundary-based representations try to find the exact boundaries of patterns (rather than classifying the entire pattern), such noticeable sudden jumps (which are represented as large values in an interval sequence) could potentially be more easily detectable. Curiously, this only seems to be the case for a batch length of 24, as in other batch lengths we see interval pitch formats struggle to match the performance of raw pitch formats.

5.3.2 Output Samples

This section will discuss some output examples from this stage of tests and compare them to the original file in order to give a concrete example of the system's performance and better demonstrate some details about the output provided by the network. Figure 5.15 shows one of the files used for testing with notes coloured in the `class` representation. In this stage, there are three possible classes. The classes are given a unique ID number (0, 1, 2) as internal representation, such that each note in the sequence is attributed with one number corresponding to the pattern class it belongs to. For example, a MIDI file such as the one in Figure 5.15 loaded in the `class` representation is internally represented as a string like so:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2
```

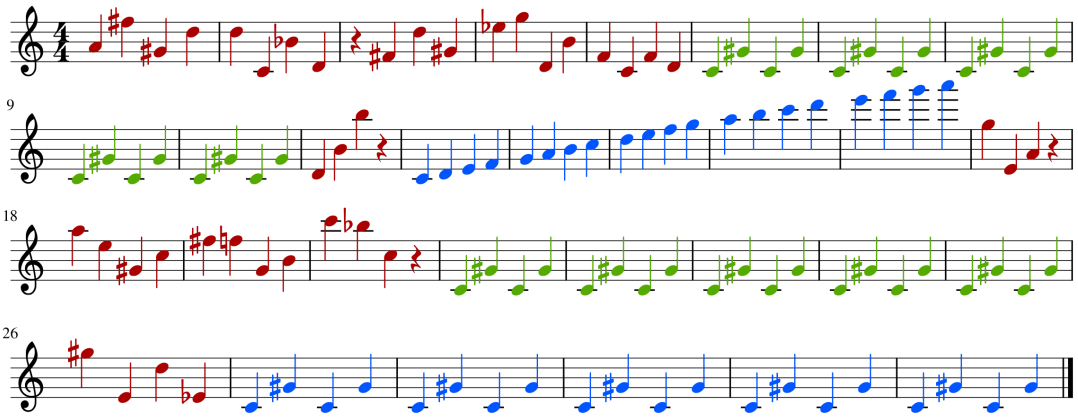


FIGURE 5.15: MIDI representation of the expected output where red notes are of class `Ran`, blue notes are of class `pat1`, and green notes are of class `pat2`.

The output that the system provides is then generated in the same format. The network will output a string of class IDs, such as the one above, and match each one against the expected output. Note that, due to the use of an LSTM, the system requires a full batch sequence as input in order to guess the next note. The model can only make a guess based on a sequence, not a single note input. This means that the system can only start guessing from the first note after the length of the first batch. For example, if the batch length is 16, the first note of the actual output is note 17 from the whole file. To show this, notes that are not part of the system's output are greyed out in the example figures.

Figure 5.16 shows the output of test `s1t001a` on the above file. This test uses the `class/raw` representation. Since the batch length is 8, the first 8 notes are greyed out as they are not part of the output. For comparison against the above, the actual output of the model is as follows:

```
- - - - - - - - 0 0 0 0 0 0 1 1 1 1 1 2 2 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 0 0 0 0 2 0 2 2 1 2 2 2 2 2 2 2 1 0 0 0 1 1 1 1 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2
```

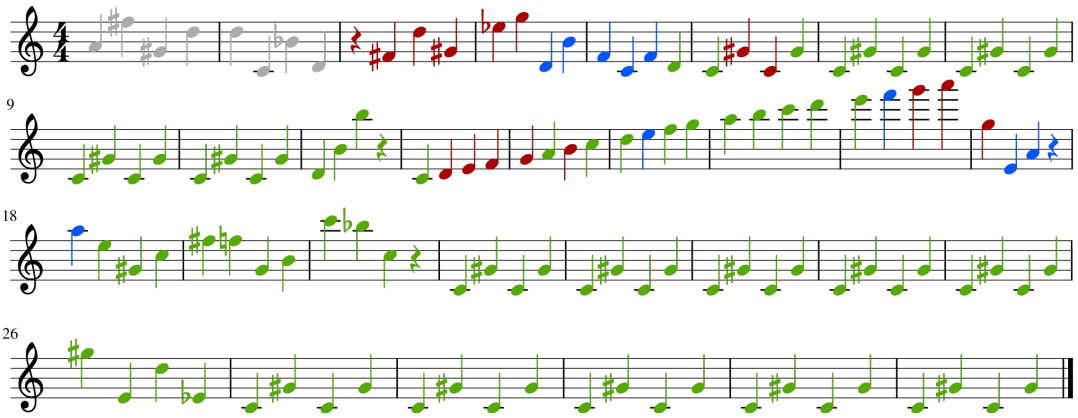


FIGURE 5.16: MIDI representation of the actual output with batch length 8 where red notes are of class `ran`, blue notes are of class `pat1`, and green notes are of class `pat2`.

Note that with this batch length the model struggles to successfully capture full patterns correctly. The reason for this is likely because the majority of patterns on which the model was trained are longer (in terms of notes) than the batch length. This means that a single batch length is unlikely to capture a full pattern in one go for training, which in turn results in a model that does not have a proper understanding of what a full pattern consists of. For perspective, this would be akin to attempting to guess the last word in a sentence without knowing the beginning and middle of that same sentence.

Figure 5.17 shows the output of test `s1t002a` on the above file. This test also uses the `class/raw` representation, but with a batch length of 16. For comparison against the above, the actual output of the model is as follows:

```
- - - - - - - - - - - - - - - - 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2
```

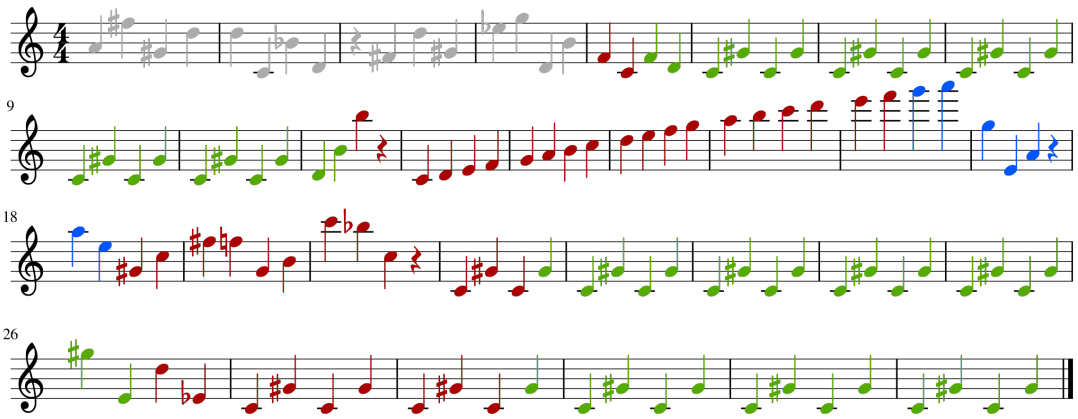


FIGURE 5.17: The MIDI representation of the actual output with batch length 16 where red notes are of class `ran`, blue notes are of class `pat1`, and green notes are of class `pat2`.

In this example the model performs much better than the previous test. The batch length of 16 is almost exactly the average length of patterns in the generated data. We observe nearly entire patterns classified correctly, bar some fuzziness in the boundaries. Note that in some cases the pattern classification bleeds into the next pattern, such that the first few notes of the following pattern are classified as part of the previous. The reason for this is once more due to the nature of LSTMs and what is essentially "inertia" in the prediction. Since an LSTM classifies a note based on the previous batch sequence of notes, if the majority of the batch sequence is made up of one pattern, the LSTM is likely to anticipate the next note to be of that same pattern. Within a few notes, it realises that the pattern has changed, and begins making different guesses once again. This is clearly apparent in bar 21 of Figure 5.17 for example, when switching from red notes to green. The first three notes in this case should be green, but carry over some delay from the previous predictions.

Finally, figure 5.18 shows the output of test `s1t003a` on the same file. The batch length is now 24, still using the `class/raw` representation. For comparison against the above, the actual output of the model is as follows:

```
- - - - - - - - - - - - - - - - - - - - - - - - 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2
```

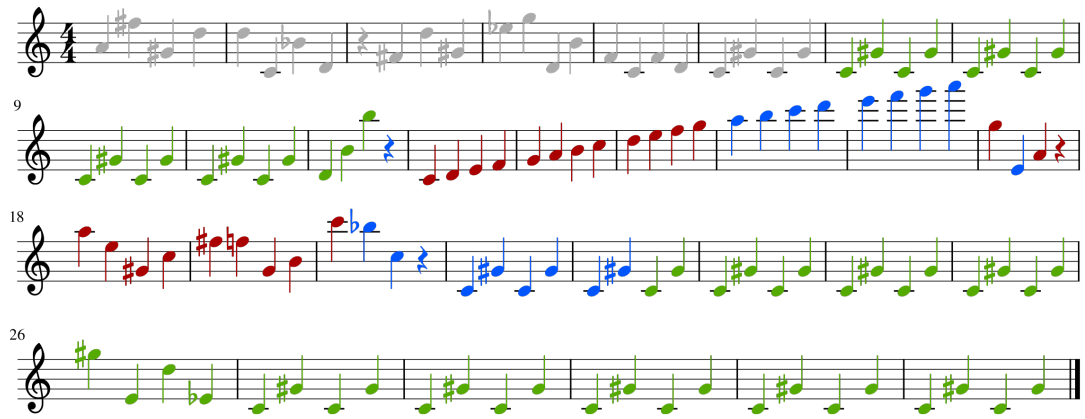


FIGURE 5.18: The MIDI representation of the actual output with batch length 24 where red notes are of class `ran`, blue notes are of class `pat1`, and green notes are of class `pat2`.

In this example we get a mixture of the two problems demonstrated earlier. On the one hand, the batch length is now long enough to fully capture all patterns. However, it is also likely to capture multiple shorter patterns in one batch, meaning that the LSTM can struggle to get a specification of one class, rather than a specification of one class with extra notes attached. Moreover, the increased batch length causes the delay in switching between patterns to increase, since it takes longer for the new pattern to overrule the previous.

Figure 5.19 shows the same files as above, but coloured in the `bounds` representation. In this class, the classes are general pattern boundary classifiers (`pattern_continue`, `pattern_start`, `pattern_end`), rather than a specific type of pattern. They are still internally represented as integer IDs, such that each note in the sequence is attributed with one number corresponding to the pattern class it belongs to. For example, a MIDI

file such as the one in Figure 5.19 loaded in the `bounds` representation is internally represented as a string like so:

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 2 1 0 0 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0 2 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2
```

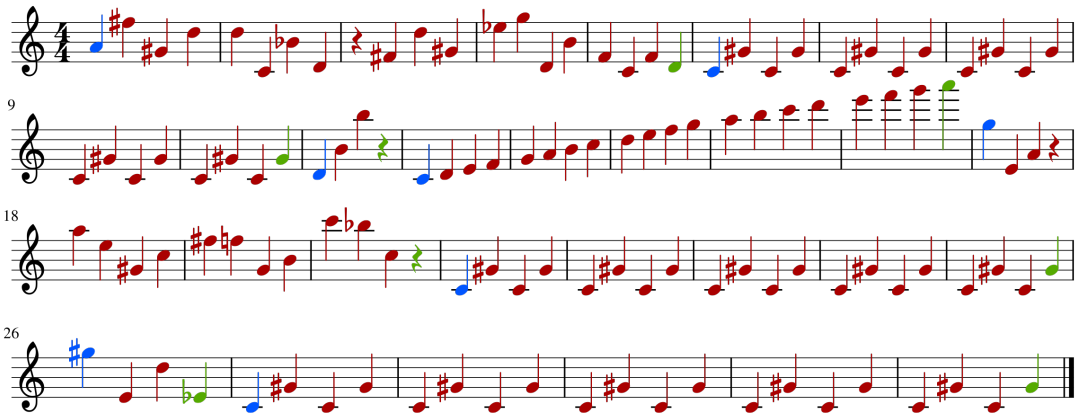


FIGURE 5.19: MIDI representation of the expected output where red notes are `pattern_continue`, blue notes are `pattern_start`, and green notes are class `pattern_end`.

Using the same output format, figure 5.20 shows the output of test `s1t001c` on the above file. This test uses the `bounds/raw` representation. Since the batch length is 8, the first 8 notes are greyed out as they are not part of the output. For comparison against the above, the actual output of the model is as follows:

```
- - - - - - - - 0 0 0 0 0 0 2 2 2 2 2 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0 2
2 2 2 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 1 1 0 0 0 0
0 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 2 2
```

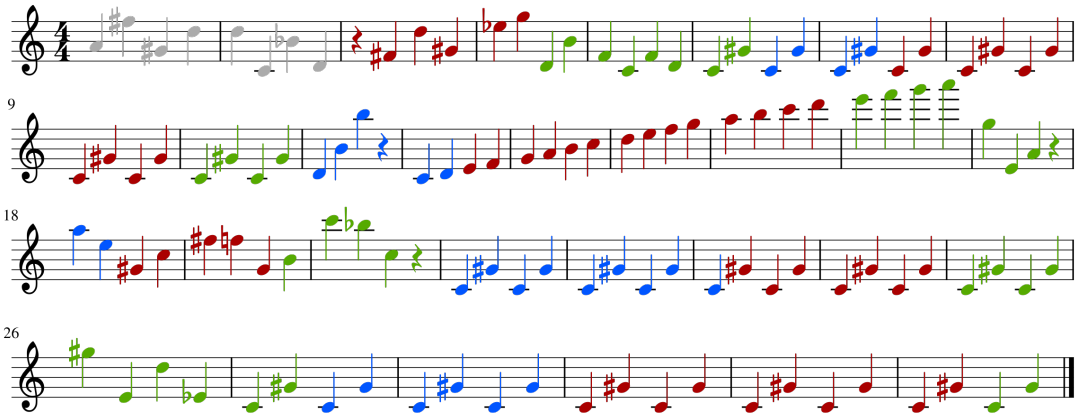


FIGURE 5.20: MIDI representation of the actual output with batch length 8 where red notes are `pattern_continue`, blue notes are `pattern_start`, and green notes are class `pattern_end`.

We observe that the model does actually seem to recognise the general neighbourhood of pattern boundaries in this representation, but struggles with pinpointing the exact note where that boundary is located. As a result, the predicted boundaries become fuzzy areas instead of a single note. The reason for this is likely that the model does not have a general definition of what a pattern start or end look like. Entirely different types of patterns can have entirely different start and end points, so the model struggles to isolate the properties that define a pattern start or end.

5.3.3 Batch Length: 8

Tests 001*a* to 001*d* carry out training using a batch length of 8 alongside the parameter combination determined in Stage 0. As explained previously, the letters in the test ID correspond to the four different pattern/pitch representations, where test `s1t001a` uses `class/raw`, test `s1t001b` uses `class/interval`, test `s1t001c` uses `bounds/raw`, and test `s1t001d` uses `bounds/interval`. Figure 5.21 shows a graph of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

We observe that the categorical accuracy of the `class/raw` representation reaches a final accuracy of 0.3435, and is higher than other representations almost throughout the entire 25 epochs. The `class/interval` representation ends at a lower value of 0.2540, the `bounds/raw` representation converges to an accuracy of 0.3127, and the `bounds/interval` representation performs similarly with an accuracy of 0.3077. It is worth nothing that switching to an `interval` pitch format when using the `class` representation takes accuracy well below the baseline performance. This is not the case when using the `bounds` representation, where the `interval` pitch format performed comparably with `raw` pitch.

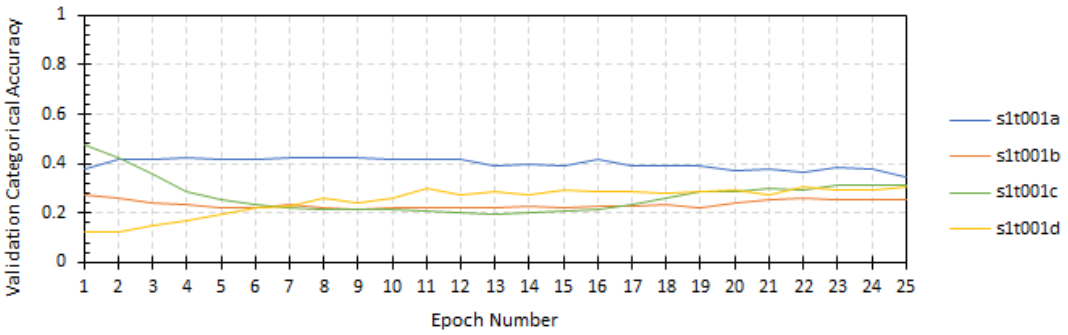


FIGURE 5.21: A graph of validation categorical accuracy on Generated Data across 25 epochs using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow) with batch length of 8.

In terms of performance metrics, the `class/raw` representation performs better in general than all other representations, with an average F1-score of 0.3854, in comparison to an average F1-score of 0.2829 when using the `class/interval` representation. In both cases, precision and recall are evenly spread across all 3 classes. Compared to baseline metrics, the `class/raw` representation performs better on average by a small margin, while the opposite is true for the `class/interval` representation.

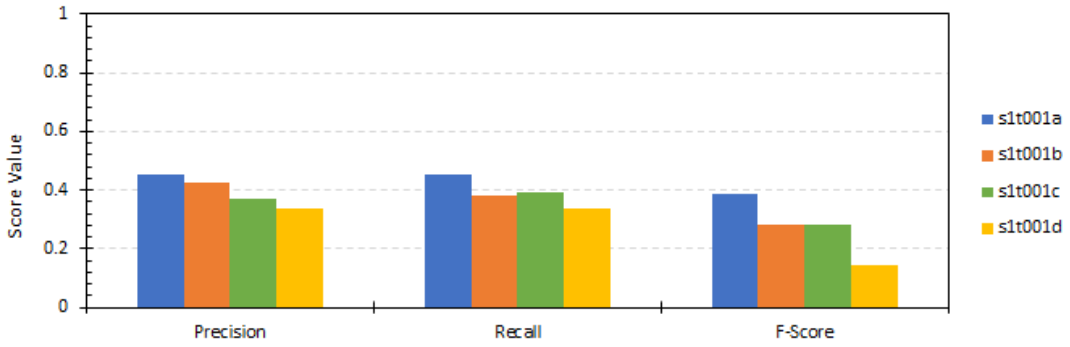


FIGURE 5.22: The average precision (left), recall (middle), and F1-score (right) on Generated Data using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow).

For the `bounds/raw` and `bounds/interval` the performance metrics are skewed (similarly to the random guessing above) towards `pattern_continue`, with very high precision for that value but very low precision for both others. As a result, the average F1-score suffers, dropping to 0.2796 for the `bounds/raw` representation, and to 0.1441 for the `bounds/interval` representation. Table 5.13 shows this skew and how it affects the averages as a result. Note that similarly to the `class` representation, using a `raw` pitch format causes the average F1-score to be above baseline metrics, while using an `interval` pitch format causes it to fall below the baseline.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Generated Data/Bounds (`s1t001c`)	0.3716	0.3935	0.2796
pattern_continue	0.9345	0.3923	0.5526
pattern_start	0.0450	0.3516	0.0798
pattern_end	0.1351	0.4364	0.2063

TABLE 5.13: Table of performance metrics for test `s1t001c`.

5.3.4 Batch Length: 16

Tests 002*a* to 002*d* carry out training using a batch length of 16 alongside the parameter combination determined in Stage 0. Test `s1t002a` uses a `class/raw` representation, test `s1t002b` uses `class/interval`, test `s1t002c` uses `bounds/raw`, and test `s1t002d` uses `bounds/interval`. Figure 5.23 shows a graph of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

With the increased batch length, we observe that all four representations see improved categorical accuracy by the end of training. The `class/raw` representation now reaches an accuracy of 0.4469 in comparison to the accuracy of 0.3435 achieved in the previous test with batch length of 8. Other representations improve by a similar amount, but all still perform worse overall than the `class/raw` representation. The `class/interval` representation reaches an accuracy of 0.3002, the `bounds/raw` representation reaches an accuracy of 0.3276, and the `bounds/interval` representation reaches an accuracy of 0.3850. It is worth noting that the `bounds` representation seems to perform better when

using an `interval` pitch format, while the opposite is true for the `class` representation. Also, more so than in the previous batch length, we see notable improvement in categorical accuracy over time. This suggests that further testing with increased epoch lengths could continue to see improvement leading to better performance.

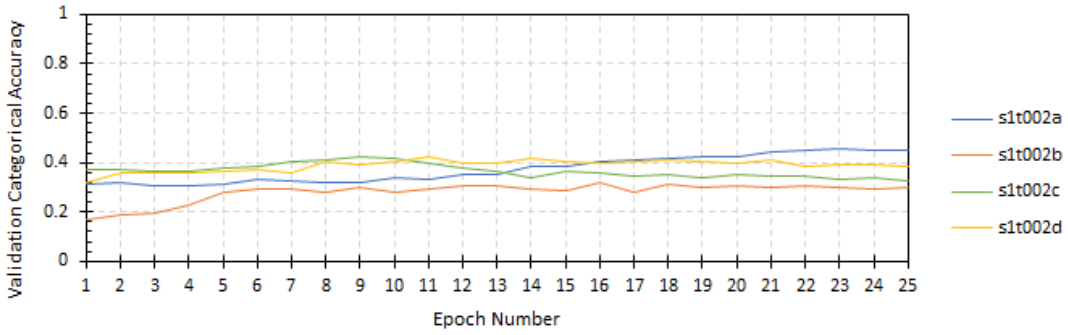


FIGURE 5.23: A graph of validation categorical accuracy on Generated Data across 25 epochs using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow) with batch length of 16.

In terms of performance metrics, the `class/raw` representation once again performs better in general than all other representations, with an average F1-score of 0.4696, in comparison to the average F1-score of 0.3854 achieved in the previous test using a smaller batch length. The increase in performance is not the case for other representations however which all saw drops in average F1-score from their counterparts in the previous series of tests, also dropping below baseline metrics for this stage.

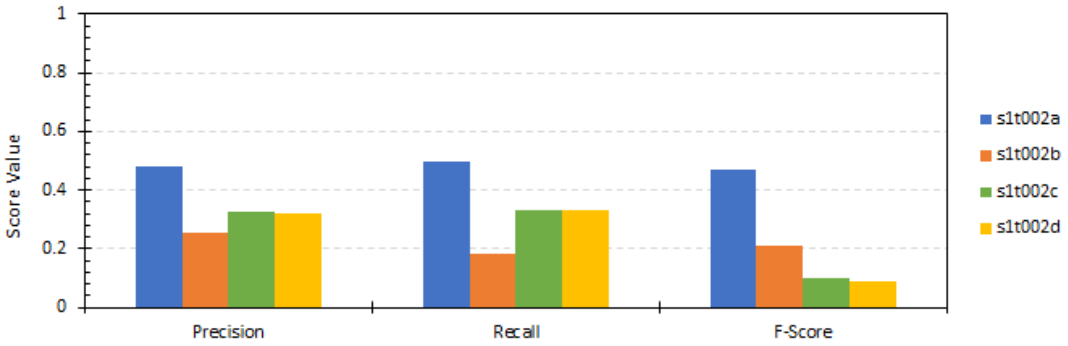


FIGURE 5.24: The average precision (left), recall (middle), and F1-score (right) on Generated Data using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow).

5.3.5 Batch Length: 24

Tests 003*a* to 003*d* carry out training using a batch length of 24 alongside the parameter combination determined in Stage 0. Test `s1t003a` uses a `class/raw` representation, test `s1t003b` uses `class/interval`, test `s1t003c` uses `bounds/raw`, and test `s1t003d` uses `bounds/interval`. Figure 5.25 shows a graph of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

With the increased batch length, categorical accuracy drops again for all representations except `bounds/interval`. This representation now reaches an accuracy of 0.5290, which is significantly higher than in previous tests. However, other representations drop to accuracy levels comparable to those achieved with a batch length of 8. The `class/raw` representation reaches an accuracy of 0.3662, the `class/interval` representation reaches an accuracy of 0.2553, and the `bounds/raw` representation reaches an accuracy of 0.3585.

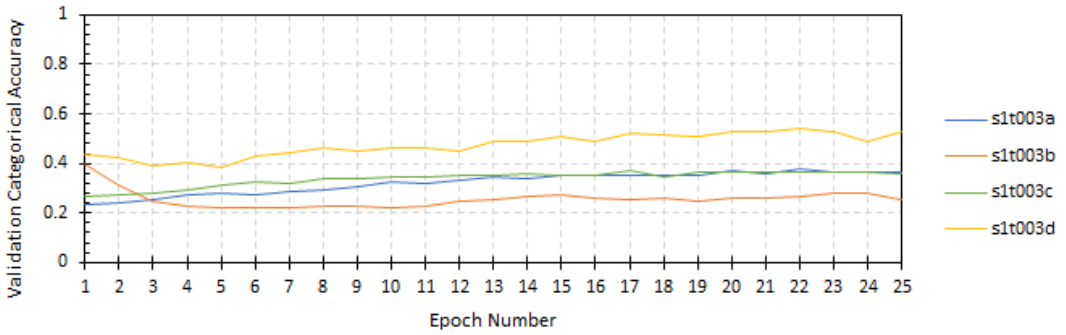


FIGURE 5.25: A graph of validation categorical accuracy on Generated Data across 25 epochs using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow) with batch length of 24.

Along with drops in categorical accuracy, we also see a decline in performance metrics for the representations using `raw` pitch formats. The `class/raw` representation in particular drops to an average F1-score of 0.2247 which is markedly lower than the metrics achieved in previous tests. However, `interval` representations actually seem to perform better with an increased batch length, climbing up to above baseline metrics. The `class/interval` representation has an average F1-score of 0.2617 and the `bounds/interval` representation has an average F1-score of 0.2868. The improved performance can be attributed to a reduced skew towards the `pattern_continue` class, as shown in Table 5.14.

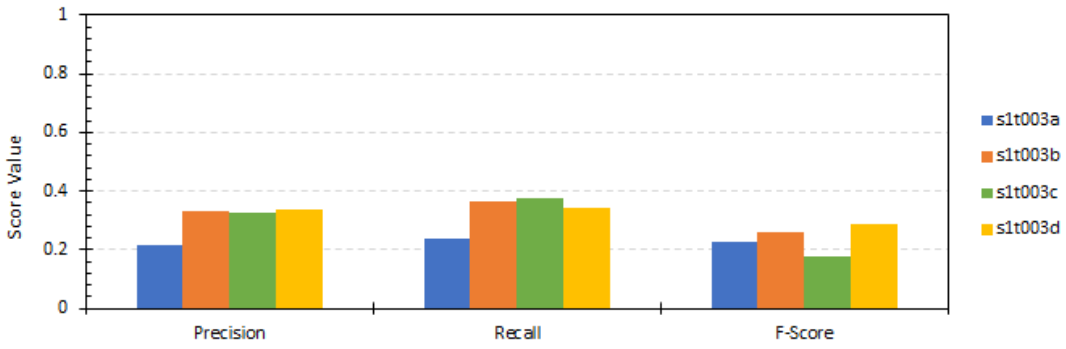


FIGURE 5.26: The average precision (left), recall (middle), and F1-score (right) on Generated Data using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow).

Dataset	Average Metrics		
	Precision	**Recall**	**F1-score**
Generated Data/Bounds (`s1t003d`)	0.3382	0.3431	0.2868
pattern_continue	0.8866	0.5634	0.6889
pattern_start	0.0602	0.4001	0.1047
pattern_end	0.0678	0.0659	0.0669

TABLE 5.14: Table of performance metrics for test `s1t003d`.

5.3.6 Summary

Our findings indicate that a model trained on data that has been generated artificially (to be less complex than real musical data) can classify certain note sequences correctly when provided with enough training data. The network model seems to perform better in a classification oriented task and manages to abstract enough information for each class to make accurate guesses on similarly structured test data. In a boundary detection task, the proposed system struggles due to the training skew towards non-boundary notes. This is a problem inherent in the representation and is expected to persist as a problem throughout all 3 stages. We also observe improved performance when the length of input batch sequences is closer to the length of the learned pattern classes.

The tests are fairly successful in this stage and serve as a proof of concept for the function of the proposed system. In the next stage, we increase complexity by using patterns taken from real music data, specifically the MTC-ANN dataset as described in Section 4.2.3, and observing the performance of the model on this new training data.

5.4 Pattern Discovery on Synthetic MTC-ANN

The second stage of training introduces an additional layer of complexity by using patterns taken from real-life music. These patterns are taken from the MTC-ANN dataset as described in Section 4.2.3. For the purposes of having enough data to train on per class, without bloating the training set too much and thereby increase training time to infeasible levels, we reduce the number of pattern classes to five. This is done by taking the five most common motifs from the MTC-ANN dataset and using them to generate data by randomly combining them into short musical pieces.

This stage of training will continue to use the parameter combinations determined in Section 5.2: a hidden size of 128 over 25 epochs, with a 10 n-fold cross-validation split. The series of tests uses the same format as in the previous stage, with increasing batch lengths and different combinations of pattern representation and pitch format. Tables 5.15 and 5.16 show the baseline precision, recall, and F1-score of this stage. In the `class` representation, we observe an average F1-score rating of $\sim$20%), which is the expected average for a class-based random guessing trial with 5 classes. In the `bounds` representation, we see the same skew towards `pattern_continue` that has been explained previously.

Dataset	Average Metrics		
	Precision	**Recall**	**F1-score**
Synthetic MTC-ANN/Class (Stage 2)	0.2003	0.2001	0.1911
Class `1:dg`	0.0725	0.1993	0.1299
Class `3:bag`	0.2192	0.1998	0.1984
Class `2:ba`	0.0964	0.1992	0.2706
Class `1:dbdg`	0.1953	0.2016	0.1573
Class `1`	0.4178	0.2001	0.1991

TABLE 5.15: Table of performance metrics for random guess on Class representation in Stage 2. Class names are taken from the MTC-ANN dataset (Kranenburg et al., 2016).

Dataset	Average Metrics		
	Precision	**Recall**	**F1-score**
Synthetic MTC-ANN/Bounds (Stage 2)	0.3329	0.3327	0.2034
`pattern_continue`	0.9334	0.3329	0.4908
`pattern_start`	0.0315	0.3351	0.0576
`pattern_end`	0.0341	0.3299	0.0618

TABLE 5.16: Table of performance metrics for random guess on Bounds representation in Stage 2

5.4.1 Overview of Results

This section provides a quick overview and summary of the detailed results given in the next few sections. We observe varying improvement between different representations and batch lengths. This stage mainly aims to answer the first and second research questions posed in Section 1.5: "*Is an artificially-generated fixed ground truth sufficient for training pattern discovery?*" and "*Can a model trained on artificial data correctly identify patterns in realistic music to a level comparable with human-annotated counterparts?*"

Our findings continue to confirm that a model trained on artificially-generated data is capable of pattern discovery to a level above baseline performance, even when the data is generated by synthetically joining patterns from real musical data. Some examples of this are given in Section 5.4.2. The complexity of the musical data is increased in this stage, but still yields relatively accurate predictions in class-based representations. Performance does decrease from the results of Stage 1, but this is to be expected considering that not only is data more realistic (and therefore more complex), but there are more pattern classes for the classification to deal with.

Furthermore, one of the major additional complexities in this stage is that each pattern class can take multiple similar (but not identical) forms. For example, the three note sequences shown in Figure 5.27 are all classified as part of class `1:dbdg` (one of the five common classes used to create the synthetic dataset). There are obvious similarities between these patterns to the trained human eye (matching aspects in rhythm or melody in this case), but a neural network has more trouble finding common ground to a sufficient level to identify the pattern class in a general manner. This is a more realistic characteristic of musical patterns, but adds an additional layer of complexity where the model not only needs to learn a model for each pattern class, but also needs to abstract that model from inexact pattern matches. In Stage 1, this was not the case because each occurrence of each pattern class was identical (except for the `ran` class).

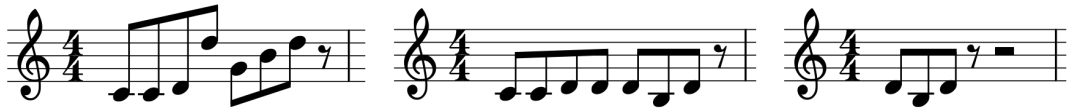


FIGURE 5.27: Some examples of note sequences classified as part of Class `1:dbdg` in the MTC-ANN dataset.

To answer the second research question, we tested the trained models on some unchanged files from the MTC-ANN dataset. In these files we considered only the notes that belonged to the pattern classes we had trained on, as the model currently has no way of filtering out notes that do not belong to a pattern that it has learned. As such, it assumes that every note has to belong to one of the classes it knows about and will attempt to classify every note. In the results for this test (shown in Section 5.4.2), we observe promising performance of the model, where it correctly identifies classes it knows about even when they are located amidst sequences of notes that it has never seen examples of. This suggests that yes, a model trained on artificially-joined data can learn enough about the pattern classes to identify them in non-artificially-joined data, provided it has a means to filter out notes that it does not think should belong to any class. Currently this filtering out is done manually, but given more time to work on this model a means could be devised for the system to do this itself.

5.4.2 Output Samples

This section will discuss some output examples from this stage of tests and compare them to the original file in order to give a concrete example of the system's performance and better demonstrate some details about the output provided by the network. Figure 5.28 shows one of the files used for testing with notes coloured in the `class` representation. In this stage, there are five possible classes. Like in previous stages, the classes are given a unique ID number (0, 1, 2, etc.) as internal representation. However, in this stage, the `slots_per_quarter` approach is implemented as described in Section 4.2.5. Since the MTC-ANN dataset includes multiple types of notes (not just quarter notes), the least common multiple of the duration of those notes is taken as the `slots_per_quarter` value. In this case, that value is 12. This means that each 12 slots in the array correspond to a quarter note, each six correspond to an eighth note, each four correspond to a triplet note, and so on. For example, a MIDI file such as the one in Figure 5.28 loaded in the `class` representation is internally represented as a string like so:

```
2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0
0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4
```

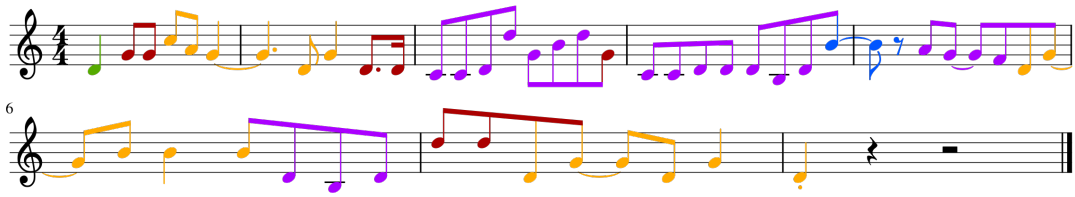


FIGURE 5.28: MIDI representation of the expected output with notes coloured by class: `1:dg` (red), `3:bag` (blue), `2:ba` (green), `1:dbdg` (purple), and `1` (orange).

Figure 5.29 shows the output of test `s2t001a` on the above file. This test uses the `class/raw` representation. Since the batch length is 8, the first 8 slots are greyed out as they are not part of the output. With the given `slots_per_quarter` value, 8 slots is not enough to capture a full quarter note but is enough to capture anything smaller than an eighth note. Notes are classified based on the majority of their respective slots (that is, if a quarter note has 9 out of 12 slots classified as Class `1:dg`, the note is given Class `1:dg` and the remaining slots in that note are ignored).

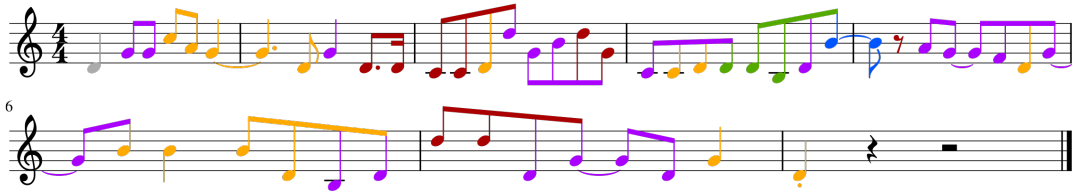


FIGURE 5.29: MIDI representation of the actual output with batch length 8 with notes coloured by class: `1:dg` (red), `3:bag` (blue), `2:ba` (green), `1:dbdg` (purple), and `1` (orange).

Despite the observable difference in data (different note types and messier patterns), we observe similar behaviour of the network in this stage. Patterns that are much longer than the batch length are less likely to be classified correctly, whereas shorter patterns are more accurately pinpointed. That being said, this could also have to do with the fact that shorter patterns are typically simpler, and therefore easier for the network to learn.

Figure 5.30 shows the output of test `s2t002a` on the above file. This test also uses the `class/raw` representation, but with a batch length of 16. We observe the expected improvement as the batch length increases and comes closer to the average pattern length. More notes in the sequence are correctly classified, but we observe the same fuzziness near the boundaries that was prominent in Stage 1.

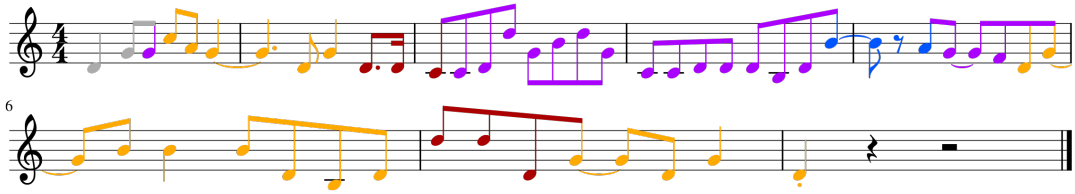


FIGURE 5.30: MIDI representation of the actual output with batch length 8 with notes coloured by class: `1:dg` (red), `3:bag` (blue), `2:ba` (green), `1:dbdg` (purple), and `1` (orange).

Finally, Figure 5.31 shows the output of test `s2t003a` on the same file. The batch length is now 24, still using the `class/raw` representation. The same problems arise that we observed in the examples of Section 5.3.2. The batch length is now too large and tends to carry over predictions past pattern boundaries by a wider margin. Interestingly we observe more circumstances where a sub-part in a longer pattern that is rhythmically and melodically similar to a smaller pattern is identified as the smaller pattern. An example of this is visible in Bar 4 in Figure 5.31, where the first four notes are identified as belonging to Class `1:dg` because they are very similar to the shape of the actual class (as seen at the end of Bar 2).

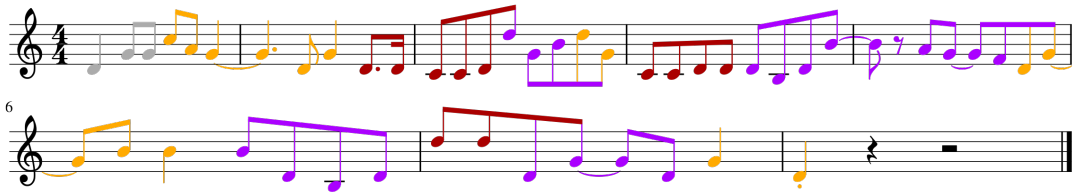


FIGURE 5.31: MIDI representation of the actual output with batch length 8 with notes coloured by class: `1:dg` (red), `3:bag` (blue), `2:ba` (green), `1:dbdg` (purple), and `1` (orange).

Another question we aimed to answer with this stage was whether or not a model trained on artificial data could identify the same patterns in real data. To test this, we took a few files from the MTC-ANN dataset that had instances of at least one of the five pattern classes that this stage was trained on. Figure 5.32 shows one such file.

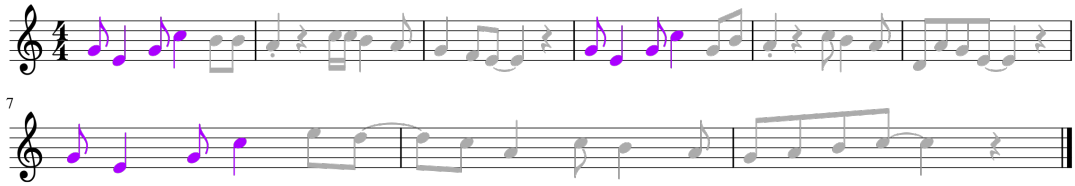


FIGURE 5.32: MIDI representation of the expected output for file NLB134474_01.mid when considering only the classes: `1:dg` (red), `3:bag` (blue), `2:ba` (green), `1:dbdg` (purple), and `1` (orange).

Greyed out notes in this figure are notes that either do not belong to any pattern class, or do not belong to a pattern class the model has been trained on. Such notes are ignored when evaluating the output. The model currently has no way of filtering out notes that do not belong to a pattern that it has learned. As such, it assumes that every note has to belong to one of the classes it knows about and will attempt to classify every note. This is obviously problematic in a real-life usage scenario, but for now the purpose of this test is to find out if it can correctly classify notes that belong to trained pattern classes when they are inside unchanged musical data without synthetically created pattern boundaries.

Figure 5.33 shows the output of the model trained in test `s2t001a` using the `class/raw` representation with a batch length of 8 on the same file. Notes that are faded out are notes that are not part of the test. The output of the model on these notes is shown for posterity and to show the similar delay characteristics that we have observed in previous tests also occurring here. The actual string output of the model on this file is as follows:

```
- - - - - - - - - 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 3 3 3 3 3 3 3 -
- - - ...  - - - 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 3
3 3 3 3 3 3 - - - ...  - - - 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 4 4 4 4 3 3 3 3 3 3 3 - - - ...  - - -
```

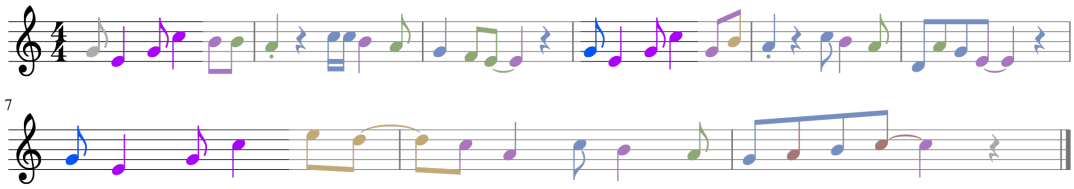


FIGURE 5.33: MIDI representation of the output of test `s2t001a` for file NLB134474_01.mid where classes are coloured as: `1:dg` (red), `3:bag` (blue), `2:ba` (green), `1:dbdg` (purple), and `1` (orange).

We observe that the model succeeds at classifying the three instances of class `1:dbdg` almost entirely accurately. This suggests that the model is learning enough of a definition for the pattern class that it can correctly identify it between sequences of data that it has no knowledge about. If we can find a way to filter out notes that the model should not be classifying, this would mean that the model is capable of pattern discovery on real music data even when the complexity of such data is re-introduced, provided it has had sufficient training on the pattern classes it is attempting to identify. One way to accomplish this filtering would be to introduce a confidence threshold, where a note is only given an output if the model is entirely certain that it belongs to one of the classes it trained on.

5.4.3 Batch Length: 8

Tests 001*a* to 001*d* carry out training using a batch length of 8 alongside the parameter combination determined in Stage 0. Test `s2t001a` uses a `class/raw` representation, test `s2t001b` uses `class/interval`, test `s2t001c` uses `bounds/raw`, and test `s2t001d` uses `bounds/interval`. Figure 5.34 shows a graph of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

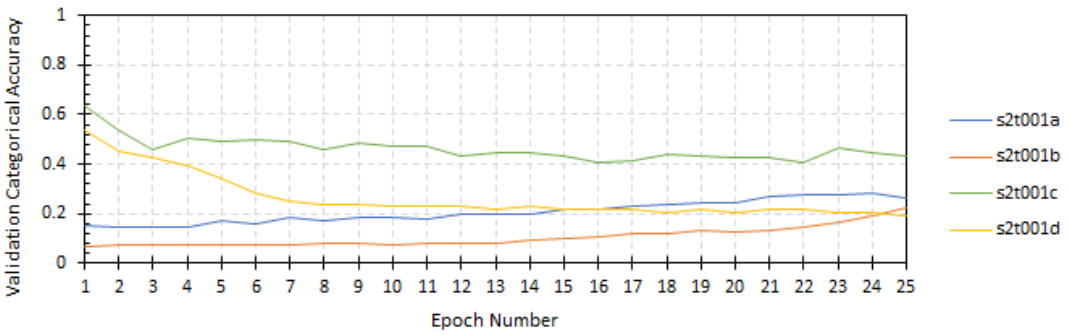


FIGURE 5.34: A graph of validation categorical accuracy on Synthetic MTC-ANN across 25 epochs using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow) with batch length of 8.

We observe that the categorical accuracy of the `class/raw` and `class/interval` representations reaches lower levels than in Stage 1. This is likely due to the increased number of pattern classes in the new dataset. The `class/raw` representation reaches a categorical accuracy of 0.2605 while the `bounds/interval` representation converges to an accuracy of 0.2234. We observe gradual climb in performance over time in both cases. In the `bounds/raw` and `bounds/interval` representations, we see similar values as in previous stages, but observe a drop in performance over time. One possible reason for this could be over-fitting to the `pattern_continue` class, which we observe to be dominantly occurrent in the dataset once again.

Looking at the performance metrics in Figure 5.35, we observe performance that is above baseline metrics in both `class` representations. The `class/raw` representation yields an average F1-score of 0.2483 while the `class/interval` representation yields an average F1-score of 0.2156. By comparison we see declining performance in `bounds` representations, with the `bounds/interval` representation in particular dropping well below baseline values with an average F1-score of 0.1072.

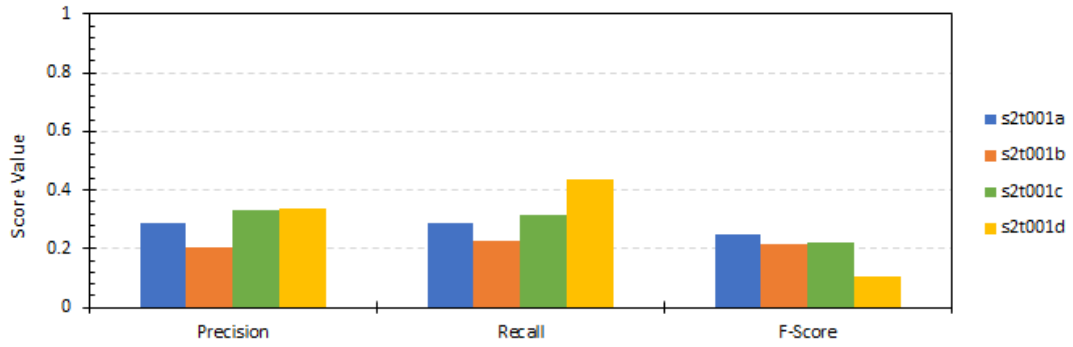


FIGURE 5.35: The average precision (left), recall (middle), and F1-score (right) on Synthetic MTC-ANN using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow).

Table 5.17 shows the results of test `s2t001a` in more detail. We observe a relatively balanced spread across 5 classes, though there is a noticeable skew in recall performance towards Class `1:dg`. Reasons for this are unclear, but can have to do with the class being simpler to identify or more closely matching the batch length. In Stage 1 we learned that performance is improved when the batch length is closer to the pattern length, and the same reasoning could be applicable when using a more complex dataset.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
Synthetic MTC-ANN/Class (Stage 2)	0.2866	0.2868	0.2483
Class `1:dg`	0.2664	0.5915	0.2055
Class `3:bag`	0.2913	0.2836	0.2587
Class `2:ba`	0.3072	0.1545	0.2014
Class `1:dbdg`	0.2636	0.2540	0.3051
Class `1`	0.3046	0.1504	0.2706

TABLE 5.17: Table of performance metrics for test `s2t001a`.

5.4.4 Batch Length: 16

Tests 002*a* to 002*d* carry out training using a batch length of 16 alongside the parameter combination determined in Stage 0. Test `s2t002a` uses a `class/raw` representation, test `s2t002b` uses `class/interval`, test `s2t002c` uses `bounds/raw`, and test `s2t002d` uses `bounds/interval`. Figure 5.36 shows a graph of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

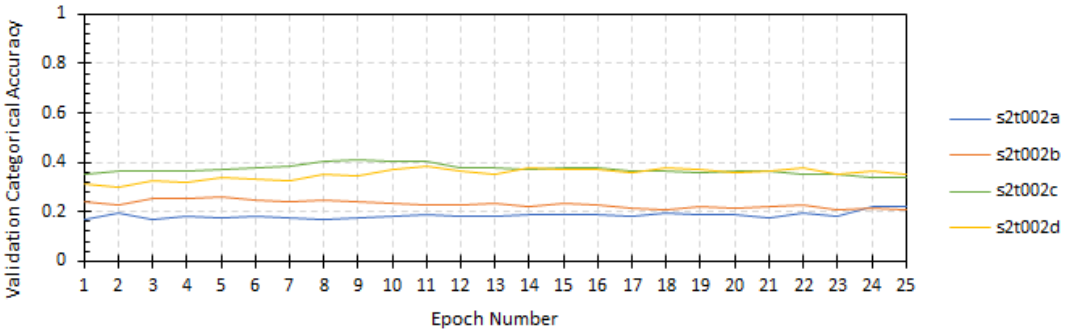


FIGURE 5.36: A graph of validation categorical accuracy on Synthetic MTC-ANN across 25 epochs using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow) with batch length of 16.

With the increased batch length, we observe decreased categorical accuracy in all representations except `bounds/interval`. We also observe that the accuracy remains more or less level throughout the training process, showing very little improvement over time. This suggests that training with this batch length may have gotten stuck in a local minimum, or else that the data batches are simply too difficult to learn enough information from. The `class/raw` representation and `class/interval` representations reach accuracies of 0.2227 and 0.2074 respectively. We see marked improvement in the `bounds/interval` representation which climbs to an accuracy of 0.3512, while the `bounds/class` representation drops to an accuracy of 0.3407.

In terms of performance metrics as shown in Figure 5.37 , we observe a similar increase in performance for the `class/raw` representation as in Stage 1. The average F1-score is 0.2839, in comparison to the average F1-score of 0.2483 achieved in the previous test using a smaller batch length. The increase in performance is not the case for other representations however which all saw drops in average F1-score from their counterparts in the previous series of tests, also dropping below baseline metrics for this stage.

5.4.5 Batch Length: 24

Tests 003*a* to 003*d* carry out training using a batch length of 24 alongside the parameter combination determined in Stage 0. Test `s2t003a` uses a `class/raw` representation, test `s2t003b` uses `class/interval`, test `s2t003c` uses `bounds/raw`, and test `s2t003d` uses `bounds/interval`. Figure 5.38 shows a graph of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).

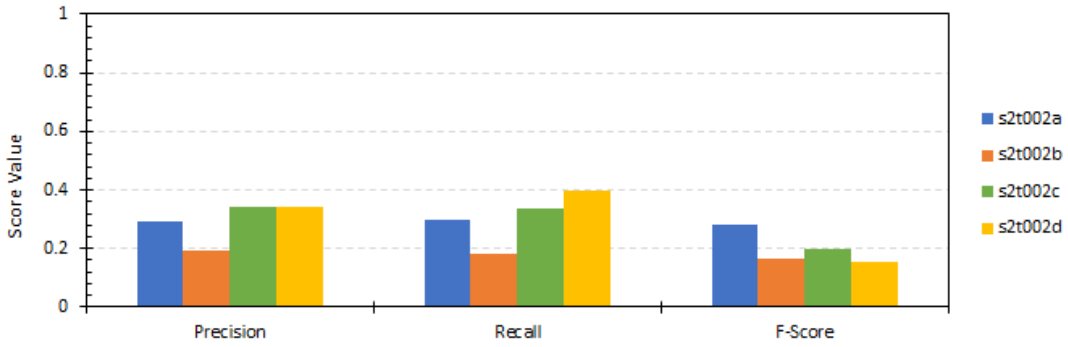


FIGURE 5.37: The average precision (left), recall (middle), and F1-score (right) on Synthetic MTC-ANN using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow).

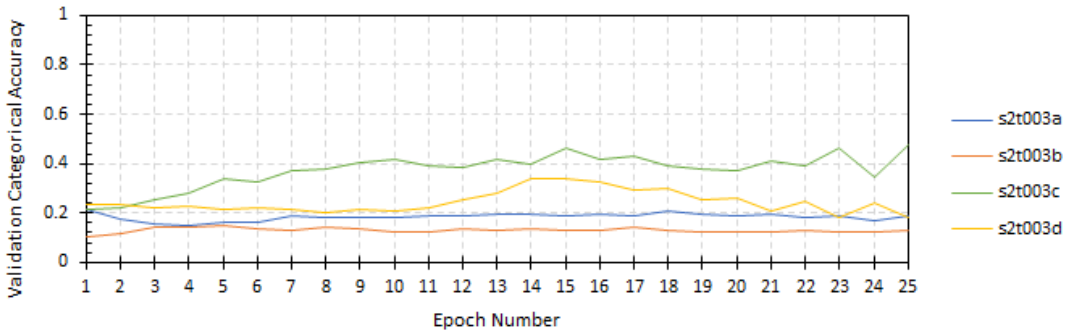


FIGURE 5.38: A graph of validation categorical accuracy on Synthetic MTC-ANN across 25 epochs using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow) with batch length of 24.

With the increased batch length, we see another decline in categorical accuracy, though the `bounds/raw` representation yields accuracies similar to those achieved when using a batch length of 8, with an average F1-score of 0.4756. The `class/raw` representation reaches an accuracy of 0.1851, the `class/interval` representation reaches an accuracy of 0.1285, and the `bounds/interval` representation reaches an accuracy of 0.1800.

Along with drops in categorical accuracy, we also see a decline in performance metrics in Figure 5.39. Both `class` and `bounds` representations show a decrease in average F1-score in comparison to previous tests. The `class/raw` representation has an average F1-score of 0.2357, remaining above baseline metrics. In comparison, the `class/interval` representation has an average F1-score of 0.1599, the `bounds/raw` representation has an average F1-score of 0.2192, and the `bounds/interval` representation has an average F1-score of 0.1507.

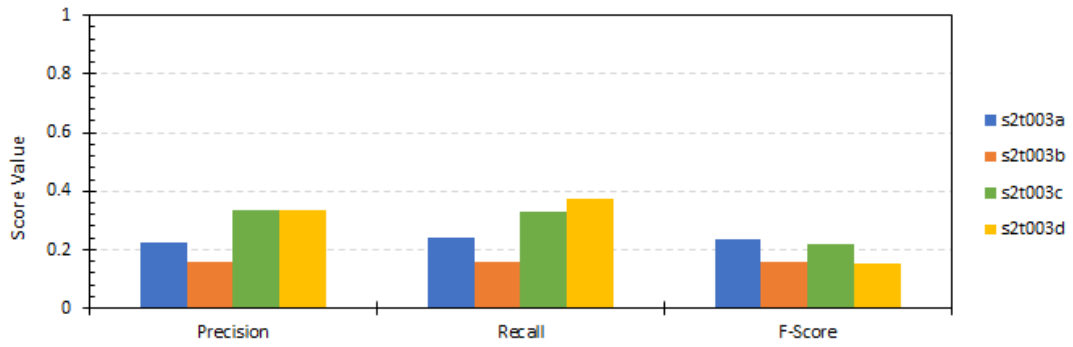


FIGURE 5.39: The average precision (left), recall (middle), and F1-score (right) on Generated Data using `class/raw` (blue), `class/interval` (orange), `bounds/raw` (green), and `bounds/interval` (yellow).

5.4.6 Summary

Our findings throughout the second stage of testing continue to show that a model with sufficient training data can perform remarkably well at classifying the pattern classes it has knowledge of. This is to be expected in some ways as neural networks have already proved very strong at this type of classification task in other media. However, considering the complexities of musical data the performance of the system is admirable. We observed promising results both when the patterns are tested in synthetically-joined data and when the same patterns are classified inside real musical data. The accuracy remains consistently above baseline despite the increased complexity in this stage of the project. Performance does decrease from the results of Stage 1, but this is to be expected considering that not only is data more realistic (and therefore more complex), but there are more pattern classes for the classification to deal with. The limitations of the system remain that it cannot classify patterns it has no knowledge about, but this is a problem inherent to neural networks more than it is the proposed system. Some possible future solutions for this problem are discussed in Section 6.2

The tests are again fairly successful in this stage and serve as the main basis for our conclusions, presented in Chapter 6. In the next stage, we further increase complexity by using an unchanged MTC-ANN dataset as described in Section 4.2.3 to investigate the performance of the system on a significantly more complex and much larger corpus of training data.

5.5 Pattern Discovery on MTC-ANN

The third and final stage of training uses unchanged real-life musical samples for training and testing. The MTC-ANN dataset as described in Section 4.2.3 is used in its raw format. This means that no classes are ignored, and pattern boundaries are those annotated by human musical experts with no shuffling or synthetic joining. The aim of this section is not necessarily to create a model that functions flawlessly, but to observe the behaviour of a model when using a significantly larger number of classes.

This stage of training will continue to use the parameter combinations determined in Section 5.2: a hidden size of 128 over 25 epochs, with a 10 n-fold cross-validation split. The series of tests uses the same format as in the previous stages, with increasing batch lengths and different combinations of pattern representation and pitch format. Table 5.18 shows the baseline precision, recall, and F1-score of this stage. In the `class` representation, we now observe a drastically lower average F1-score. The reason for this is that the unchanged MTC-ANN dataset has 94 pattern classes in total. Random guessing should therefore yield an accuracy rate of $\sim$1%, which is what we observe here. In the `bounds` representation, we see the same skew towards `pattern_continue` that has been explained previously.

Dataset	**Average Metrics**		
	Precision	**Recall**	**F1-score**
MTC-ANN/Class (Stage 3)	0.01023	0.01014	0.01019
MTC-ANN/Bounds (Stage 3)	0.3334	0.33513	0.1868
pattern_continue	0.9661	0.3330	0.4953
pattern_start	0.0159	0.3351	0.0304
pattern_end	0.0182	0.3373	0.0346

TABLE 5.18: Table of performance metrics for random guess in Stage 3

5.5.1 Overview of Results

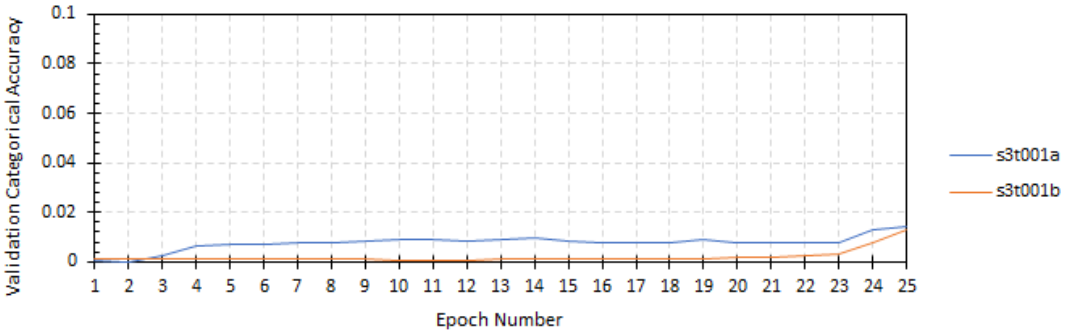
This section provides a quick overview and summary of the detailed results given in the next few sections. We observe poor performance when using class-based representations and a general drop in performance across all tests. This stage mainly aims to answer the second part of the first research question posed in Section 1.5: "*Does performance drop when a fixed ground truth is not available ?*".

Our findings suggest that the approach used in this project is not yet ready for training on unchanged real-life musical data. While we observed good performance in the previous two stages where the complexity and the number of classes were reduced, this performance dropped by a large margin in this stage, which attempted to train a model from the raw MTC-ANN dataset. The reason for this is likely because, due to the large amount of classes, the model has very few occurrences of each pattern class to train on, leading it to be stretched too thin across the available set of pattern classes. This problem, compounded with the difficulties that we already explained in Section 5.4.1 regarding each class having several different possible forms, results in a model that is incapable of making clear classification choices on the given data. This creates a scenario where output strings are relatively incoherent compared to the clearer results we obtained in previous stages.

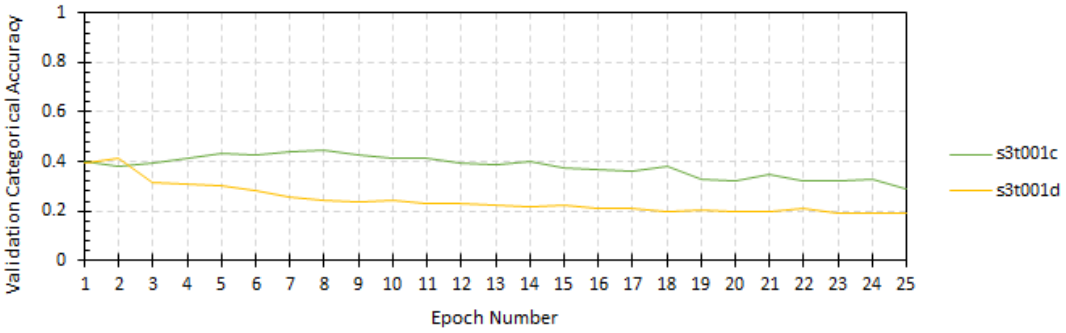
Naturally, there are plenty of possible improvements and tweaks that can be made to the system to have it perform better, not least of which is augmenting the dataset to have more trainable occurrences of each class and performing more training in general. More discussion regarding this matter is given in Section 6.2.

5.5.2 Batch Length: 8

Tests 001*a* to 001*d* carry out training using a batch length of 8 alongside the parameter combination determined in Stage 0. Test `s3t001a` uses a `class/raw` representation, test `s3t001b` uses `class/interval`, test `s3t001c` uses `bounds/raw`, and test `s3t001d` uses `bounds/interval`. Figures 5.40a and 5.40b show graphs of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation). Note that the scale of the y-axis in Figure 5.40a is smaller than in Figure 5.40b, due to the drop in accuracy for the two class-based representations.



(A) Using `class/raw` (blue) and `class/interval` (orange).



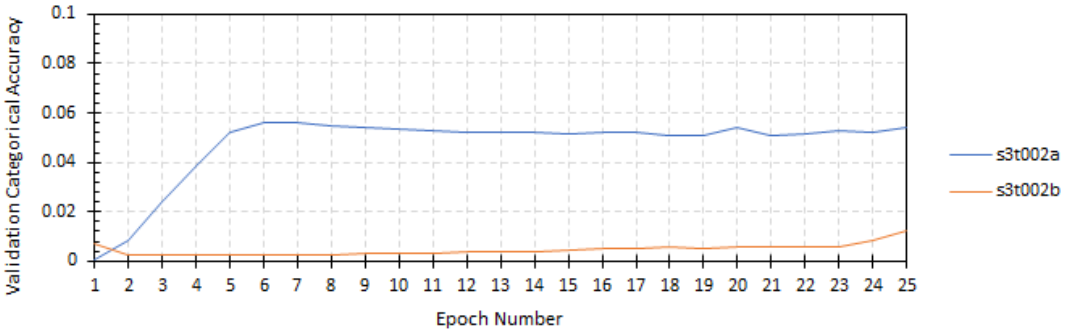
(B) Using `bounds/raw` (green) and `bounds/interval` (yellow).

FIGURE 5.40: A graph of validation categorical accuracy on MTC-ANN across 25 epochs using with batch length of 8.

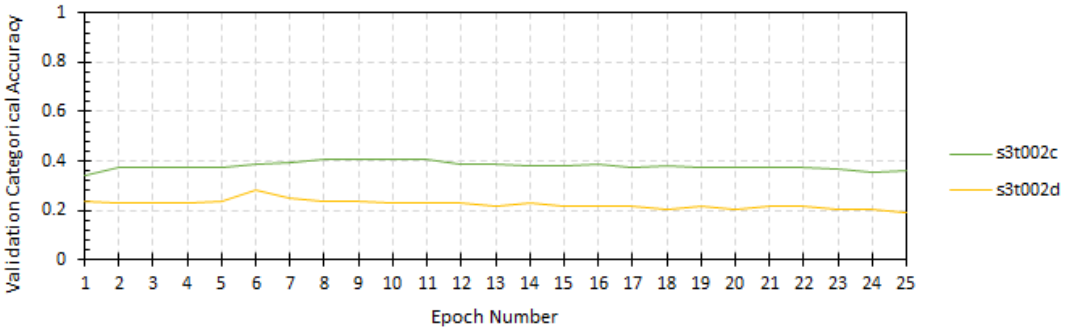
We observe that the categorical accuracy of the `class/raw` and `class/interval` representations reaches much lower levels than in Stages 1 and 2. This is due to the significantly increased number of pattern classes in the MTC-ANN dataset. The `class/raw` representation reaches a categorical accuracy of 0.0142 while the `bounds/interval` representation converges to an accuracy of 0.0129. We observe very marginal gradual climb in performance over time in both cases. In the `bounds/raw` and `bounds/interval` representations, we see similar values as in previous stages, but observe a drop in performance over time. Once again this is likely due to over-fitting of the model to the `pattern_continue` class.

5.5.3 Batch Length: 16

Tests 002*a* to 002*d* carry out training using a batch length of 16 alongside the parameter combination determined in Stage 0. Test `s3t002a` uses a `class/raw` representation, test `s3t002b` uses `class/interval`, test `s3t002c` uses `bounds/raw`, and test `s3t002d` uses `bounds/interval`. Figures 5.41a and 5.41b show graphs of the validation categorical accuracy over 25 epochs for the four representations. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).



(A) Using `class/raw` (blue) and `class/interval` (orange).



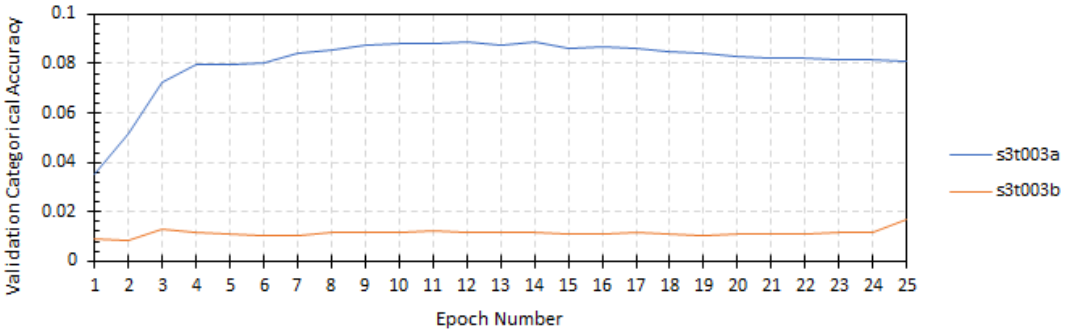
(B) Using `bounds/raw` (green) and `bounds/interval` (yellow).

FIGURE 5.41: A graph of validation categorical accuracy on MTC-ANN across 25 epochs using with batch length of 16.

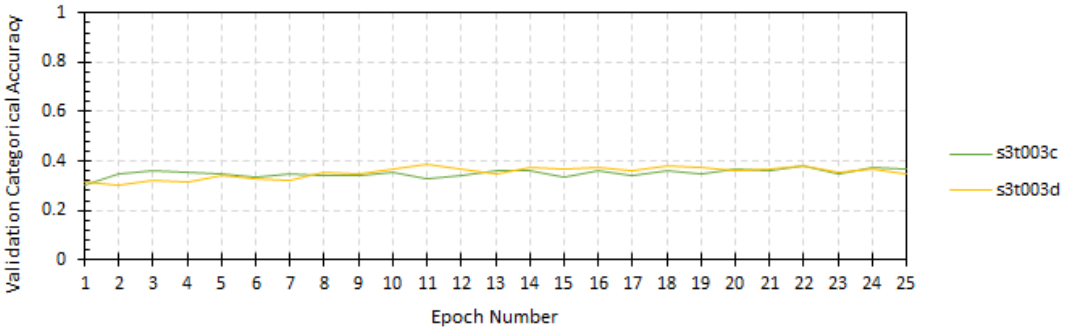
With the increased batch length, we observe a large increase in categorical accuracy in the `class/raw` representation, although the overall accuracy still remains below 0.1. This rapid improvement is not the case for the `class/interval` representation. The `class/raw` representation and `class/interval` representations reach accuracies of 0.0538 and 0.0125 respectively. We see some improvement in the `bounds/interval` and `bounds/class` representations which reach accuracies of 0.3635 and 0.1895 respectively.

5.5.4 Batch Length: 24

Tests 003*a* to 003*d* carry out training using a batch length of 24 alongside the parameter combination determined in Stage 0. Test `s3t003a` uses a `class/raw` representation, test `s3t003b` uses `class/interval`, test `s3t003c` uses `bounds/raw`, and test `s3t003d` uses `bounds/interval`. Figures 5.42a and 5.42b show graphs of the validation categorical accuracy over 25 epochs for the four representations.. The value of the categorical accuracy at each epoch is an average over the 10 n-folds (cross-validation).



(A) Using `class/raw` (blue) and `class/interval` (orange).



(B) Using `bounds/raw` (green) and `bounds/interval` (yellow).

FIGURE 5.42: A graph of validation categorical accuracy on MTC-ANN across 25 epochs using with batch length of 24.

With the increased batch length, we observe decreased categorical accuracy in all representations. We also observe that the accuracy remains more or less level throughout the training process, showing very little improvement over time. This suggests that training with this batch length may have reached a local minimum, or else that the data batches are simply too difficult to learn enough information from. The latter is more likely the case due to the complexity of the data in this stage. The `class/raw` representation and `class/interval` representations reach accuracies of 0.0812 and 0.0170 respectively. We see some more improvement in the `bounds/interval` representation which climbs to an accuracy of 0.3647, while the `bounds/class` representation drops to an accuracy of 0.3512.

Chapter 6
Conclusion

6.1 Research Summary

In the introduction, we stated three research questions regarding the research goals of this project and the use of machine learning to create a pattern discovery algorithm capable of learning from human annotations.

Research Question 1
Is an artificially-generated fixed ground truth sufficient for training pattern discovery (stages 1 and 2), and does performance drop when this fixed ground truth is not available (stage 3)?

Research Question 2
Can a model trained on artificial data (stage 2) correctly identify patterns in realistic music to a level comparable with human-annotated counterparts?

Research Question 3
Can the proposed system perform comparably with other pattern discovery algorithms that utilise differing approaches?

The first two questions have already been answered in part in Sections 5.3.1, 5.4.1, and 5.5.1, but in this section we will summarise everything and discuss the results of the entire training and testing, as well as look at our answer for the third question. In each subsection, we give the answer to the corresponding research question based on the results from the previous chapter.

6.1.1 Influence of Synthetic Ground Truths and Musical Complexity

This research question aims to look at the differences between training on less complex musical data and training on real musical data. In order to do this, the results of Stages 1 and 2 as described in 1.5 are compared to those of Stage 3.

In Stage 1, as shown in Section 5.3.2, we observe that a model trained on artificially-generated data is capable of pattern discovery above baseline level (random guessing) on similar artificially-generated data. The model performs better in the class-based representations, suggesting that a classification approach is a better solution for the type of learning than a boundary identification approach. This makes sense, because identifying common characteristics of a single pattern class is easier than trying to come up with a general definition of what is and is not a pattern boundary. We observe low performance of the boundary representation tests even when the boundaries are clearly defined (as they are in this stage), suggesting that the boundary identification task might be too difficult for the network to learn effectively. Notably, we observed

that representing pitch as intervals did improve the performance of boundary representations somewhat. It is possible that using more advanced note representations might yield different results, but that was not within the scope of this project.

In Stage 2, as shown in Section 5.4.2, we continue to confirm that a model trained on artificially-generated data is capable of pattern discovery to a level above baseline performance, even when the data is generated by synthetically joining patterns from real musical data. Despite the increased complexity and number of pattern classes, we observed that the model was capable of learning a sufficient definition of each pattern class to identify that class in synthetically joined musical pieces. This implies that the complexity of a pattern does not necessarily matter to the training process given that enough data about that pattern is provided. This data would ideally come in the form of varied occurrences of each pattern class mixed along with other classes such that the model can learn separation between different pattern classes in sequence.

In Stage 3, as described in Section 5.5.1, we see the limitations of this approach. When trained on an unchanged dataset with a realistically large and diverse amount of pattern classes, the model struggles to learn sufficient information to properly classify these results. Performance drops by a significant margin, and output is incoherent when compared to the results obtained from the previous stages. In theory, a boundary detection algorithm would be more successful in such a case, but the boundary representations used in this project still exhibited the same type of problems that had been evident in previous stages.

Across the three stages, we see that performance remains above the baseline of random guessing by a steady margin provided that sufficient data about each pattern class is provided to the training stage. The results of stages 1 and 2 show that the network can successfully classify patterns with a margin of error, and we believe that with further tuning and some improvements to the model it would be capable of higher accuracy rates. We observe that performance does drop when a clearly-defined pattern separation is not available (stage 3), but this likely has more to do with the rapid increase in class number (and by extension the decrease in available data for each class without augmentation) rather than a lack of clear pattern boundaries. It is difficult to gauge the effectiveness of the network training on complex pattern boundaries without further testing and a larger dataset. This problem ties into the response to the second research question, as discussed in the next section. Moreover, training took significantly longer due to the increased size of the dataset, which was still not complete enough to provide sufficient data for the network to learn. Increasing this data to an effective level would likely involve significantly increased training times.

We also observed that performance was generally better when the input sequence of data being processed by the network was closer in length to the average length of pattern classes. In some cases, this improvement in performance was substantial. However, it is important to bear in mind that by tuning the length of input to fit a specific dataset, one can also potentially omit outlier pattern classes that are further away from the average pattern length. In a realistic dataset, patterns can vary greatly in length, so it is important to keep in mind that the batch length should be able to cater for a broader set of inputs. One other factor to consider is overlap in the batch inputs (for example, if using a batch length of 8, Batch B would start halfway through Batch A rather than at the end). We did not have time to properly investigate this in this project, however it is possible that overlapping the input batches would allow for longer term recognition between batches by the model.

6.1.2 Using Training on Artificial Data to Classify Real Data

This research question aims to determine if the training done to learn about pattern classes requires those pattern classes to be placed in a real scenario, or if it is sufficient to train a network on synthetically-joined patterns. The goal is to analyse if the models trained in Stage 2 (using synthetically joined patterns from MTC-ANN with a reduced number of classes) are capable of identifying occurrences of the trained patterns even when they are placed in different circumstances, namely when surrounded by notes that do not belong to any other trained pattern class. To do this, the models are tested against unchanged files pulled directly from the MTC-ANN dataset. An example of the results of this are shown in Section 5.4.2.

We observe fairly promising performance of the system in this scenario. It correctly identifies classes it knows about even when they are located amidst sequences of notes that it has never seen examples of. This suggests that a model trained on artificially-joined data can learn enough about the pattern classes to identify them in non-artificially-joined data. The difficulty comes in recognising which parts of the data *do not* belong to any of the classes the model has been trained on. Currently, the system is incapable of doing so. Instead, it simply learns the "none" class as if it were any other pattern class, and attempts to classify notes into it. This is problematic because the "none" class is generally dominantly present in a realistic dataset (most notes do not necessarily belong to a pattern), and the learning of the model can therefore skew towards the "none" class at the expense of other proper pattern classes.

This aspect of learning is an inherent problem with neural networks. When giving a network training data, we teach it what constitutes each class, but teaching it what is *not* in a class is a more difficult task because doing so would involve providing examples of every possible type of data that does not belong in the class. Essentially, the network can only provide predictions about data it has knowledge about, and if exposed to data it has no knowledge about it will attempt to make predictions based on what it knows. There are partial solutions to this, but implementing and testing them would have required more time and effort that was beyond the scope of this project. We provide more discussion about this in Section 6.2.

6.1.3 Comparison with Pattern Discovery Algorithms

This research question aims to determine if the results of this approach on a given unchanged dataset (stage 3 of training) are of a level that is comparable to other pattern discovery algorithms, such as the ones discussed in Section 3.4. While we had hoped to have sufficient results from stage 3 to make such a comparison, we instead determined that the model was not yet capable of handling data with that level of complexity. Improving the proposed system to a level comparable to state-of-the-art pattern discovery algorithms would require significant upgrades to multiple parts of the overall system.

That being said, our findings show promise despite the poor performance of the system on a full realistic dataset. We observe relatively high classification accuracy when training on a reduced number of classes, and observe that accuracy to carry over when identifying those classes in a realistic dataset. This performance is well above random guessing and with further improvement could potentially prove to be very effective.

Our conclusion regarding this matter is that it is currently difficult to give a concrete answer regarding how the proposed system would perform in comparison to other state-of-the-art algorithms. We would require more extensive training to observe how the system improves when provided with sufficient musical data. The main drawback of the system is its reliance on extensive training time. Without significant computational power which we did not have at our disposal, the system simply takes too long to learn anything meaningful from a realistic (and therefore larger) corpus of musical data and musical pattern classes. We hypothesise that with more time to thoroughly train the model on each pattern class it could very well perform to a much higher level. This of course brings up the problem that training will always take a near-infeasible amount of time in order for the system to perform at a sufficient level. Using the proposed system and trained models on realistic corpora will continue to have certain limitations in how many pattern classes the system can learn to keep training time feasible. However, there does seem to be potential in detecting musical repetition, and with advances in hardware and overall computational speed, as well as improvements to the framework of the proposed system, it is possible that the suggested training scheme could be feasible and successful on a state-of-the-art level.

6.2 Future Work

In light of our findings, there are some major improvements that can be used as a starting point to build upon the system outlined in project. Firstly, the most obvious improvement would be to provide the system with more data regarding the pattern classes it aims to learn about. This means more occurrences of each pattern class in sequence with other pattern class. For example, if we have N pattern classes, we ideally want training examples where each pattern class n has a pattern occurrence in one of the data files such that patterns of classes 0 to N precede it, follow it, or both. The more combinations of pattern occurrences we can provide, the better the expected results of the system given substantial training time.

In our results, we showed that when given sufficient data about 5 pattern classes from the MTC-ANN dataset (stage 2), the system is capable of identifying those classes even within more complex scenarios. We believe that, if provided with exhaustive amounts of data for every class (as aforementioned), the model is capable of applying this successful performance to a much larger selection of pattern classes, including potentially even more complex classes that are longer or involve more complex musical texture.

Our findings also indicate that this data does not necessarily need to *all* be in the form of manually-annotated musical pieces. The good performance of training on synthetically-joined musical pieces suggests that data augmentation techniques could be used to make up for the overall lack of available music data that is a general problem in MIR. If we have a good selection of pattern occurrences for each pattern class, we can use these augmentation techniques to create our own dataset with a balanced amount of occurrences between classes and exhaustive combinations and sequences. Obviously, providing an effective amount of training to such a system would require significantly more time than is generally feasible, unless the available hardware has much higher computational power than was available in this project. Tuning the parameters to better fit the dataset would also help to further improve performance.

Finding a way to have the model filter out notes that do not belong to any recognised class is another important next step for such a system. Doing so would allow the model to successfully point out pattern occurrences without the need of foreknowledge about where the pattern starts and ends. In the example we show in Section 5.4.2, we know where the pattern exists and can therefore manually determine how accurate the classification is being. By allowing the model to automatically point out patterns and filter out everything else, it takes a step closer to being a potential replacement for human annotation. One simple way to include such a mechanic would be to introduce a *confidence threshold*. As the system currently stands, it will give a value between 0 and 1 for each available class, which is effectively it's *confidence* that the note belongs to each respective class. The class in which the system's confidence is highest (has the highest value between 0 and 1) is picked as the prediction of the model. Introducing a *confidence threshold* would therefore mean deciding on a value where if the confidence for all classes is below that value, the note is classified as not belonging to any class. Naturally, determining such a value is non-trivial and would require further testing and tuning. Other more advanced options are also a possibility, but doing more research into this aspect of the system was outside the scope of this project.

There are of course a number of other improvements that can be made to the proposed system that we cannot all list here. Looking at different aspects of the framework and improving on those aspects in some way or another can more than likely make iterative contributions towards increased performance. For example, looking at the input data, converting the input note sequences into more informative representations (such as maybe using the viewpoint system suggested by Conklin and Witten (1995)) instead of simply using the sequence of notes could lead to faster and better training of network models that can extrapolate an understanding of pattern classes based on information that is not present in the raw note sequences we used (information like relative note offsets, combining interval values with raw pitch, or even complex hierarchical structures such as the parts used by SymCHM (Pesek et al., 2017b)).

Alternatively, looking at the machine learning part of the system, using different network topologies and more complex layer setups that are tailored to the task at hand can further augment the system's ability to learn the required information. In this project, we used a very rudimentary layer topology that served to give a baseline of LSTM performance, but researching further into network structures used by other studies and using more advanced techniques to guide the learning process would almost definitely prove beneficial. Neural networks are a popular subject in research (a subject that is still growing with time) and have been used extensively in other fields of information retrieval, which means that it should be easy to find other examples of their use in similar retrieval tasks and attempt to apply them to the field of MIR. Both current studies and future advancements can provide beneficial knowledge and technology for this kind of learning-based approach. In short, more research could shed light on a great deal of possible improvement to the proposed system, but this is outside the scope of this current project.

6.3 Final Remarks

To conclude, we have created a system that has a lot of potential for future research and development. There is presently a large gap in performance between the classification accuracy using the proposed system and the state-of-the-art Pattern Discovery algorithms, but the models we trained showed signs of improvement with more time and model tuning. We recommend that future research into this topic allows for a more extensive training process with potentially more computational power to explore performance when the system is allowed to learn for a much longer period of time. We have learned that it is possible for an LSTM neural network to establish a definition for pattern classes given enough training time and training data, whether that data is constructed from simple musical sequences (Section 4.2.2) or from realistic musical data (Section 4.2.3). We have also learned that it is possible to train an LSTM neural network on patterns combined synthetically in sequence that are taken from a realistic dataset and have the learned model be able to identify those patterns in their original scenarios.

Ultimately, we have determined that LSTMs can be an effective tool for pattern discovery in MIR, but that more work is required before they can be applied in broader scenarios. We hope that the base work we provide serves as a beneficial starting point and serves to open the door towards further developing and improving Pattern Discovery algorithms.

Acknowledgements

Writing a thesis is always hard, but it was made significantly less so by all the people who helped me along the way, who provided feedback, suggestions, constructive criticism, and advice without which this thesis would never be what it is. First of all, I would like to express my deepest gratitude to my supervisors: Anja Volk and Iris Yuping Ren for all their patience and assistance over the course of the year and the many meetings and late-night emails. Anja's insights and experience in MIR were invaluable, and she always came up with relevant improvements and additions to my work, while Iris was equally helpful with feedback and in helping set up the experiment framework and guiding me through the often overwhelming amount of results. I am very thankful for their help in all matters, and for their expertise and passion for this field and for our work.

Secondly, I want to thank Peter van Kranenburg for his valuable feedback as my second examiner on this thesis. His insights and fresh perpective helped me to finalise and tweak the thesis into a better, clearer, and more understandable end product, and his knowledge of the MTC-ANN dataset helped guide ideas about our conclusions and thoughts for future work.

Finally, I want to thank all my friends and family who have been incredibly supportive over the course of this thesis and beyond. My parents for their advice and support, not only in this year but throughout my entire education, my sister for always being willing to listen to my rambling thoughts and help me figure things out, and last but not least all my close friends for their motivation, feedback, and proof-reading.

Thank you all.

fine

Appendix A

Overview of Tests

Test ID	Data Format		Parameters				
	Pattern	Pitch	batch length	hidden size	num epochs	nfold splits	optimiser
s0t001a	class	raw	8	**512**	10	10	adam
s0t001b	class	interval	8	**512**	10	10	adam
s0t002a	class	raw	8	**128**	10	10	adam
s0t002b	class	interval	8	**128**	10	10	adam
s0t003a	class	raw	8	**64**	**10**	10	adam
s0t003b	class	interval	8	**64**	**10**	10	adam
s0t004a	class	raw	8	128	**5**	10	adam
s0t004b	class	interval	8	128	**5**	10	adam
s0t005a	class	raw	8	128	**25**	**10**	adam
s0t005b	class	interval	8	128	**25**	**10**	adam
s0t006a	class	raw	8	128	25	**5**	adam
s0t006b	class	interval	8	128	25	**5**	adam
s0t007a	class	raw	8	128	25	**15**	adam
s0t007b	class	interval	8	128	25	**15**	adam

Test ID	Data Format		Parameters				
	Pattern	Pitch	batch length	hidden size	num epochs	nfold splits	optimiser
s0t008a	class	raw	8	**512**	10	10	sgd
s0t008b	class	interval	8	**512**	10	10	sgd
s0t009a	class	raw	8	**128**	10	10	sgd
s0t009b	class	interval	8	**128**	10	10	sgd
s0t010a	class	raw	8	**64**	**10**	10	sgd
s0t010b	class	interval	8	**64**	**10**	10	sgd
s0t011a	class	raw	8	128	**5**	10	sgd
s0t011b	class	interval	8	128	**5**	10	sgd
s0t012a	class	raw	8	128	**25**	**10**	sgd
s0t012b	class	interval	8	128	**25**	**10**	sgd
s0t013a	class	raw	8	128	25	**5**	sgd
s0t013b	class	interval	8	128	25	**5**	sgd
s0t014a	class	raw	8	128	25	**15**	sgd
s0t014b	class	interval	8	128	25	**15**	sgd

Test ID	Data Format		Parameters				
	Pattern	**Pitch**	batch length	hidden size	num epochs	nfold splits	optimiser
s1t001a	class	raw	8	128	25	10	sgd
s1t001b	class	interval	8	128	25	10	sgd
s1t001c	bounds	raw	8	128	25	10	sgd
s1t001d	bounds	interval	8	128	25	10	sgd
s1t002a	class	raw	16	128	25	10	sgd
s1t002b	class	interval	16	128	25	10	sgd
s1t002c	bounds	raw	16	128	25	10	sgd
s1t002d	bounds	interval	16	128	25	10	sgd
s1t003a	class	raw	24	128	25	10	sgd
s1t003b	class	interval	24	128	25	10	sgd
s1t003c	bounds	raw	24	128	25	10	sgd
s1t003d	bounds	interval	24	128	25	10	sgd

Test ID	Data Format		Parameters				
	Pattern	**Pitch**	batch length	hidden size	num epochs	nfold splits	optimiser
s2t001a	class	raw	8	128	25	10	sgd
s2t001b	class	interval	8	128	25	10	sgd
s2t001c	bounds	raw	8	128	25	10	sgd
s2t001d	bounds	interval	8	128	25	10	sgd
s2t002a	class	raw	16	128	25	10	sgd
s2t002b	class	interval	16	128	25	10	sgd
s2t002c	bounds	raw	16	128	25	10	sgd
s2t002d	bounds	interval	16	128	25	10	sgd
s2t003a	class	raw	24	128	25	10	sgd
s2t003b	class	interval	24	128	25	10	sgd
s2t003c	bounds	raw	24	128	25	10	sgd
s2t003d	bounds	interval	24	128	25	10	sgd

Test ID	Data Format		Parameters				
	Pattern	**Pitch**	batch length	hidden size	num epochs	nfold splits	optimiser
s3t001a	class	raw	8	128	25	10	sgd
s3t001b	class	interval	8	128	25	10	sgd
s3t001c	bounds	raw	8	128	25	10	sgd
s3t001d	bounds	interval	8	128	25	10	sgd
s3t002a	class	raw	16	128	25	10	sgd
s3t002b	class	interval	16	128	25	10	sgd
s3t002c	bounds	raw	16	128	25	10	sgd
s3t002d	bounds	interval	16	128	25	10	sgd
s3t003a	class	raw	24	128	25	10	sgd
s3t003b	class	interval	24	128	25	10	sgd
s3t003c	bounds	raw	24	128	25	10	sgd
s3t003d	bounds	interval	24	128	25	10	sgd

Bibliography

Basu, J. K., Bhattacharyya, D., and Kim, T.-H. (2010). "Use of Artificial Neural Network in Pattern Recognition". In: *International Journal of Software Engineering and Its Applications* 4.2, pp. 23–34. URL: https://pdfs.semanticscholar.org/0c2b/61e35b876462d73e2454f62769ecba905dab.pdf.

Bayard, S. P. (1950). "Prolegomena to a Study of the Principal Melodic Families of British-American Folk Song". In: *The Journal of American Folklore* 63.247, pp. 1–44. ISSN: 00218715. URL: http://www.jstor.org/stable/537347.

Bengio, Y., Simard, P., and Frasconi, P. (1994). "Learning Long-term Dependencies with Gradient Descent is Difficult". In: *Trans. Neur. Netw.* 5.2, pp. 157–166. ISSN: 1045-9227. URL: https://ieeexplore.ieee.org/document/279181/.

Benward, B. and Saker, M. N. (2015). *Music in Theory and Practice*. Dubuque, IA: McGraw-Hill Education. ISBN: 007802515X.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. New York, NY: Oxford University Press. ISBN: 0198538642.

Boot, P., Volk, A., and Haas, W. B. de (2016). "Evaluating the Role of Repeated Patterns in Folk Song Classification and Compression". In: *Journal of New Music Research* 45.3, pp. 223–238. URL: https://doi.org/10.1080/09298215.2016.1208666.

Chou, T.-C., Chen, A. L. P., and Liu, C.-C. (1996). "Music Databases: Indexing Techniques and Implementation". In: *Proceedings of the 1996 International Workshop on Multi-Media Database Management Systems (IW-MMDBMS '96)*, p. 46. ISBN: 0-8186-7469-5. URL: http://dl.acm.org/citation.cfm?id=524817.858197.

Collins, T. (2017). *Discovery of Repeated Themes & Sections*. URL: http://www.music-ir.org/mirex/wiki/2017:Discovery_of_Repeated_Themes_&_Sections.

Collins, T., Thurlow, J., Laney, R., Willis, A., and Garthwaite, P. (2010). "A comparative evaluation of algorithms for discovering translational patterns in Baroque keyboard works". In: *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*. URL: http://oro.open.ac.uk/id/eprint/21837.

Conklin, D. (2009). "Melody Classification Using Patterns". In: *2nd International Workshop on Machine Learning and Music*, pp. 37–41. URL: http://www.ehu.eus/cs-ikerbasque/conklin/papers/mml09conklin.pdf.

Conklin, D. (2010). "Discovery of Distinctive Patterns in Music". In: *Intell. Data Anal.* 14.5, pp. 547–554. ISSN: 1088-467X. URL: http://dl.acm.org/citation.cfm?id=1859240.1859243.

Conklin, D. and Bergeron, M. (2008). "Feature Set Patterns in Music". In: *Comput. Music J.* 32.1, pp. 60–70. ISSN: 0148-9267. URL: http://dx.doi.org/10.1162/comj.2008.32.1.60.

Conklin, D. and Witten, I. H. (1995). "Multiple Viewpoint Systems for Music Prediction". In: *Journal of New Music Research* 24.1, pp. 51–73. URL: https://doi.org/10.1080/09298219508570672.

Cook, N. J. (1994). *A Guide to Musical Analysis*. Oxford, New-York: Oxford University Press. ISBN: 9780198165088.

Dannenberg, R. B. (1993). "Music Representation Issues, Techniques, and Systems". In: *Computer Music Journal* 17.3, pp. 20–30. ISSN: 01489267. URL: http://www.jstor.org/stable/3680940.

Downie, J. S., Byrd, D., and Crawford, T. (2009). "Ten Years of ISMIR: Reflections on Challenges and Opportunities". In: *Proceedings of the 10th International Society for Music Information Retrieval Conference (ISMIR 2009)*, pp. 13–18. ISBN: 978-0-9813537-0-8. URL: http://ismir2009.ismir.net/proceedings/keynote1.pdf.

Fidler, S., Boben, M., and Leonardis, A. (2009). "Learning hierarchical compositional representations of object structure". In: *Object categorization: Computer and human vision perspectives*, pp. 196–215. URL: https://doi.org/10.1017/CBO9780511635465.012.

Greff, K., Srivastava, R. K., Koutnik, J., Steunebrink, B. R., and Schmidhuber, J. (2017). "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10, 2222—2232. ISSN: 2162-2388. URL: http://dx.doi.org/10.1109/TNNLS.2016.2582924.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. New York, NY, USA: Cambridge University Press. ISBN: 0-521-58519-8.

Helmholtz, H. von (1913). *Die Lehre von den Tonempfindungen als Physiologische Grundlage fur die Theorie der Musik*. Wiesbaden: Brauschweig. ISBN: 978-3-663-18482-9.

Hochreiter, S. and Schmidhuber, J. (1997). "Long Short-Term Memory". In: *Neural Comput.* 9.8, pp. 1735–1780. ISSN: 0899-7667. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Janssen, B., Haas, W. B. de, Volk, A., and Kranenburg, P. van (2013). "Discovering Repeated Patterns in Music: State of Knowledge, Challenges, Perspectives". In: *Proceedings of the 10th International Symposium on Computer Music Modeling and Retrieval (CMMR 2013)*. URL: http://beritjanssen.com/Texts/Janssen2013a.pdf.

Jones, G. T. (2002). *Music Theory*. New York, NY: HarperResource. ISBN: 0064671682.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). "An Empirical Exploration of Recurrent Network Architectures". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, pp. 2342–2350. URL: http://dl.acm.org/citation.cfm?id=3045118.3045367.

Karpathy, A. (2015). *The Unreasonable Effectiveness of Recurrent Neural Networks*. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Karydis, I., Nanopoulos, A., and Manolopoulo, Y. (2006). "Symbolic Musical Genre Classification Based on Repeating Patterns". In: *Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia*, pp. 53–58. ISBN: 1-59593-501-0. URL: http://doi.acm.org/10.1145/1178723.1178732.

Karydis, I., Nanopoulos, A., and Manolopoulos, Y. (2007). "Finding Maximum-length Repeating Patterns in Music Databases". In: *Multimedia Tools Appl.* 32.1, pp. 49–71. ISSN: 1380-7501. URL: http://dx.doi.org/10.1007/s11042-006-0068-5.

Kim, S., Kim, D., and Suh, B. (2016). "Music Genre Classification Using Multimodal Deep Learning". In: *HCIK 2016: Proceedings of HCI Korea*, pp. 389–395. ISBN: 978-89-6848-791-0. URL: https://doi.org/10.17210/hcik.2016.01.389.

Kingma, D. P. and Ba, J. (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations (ICLR 2015)*. URL: http://arxiv.org/abs/1412.6980.

Klapuri, A. (2010). "Pattern Induction and Matching in Music Signals". In: *Proceedings of the 7th International Symposium on Computer Music Modeling and Retrieval (CMMR 2010)*, pp. 188–204. ISBN: 978-3-642-23125-4. URL: http://dl.acm.org/citation.cfm?id=2040789.2040806.

Kranenburg, P. V., Janssen, B., and Volk, A. (2016). *The Meertens Tune Collections: The Annotated Corpus (MTC-ANN)*. URL: https://www.researchgate.net/publication/310624600_The_Meertens_Tune_Collections_The_Annotated_Corpus_MTC-ANN_Versions_11_and_201.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM* 60.6, pp. 84–90. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/3065386.

Lerdahl, F. and Jackendoff, R. S. (1985). *A Generative Theory of Tonal Music*. Cambridge, MA: MIT Press Cambridge, Mass. ISBN: 0262120941.

Li, M., Chen, X., Li, X., Ma, B., and Vitanyi, P. M. B. (2004). "The Similarity Metric". In: *IEEE Transactions on Information Theory* 50.12, pp. 3250–3264. ISSN: 0018-9448. URL: https://ieeexplore.ieee.org/document/1362909.

Manning, C. D., Raghavan, P., and Schutze, H. (2008). *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press. ISBN: 0521865719.

Meadows, J. and Cronin, B. (2003). "Annual Review of Information Science and Technology (ARIST) Volume 37". In: *Journal of Documentation* 59.4, pp. 494–496. URL: https://doi.org/10.1108/00220410310485848.

Meredith, D. (2015). "Music Analysis and Point-Set Compression". In: *Journal of New Music Research* 44.3, pp. 245–270. URL: https://doi.org/10.1080/09298215.2015.1045003.

Meredith, D. (2018). "Music Analysis and Point-Set Compression". In: *Journal of New Music Research* 44.3, pp. 245–270. URL: https://doi.org/10.1080/09298215.2015.1045003.

Meredith, D., Lemström, K., and Wiggins, G. A. (2002). "Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music". In: *Journal of New Music Research* 31.4, pp. 321–345. ISSN: 09298215. URL: https://www.tandfonline.com/doi/abs/10.1076/jnmr.31.4.321.14162.

Mirka, D. (2010). "Metric Manipulations in Haydn and Mozart: Chamber Music for Strings". In: *Music Perception: An Interdisciplinary Journal* 28.2, pp. 201–204. ISSN: 07307829. URL: https://www.jstor.org/stable/10.1525/mp.2010.28.2.201.

Muller, M. and Clausen, M. (2007). "Transposition-Invariant Self-Similarity Matrices". In: *Proceedings of the 8th International Society for Music Information Retrieval Conference (ISMIR 2007)*, pp. 47–50. URL: http://ismir2007.ismir.net/proceedings/ISMIR2007_p047_mullermuller.pdf.

Nieto, O. and Farbood, M. M. (2014). "Identifying polyphonic musical patterns from audio recordings using music segmentation techniques". In: *Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR 2014)*, pp. 411–416. URL: http://www.terasoft.com.tw/conf/ismir2014/proceedings/T074_150_Paper.pdf.

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: *CoRR* abs/1811.03378, 2222—2232. ISSN: 1811.03378. URL: https://dblp.org/rec/bib/journals/corr/abs-1811-03378.

Olah, C. (2015). *Understanding LSTM Networks*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Pesek, M., Leonardis, A., and Marolt, M. (2014). "Compositional hierarchical model for music information retrieval". In: *PhD Thesis for the University of Ljubljana*, p. 132. URL: http://eprints.fri.uni-lj.si/id/eprint/4257.

Pesek, M., Leonardis, A., and Marolt, M. (2017a). "Robust Real-Time Music Transcription with a Compositional Hierarchical Model". In: *PLOS ONE* 12.1, pp. 1–21. URL: https://doi.org/10.1371/journal.pone.0169411.

Pesek, M., Leonardis, A., and Marolt, M. (2017b). "SymCHM-An Unsupervised Approach for Pattern Discovery in Symbolic Music with a Compositional Hierarchical Model". In: *Applied Sciences* 7.11, pp. 1–21. URL: https://www.mdpi.com/2076-3417/7/11/1135.

Pohlmann, K. C. (2000). *Principles of Digital Audio*. New York, NY: McGraw-Hill Professional. ISBN: 0071348190.

Reidma, D. and Akker, R. op den (2008). "Exploiting Subjective Annotations". In: *Coling 2008: Proceedings of the workshop on Human Judgements in Computational Linguistics*, pp. 8–16. ISBN: 978-1-905593-47-7. URL: http://dl.acm.org/citation.cfm?id=1611628.1611631.

Ren, I. Y., Volk, A., Swierstra, W., and Veltkamp, R. C. (2018a). "Analysis by Classification: A Comparative Study of Annotated and Algorithmically Extracted Patterns in Symbolic Music Data". In: *Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018, Paris, France, September 23-27, 2018*, pp. 539–546. ISBN: 978-2-9540351-2-3. URL: http://ismir2018.ircam.fr/doc/pdfs/75_Paper.pdf.

Ren, I. Y., Janssen, B., and Collins, T. (2018b). *Patterns for Prediction*. URL: https://www.music-ir.org/mirex/wiki/2018:Patterns_for_Prediction.

Rolland, P.-Y. (1999). "Discovering Patterns in Musical Sequences". In: *Journal of New Music Research* 28.4, pp. 334–350. URL: https://www.tandfonline.com/doi/abs/10.1076/0929-8215.

Sadie, S. and Tyrrell, J. (2002). *The New Grove dictionary of Music and Musicians*. New York, NY: Grove. ISBN: 1-56159-239-0.

Schaffrath, H. (1997). "Beyond MIDI". In: ed. by E. Selfridge-Field. Cambridge, MA, USA: MIT Press. Chap. The Essen Associative Code: A Code for Folksong Analysis, pp. 343–361. ISBN: 0-262-19394-9. URL: http://dl.acm.org/citation.cfm?id=275928.275972.

Schedl, M., Gómez, E., and Urbano, J. (2014). "Music Information Retrieval: Recent Developments and Applications". In: *Foundations and Trends® in Information Retrieval* 8.2–3, pp. 127–261. ISSN: 1554-0669. URL: http://dx.doi.org/10.1561/1500000042.

Schoenberg, A. (1999). *Fundamentals of Musical Composition*. New York, NY, USA: Faber and Faber. ISBN: 9780571196586.

Selfridge-Field, E. (1997). *Beyond MIDI: The Handbook of Musical Codes*. Cambridge, MA: The MIT Press. ISBN: 0262193949.

Simonyan, K. and Zisserman, A. (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556, pp. 84–90. ISSN: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

Swierstra, W., Ren, I. Y., Nieto, O., Koops, H. V., and Volk, A. (2018). "Investigating Musical Pattern Ambiguity in a Human Annotated Dataset". In: *International Conference on Music Perception and Cognition/European Society for the Cognitive Sciences of Music*. URL: https://dspace.library.uu.nl/handle/1874/373437.

Szeto, W. M. and Wong, M. H. (2006). "A graph-theoretical approach for pattern matching in post-tonal music analysis". In: *Journal of New Music Research* 35.4, pp. 307–321. ISSN: 0929-8215. URL: https://doi.org/10.1080/09298210701535749.

Tsaptsinos, A. (2017). *Optimizing the Shazam backend structure via Genetic Algorithms*. URL: https://blog.shazam.com/optimizing-the-shazam-backend-structure-via-genetic-algorithms-4ade88898972.

Volk, A. and Kranenburg, P. van (2012). "Melodic similarity among folk songs: An annotation study on similarity-based categorization in music". In: *Musicae Scientiae* 16, pp. 317–339. URL: https://pure.knaw.nl/ws/files/479778/AnnotationArticle_final_submission.pdf.

Young, R. W. (1939). "Terminology for Logarithmic Frequency Units". In: *The Journal of the Acoustical Society of America* 11.1, pp. 134–139. URL: https://doi.org/10.1121/1.1916017.