Using Recurrent Neural Networks to Generate Music

Liam Devlin

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Information and Library Studies

DEPT. OF COMPUTER AND INFORMATION SCIENCES UNIVERSITY OF STRATHCLYDE

August 2019

Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive. (please tick) Yes [ ] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 14800 words.

I confirm that I wish this to be assessed as a Type 5 dissertation.

Signature:

Date:

Abstract

This project focuses on the application of Recurrent Neural Networks (RNN) to the field of generative symbolic music. The following paper discusses the current state of RNN music generation and proposes two representations for compositions with a shared underlying data structure. Once the compositions had been generated, they were compared against human-written compositions by using an empirical MIR analysis system, as well as a subjective test in which participants offered perceptual responses to the compositions. Overall, while the system was able to produce some melodically pleasant results, it failed to capture the long term structure and complexity of human compositions.

Acknowledgements

I would like to thank my amazing partner, Rachel, for her incredible support and unwavering patience during my studies: I truly could not have done this without your support. I'd also like to thank my Mum for being incredibly understanding during this project, as well as my good friend Thor who was always willing to listen to my problems, moans and groans, and motivated me to continue my work.

Table of Contents

1. **Introduction** 7
2. **Aims** 10
3. **Background / Key Literature** 11
   - *3.1 Recurrent Neural Networks* 11
   - *3.2 Music Theory and Data Representation* 15
   - *3.3 Current State of Generative Music* 19
   - *3.4 Computational Music Analysis* 20
   - *3.5 Subjective Music Analysis* 24
4. **Methodology** 26
   - *4.1 Dataset Selection and Preparation* 26
   - *4.2 Designing Structure of Neural Network(s)* 37
   - *4.3 Synthesising Symbolic Data* 43
   - *4.4 Computational Music Analysis* 48
   - *4.5 Subjective Music Analysis* 52
5. **Analysis of the System** 54
   - *5.1 Analysis of Network Training* 54
   - *5.2 Symbolic Analysis* 55
   - *5.3 Subjective Analysis* 67
6. **Conclusion** 72
7. **Appendices** 74

List of Figures

Fig No.	*Figure Description*	*Page No.*
3.1.1	Recurrent Neural network architecture compared against feed-forward neural network	11
3.1.2	Unrolling a recurrent neuron in RNN	13
3.2.1	Variety of Musical Time Signatures	16
3.4.1	List of jSymbolic 2.2 MIR Descriptors	23
4.1.1	loadMidiData function	28
4.1.2	Function to retrieve keras data format	29
4.1.3	Method A network data processing function	31
4.1.4	Method B network data processing function	33
4.1.5	Function to generate random comparison data	36
4.2.1	Method A Network Architecture	40
4.2.2	Method B Network Architecture	41
4.3.1	Function to convert Method A data back to MIDI	43
4.3.2	Function to convert Method B data back to MIDI Pt.1	44
4.3.3	Function to convert Method B data back to MIDI Pt.2	45
4.3.4	Function to convert Method B data back to MIDI Pt.3	46
4.3.5	Ableton Live 9 User Interface	47
4.4.1	jSymbolic2 User Interface	48
4.4.2	Contrary Motion	49
4.4.3	Similar Motion	49
4.4.4	Chromatic Motion	50
4.4.5	Stepwise Motion	50
4.5.1	Listening Questionnaire Predisposition	52

4.5.2	Listening Questionnaire Sample Question	52
5.1.1	Network Training Period Loss	54
5.1.2	Network Validation Loss	54
5.1.3	Final Hyperparameters for both networks	55
5.2.1	Amount of Arpeggiation Comparison	56
5.2.2	Average Note Duration Comparison	57
5.2.3	Average Simultaneous Pitches Comparison	58
5.2.4	Chromatic Motion Comparison	59
5.2.5	Contrary Motion Comparison	60
5.2.6	Stepwise Motion Comparison	61
5.2.7	Similar Motion Comparison	62
5.2.8	Repeated Notes Comparison	63
5.2.9	Melodic Embellishments Comparison	64
5.2.10	Melodic Pitch Variety Comparison	65
5.2.11	Variability of Note Durations Comparison	65
5.2.12	Variation of Dynamics Comparison	66
5.3.1	Participants Skill Level	67
5.3.2	Participants Skill Statistics	68
5.3.3	Participant Accuracy	68
5.3.4	Participants Perceived Complexity	69
5.3.5	Participants Perceived Enjoyment	69
5.3.6	Participant Replayability	70
6.1	Sample Method A Composition in Music Notation	74
6.2	Sample Method B Composition in Music Notation	74

1. Introduction

Neural networks belong to a family of computer algorithms known as Machine Learning algorithms, which attempt to automate a solution to a problem. Machine learning tasks usually concentrate on tasks that are either extremely tedious for programmers, or where the time required to implement the algorithm traditionally is unfeasible. With neural networks, the computer is fed massive quantities of data to "train" the network, which is to say, the network tries to formulate a pattern from the data, such that it is able to predict unseen classes/categories, or generate new data based on *some* input. Neural networks specifically have seen a massive rise in use due in large part to the advent of extremely sophisticated Graphics Processing Units (GPUs). Originally intended for video games, modern GPUs often contain thousands of cores, all of which are specifically engineered to perform vector and matrix algebra extremely quickly: structures fundamental to the operation of neural networks (Paine et al, 2013). From the advent of this newfound power, neural networks have been applied to a variety of problems, including: image classification, feature extraction from audio signals, time-series analysis of financial data and much more. With this, a variety of neural network architectures have emerged, many of which have been fine-tuned for a particular area of application. Convolutional neural networks have seen widespread use in image classification due to the similarity in structure to that of the visual cortex; feed-forward neural networks are some of the most simple ANN architectures and are generally quite accurate at most machine learning tasks, including image processing and natural language processing (Graves et al, 2013) tasks which, until recently, were among the most challenging for traditional algorithms to approach.

One traditionally challenging problem, however, is the generative composition of music. Studies by (Eck & Schmidhuber, 2002) and (Franklin, 2006) have shown traditional neural network architectures to be unfit for generating music due to their inability to effectively deal with time-series data, that is, data where the ordering of data-points is important. In music the order of notes is fundamental to giving music a 'human' feel,

and in most music, many abstractions of ordering take place. In music theory and notation, a bar (also known as a measure) is a series of notes which can be loosely thought of as a complete musical phrase. Bars have sub-divisions of time known as “beats” which vary based on the time-signature of the piece. Time signatures have the largest impact on the rhythmic “feel” of the track and inform the order and spacing between notes for at least the remainder of that bar (though in most non-classical music, compositions will stick to one or a small number of time-signatures). From here further abstractions may be applied to the composition’s temporal structure, a verse-chorus song structure may be composed of a number of separate, yet closely related bars of music, which are ordered in such a way as to have an emotional impact in the listener.

Recurrent Neural Networks (RNN) are neural networks that are able to keep track of information from a previous timestep, and thus have some rudimentary form of memory. This is useful for keeping track of so called ‘long term dependencies’, take for example a motif from a classical piece, the composition may be over an hour long, however specific themes and phrases from parts of the music are reused for dramatic and emotional effect (Sutskever et al, 2014).

Another problem generative music currently faces is that while current solutions may be aware of the *form* of music, e.g. they are able to capture the basic structure of music, they are not however able to capture the semantics of music. (Langston, 1988) describes the semantic meaning of a composition as being influenced by “experiential, cultural and historic information (the result of activities we loosely call experience, acculturation and learning) that does not yet exist in any machine manipulable form”.

Analysis of music (whether a generative or bespoke composition) is a challenging task for computers. Basic statistics about the composition, such as it’s key or tempo are relatively easy to extract with modern tools (Antti & Eronen, 2017), and

can even be extended to work out more involved properties of a composition, such as it's time-signature or key changes that occur throughout. Empirically analysing the emotional impact or perceived 'quality' of a piece of music is extremely challenging (if not impossible) due in no small part to research which suggests that human's assessment of music and their stylistic preferences are based almost entirely on individual cognitive ability and past life experiences (Krumhansl, 2002). Theories such as the Generative Theory of Tonal Music (GTTM) (Lerdahl and Jackendoff, 1983) have made strides in making meaningful semantic analysis of music a reality. GTTM is different from most other music analyses in that it constructs a representation of the composition that is informed by music theory, but also by the listeners cognitive behaviour when listening to music (Jackendoff, 1994).

Based on these factors, a reasonable assumption could be made that a neural network with strong time-domain capabilities could prove insightful, if not useful, in the field of generative music. Though a rather niche area of computing, generative music sees widespread use in the video-game sector and can lead to a more optimal experience for the consumer, as they are able to make actions which will affect the music, thus increasing player agency (Collins, 2009). Artists such as Brian Eno (who originally coined the phrase "Generative Music") have also shown interest in the possibilities of this technique , and its application in musical performances, specifically avant-garde performances (Darko, 2009). As such, this project will discuss the experimental design of a RNN system which generates music.

2. Aims

2.1 The overall aim of this project is to develop a Recurrent Neural Network that will be capable of producing “human” sounding music. To achieve this system, the following aims must be met:

- Investigate the current state of Recurrent Neural Networks in the field of music generation. This will involve analysing the strengths and weaknesses of currently existing generative music systems using RNNs, with particular attention paid to the structure of the generated songs.
- Develop and train a RNN with the aim of generating harmonious and temporally structured music.
- Subjectively and empirically assess the quality of generated music.

2.2 The resources required to undertake this project are as follows:

- Computer with access to Python, Tensorflow, jSymbolicand Keras. A laptop with a capable GPU will help speed up the training of the neural network significantly.
- Music dataset for training, in MIDI file format.
- Access to human participants for subjective analysis of generated music.

3. *Background / Key Literature*

In order to properly and effectively build the proposed neural network model, it is of vital importance to review the existing body of RNN, generative music and surrounding material. The following subjects were identified as the most relevant to the development of the system.

3.1 *Recurrent Neural Networks*

Neural networks are a family of machine-learning algorithms which have seen widespread adoption in the advent of new-found computer processing speed, with applications in the field of deep learning and medical research (Paine et al, 2013). Some of the most common neural network architectures are Feed-forward networks and Convolutional Neural Networks: the former is among the most primitive neural network configuration and is often used as a naive starting point, while convolutional neural networks see extensive use in image recognition applications(Krizhevsky, Sutskever and Hinton, 2012). One of the fundamental flaws of both Feed-Forward and Convolutional neural networks is that they are unable to deal with time-series data, which is any data in which the ordering of data points is important (Franklin, 2006). Recurrent Neural Networks differ from other Neural Network architectural styles in that the output of each neuron at any given time-step is fed back into the neuron at the next timestep, meaning the summed input of the neuron is based on the current incoming input, as well as the output of the neuron at the last time-step (Bown and Lexer, 2006)

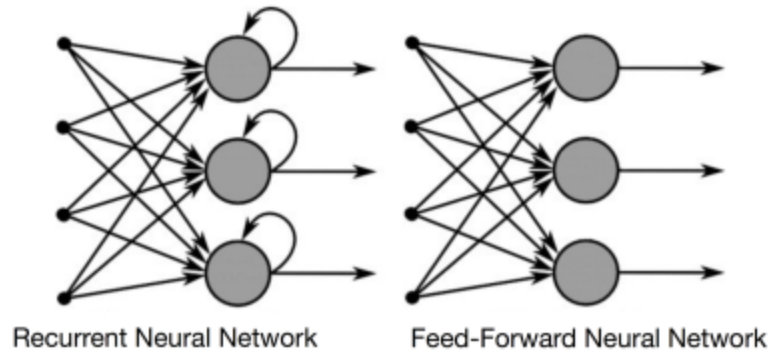


Figure 3.1.1 *Recurrent Neural Network architecture compared against Feed Forward neural network.*

Figure 3.1.1 illustrates the difference between a recurrent neural network and a feed forward neural network. In feed forward networks, the output from each layer *only* moves forward to the next layer until it eventually reaches the output terminal. This means that the network is completely unaware of the output from the last timestep, not to mention the networks state. This makes feed-forward networks sub-optimal for music as they are not able to reproduce or capture any kind of long-term musical structure. Recurrent neural networks help to alleviate this problem by taking the output from each layer at time t and feeding it back into itself at time t+1 . This means that for all recurrent neurons in the layer (more on this shortly) the neuron is aware of the current incoming input, as well as the output from this neuron from the previous time step. (Franklin, 2006).

Traditionally, Recurrent Neural Networks have proven challenging to train due in no small part to the *vanishing* and *exploding* gradient problem. In short, the exploding gradient problem describes the situation where long-term temporal dependencies cause massive updates to the weights of the network, leading to unstable training, or weights with NaN values. The vanishing gradient problem describes precisely the opposite problem, where many short-term dependencies can cause the loss curve to become so small that it is unable to make any effective change to the network and causes the rate of learning to slow to a crawl (Pascanu et al, 2013). The Backpropagation algorithm used to train nearly all neural networks is in itself unfit for training RNNs due to its inability to backpropagate errors from multiple time-steps. Backpropagation Through Time (BPTT) allows the fundamental backpropagation algorithm to be applied to RNNs by “unrolling” the network. As can be seen from Figure 3.1.2, unrolling a recurrent network involves representing each recurrent neuron as a series of neurons (the length of the series is equal to the number of timesteps), thus explicitly representing the weight of a neuron at each timestep. This process is applied to the entire network, with the result of this process being the original RNN represented

as a feedforward neural network, allowing traditional backpropagation to be performed on the networks weights (Saon et al, 2014).

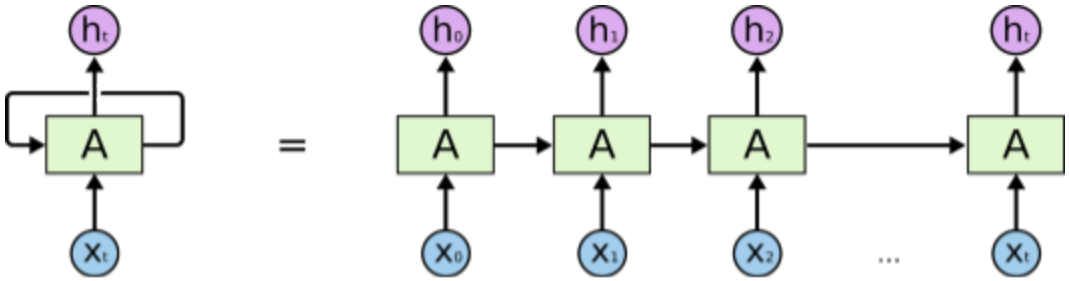


Figure 3.1.2 *Unrolling a recurrent neuron in a recurrent neural network,*

One major flaw with BPTT however, is that it is prone to becoming stuck in local optima during training. This proves problematic in its application to training recurrent networks as by their nature of repeating input from previous time-steps, RNNs naturally also suffer from the problem of local optima (Cuellar et al , 2006). Were this problem left unfixed for the proposed system, at some point in the generation of a composition the network would converge on a single note and repeat indefinitely. A problem that more generally plagues RNNs is the issue of so called “long term dependencies”. Take, for instance, an RNN solving a NLP problem of predicting the next word in a given sentence; for the sentence “I am from france, I speak _” , the obvious answer to a human is ‘french’ as *most* humans will understand that people who live in france, must by extension speak french. For a standard recurrent neural network however, there is no way for the network to intrinsically understand the significance of the word “france” in the context of the sentence, and a poorly trained RNN may not even recognise that the next word should be some kind of language, or even a word relating to france at all. Both of these problems are addressed by LSTM units.

LSTM networks use a special kind of recurrent connection which significantly aids an RNNs ability to recognise and deal with long-term dependencies. The core principle of an LSTM cell is that each cell maintains some “state” which is consistent between

timesteps. The inner workings of the LSTM cell state are extremely complex and fall outwith the scope of this project, but an LSTM cell state can loosely be thought of as a conveyor belt which flows through all-timesteps of an LSTM cell, carrying different configurations at each point in time. At each timestep, the LSTM can be trained to either remember or forget part of the information it is currently holding - this modified information then becomes the state for the next timestep (Gers, Schmidhuber and Cummins, 2000). The other components of an LSTM cell are used purely to control what data is modified in the cell's state. The forget gate dictates how much of the current state should be discarded to allow for new information to flow in. As its output, the forget gate return a number in the range 0 to 1, with 0 representing "forget nothing" and 1 representing "forget everything". The input gate provides the opposite behaviour, dictating how much of the new input data should be fetched and stored in the cells state (Olah, 2015). A simpler alternative to LSTM networks exist in GRU (Gated Recurrent Unit) networks. GRU units have only two gates: the forget gate - which decides how much of the current state should be forgotten, and the update gate - which dictates how much of the current incoming activation should overwrite the previous timestep's activation. The most salient difference between GRUs and LSTMs is that GRUs always expose the full state during updates, where LSTM units are able to control the degree to which the state is exposed (Chung et al., 2014). By allowing the network to retain information over long periods of time as well as being able to tune how much information should be forgotten and retained at each time-step, both LSTM and GRU networks are able to mitigate the problem of long-term dependencies as well as helping to deal with both vanishing and exploding gradients.

3.2 Music Theory and Data Representation

In approaching a generative music application, it is of crucial importance to consider at least a small amount of music theory; this section will primarily discuss western music theory as the training data is exclusively comprised of western music, however it should be noted that there are slight variations in music theory throughout different parts of the world.

A note is a single musical excitement which has only 2 salient features: Pitch and Duration. Pitch describes the tonality of the musical excitement, and is measured (in western music theory) using the chromatic scale. This is a musical scale which represents a mapping of Note Pitches (often simply referred to as notes) to various frequencies, for example, take the note pitch A4, which lies at 440hz. The chromatic scale is composed of 12 evenly-spaced pitch classes over 8 octaves, hence 'A4' meaning the pitch-class 'A' at octave 4. Increasing the octave of a note doubles the note's current frequency, e.g. A4 = 440Hz and A5 = 880Hz. The chromatic scale is used as the basis for western musical scales due to its versatility, stability across many octaves and wide-range of harmonious possibilities (Schellenberg et al., 2005).

The most fundamental collection of notes which can be considered a complete musical passage is a bar. Each bar may have an arbitrary number of notes which compose a musical phrase, however it is primarily the time signature (more on this shortly) which informs the rhythmic properties of the musical expression. Note durations are relative to the tempo (ordinarily measured in beats-per-minute), and have predetermined intervals to denote various durations in a readable format. The most common note durations are a crotchet, quaver and semi-brieve which represent ¼ , ⅛ and whole note lengths. It is impossible to explain the meaning of these time designations without first explaining the concept of a time-signature, which in western music theory, is the primary method of expressing the rhythmic properties of a musical phrase.

Figure 3.2.1: Variety of musical time signatures, written as they would appear in western music notation.

As can be seen from Figure 3.2.1, time signatures are represented by two numbers, which we will refer to as the enumerator and denominator for the sake of ease. The enumerator refers to the number of pulses (more commonly referred to as beats) per bar, while the denominator refers to the value - or duration - of each beat. The tempo and time signature's denominator are the primary methods for informing the reader of the "speed" of a musical phrase.

In order to achieve the goal of "human feeling" music, the system would ideally understand the distinction between sections of music; these segments of music are usually broken down into verse, chorus and bridge components (Sloboda, 1991). While this seems extremely simple, most popular music is written exclusively within the confines of this structure denoted "verse-chorus form", with minor variations introduced in each section to minimise the repetition and elicit an emotional response from the listener (Doll, 2011). With this being said, there are vastly more complicated forms of music such as freeform jazz, which can almost be said to contain no discernable structure. There are also many examples of popular music which do not conform to the verse-chorus form, but due to its prevalence in music and its relatively simple rules, the verse-chorus form will be used as the basis for all general structure we wish the system to be able to replicate.

Beyond the basic music theory involved in designing a generative music system, a huge consideration in any neural network system is the form of the representation.

One of the most common ways to digitally represent music is MIDI (Musical Instrument Digital Interface). MIDI does not seek to recreate the exact sound of a piece of music, MIDI instead captures musical events for an arbitrary number of instruments, which can be layered within a single file. An example of an event would be a “note on” event, which would correspond to a given instrument, and would be passed through a note (or pitch) to play at a given velocity, e.g. how hard the note was played. MIDI was originally developed in 1983 with the goal of allowing synthesizers from multiple different companies to communicate; it has since become a ubiquitous way of storing music. It has since seen use in MIR (Music Information Retrieval) domains as a supplementary asset to a given musical signal, for the purposes of both musical feature extraction and classification (Cataltepe et al, 2005).

In MIDI, a note’s pitch is represented as a number in the range 0 to 127. For example, MIDI note 72 represents the note C in the 5th octave, which is approximately equivalent to 524Hz. MIDI notes are modelled to represent all available notes on the chromatic scale, which means that certain musical styles such as microtonal music cannot be accurately represented using MIDI. Due to this limitation, MIDI is primarily used to represent western music as it simply lacks the infrastructure to effectively deal with non-western pitch representations. Velocity describes how hard a note was excited by an instrument, and can be loosely thought of as a measure of loudness and dynamics. The velocity value is also ranged 0-127, with 0 representing no excitement at all, and 127 representing the maximum volume for a note. For this project, the velocity will primarily be thought of as a loudness metric (as this project only seeks to generate music for a piano), however for other instruments, velocity impacts not only the volume of the sound, but the way the excitement sounds tonally. Take, for example, a drum kit; while increasing velocity will make the drum hits louder, this will also cause the drum hits to sustain longer, and potentially even change the tone depending on how hard the drum was hit (Dannenberg, 2006).

In terms of representation, MIDI is often used as a starting point for data preparation, as all of the information required for the network can either be directly sampled from the MIDI file, or extrapolated from the existing data. Common to all representations is the inclusion of Pitch and Velocity as features. Time features such as note duration and start are usually less consistently defined, but can generally be split into one of two approaches: relative timing and absolute timing. In relative timing, the note's start time (not duration) is relative to the start time of the last note, in absolute timing , the note's start time is the absolute time it should start, in seconds (Eck and Schmidhuber, 2002b)

Tokenization is a process ordinarily used in Natural Language Processing (NLP) applications, in which a dataset is encoded into a symbolic representation where each symbol represents some *facet* of a dataset, for instance, words in a NLP neural network. Tokenization makes use of a concept known as a 'dictionary' which holds all of the tokens and the original data they represent. Dictionaries are almost always a fixed length, meaning some of the original data may no longer be representable; this is a necessary sacrifice as large dictionaries can lead to exponential increases in training time, or the possibility of divergence if the given dictionary is sufficiently large (Luong et al., 2015). In NLP applications, data usually takes the form of a list of n-length vectors of integers (with each integer representing a word from the dictionary, and thus the vector representing a sentence). Tokenization has seen application to sentiment analysis of user reviews (Maas et al., 2011), real-time machine translation (Devlin et al., 2014) as well as password security analysis (Melicher et al., 2016). The application of this approach to generative music is not a novel concept: a study which used a Transformer decoder to generate music also used a tokenized symbolic representation with extremely promising results (Cheng-Zhi et al., 2018). Another study proposes fundamental similarities between language and music, specifically with regard to metrical structure and phrasing (Jackendoff, 2009).

3.3 Current State of Generative Music

There have been many successes of the application of RNNs to a particular subset of a musical piece, such as melody composition (Chen & Miikkulainen, 2001), drum grooves (Hutchings, 2017) and even small segments of music using a multi-layer system which trains one network on chord structures, and another to compose melody within the confines of the generated chord structure (Eck and Schmidhuber, 2001).

One potential problem highlighted by (Eck and Schmidhuber, 2002a) is that many neural networks use multiple time-steps to one note, which needlessly complicates both the network as a whole and the training process. The same study also suggests for the sake of simplicity and efficiency: a predetermined note distribution should be established for the entire dataset, e.g. notes in a song will have a minimum distribution of pitch (jumps in 3 semitones, rather than 1 semitone for the chromatic scale). This was seconded by (Franklin, 2006) who also suggested that a traditional mapping of pitch, e.g. 12 notes per scale and 8 octaves of notes, may be a poor representation for pitch, with respect to training accuracy. This was, however, disproven by (Liu and Ramakrishnan, 2014) who successfully trained a model using data which took the form of a 2D vector of pitch activations over many timesteps.

Another approach highlighted by (Skuli, 2017) takes a simultaneously simpler and more sophisticated approach to representation. This approach removes all timing features with a predetermined interval between every note, drastically limiting the expressive capabilities of the network. The network does however distinguish between a single note as a musical excitement and a chord,

One significant challenge still facing generative music solutions is capturing the so-called “long-term structure” of music, which can be thought of as the culmination of long-term dependencies which govern the style of a particular composition (Eck and Schmidhuber, 2002b). One recent approach looks at tackling the problem of ‘long-term

structure' through the use of a Transformer Decoder network, which significantly improved the quality of prediction compared to other state-of-the-art models, specifically with regard to note duration and timing. This representation for this approach utilised a symbolic approach for pitch and velocity, where note duration and activation were dictated by a piano-roll like time-series structure (Cheng-Zhi et al., 2018).

3.4 Computational Music Analysis

A significant challenge in computationally analysing music, and its perceived quality, is that a human's appreciation for music is hugely dependent on their life and the context in which they listen to the piece (Ariza, 2009). As such, music's perceived quality can be thought of as a culmination of the listener's emotional response and their cognitive ability / experiences (Krumhansl, 2002). Some formative work by (Lerdahl and Jackendoff, 1983) attempts to disambiguate a musical piece into the way humans purportedly perceive music. Labelled GTTM (Generative Theory for Tonal Music), this technique is intended as a rule-set that governs the generation of music (Dodd, 2015) but has also been applied to perceptual music analysis systems, and works by splitting the music into 4 discrete components: the prolongonal tree, time-span tree, metrical structure and grouping structure (Eck and Schmidhuber, 2002a). *Grouping structure* describes the relationship between a grouping of notes (or a phrase), the *metrical structure* describes the rhythmic feel of a phrase, the *time-span tree* is a hierarchical tree which allocates an importance to a given note in a phrase, and finally, the *prolongonal tree* expresses the notion of tension and release within a given section. As both a method of analysis and generation, GTTM has amazing potential to generate music which mimics human emotion. One major caveat is that no standard implementation of GTTM exists and as such, must be implemented by the programmer.

An approach by (Hamanaka et al, 2015) highlighted many potential issues with GTTM, not least of which was the lack of a formal implementation for computers, as well as

many of the proposed rules (specifically how the music generation is started) being ambiguously described. The approach Hamanaka et al took was to take a sheet music representation (similar to that provided by Sibelius or other music authoring software), and apply a subset of GTTM recursively over increasing sizes of musical phrase. This technique denoted ATTA, or Automatic Timespan Tree Analyzer (Hamanaka et al, 2015), looks at a small phrase of music and recursively looks at its position, relative to a larger and larger subsection of music. This allows the analysis system to be more perceptually aware of the intended tone of a given phrase; for example, many happy songs will have minor chords, or even minor sections which provide a different *feel* to the music, without necessarily making the song sound inherently '*sad*'.

Unfortunately, no solid computational implementation for GTTM analysis yet exists. As noted by (Hamanaka, Hirata and Tojo, 2007), GTTM has proven difficult to interpret as a computational algorithm due to vague rules in the original specification, and as such, very few applications have made attempts to use GTTM for analysis. For this system, analysing the emotional response of a listener to the generated compositions is out of scope, but some method of analysing the musicality of a generated piece is essential to ascertain if the system has been able to produce a pleasant musical composition.

Computational Musicology (CM) is a multidisciplinary subject which aims to combine computer science and music theory; it covers a broad range of well-documented research areas, such as mathematical music theory, music information retrieval (MIR) and computer generated music (Volk, Wiering and van Kranenburg, 2011). For this project, we are primarily interested in the analytical aspects of CM and will be focussing on MIR, as it has seen the most widespread adoption (and therefore, development) following the mass digitisation of popular music in the late 1990s and early 2000s. MIR has a wide variety of applications, from forming music recommendations based on a corpus of existing music, to automatic music transcription (Klapuri and Davy, 2011)**.** Two main approaches exist for MIR: Signal-based MIR, which operates on the raw

audio signal (or signal files such as .wav, .mp3), and symbolic MIR which operates on binary formats such as MIDI (Li et al., 2017). This project will focus on symbolic MIR, as our output representation will be a binary MIDI file. A small handful of MIR toolkits exist for a variety of languages including music21 for python (Cuthbert and Ariza, 2010), Essentia for C++ and Python (Bogdanov et al., 2013), and the framework this project will utilise, jSymbolic.

jSymbolic is part of a collection of MIR tools known as jMIR; it is a symbolic MIR tool which operates on MIDI and MusicXML data, providing a massive collection (see Figure 3.4.1) of musical descriptors such as: pitch and harmony, melodic intervals, rhythmic intervals, musical texture and has seen use in a variety of MIR studies (Raffel, 2016) (Odekerken, 2018) (Wickland, Calvert and Harley, 2018). The framework was originally intended with the goal of automating user suggestions and classification of music; more recently, however, the framework has seen use in the field of Machine Learning (Mckay, Cumming and Fujinaga, 2018) as well as Musicology, where researchers used jSymbolic analysis data to predict the emotion of a composition (Lu et al, 2010)

Overall Pitch Statistics
(128) Basic Pitch Histogram
(12) Pitch Class Histogram
(12) Folded Fifths Pitch Class Histogram
Number of Pitches
Number of Pitch Classes
Number of Common Pitches
Number of Common Pitch Classes
Range
Importance of Bass Register
Importance of Middle Register
Importance of High Register
Dominant Spread
Strong Tonal Centres
Mean Pitch
Mean Pitch Class
Most Common Pitch
Most Common Pitch Class
Prevalence of Most Common Pitch
Prevalence of Most Common Pitch Class
Relative Prevalence of Top Pitches
Relative Prevalence of Top Pitch Classes
Interval Between Most Prevalent Pitches
Interval Between Most Prevalent Pitch Classes
Pitch Variability
Pitch Class Variability
Pitch Class Variability After Folding
Pitch Skewness
Pitch Class Skewness
Pitch Class Skewness After Folding
Pitch Kurtosis
Pitch Class Kurtosis
Pitch Class Kurtosis After Folding
Major or Minor
First Pitch
First Pitch Class
Last Pitch
Last Pitch Class
Glissando Prevalence
Average Range of Glissandos
Vibrato Prevalence
Microtone Prevalence

Melodic Intervals
(128) Melodic Interval Histogram
Most Common Melodic Interval
Mean Melodic Interval
Number of Common Melodic Intervals
Distance Between Most Prevalent Melodic Intervals
Prevalence of Most Common Melodic Interval
Relative Prevalence of Most Common Melodic Intervals
Amount of Arpeggiation
Repeated Notes
Chromatic Motion
Stepwise Motion
Melodic Thirds
Melodic Perfect Fourths
Melodic Tritones
Melodic Perfect Fifths
Melodic Sixths
Melodic Sevenths
Melodic Octaves
Melodic Large Intervals
Minor Major Melodic Third Ratio
Melodic Embellishments
Direction of Melodic Motion
Average Length of Melodic Arcs
Average Interval Spanned by Melodic Arcs
Melodic Pitch Variety

Chords and Vertical Intervals
(128) Vertical Interval Histogram
(12) Wrapped Vertical Interval Histogram
(11) Chord Type Histogram
Average Number of Simultaneous Pitch Classes
Variability of Number of Simultaneous Pitch Classes
Average Number of Simultaneous Pitches
Variability of Number of Simultaneous Pitches
Most Common Vertical Interval
Second Most Common Vertical Interval
Distance Between Two Most Common Vertical Intervals
Prevalence of Most Common Vertical Interval
Prevalence of Second Most Common Vertical Interval
Prevalence Ratio of Two Most Common Vertical Intervals
Vertical Unisons
Vertical Minor Seconds
Vertical Thirds
Vertical Tritones
Vertical Perfect Fourths
Vertical Perfect Fifths
Vertical Sixths
Vertical Sevenths
Vertical Octaves
Perfect Vertical Intervals
Vertical Dissonance Ratio
Vertical Minor Third Prevalence
Vertical Major Third Prevalence
Chord Duration
Partial Chords
Standard Triads
Diminished and Augmented Triads
Dominant Seventh Chords
Seventh Chords
Non-Standard Chords
Complex Chords
Minor Major Triad Ratio

Rhythm
(2) Initial Time Signature
Simple Initial Meter
Compound Initial Meter
Complex Initial Meter
Duple Initial Meter
Triple Initial Meter
Quadruple Initial Meter
Metrical Diversity
Total Number of Notes
Note Density per Quarter Note
Note Density per Quarter Note per Voice
Note Density per Quarter Note Variability
(12) Rhythmic Value Histogram
Range of Rhythmic Values
Number of Different Rhythmic Values Present
Number of Common Rhythmic Values Present
Prevalence of Very Short Rhythmic Values
Prevalence of Short Rhythmic Values
Prevalence of Medium Rhythmic Values
Prevalence of Long Rhythmic Values
Prevalence of Very Long Rhythmic Values
Prevalence of Dotted Notes
Shortest Rhythmic Value
Longest Rhythmic Value
Mean Rhythmic Value
Most Common Rhythmic Value
Prevalence of Most Common Rhythmic Value
Relative Prevalence of Most Common Rhythmic Values
Difference Between Most Common Rhythmic Values
Rhythmic Value Variability
Rhythmic Value Skewness
Rhythmic Value Kurtosis
(12) Rhythmic Value Median Run Lengths Histogram
Mean Rhythmic Value Run Length
Median Rhythmic Value Run Length
Variability in Rhythmic Value Run Lengths
(12) Rhythmic Value Variability in Run Lengths Histogram
Mean Rhythmic Value Offset
Median Rhythmic Value Offset
Variability of Rhythmic Value Offsets
Complete Rests Fraction
Partial Rests Fraction
Average Rest Fraction Across Voices
Longest Complete Rest
Longest Partial Rest
Mean Complete Rest Duration
Mean Partial Rest Duration
Median Complete Rest Duration
Median Partial Rest Duration
Variability of Complete Rest Durations
Variability of Partial Rest Durations
Variability Across Voices of Combined Rests
(161) Beat Histogram Tempo Standardized
Number of Strong Rhythmic Pulses - Tempo Standardized
Number of Moderate Rhythmic Pulses - Tempo Standardized
Num. Relatively Strong Rhythmic Pulses - Tempo Standardized
Strongest Rhythmic Pulse - Tempo Standardized
Second Strongest Rhythmic Pulse - Tempo Standardized
Harmonicity of Two Strongest Rhythmic Pulses - Tempo Stand.
Strength of Strongest Rhythmic Pulse - Tempo Standardized
Strength of Second Strongest Rhythmic Pulse - Tempo Stand.
Strength Ratio of Two Strongest Rhythmic Pulses - Tempo Stand.
Combined Strength of 2 Strongest Rhyth. Pulses - Tempo Stand.
Rhythmic Variability - Tempo Standardized
Rhythmic Looseness - Tempo Standardized
Polyrhythms - Tempo Standardized
Initial Tempo
Mean Tempo
Tempo Variability
Duration in Seconds
Note Density
Note Density Variability
Average Time Between Attacks
Average Time Between Attacks for Each Voice
Variability of Time Between Attacks
Average Variability of Time Between Attacks for Each Voice
Minimum Note Duration
Maximum Note Duration
Average Note Duration
Variability of Note Durations
Amount of Staccato
(161) Beat Histogram
Number of Strong Rhythmic Pulses
Number of Moderate Rhythmic Pulses
Number of Relatively Strong Rhythmic Pulses
Strongest Rhythmic Pulse
Second Strongest Rhythmic Pulse
Harmonicity of Two Strongest Rhythmic Pulses
Strength of Strongest Rhythmic Pulse
Strength of Second Strongest Rhythmic Pulse
Strength Ratio of Two Strongest Rhythmic Pulses
Combined Strength of Two Strongest Rhythmic Pulses
Rhythmic Variability
Rhythmic Looseness
Polyrhythms

Instrumentation
(128) Pitched Instruments Present
(47) Unpitched Instruments Present
(128) Note Prevalence of Pitched Instruments
(47) Note Prevalence of Unpitched Instruments
(128) Time Prevalence of Pitched Instruments
Variability of Note Prevalence of Pitched Instruments
Variability of Note Prevalence of Unpitched Instruments
Number of Pitched Instruments
Number of Unpitched Instruments
Unpitched Percussion Instrument Prevalence
String Keyboard Prevalence
Acoustic Guitar Prevalence
Electric Guitar Prevalence
Violin Prevalence
Saxophone Prevalence
Brass Prevalence
Woodwinds Prevalence
Orchestral Strings Prevalence
String Ensemble Prevalence
Electric Instrument Prevalence

Texture
Maximum Number of Independent Voices
Average Number of Independent Voices
Variability of Number of Independent Voices
Voice Equality - Number of Notes
Voice Equality - Note Duration
Voice Equality - Dynamics
Voice Equality - Melodic Leaps
Voice Equality - Range
Importance of Loudest Voice
Relative Range of Loudest Voice
Relative Range Isolation of Loudest Voice
Relative Range of Highest Line
Relative Note Density of Highest Line
Relative Note Durations of Lowest Line
Relative Size of Melodic Intervals in Lowest Line
Voice Overlap
Voice Separation
Variability of Voice Separation
Parallel Motion
Similar Motion
Contrary Motion
Oblique Motion
Parallel Fifths
Parallel Octaves

Dynamics
Dynamic Range
Variation of Dynamics
Variation of Dynamics in Each Voice
Average Note to Note Change in Dynamics

MEI-Specific
Number of Grace Notes
Number of Slurs

Figure 3.4.1 Complete list of jSymbolic 2.2 MIR Descriptors

3.5 Subjective Music Analysis

As humans are the intended target for the output of the system, subjective analysis of the compositions will be performed (as a supplement to computational analysis) to assess the human reaction to the pieces. In setting out to design our subjective music assessment, our most salient goal is to identify, to what extent, test subjects are able to discern between the generated music and bespoke compositions. As previously discussed, perception and enjoyment of music is highly subjective and is informed hugely by an individual's life experiences (Krumhansl, 2002). Another study expanded up on this by proposing that one's age and cultural background may have an even more significant perceptual impact on music than life experiences. The same study also showed that music's perceived 'complexity' varies with age, but common to both young and old, cultural differences can create perceptual complexity (Morrison et al, 2008); for example, an Indian pop song perceived as simple by Indian listeners may be perceived as complex by a naive western listener who has no frame of reference for what "simple" Indian pop music should sound like.

Upon designing the listening test for participants to assess the perceived quality of a given composition, the primary focus was to ensure that data analysis would yield meaningful results that could be interpreted to gauge overall enjoyment of a composition. As a full listening test with trained musicians was out of the scope (and budget) of this project, it was decided an informal questionnaire would be used to subjectively analyse a human's reaction to the generative composition. One commonly used method for collecting perceptual data is the use of the Likert Scale, a psychometric scale which ordinarily poses a statement to the participant and offers 5 categorised responses: Strong Disagree, Disagree, Neutral, Agree, Strong Agree (Nemoto and Belgar, 2013). Likert-style questionnaires can be split in to two varieties: one in which the questions use the likert scale categories as a response but are disparate from other questions, and the other in which multiple questions are related to one subject or trait which we are trying to measure (Boone and Boone, 2012). This study will use the latter

approach as we are trying to gauge an overall enjoyment of the composition, rather than answer distinct questions about the properties of the music. The counterpart to the questionnaire will be the accompanying compositions, and although the format is not that of a listening test, some insight can be gained into the ideal listening conditions for participants. One study evaluated the performance of 6 different listening test methodologies and their effectiveness in evaluating the perceived “pleasantness” of 10 different engine sounds. The study asserts that a likert-type scale is a good choice for listening tests in which the goal is to assess the pleasantness of a single sound for each question (Parizet, Hamzaoui and Sabatié, 2005). The proposed method has some additional stipulations; in order to minimise the chance of bias, subjects must be allowed to hear all sounds before making a decision and must have the ability to repeat a sound.

4. Methodology

This section will discuss the methodology used to undertake this project and implementation of the system. When approaching the implementation of any project, it is important to keep in mind the original aims of the project, whilst simultaneously allowing crucial insight gained from the literature review to be utilised effectively.

4.1 Dataset Selection and Preparation

Due to the fact that 2 representations will be discussed and compared, the preparation of data will be discussed in detail before tackling any other stage of the implementation. The network architecture will be discussed along with the python implementation using keras, and finally, the testing methodology will be outlined. As with any machine learning task, having access to a high quality and quantity data-set is fundamental to the success of the overall system. The identified ideal format for the data would be MIDI, due to the fact that each note of a midi file can be seen as a time-step. In comparison, time-series analysis of an audio file is extremely complicated, due to the fuzzy timestep size as well as the extreme complexity of extracting specific musical instruments, such as drums, lead vocals, keyboards etc from the original signal. Other symbolic formats such as MusicXML may also have been considered if not for the lack of widespread support, thus availability to a sufficiently large quantity of MusicXML was not available. The ideal data-set was to belong to one genre of music, and while including music from multiple genres may be possible, it complicates the process needlessly and vastly increases the search space for the network, leading to longer training times and poorer overall results. The final dataset was taken from www.piano-midi.de and consists purely of classical music from composers such as Tchaikovsky and Mozart, with a total of approximately 300 unique compositions in the entire set. This may seem like a fairly small dataset, but considering that the network is only attempting to emulate one style, and given that each song may contain thousands of notes, we end up with a small but well suited and clean dataset.

Two methods of representation are proposed (henceforth referred to as **Method A & Method B**): Method A proposes that each note of the song be considered its own timestep, with pitch, velocity and timing information stored as the timestep's features; Method B proposes a piano-roll like representation, where each timestep represents a designation of time containing a list of notes that are active at this point in time. Both of these approaches have seen prior use as highlighted by the literature, and both have their own set of advantages and disadvantages. As previously mentioned, mapping a single note to multiple timesteps can lead to a lack of comprehension, while Method A's approach of treating each new note as a new timestep can fail to capture the rhythmic structure of a composition. This project will look at evaluating the effectiveness of both approaches, and conclude with a recommendation. Before discussing these methods further, the following tools were used to aid the implementation for this part of the system.

Pretty_midi

Pretty_midi is a python library which provides a wide variety of helper functionality for interacting with and altering MIDI data. Pretty_midi is used for loading in the raw midi data and converting it to a format appropriate for the neural network, as well as for converting neural network predictions back into midi data. The framework splits MIDI data up into an extremely simple structure, with a midi file containing N number of instruments, which themselves contain N number of notes. A note contains the pitch and velocity, as well as the start and end time, making conversion to the network representation a much more streamlined process.

Numpy

A ubiquitous mathematics and scientific computation library which provides a massive collection of functionality as well as memory-efficient and performant data containers. Numpy's data containers (array, ndarray, matrix) also work extremely well with tensorflow (discussed further shortly) and have the added benefit that they can be

saved to and loaded from a text file, meaning computation and conversion to the network format need only be performed once, after which the data-set can simply be reloaded with a few simple lines of python.

Before converting the input MIDI data to the representation proposed for Method A and B, a common format is introduced to make conversion to both representations easier and the system more modular overall. The first stage in this process is to load in the raw midi data using pretty_midi.

```
def loadMidiData(cleanfiles, minOctave, maxOctave):
    print("Loading Midifiles")
    midifiles = {}
    key_distributions = {}
    for c in tqdm(cleanfiles):
        try:
            mid = pretty_midi.PrettyMIDI(c)
            limitMidiOctaves(mid, minOctave, maxOctave)
            midifiles[c] = mid
            total_velocity = sum(sum(mid.get_chroma()))
            key_weight_distribution = [
                sum(semitone) / total_velocity for semitone in mid.get_chroma()
            ]
            key_distributions[c] = key_weight_distribution

        except OSError:
            print("Midifile: " + c + " could not be loaded)")
        except ValueError:
            print("Midifile: " + c + " could not be loaded)")
        except IndexError:
            print("Midifile: " + c + " could not be loaded)")
        except KeyError:
            print("Midifile: " + c + " could not be loaded)")
        except mido.KeySignatureError:
            print("Midifile: " + c + " could not be loaded)")
        except EOFError:
            print("Midifile: " + c + " could not be loaded)")


    return midifiles, key_distributions
```

Figure 4.1.1 Function to load in MIDI data

Figure 4.1.1 shows the method which loads in the raw pretty_midi data and stores each object in a dictionary for further processing. It is passed a list of paths and attempts to load these paths as pretty_midi objects. Furthermore, the function takes a minOctave and maxOctave parameter which allows us to limit the range of available pitches in our dataset, which has two distinct advantages. The first advantage is that there are more tokens available for the time-features of the note, allowing for greater rhythmic diversity. Secondly, as fewer pitches exist across the dataset, the search space is narrowed significantly which (in theory) should allow the network to better understand the relation between pitches. After the initial raw data has been loaded in, the next step in the system is to convert this data into a general format used in both Methods A & B.

```
def getKerasData(mididict, key_distributions):
    binaryVecs = []
    data = []

    for k, v in tqdm(mididict.items()):
        raw_notes = v.instruments[0].notes
        notes = []
        # not being used right now
        binaryKeyVector = getBinaryKeyVector(key_distributions[k])
        lastNote = raw_notes[0]
        for note in raw_notes:
            noteTime = note.start - lastNote.start
            noteDuration = note.end - note.start
            noteVelocity = note.velocity
            notePitch = note.pitch
            # do some stuff
            # 1st Start Time 2nd Duration of Note, 3rd Note Velocity, 4th Note Pitch
            notes.append((noteTime, noteDuration, noteVelocity, notePitch))
            # for normalisation in neural network

            lastNote = note

        data.append(notes)
        binaryVecs.append(key_distributions[k])

    return (
        data,
        binaryVecs
    )
```

Figure 4.1.2 : Function to retrieve common data-format used in methods A & B.

From Figure 4.1.2, we can see that the common format is retrieved by passing through a dictionary containing all the pretty-midi objects as well as a key_distributions object, which was originally planned for use in a post-processing pass, however this post-processing technique was later abandoned and as such the key distribution vector goes unused (The decision was made to leave the key-distributions in the code-base, as although they are currently unused, continued development of the system could likely make use of these vectors for refinements relating to melody and chordal structures.). For each note in each composition, we record its pitch and velocity (which are attributes of the pretty_midi.Note class) we then record the duration of the note (calculated as the note's end time - the note's start time) and finally, an 'offset', which is the amount of time (in seconds) that has elapsed since the last note was activated. The shape of the data returned from getKerasData() is (Total Number of Songs, Number of Notes in Song, 4). The next sections will discuss the method-specific data-preparation process.

Method A

As previously discussed, Method A treats each timestep as a new note in the composition. Each note contains 4 features: pitch, velocity, duration and offset - which indicates the amount of time that has passed between this note and the last activated note. The labels for this approach then are a list of all notes from every midi file, with the data being a list of *N* notes which corresponds (and leads to) the current label note. The sequence size for the training data was experimented with, and as such, warrants further discussion in the analysis section.

Each feature of the note was tokenized and stored in a shared dictionary, which had a maximum size of 500 tokens. While limiting the musical possibilities the system would be able to reproduce, limiting the dictionary size is a common practice for increasing the performance, accuracy and training time of a neural network.

Figure 4.1.3 shows us the function which takes in our raw keras_data object and converts this into the training labels and data. This process starts by collecting every pitch, velocity, offset and duration that occurs throughout the entire dataset and counting the occurences of this feature. Using the occurrence dictionaries, we are able to establish which durations and offsets occur most often and use these as our tokens. The getTokensA function uses these occurrences to produce the final vocabulary, which is a mapping of int indices to MIDI messages e.g. (1 = 81, 2 = 0.0013 etc.) Having collected the final set of tokens, we then iterate through keras data object, converting each MIDI message to a token using the getNoteTokenA method, forming the label data. This method takes a MIDI message and token dictionary and returns the nearest possible value to the original MIDI message as a token.

```
def processKerasDataMethodA(data, sequenceSize):
    newNotes = []
    newLabels = []
    print("Converting to LSTM Format")
    counter = 0

    pitches_occurences, offsets_occurences, durations_occurences, velocities_occurences = countTokenOccurences(
        data
    )

    tokens, token_cutoffs = getTokensA(pitches_occurences, durations_occurences, offsets_occurences, 500)
    inv_tokens = invertDictionary(tokens)
    pitch_tokens = invertDictionary(slice_odict(inv_tokens,1, token_cutoffs["last_pitch"] + 1))
    velocity_tokens = invertDictionary(slice_odict(inv_tokens,token_cutoffs["last_pitch"] + 1, token_cutoffs["last_velocity"] +1))
    duration_tokens = invertDictionary(slice_odict(inv_tokens,token_cutoffs["last_velocity"] + 1, token_cutoffs["last_duration"] + 1))
    offset_tokens = invertDictionary(slice_odict(inv_tokens,token_cutoffs["last_duration"]+1, token_cutoffs["last_offset"] + 1))

    for i in tqdm(range(len(data))):
        for j in range(len(data[i])):
            if j < sequenceSize:
                j = sequenceSize
            note = data[i][j]

            note_time = note[0]
            note_duration = note[1]
            note_velocity = note[2]
            note_pitch = note[3]

            token_offset = getNoteTokenA(note_time, offset_tokens)
            token_duration = getNoteTokenA(note_duration, duration_tokens)
            token_velocity = getNoteTokenA(note_velocity, velocity_tokens)
            token_pitch = getNoteTokenA(note_pitch, pitch_tokens)

            newLabels.append((token_pitch, token_offset, token_duration, token_velocity))

            priorNotes = []
            for k in range(sequenceSize):
                pnote = data[i][j - (k + 1)]
                pnote_time = pnote[0]
                pnote_duration = pnote[1]
                pnote_velocity = pnote[2]
                pnote_pitch = pnote[3]

                ptoken_offset = getNoteTokenA(pnote_time, offset_tokens)
                ptoken_duration = getNoteTokenA(pnote_duration, duration_tokens)
                ptoken_velocity = getNoteTokenA(pnote_velocity, velocity_tokens)
                ptoken_pitch = token_pitch = getNoteTokenA(pnote_pitch, pitch_tokens)
                priorNotes.append((ptoken_pitch, ptoken_offset, ptoken_duration, ptoken_velocity))

            newNotes.append(priorNotes)

    return (np.array(newNotes), np.array(newLabels), tokens, token_cutoffs)
```

Figure 4.1.3 : Method A Data Processing Method

Having collected the current label, that is the note at the current timestep, we next wish to collect a list of N previous notes, to form the associated data with our label. As previously mentioned, the data for this system will take the form of a list of N notes (N denoted in code as sequenceSize). This is achieved by creating a loop which iterates over the previous N notes from the position of the current label note, tokenizing each previous note in the same method as used for the label, before and adding it to a list. Once the list has been filled, it is added to the data array. The final shape of the labels is then = Total number of Notes x 4 (pitch, velocity, offset, duration), with the final shape of the data being = Total number of notes x Sequence Length x 4.

Method B

For this approach, each time-step denotes a segment of time, which contains N active notes (the number of concurrently active notes was also experimented with and will be discussed further in the analysis portion of this project). The number of time-steps per second was also parameterised to aid in the experimental part of this project. The label representation for this approach would be an individual timestep, with the training-data being a list of *N* timesteps which corresponds (and leads to) the current label timestep. Duration and start-time are baked into the representation for this approach: if a note was not in the previous timestep but appears in the current timestep, this is considered a new note starting; conversely, if a note was in the previous timestep but does not appear in the current timestep, this is considered the end of a note. The number of timesteps per second (referred to as timestep resolution) was determined through experimentation (and will be discussed further in the analysis section) however a rough guideline kept this number between 10 and 30. Limiting the timestep resolution too heavily would result in little to no rhythmic diversity or complexity, however a resolution too large could lead to the neural network failing to comprehend the sequence at all.

Due to the removal of time as a note feature, this method benefits from a significantly reduced dictionary size, only representing the available pitches for a small number of

velocities. As such, a note is composed of a single token, where the token represents a pitch at a specific velocity.

```
def processKerasDataMethodB(data, sequence_size, num_timesteps, timestep_resolution, num_simultaneous_notes):
    processedData = []
    pitches_occurences, offsets_occurences, durations_occurences, velocities_occurences = countTokenOccurences(data)
    tokens, velocities = getTokensB(pitches_occurences, velocities_occurences, 500)

    print("\nConverting data to final network representation\n")

    y = []
    for song in tqdm(data):
        lastStartTime = 0
        song_data = np.zeros((num_timesteps, num_simultaneous_notes), dtype=int)

        for note in song:
            time = note[0]
            duration = note[1]
            velocity = note[2]
            pitch = note[3]
            lastStartTime += time

            note_token = getNoteTokenB(pitch, velocity, tokens, velocities)
            note_duration = int(timestep_resolution * duration)
            note_start = int((lastStartTime) * timestep_resolution)

            if note_start + note_duration >= num_timesteps:
                break
            else:
                # place the note
                for i in range(note_duration):
                    for j in range(num_simultaneous_notes):
                        value = song_data[note_start + i][j]
                        if value == 0:
                            song_data[note_start + i][j] = note_token
                            break
        y.append(song_data)

    x = []
    for song in tqdm(y):
        song_train = []
        for i in range(len(song)):
            if i < sequence_size:
                i = sequence_size

            lastNotes = []
            for j in range(sequence_size):
                lastNotes.append(song[i - j])
            song_train.append(lastNotes)
        x.append(song_train)

    return x, y, tokens
```

Figure 4.1.4 : Method B Data Processing Method

Figure 4.1.4 shows the implementation of the data processing for Method B's representation, which takes in a few extra parameters which alter the properties of the

final Method B data. Parameterisation of the data preparation stage made it much easier to test new configurations and sped up development of the system by a significant margin. The new parameters introduced are: num timesteps, timestep resolution and num simultaneous notes. The number of timesteps is simply the total number of timesteps that we wish to compose our training instance from, meaning how many seconds of the training instance will be used. This parameter was added due to the fact that the researcher's computer did not have enough RAM or GPU VRAM to store the entire Method B dataset when the entire songs were sampled. The timestep resolution denotes the number of samples that will be taken per second, e.g. a resolution of 10 means that for every second of symbolic music, 10 seconds will be taken. The number of simultaneous notes describes how many different notes can be active at any given timestep. We start the process in a similar way to that of the first method, collecting all of the occurrences for pitch, velocity and timing features (though offset and duration occurrences were not used for this method). We next generate the tokens which, in this method, encapsulate both a pitch and velocity. As mentioned in the introduction, the compositions have been limited to a range of 3 octaves, meaning that a total of only 36 pitches exist throughout the dataset. Furthermore, to reduce the search-space and reduce the complexity of the network, only 4 velocity values are sampled per note, meaning the total number of tokens available is just 144. The function specifies a max tokens parameter which is unnecessary for the current approach, however, should more pitches / velocities be required during further development of the system, this functionality may prove useful.

Having retrieved the token, we are now able to begin creating our data and labels for Method B. We start by iterating through the songs in the raw keras data format, creating an empty 2D numpy array and filling this with 0s, indicating no active note (0 is used as the null token for both approaches). The final shape of this array is The total number of timesteps x the number of simultaneous notes. We also create a

lastStartTime variable, which is used to calculate where the current note should be placed, with respect to the placement of the last note

The next step is to calculate the note's position and duration in the numpy array: the start position of the note is added to the lastNoteCounter, which in turn is multiplied by the timestep resolution and cast to an int to find the concrete starting index of this note. The duration is calculated in a similar manner, multiplying the timestep resolution by the length of the note and casting the result to an int. Having found the indices where our note should start and the amount of timesteps the note should occupy, we must next check that we are able to add this new note to the song's numpy array: this is achieved by iterating through each identified timestep and checking for any null tokens (0). If a null token is found, the new note replaces the null token and the loop breaks to continue onto the next timestep to be checked, otherwise the note is not added to the dataset.

We then iterate through every note in the unprocessed data format and extract the pitch, velocity, offset and duration, taking the pitch and offset and feeding this into the vocabulary to find the closest available token. The resulting dataset form the labels for this system. As previously stated, the data for this system is a series of N timesteps which correspond to the current label timestep. This is achieved by iterating through the label data, sampling the previous N timesteps from the current label timestep and storing this in an array. Having completed the training data, the function finally returns the data, labels and tokens, for later use converting the predictions back to MIDI format. The final shape of the labels is = Total Number of Timesteps x Num Simultaneous Notes, and the final shape of the data is = Total Number of Timesteps x Sequence Length x Num Simultaneous Notes.

Random Comparison

A common approach when evaluating a ML application is to perform a comparison against random data. This is extremely beneficial when performing analysis of the

generated music as we are better able to evaluate the extent to which the neural network is able to reproduce “human-sounding” music.

```
import random
import pretty_midi
from DataGen import *

def genRandomData(num_songs_to_compose):
    for i in range(num_songs_to_compose):
        num_notes = random.randint(1, 1000)
        print("Num notes = %d" % num_notes)
        pm = pretty_midi.PrettyMIDI()
        inst = pretty_midi.Instrument(0)
        counter = 0.0
        for j in range(num_notes):
            pitch = random.randint(0, 127)
            velocity = random.randint(0, 127)
            duration = random.random() * 3
            offset = random.random() * 3
            counter += offset
            n = pretty_midi.Note(velocity, pitch, counter, counter + duration )
            inst.notes.append(n)
        pm.instruments.append(inst)
        pm.write(getMidiRunName("Random"))

genRandomData(5)
```

Figure 4.1.5 : Random comparison Data Generation Method

From figure 4.1.5 the function to generate random MIDI data can be seen. The function takes in the number of random MIDI files to generate as it’s only parameter. For each random composition, we start by randomly selecting a number of notes to generate as well as creating a PrettyMIDI and Instrument object to contain our generated notes. For each note in each composition, we randomly generate an Int in the range 0-127 for the Pitch and velocity,. For offset and duration, we reuse the occurrences dictionary previously used in the tokenisation process, and sample the minimum and maximum for each feature, using these values as the bounds for our random data. Having generated the required data for a note, we create a new pretty_midi Note object and append this to the instrument’s note array. Finally, we add the instrument to our PrettyMIDI object and write this file out to the random MIDI directory.

4.2 Designing Structure of Neural Network(s)

This section will discuss the design and implementation of the final network architectures. The design of the network draws inspiration from various sources, however the primary influence for the design of this network came from a GitHub repository (available at https://github.com/Skuldur/Classical-Piano-Composer) (Sigurður Skúli, 2017) which proposes a similar (yet more simple) representation of a note, with a similar sequence like approach used to predict new music. Programming was performed in python using a variety of frameworks, which will be briefly outlined.

Tensorflow

Tensorflow is an open source symbolic mathematics library developed by Google. It allows for extreme performance across a range of programming tasks, it is however most commonly used for machine learning tasks - specifically neural networks. The primary reason Tensorflow was used for this project is due to the researchers familiarity with Tensorflow, the possibility of offloading large tasks to the GPU as well as it's integration to the keras framework.

Keras

Keras is an open-source machine-learning framework originally created by a single developer, Francois Chollet. Keras has a strong focus on Neural Network programming, allowing for standard feed-forward neural networks, convolutional neural networks and recurrent neural networks to be developed. A strong effort has been made to make Keras a user-friendly experience, with minimal code required to setup and train a neural network, and even contains functionality intended to make data pre-processing easier (such as train-test splits and word-embedding functionality). The most important aspects of Keras to discuss are the Model, Layers and Optimisers.

The model class can be thought of as a container of arbitrary length for neural network layers. A few variants of the model class exist, but by far the most common is the

Sequential model, in which the model is entirely self contained and in which all layers feed forward. Models also contain the functionality to begin training, as well as make predictions, provided data has been fed and the network contains no errors.

The layers API provides a wide selection of neural network layers, which allows the programmer to rapidly prototype and develop new network architectures. Specifically, keras implements the LSTM and GRU cell making development of a recurrent network solution significantly easier. Keras also implements other types of layers such as Dense fully connected layers, as well as convolutional layers and dropout layers (for regularization). One consideration is that the beginning layer in any model must specify the input shape of the data, while the final layer must present the same shape as the input label.

An optimiser encapsulates the algorithm that will perform backpropagation on the network, as well as various other functionality (such as network unwrapping for recurrent layers). Keras offers a range of optimisers such as SGD, Adadelta, RMSprop, and the optimiser used by both configurations in this network, Adam.

The final end to end process for developing a neural network in Keras as such entails designing your architecture, rapid implementation thanks to the extremely simple syntax, and evaluating the results from the network.

Keras is able to run atop a variety of low-level machine learning frameworks, including Theano, Microsoft's Cognitive Toolkit as well as Tensorflow. Keras is also able to leverage the power of Tensorflow's GPU compute capabilities, vastly increasing the speed of training. Enabling GPU support for Tensorflow can be a cumbersome task, requiring specific driver versions, an outdated version of CUDA compute, CuDNN as well as a range of other Nvidia software; however, the benefit is immediately felt in terms of speed. One consideration for GPU acceleration on large datasets is that when

using GPU acceleration, the data is stored on the graphics memory rather than RAM as GPU memory is often magnitudes faster than desktop DDR3/4. As a result, for very large data-sets, GPU accelerated training may need to be performed in batches, as most modern graphics cards are limited to approximately 8GB.

Helpful Tools

As well as the core programming frameworks previously discussed, the following tools were employed for a variety of tasks which significantly aided the development of this project. The primary IDE for this project was Spyder, due to its ability to run small snippets of code as well as the extremely useful Variable Explorer feature. Variable Explorer made visualising and debugging data significantly less frustrating than it could have been, especially considering native support for most Numpy containers.

Tensorboard is a sub-module of Tensorflow which allows for easy visualisation of the training process for a neural network. Furthermore, it allows for deep statistical analysis of the network training period, as well as other metrics to assess the quality of prediction made by the network. It was primarily used to analyse the training period performance of both networks for this project.

A note on Compute Power

One significant factor to consider when designing any neural network is the available compute power of the development system. In this case, the computer was equipped with a 6-core CPU, 16GB RAM and a 4GB GPU (3GB usable) and as such, very large networks with vast data-sets would likely not be feasible. During the experimentation stage, many networks either failed to begin training or even failed to compile at all due to a lack of memory on the GPU. As such, the network configuration may be sub-optimal for performance, as allowing a greater number of timesteps per label, or a higher dimensionality of LSTM cell may have yielded better results.

Method A Network

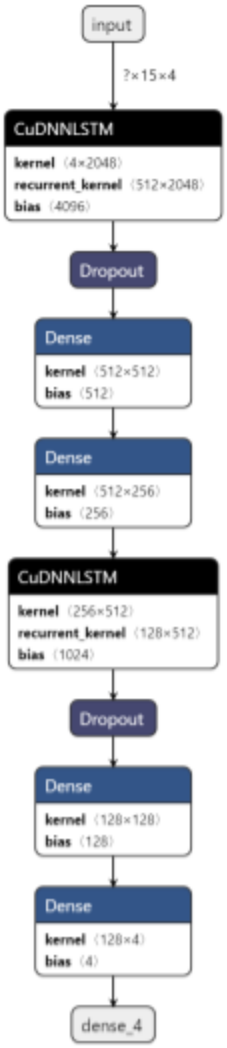


Figure 4.2.1 - Method A Final Network Architecture

The final architecture for Method A is composed of 4 dense, fully-connected layers and 2 LSTM cells (CuDNNLSTM is an LSTM cell which has been optimised for GPU training in keras). The architecture is relatively simple and was derived through experimentation. The network accepts a 15x4 tensor as it's input, with the 4 numbers representing the note features, and 15 the number of timesteps (previous note) used to predict the current label. The final output of the network is a 4 neuron hidden layer, with each neuron corresponding to a part of the note (e.g. pitch).

Method B Network

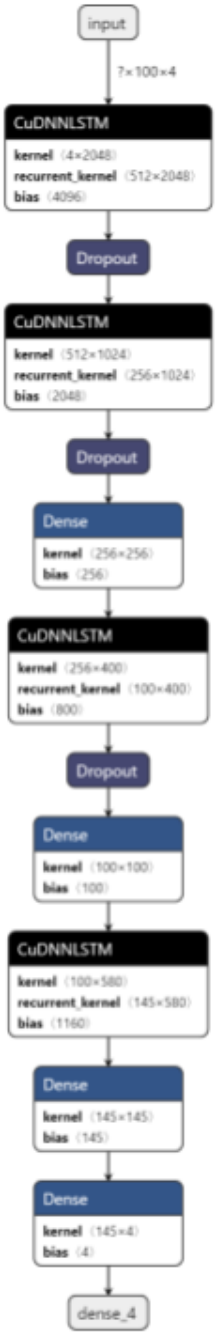


Figure 4.2.2 : Method B Final Network Architecture

The final network architecture for the Method B representation can be seen in Figure 4.2.2. The network accepts a 100x4 vector as its input (100 timesteps, 4 simultaneously active notes) and produces a 4 feature vector which correspond to the 4 active notes at the current timestep. The full network is composed of 4 LSTM cells and 4 fully-connected dense layers. Experimentation was used to dictate the exact parameters of the various network layers, however the theory is that the network has some arbitrary amount of memory which stores the incoming sequence before any manipulation of the data is performed. The sequence is then fed through another LSTM cell and dense layer, whose parameters were dictated through experimentation. The final layers of the network's dimensionality and size are proportional to the shape of the data, e.g. the final layer is a 4 neuron fully-connected dense layer corresponding to the 4 active notes at the current time-step (the current label). The second to last layers have the same number of neurons as tokens that exist in the dictionary, and above that exists a fully-connected dense layer whose number of neurons is proportional to the length of the sequence fed in to the network.

4.3 Synthesizing symbolic data

After the network has generated some predictions, a method for converting this representation back to MIDI for synthesization is required.

Method A

```
def convertMethodADataToMidi(notes, tokens, token_cutoffs, model_name):
    print("Converting raw notes to MIDI")
    mid = pretty_midi.PrettyMIDI()
    inst = pretty_midi.Instrument(0)
    # inst.program = 0
    lastNoteStart = 0
    for note in tqdm(notes):
        # remap output of neural network / normalise notes
        pitch_token = int(note[0][0])
        offset_token = int(note[0][1])
        duration_token = int(note[0][2])
        velocity_token  = int(note[0][3])

        pitch_token = int(clamp(pitch_token, 1, token_cutoffs["last_pitch"]))
        velocity_token = int(clamp(offset_token, token_cutoffs["last_pitch"] + 1, token_cutoffs["last_velocity"]))
        velocity_token = int(clamp(offset_token, token_cutoffs["last_velocity"] + 1, token_cutoffs["last_duration"]))
        velocity_token = int(clamp(offset_token, token_cutoffs["last_duration"] + 1, token_cutoffs["last_offset"]))

        pitch = np.uint8(tokens[pitch_token])
        offset = tokens[offset_token]
        duration = tokens[duration_token]
        velocity = np.uint8(tokens[velocity_token])

        start = lastNoteStart + offset
        pmnote = pretty_midi.Note(velocity, pitch, start, start + duration)
        inst.notes.append(pmnote)
        lastNoteStart = start

    mid.instruments.append(inst)
    cleanMidiA(mid)
    mid.write(getMidiRunName(model_name))
```

Figure 4.3.1 : Function to convert Method A data back to MIDI

As can be seen from Figure 4.3.1, the function to convert network data back to MIDI takes 4 arguments: the predictions made by the network (notes), the original token vocabulary, an auxiliary dictionary used to segment the token dictionary into pitch, velocity and timing section and finally the name of the model that generated the composition. We start by creating a new Pretty_Midi object and adding a Piano instrument to it (instrument code 0 is grand piano). We then create a variable to track the last note's start time and begin iterating through the data. For each note, we convert the pitch, velocity, offset and duration from their token representation to their original value, creating a new Pretty_Midi note object and using these values as the parameters to create this note. Once the note has been created, we add it to the

PrettyMIDI object and perform a cleaning pass, which keeps all notes positioned relatively, but will randomly alter the key to add diversity to the generations.

Method B

```
def convertMethodBDataToMidi(data, tokens, model_name, timestep_resolution):
    print("Converting raw notes to MIDI")
    mid = pretty_midi.PrettyMIDI()
    inst = pretty_midi.Instrument(0)
    timestepCounter = 0
    # what notes are currently being played
    current_notes = {}
     # unnormalize the data at each step
    for d in tqdm(data):
        for timestep in d:
            # for all notes in this timestep
            rounded_timestep = [round(x) for x in timestep]
            for msg in rounded_timestep:
                # if we encounter a new note start counting its duration
                if msg not in current_notes:
                    current_notes[msg] = []
                    current_notes[msg].append(timestepCounter)
                    current_notes[msg].append(1)
                else:
                    current_notes[msg][1] = current_notes[msg][1] + 1
            # if any of the currently playing notes are not
            # being played at this timestep, remove it from current notes7
```

Figure 4.3.2 : Function to convert Method B data back to MIDI Pt.1

From Figure 4.3.2, we can see the first part of our function for converting Method B data back to Midi, like with Method A, we start by creating a new PrettyMIDI object and adding a piano to the classes instrument list. We next create two new variables, the timestepCounter and current_notes dictionary. The timestepCounter is used to keep track of the current timestep, and the current notes dictionary is used to keep a record of which notes are currently being played. We begin the conversion by iterating through the predictions made by the network and converting them to ints using the round function. We next iterate through all of the messages in the current timestep and check if they are in our current_notes dictionary. If they are not, a new key is added to the dictionary and a counter is started. The counter represents the number of timesteps

that note has been active for, and will be later converted back to seconds. If the note is in the current notes dictionary, we increment the counter by 1.

```
notesToRemove = []
for note, timing in current_notes.items():
    if note not in rounded_timestep:
        # print("Logic is sound, this is being called :) ")
        currentToken = abs(int(note))
        if note >= 1:
            pitch, velocity = tokens[currentToken]
            if pitch > 127:
                pitch = np.uint8(127)
            if velocity > 127:
                velocity = np.uint8(127)
            start = timing[0] / timestep_resolution
            duration = timing[1] / timestep_resolution
            # print("Note start time = %d, note duration = %d" % (start, duration))
            new_note = pretty_midi.Note(
                velocity, pitch, start, start + duration
            )
            inst.notes.append(new_note)
            notesToRemove.append(note)

for note in notesToRemove:
    current_notes.pop(note)

timestepCounter += 1
```

Figure 4.3.3 : Function to convert Method B data back to MIDI Pt.2

We next iterate through the current notes, checking if they exist in the current notes dictionary; if the note is in the current timestep, we do nothing, otherwise we interpret this as the end of the note. We retrieve the pitch and velocity from the token dictionary, ensuring they do not exceed the maximum MIDI value of 127. The start time of the note is calculated by dividing the starting timestep by the timestep resolution, converting it back into seconds. The same process is applied to the duration - the total number of timesteps the note lasted for is divided by the timestep resolution, resulting in the duration in seconds. These values are used to create a new Pretty_Midi note object and added to the instrument. After all the finished notes have been added to the Pretty_Midi object, they are removed from the current notes dictionary.

```
if bool(current_notes) == True:
    for note, timing in current_notes.items():
        currentToken = abs(int(note))
        if currentToken != 0:
            pitch, velocity = tokens[currentToken]
            if pitch > 127:
                pitch = np.uint8(127)
            if velocity > 127:
                velocity = np.uint8(127)
            start = timing[0] / timestep_resolution
            duration = timing[1] / timestep_resolution
            # print("Note start time = %d, note duration = %d" % (start, duration))
            new_note = pretty_midi.Note(velocity, pitch, start, start + duration)
            inst.notes.append(new_note)

del current_notes
print("total number of notes = %d" % len(inst.notes))
mid.instruments.append(inst)
cleanMidiB(mid)
mid.write(getMidiRunName(model_name))
```

Figure 4.3.4 : Function to convert Method B data back to MIDI Pt.1

The final part of this function takes place after all predictions have been iterated through. We check if the current notes dictionary has any remaining notes left, and if so, add them to the instrument class. Next we get rid of our current notes dictionary as it is no longer needed, we then add the instrument to the Pretty_Midi object and perform a cleaning pass, which simply randomises the key and slows the composition down. This was performed as many of the compositions tempo were too fast, but the intervals between notes were still sound. We then finally write the midi file to disk for evaluation.

As a small aside, it should be noted that the human compositions used for both subjective and computational analysis will first be converted to either Method A or B's representation, and then processed back to MIDI. This will be performed to eliminate any potential bias in terms of disparity between representations. This is a particularly important consideration for Method B which uses a drastically reduced timestep resolution compared to standard MIDI (Standard MIDI uses 480 'ticks' per second).

The final compositions were synthesized using Ableton Live, a Digital Audio Workstation (DAW). Ableton has a wide variety of built-in digital instruments, as well as an extremely useful toolset for visualising and manipulating MIDI data. Figure 4.3.5 shows the Ableton user interface, with a sample neural network composition's piano roll show on the lower half of the screen.

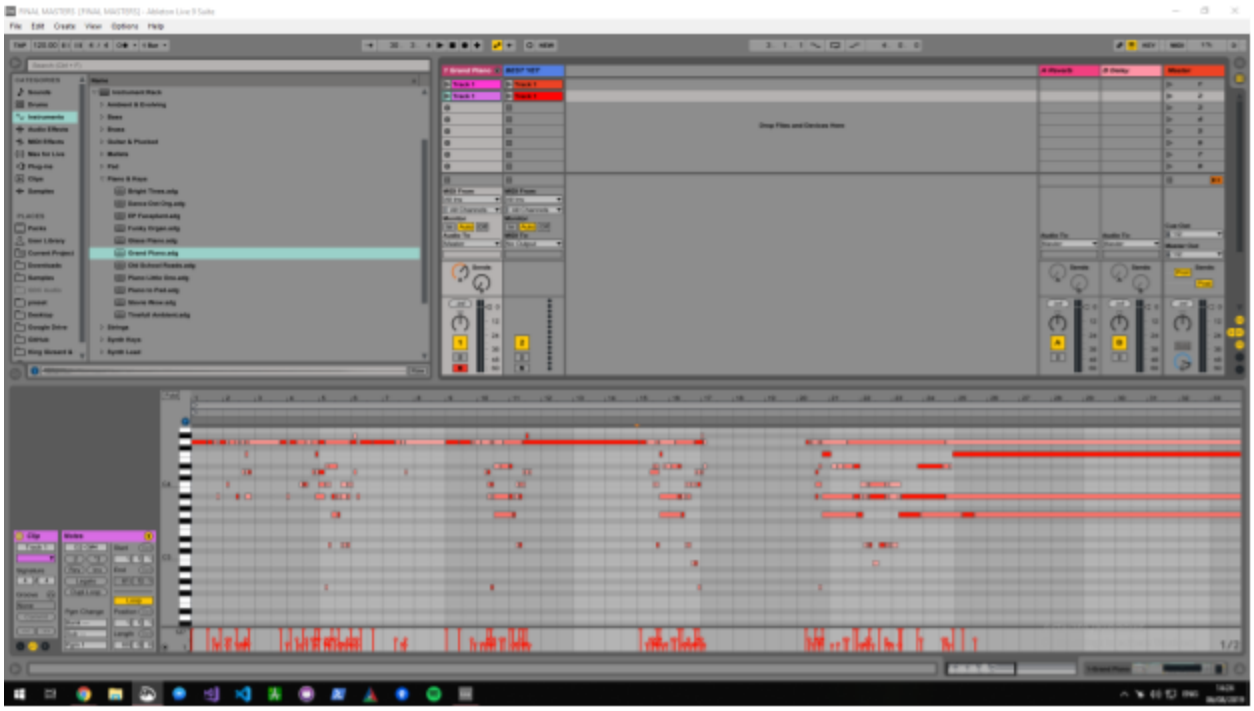


Figure 4.3.5 : Ableton Live User Interface

4.4 Computational Music Analysis

Computational analysis of the generated music was performed in order to empirically analyse the performance of the network against other solutions, such as human composed music, as well as randomly generated music, purely for the sake of comparison. Analysis was performed using jMIR, specifically the jSymbolic2 tool, which offers a vast number of music descriptors which can be dumped to XML for further analysis. Many of the descriptors are fairly self explanatory in their meaning (e.g. variability of note durations or pitches) however, some of the descriptors have a larger basis in music theory than in simple mathematics, and as such, will briefly be outlined.

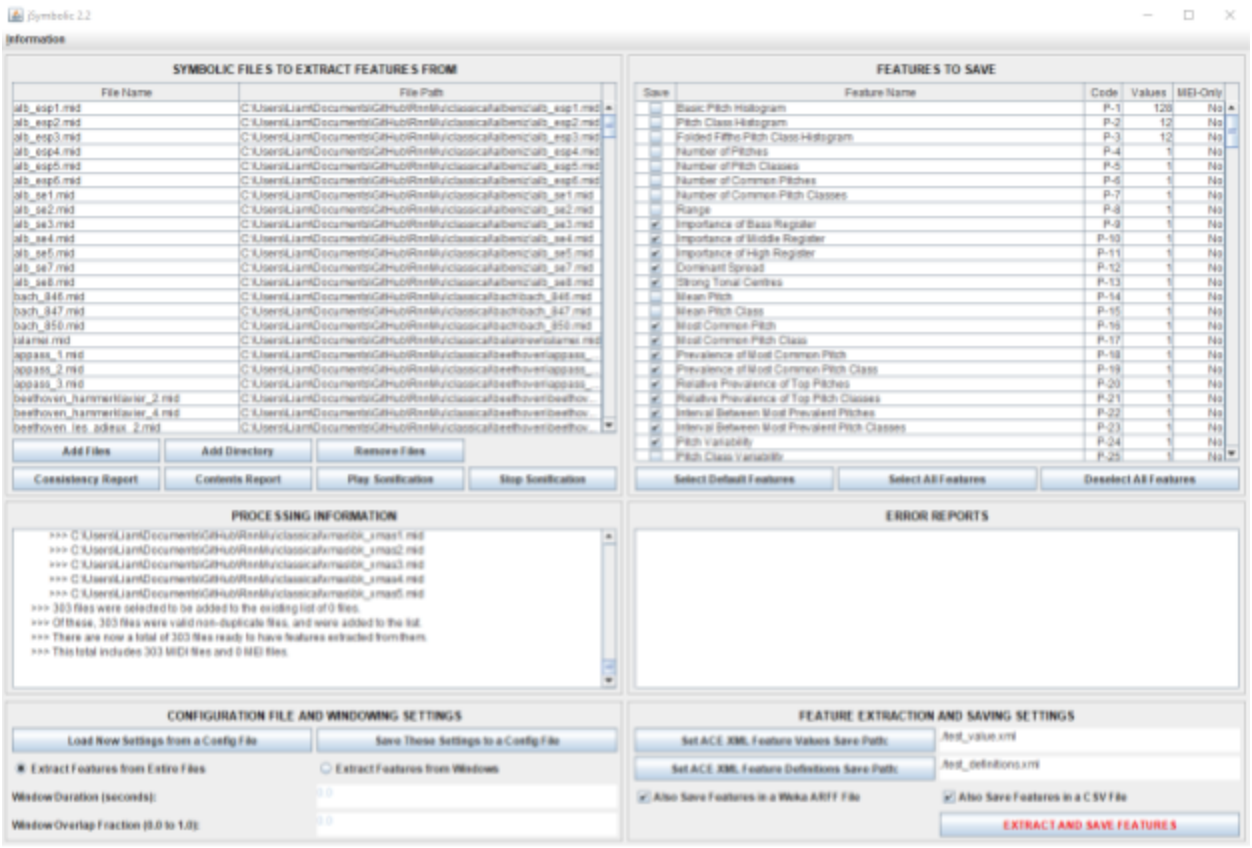


Figure 4.4.1 Screenshot of jSymbolic App

Contrary Motion

When discussing motion in the context of a musical phrase, it is first important to define precisely what motion actually means. The motion of a given musical phrase can be thought of as a direction in which the melody moves, or how the pitch and other important properties (such as velocity, duration etc.) change over the course of the passage. As such, contrary motion describes two or more musical voices which are “moving” in perfect opposite directions, where one voice is ascending a given scale or phrase, and the other is descending the same passage.

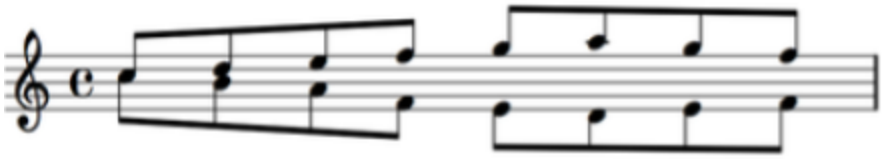


Figure 4.4.2 : Contrary Motion in Music Notation

Similar Motion

Similar motion occurs when two or more voices are playing the same shape of melody at different base pitches. The intervals between the steps need not be identical to exhibit similar motion, if all interval changes between both voices are identical, the motion is instead described as parallel.

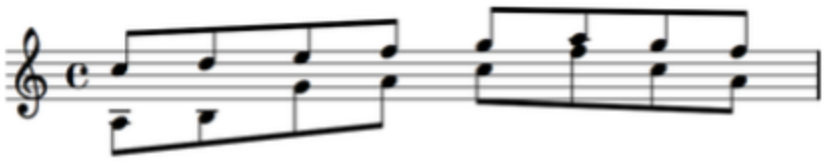


Figure 4.4.3 : Similar Motion in Music Notation

Chromatic Motion

Chromatic motion occurs when one or more musical voices is linearly ascending / descending the chromatic scale. Chromatic motion is the most basic form of melodic line, and may be erroneously considered an indicator of low complexity, however when used in moderation, it can elicit specific emotions in the listener, such as suspense or dread.

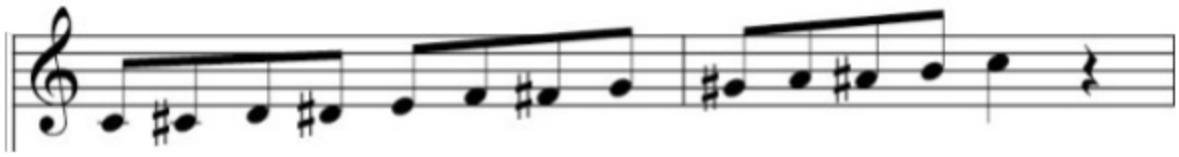


Figure 4.4.4 : Chromatic Motion in Music Notation

Stepwise Motion

Stepwise motion occurs when one or more musical voices linearly ascends or descends a given musical scale. Stepwise motion is similar to chromatic motion but will sound inherently more “musical” as the key of the musical phrase is respected with stepwise motion.

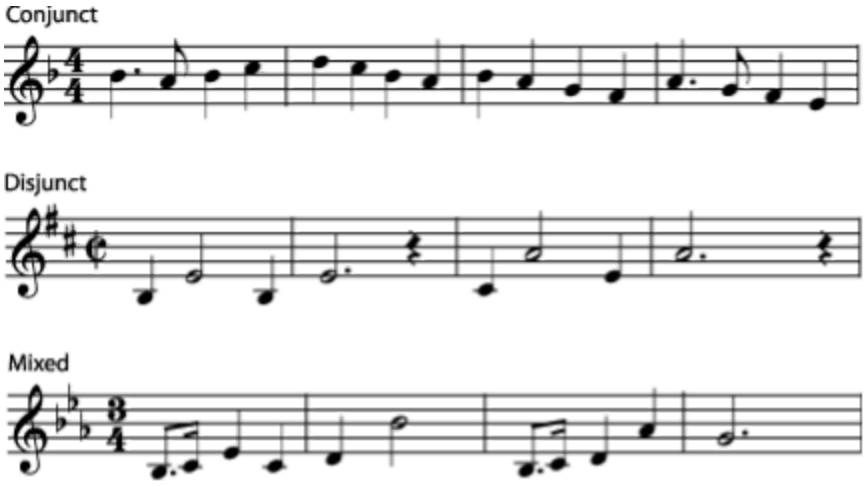


Figure 4.4.5 : Stepwise Motion (Conjunct) compared against disjunct (Skipwise Motion) and a mixture of both.

Figure 4.4.5 shows music notation for so called conjunct, disjunct and mixed melodies, with a conjunct melody being synonymous with stepwise motion. Disjunct motion has much more variety in the intervals between pitches and often jumps large amounts between the pitches of the key, compared to a conjunct melody, which as previously mentioned, moves linearly in a direction, with no major change to the interval between notes.

Amount of Arpeggiation

Arpeggiation is a type of broken chord (where each of the notes are played separately) in which the notes of the chord are played in ascending and descending order (e.g. C, E, G, E, C, E...) The amount of arpeggiation then, is the percentage of notes in the composition that exhibit arpeggiation.

Metrical Diversity

The metrical diversity descriptor indicated a high variety in rhythmic feel throughout the composition. It is distinct from variability of note durations in that it considers the duration and timing of notes in the context of the surrounding notes, where the variability of notes is considered across the entire composition.

Melodic Embellishments

A melodic embellishment (also known as an ornament) is a small musical phrase added on top of some recurring melody. The embellishment usually exists within the same key and is often slightly faster than the current tempo to highlight its effect. They are non-essential to the “core” melody of the passage, buy instead serve to decorate and reduce repetition. Melodic embellishments are a fine indicator of complexity.

Repeated Notes

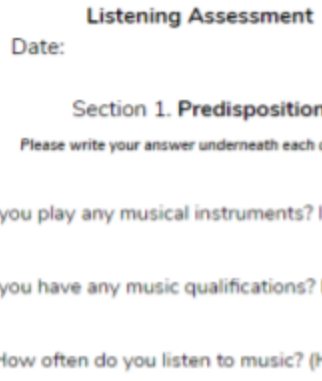
Repeated notes describes the percentage of notes in the composition which have the same pitch as the previous note. A high amount of repeated notes can generally be considered an indicator of low complexity, as complex melodies often contain many different pitches with a wide variety of intervals between pitches.

These descriptors formed the core of the computational analysis component, giving a strong baseline indicator in the corpus of human compositions to compare with the algorithmic compositions. Having performed analysis using the jSymbolic framework, the XML results were processed using python, with various plots and comparisons being drawn using the matplotlib framework.

4.5 Subjective Music Analysis

This section will detail the use of a listening questionnaire on human participants to gauge the perceived quality and complexity of the generated pieces, compared to that of human compositions. The listening questionnaire was performed in the Strathclyde Livingstone Tower Floor 11 labs, with participants being made up of friends and family.

The questionnaire was composed of two sections: a brief disposition about musical ability and knowledge and a nine song listening test.



Listening Assessment

Date:

Section 1. **Predisposition**

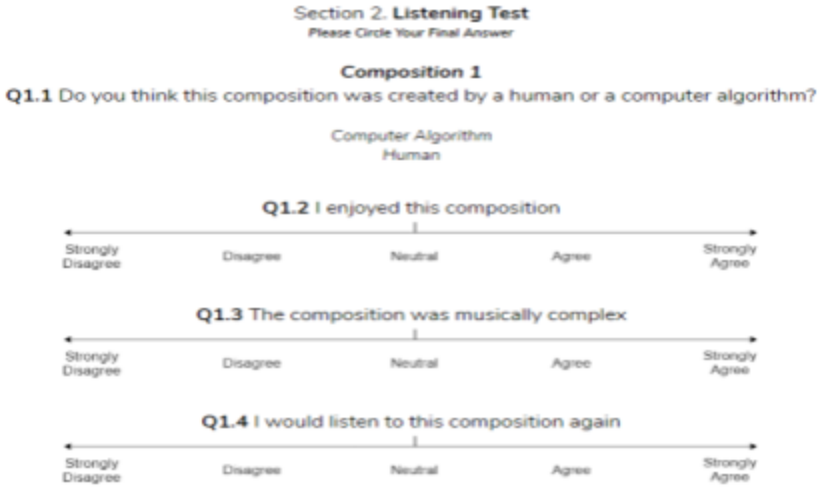
Please write your answer underneath each question

Q1. Do you play any musical instruments? If so, how many?

Q2. Do you have any music qualifications? If so, what level?

Q3. How often do you listen to music? (Hours / Week)

Figure 4.5.1 : Listening Questionnaire Predisposition



Section 2. **Listening Test**

Please Circle Your Final Answer

Composition 1

Q1.1 Do you think this composition was created by a human or a computer algorithm?

Computer Algorithm
Human

Q1.2 I enjoyed this composition

Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree

Q1.3 The composition was musically complex

Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree

Q1.4 I would listen to this composition again

Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree

Figure 4.5.2 : Listening Questionnaire sample question format

As can be seen from figure 4.5.2, the listening test portion of the questionnaire asks participants to decide if the current composition was generated by a computer or written by a human. Following this is a series of questions which asks participants whether they enjoyed the composition, how they felt about the complexity of the piece and if the participant would choose to listen to the piece again. The participants responses are recorded using a likert-style scale, where participants can respond neutrally, agree or disagree, or strongly agree / disagree. The benefit of using a likert-scale in this situation is that it provides perceptual information about how participants responded to all of the compositions. Having a suite of trained musicians perform a test with more of a music theory basis would undoubtedly provide a higher level of insight into what differentiates generated and bespoke compositions at a technical level, but this was unavailable to the researcher and well outwith the scope of the project.

The listening tests were carried out at the strathclyde Livingstone tower labs with a total of 10 participants taking the test. Subjects were given consent forms to complete and immediately return, after which the test promptly began. All subjects wore ATH-M50 headphones to listen to the same compositions, in the same order. Participants listened to a total of 9 compositions (each of which were 20 seconds in length) which were evenly divided among 3 categories: Method A, Method B and human-written compositions.

The results of the subjective analysis section were recorded to paper and later transferred to an XML document for processing and analysis in python.

5. *Analysis of the System*

5.1 Analysis of Network Training

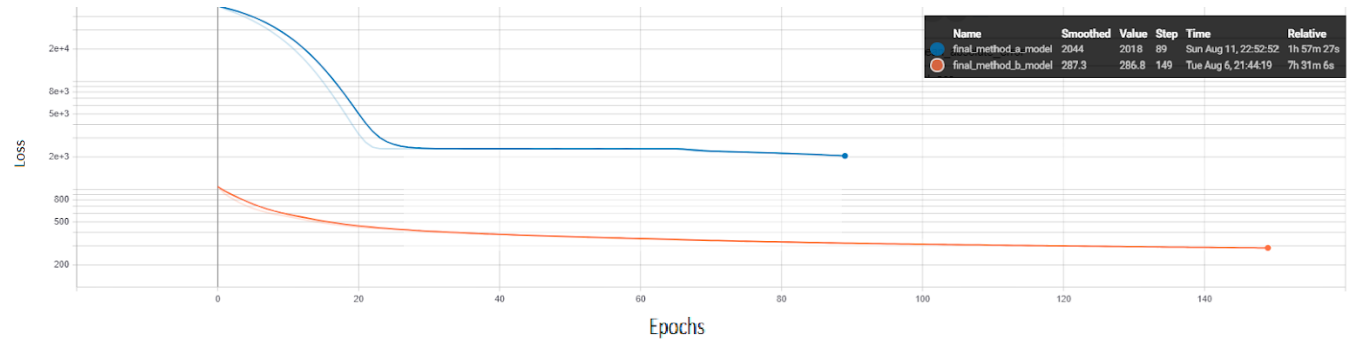


Figure 5.1.1 : Network(s) Training Period Loss curve

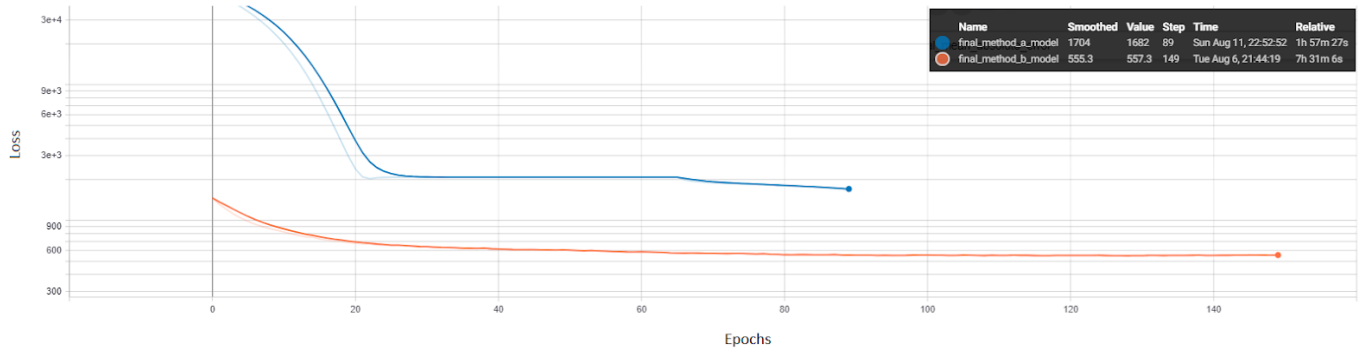


Figure 5.1.2 : Network(s) Validation Loss curve

The above figures show the loss and Mean Absolute Error (MAE) of both network architectures over the training and testing period. The final Method A network trained for 90 epochs, while the Method B network trained for 150. Method A has a steep descent in loss but unfortunately flatlines early, leading to a lack of comprehension. Method B has a more consistent loss but again, it flatlines at approximately epoch 80. This is unlikely to be explained by learning rate, as for both methods, it was already extremely small, meaning even if progress were small, some progress would be made. What is more likely is that both network architectures reached the limit of their comprehension, based on the network configuration, and simply could not learn

anything new. Furthermore, the underlying data may have some underlying noise providing a hard limit to the minimum error.

Hyper-parameters such as the learning rate, batch-size and epochs were dictated through extended experimentation, testing many configurations of both networks, parameters and even the underlying structure of the data.

Method A Parameters

```
params = {}
params["learning_rate"] = 0.000002
params["batch_size"] = 64
params["epochs"] = 90
params["dataset"] = "classical"
params["sequence_length"] = 15
params["data_amount"] = 1.0
```

Method B Parameters

```
# parameter dictionary
params = {}
params["learning_rate"] = 0.000004
params["batch_size"] = 128
params["epochs"] = 150
params["timestep_resolution"] = 10
params["composition_length"] = 60
params["dataset"] = "classical"
params["sequence_length"] = 100
params["num_simultaneous_notes"] = 4
params["data_amount"] = 1.0
```

Figure 5.1.3 : Final parameters for both networks

5.2 Symbolic Analysis

This section will detail the use of symbolic MIR data (obtained from jSymbolic2) to analyse the performance of the two methods, drawing a comparison with a baseline random measurement, as well as a corpus of human compositions. Each human composition was converted to either Method A or B's network representation, then back to MIDI which helped to reduce bias, as both networks had an inherent disadvantage in a drastically lower time-step resolution and a reduction of the number of available pitches.

50 compositions were taken from the Method A & B networks, a random solution and the training dataset. These compositions were then bulk analysed in jSymbolic.

Amount of Arpeggiation

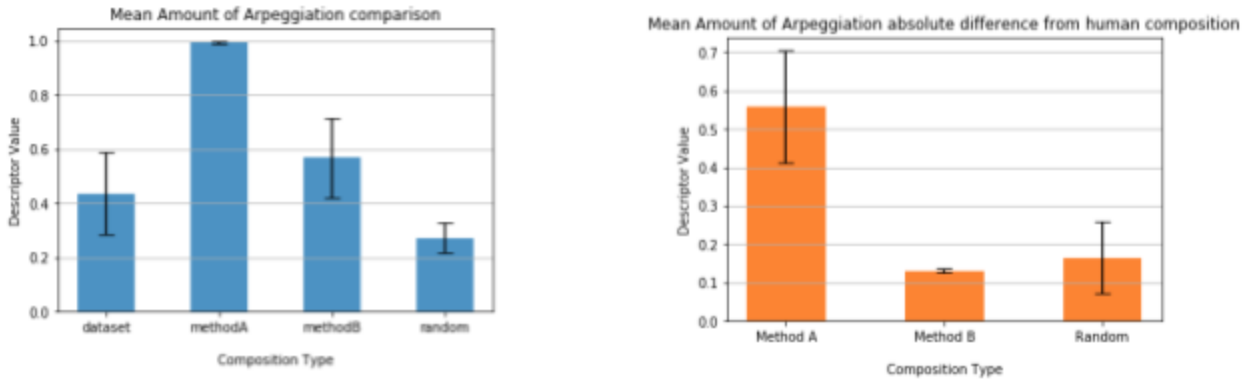


Figure 5.2.1 : Amount of Arpeggiation

Arpeggios are sequences of notes which outline a specific chord, or in some instances an entire key. They are a good indicator that the network understands chordal structures as well as timing intervals as arpeggios are usually played with fixed timing intervals between notes.

Looking at Figure 5.2.1, on the left we have the absolute percentage of arpeggiation across all tested datasets, with the right comparing random, Method A and Method B difference against human compositions. This style was adopted for all MIR metrics as it allows insight into the raw performance of the network, but also can help us to reason how close each method was to achieve a “human-like” sound. From this, we can see that Method B is the clear winner, having the most similar amount of arpeggiation with an almost identical distribution, due to the extremely low standard deviation. Method A reported the highest amount of arpeggiation, though this is likely a bug with the jSymbolic algorithm for arpeggiation, as previously discussed, Method A was unable to capture the structure of the music at all, and infinitely repeated one note across all time-steps. As expected, the random solution performed worst.

Average Note Duration

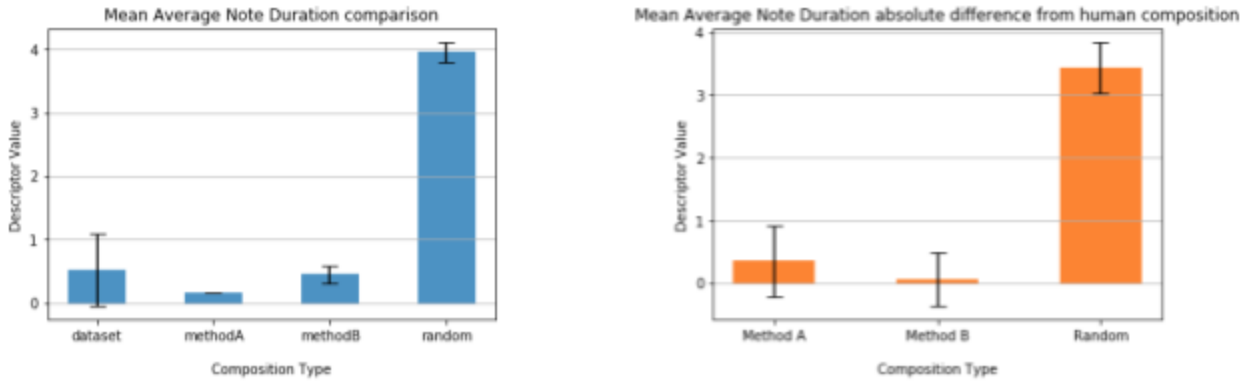


Figure 5.2.2 : Average Note Duration

The average note duration is not a useful indicator for detecting any particular musical feature by itself, however during the experimentation stage of the project, many of the early networks compositions had extremely poor temporal awareness, and compositions would often be extremely long and drawn out (the random solution also suffered from this problem and required a hard limit on the duration of a note, as to not crash jSymbolic). As such, note duration was included to clearly highlight any instances in which notes are placed extremely sporadically.

From Figure 5.2.2 we can see that Method B performs most similarly to human compositions, with a significantly reduced standard deviation, and as such a reduced variety of note durations. Method A performs extremely poorly in this category with practically zero variance to the note duration. Random average was wildly off compared to all other datasets, though variance between durations was similar to both Methods A & B.

Average Number of Simultaneous Pitches

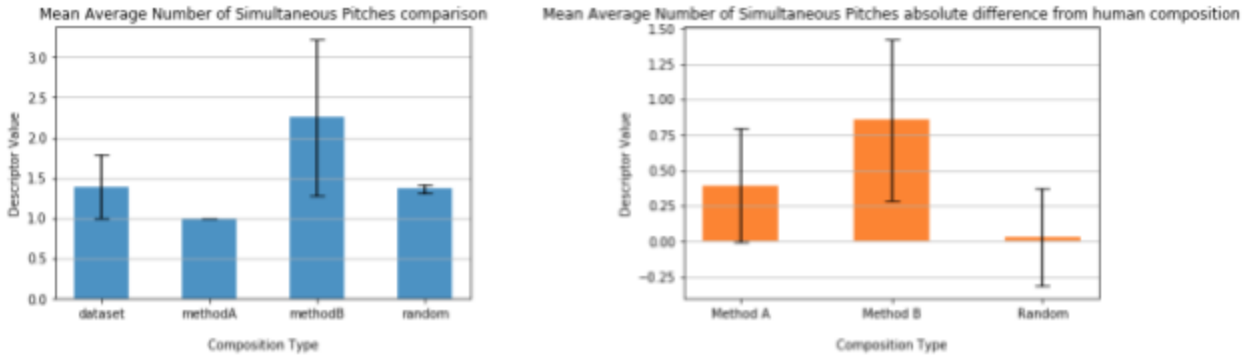


Figure 5.2.3 : Average Number of Simultaneous Pitches

The average number of simultaneous pitches describes the average number of notes active at a given time. This descriptor helps to identify the ratio of melody to chords in the composition. Though it is not a substitute for proper tonal analysis, it is simply a good indicator when compared against human compositions. As can be seen from Figure 5.2.3, surprisingly, the best performing approach is random with a tiny difference in average compared to human compositions, though it should be noted the difference in variance is quite large, with random again having a very low deviation. Method B performed poorly in this regard, with a massive deficit in both error and variance. Method A had the lowest overall difference in number of simultaneous pitches, but still had a large difference in variance due to only one note being played at a time.

Chromatic Motion

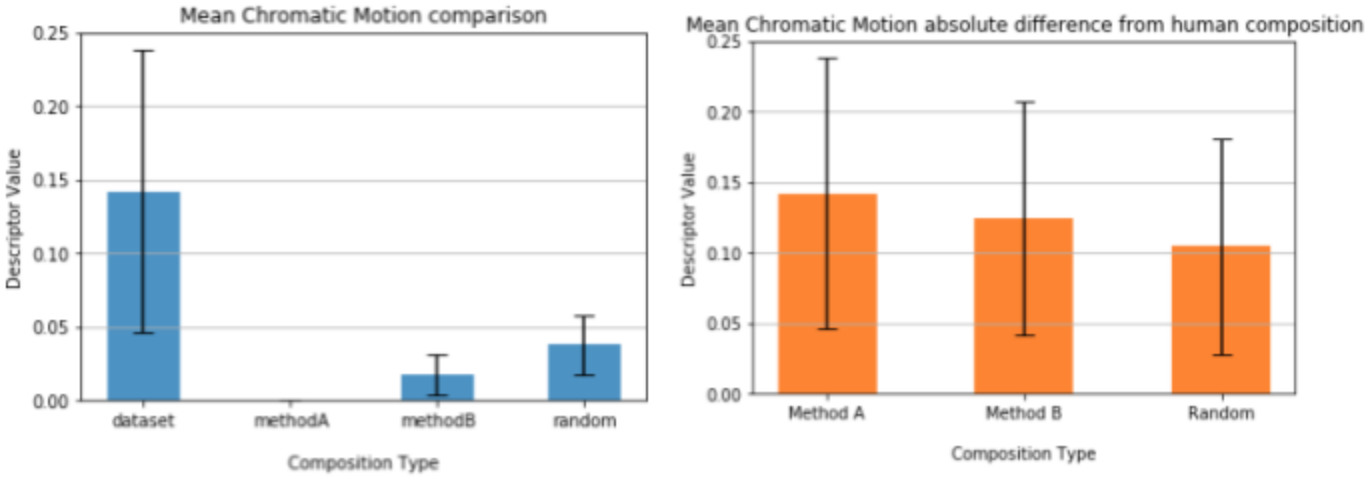


Figure 5.2.4: Chromatic Motion

Chromatic Motion describes a phrase or melody which moves up or down the chromatic scale, as opposed to following the key of the piece. Chromatic motion may often sound inharmonious due to the fact the key is not being respected whatsoever; it is, however, often used in small amounts to add tonal flavour to a musical phrase and can can elicit emotion from the listener. Specifically, chromatic notes which produce a dissonant sound can imply tension, unease and uncertainty. With this, looking at Figure 5.2.4 we can see that the corpus of human compositions had by far the largest amount of chromatic motion as well as the largest variance. The high variance shows us that the mean is not in fact typical of the dataset, and as such, chromatic motion is used across human compositions to an extremely varied degree. All other methods of composition failed to even come close to human performance in terms of chromaticism, with the random method being the closest available method. This is to be expected as random solutions have no awareness about any kind of musical structure, thus the likelihood that a musical key is respected is extremely low. Both Methods A & B performed poorly in terms of matching a human composition, with Method B performing only slightly better than Method A, which exhibited no chromatic motion at all.

Contrary Motion

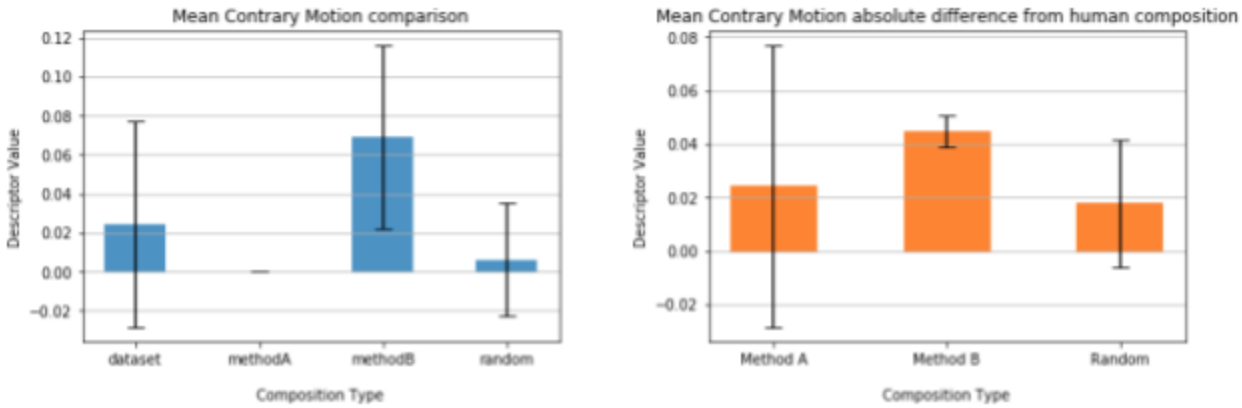


Figure 5.2.5 : Contrary Motion

Contrary motion describes the musical phenomena where two or more musical voices are moving in opposite directions, specifically in terms of pitch. Contrary motion is not specifically an indicator of complex music, however it shows at least a comprehension of key, as for contrary motion to be respected, both voices must be traversing the same scale in opposite directions. Here, the random solution was the closest to that of the human compositions in terms of mean value, however the difference in deviation was rather significant, with human compositions exhibiting a wider range of contrary motion overall. Though Method B had a much higher mean score, the difference in deviation between Method B and human compositions was relatively small, meaning the variance in contrary motion was quite similar to that of human compositions, despite having more on average. Method A was by far the worst performer compared to human compositions, exhibiting no contrary motion at all.

Stepwise Motion

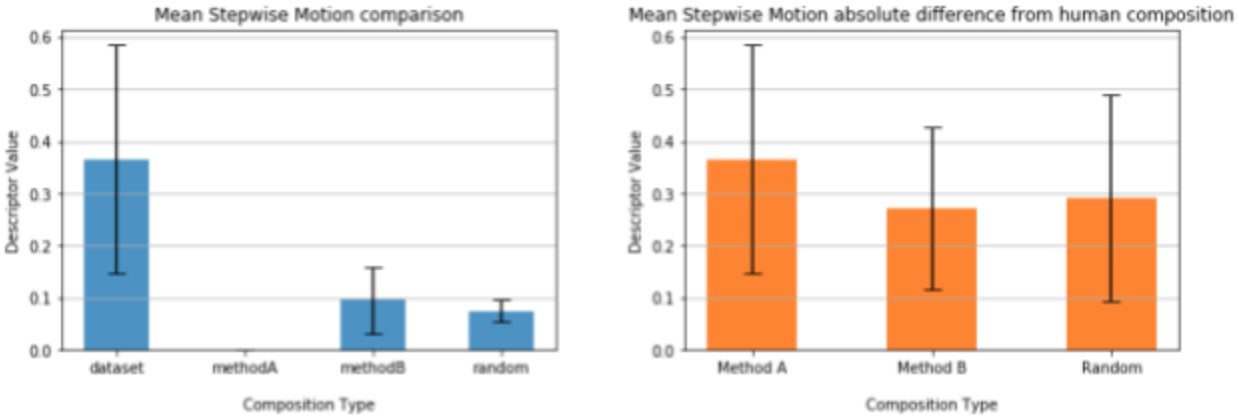


Figure 5.2.6 : Stepwise Motion

Stepwise motion refers to movement in a musical passage that is linearly ascending or descending a given musical scale. Higher values show a clear comprehension of key, though stepwise motion is not necessarily considered a universally positive indicator of complexity in music. The opposite of stepwise motion would be *skipwise* motion, though jSymbolic unfortunately does not provide this as a descriptor. Skipwise motion describes a melodic line that again, respects the key of the current section, but does not linearly ascend the scale of the key, instead skipping between the various pitch intervals. From Figure 5.2.6 we can see that no method came close to matching the human training set score, with a significantly higher mean score and variance than any other method. Again, Method B was the closest to the human compositions but not by a substantial margin over random, which is disappointing. Method A was again the worst performer having the highest deficit in both mean value and deviation.

Similar Motion

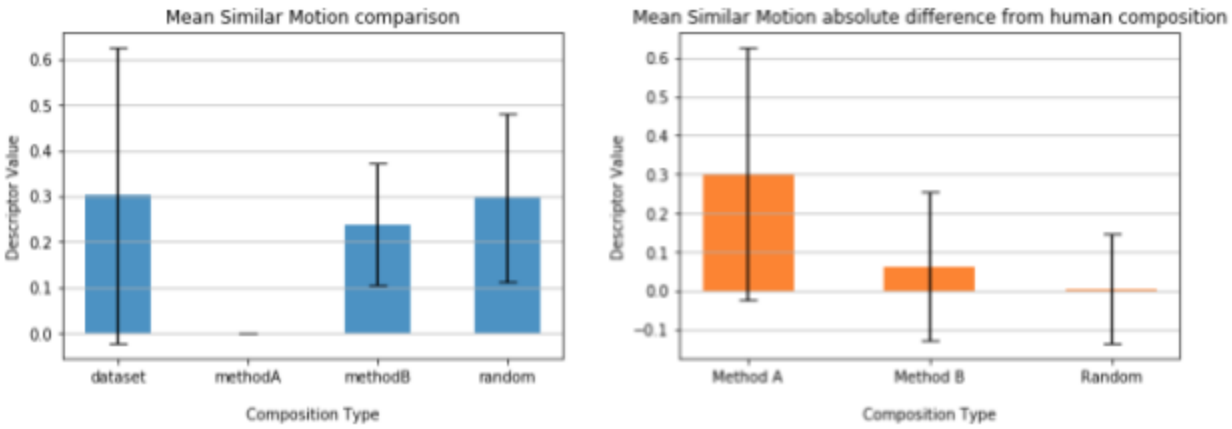


Figure 5.2.7 : Similar Motion

Similar motion describes the musical phenomena where two or more musical voices are moving in similar (though not necessarily perfectly similar) directions. Similar motion implies the presence of a counter melody: both voices move in the same approximate direction at different pitch intervals as we would expect to see with a counter-melody; we cannot, however, explicitly say that the phrases described by similar motion belong to the same key, and as such cannot guarantee a harmonious counter-melody. Unfortunately neither Method A or B were able to achieve similar results to that of human compositions, with both methods being beaten by the random solution. The significant deviation on the human composed dataset also implies that the mean is not an indicative value for the rest of the human dataset. Similar motion may be an unfit descriptor for comparison then, as without a strong human baseline comparison, the network results are rendered useless.

Repeated Notes

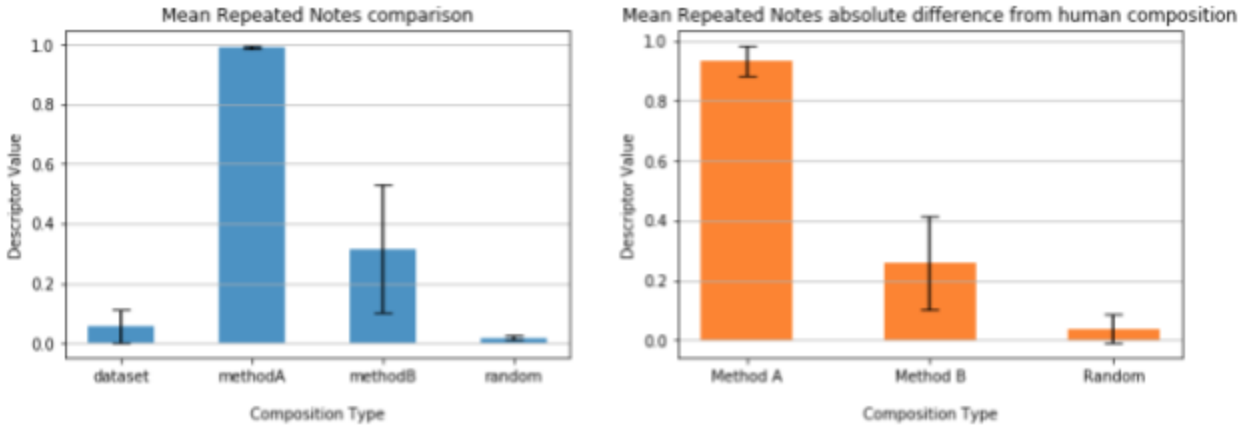


Figure 5.2.8 : Repeated Notes

The repeated notes indicator describes the number of notes that are the same as the note that came before them; this is a good indicator of low complexity. Unfortunately both methods again performed poorly against a human dataset with even the random solution outpacing both methods, with respect to achieving human like quality. One important consideration is that repeated notes refers only to the pitch, and does not account variations in timing or velocity, and as such, the random solution may be at a slight advantage, as PRNGs are by nature designed to produce unpredictable results, and the same randomly generated pitch is unlikely to appear many times in a row. Looking at both Methods A and B, the clear winner is B with a significantly lower mean repeated notes and wider variance. Here, a low variance in Method A indicates an extremely strong prevalence of repeated notes; this is true as almost all compositions were the same note repeated over and over again, with only minor differences in timing and duration.

Melodic Embellishments

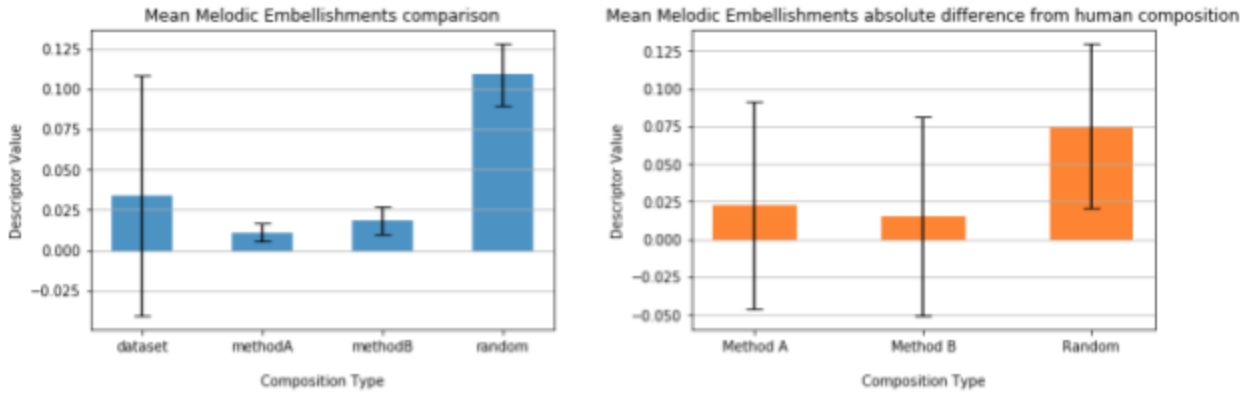


Figure 5.2.9 : Melodic Embellishments

The melodic embellishments descriptor describes the number of ornamental notes in a given composition. Embellishments serve to make a passage of music more exciting and less repetitive. As can be seen from Figure 5.2.9, the best performing solution was Method B, though the difference in variance between B and human compositions is rather substantial. The human compositions exhibited a massive variance, and as such, may be a poor baseline for comparison. It does highlight, however, that while Method B appears to be the best approach out of the other computational methods, in reality the deviation is much more constrained than human compositions, meaning that while human compositions may sometimes have many embellishments in a piece, the generative solution is likely to have a very small amount of ornamentation by comparison. Method A suffers from the same problem to an even more significant degree, with lower embellishments and an even narrower deviation. The random solution had the least accurate mean score of all generative solutions, but did in fact beat both Method A and B in terms of variance.

Melodic Pitch Variety

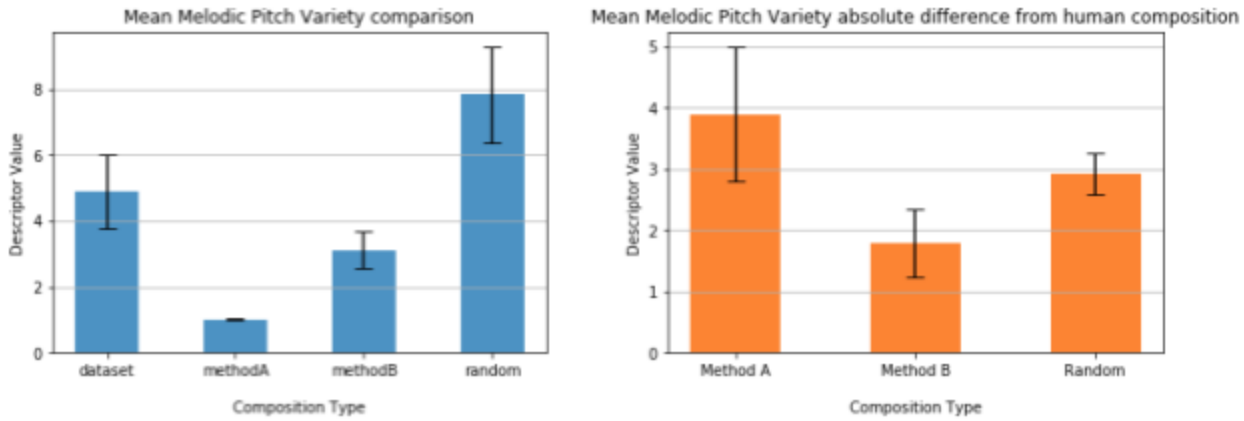


Figure 5.2.10 : Melodic Pitch Variety

Melodic Pitch Variability is exactly as it sounds, it describes the variability of melodic pitches in a given composition. Though not precisely an indicator of complexity, this metric gives us some insight into the variety of musical notes considered for the composition. The closest method to human compositions for this descriptor was Method B in terms of mean, though the variance of the random solution was slightly closer to that of the human compositions.

Variability of Note Durations

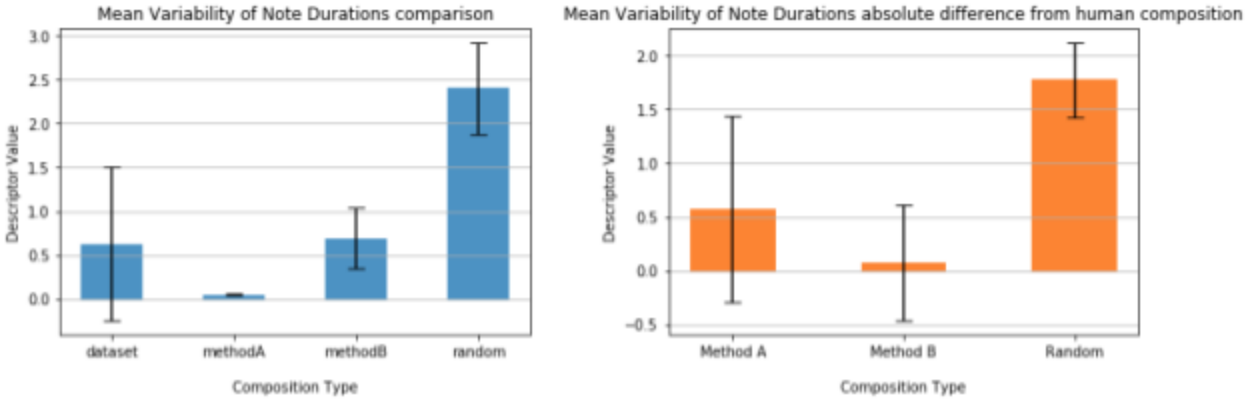


Figure 5.2.11 : Variability of Note Durations

Similar to the performance on variability of pitches, method B achieves a very similar mean score when compared to the other methods, however random again comes closest in terms of variance. Method A showed the worst performance again, with no variability to note durations whatsoever recorded by jSymbolic.

Variation of Dynamics

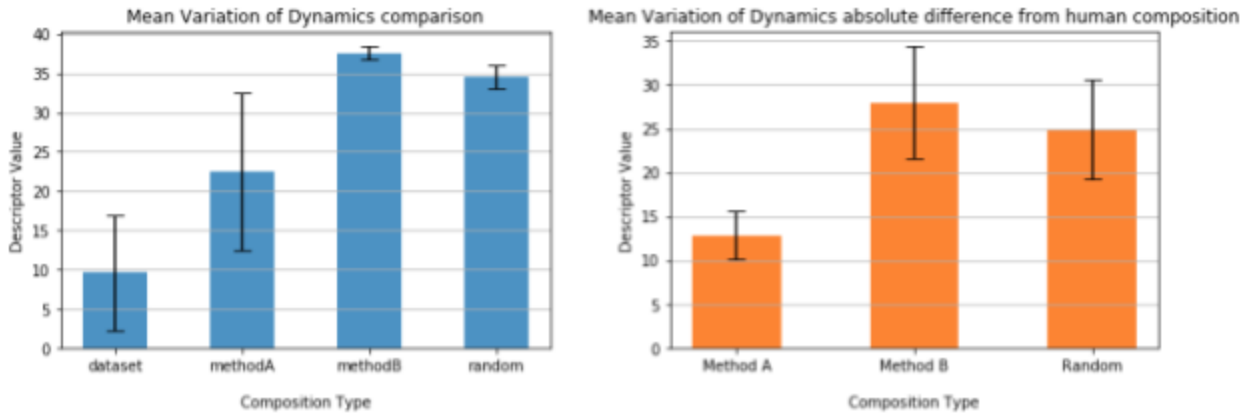


Figure 5.2.12 : Mean Variation of Dynamics

For the variability of dynamics, surprisingly, the best performance was had by Method A, which had a closer mean score and variance to human compositions than either random or Method B. Method B was the worst performer with an exceedingly high variability of dynamics, with very little variance.

Findings from symbolic analysis

From symbolic analysis alone, we can see that neither Method A or B were able to fully match the performance of the human compositions, though Method B was a better candidate exhibiting promising results in 6 of 12 symbolic descriptors. Disappointingly, Method B did not vastly outperform either Method A or the random in any of the Motion descriptors, good indicators that not only pitch but also temporal structure are being comprehended by the network. Method A was unsurprisingly a very poor performer, again, due to the fact that the network was unable to at all capture the structure of the training data, and resulted in either identical, or similar output regardless of the input.

The random solution may appear to be a rather strong candidate in theory, but based on the average note duration descriptor alone, random solutions become too unbearably slow to be considered listenable. Furthermore, the surprising results in variety of pitch may be more an indication that the descriptor was a poor analytical method more than that random compositions exhibit more melodic variability, as this simply was not the case.

5.3 Subjective Analysis

This section will evaluate the human response to the generated compositions of Method A and B. As previously discussed, the data was collected by conducting an informal listening questionnaire, where participants were asked to identify if a composition had been composed by a human or a computer. Furthermore, participants were asked to gauge their feelings on the composition by responding to a series of statements using a likert-style scale. Participants were played nine 20-second long compositions, with three compositions belonging to the human corpus, three belonging to Method A and three belonging to Method B.

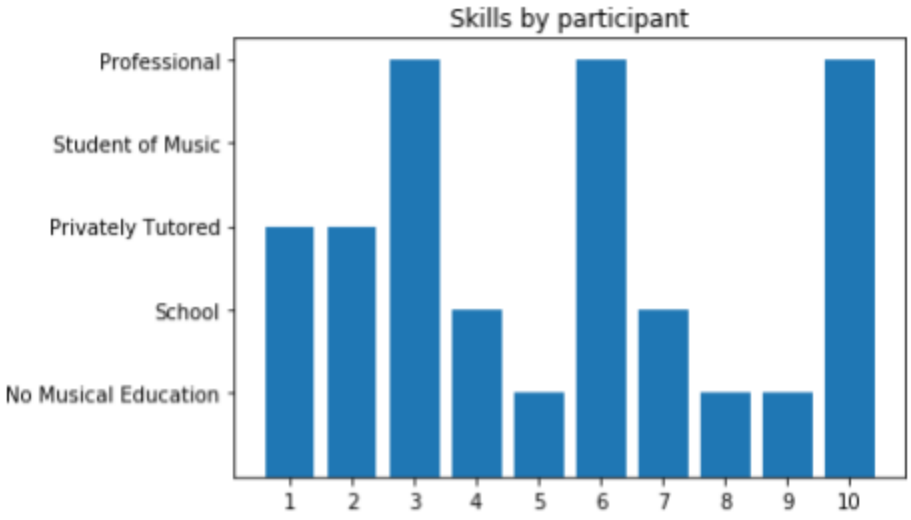


Figure 5.3.1 : All participants skill level

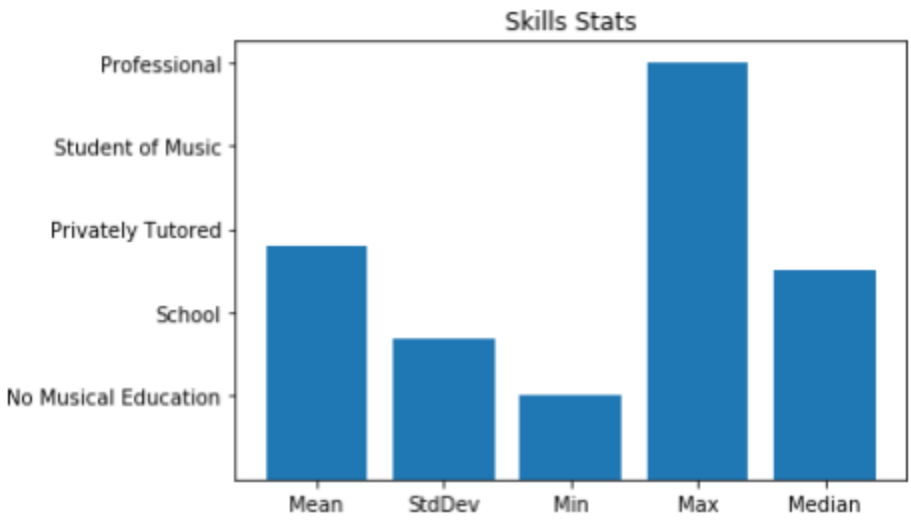


Figure 5.3.2 : Participant Skill Statistics

Figure 5.3.1 and 5.3.2 show us the skill level across the participants of the study, as well as a statistical plot showing the mean, deviation, min, max and median of participant skill level. All statistics indicate that the test group had an extremely varied level of musical skill, with two participants having no musical education whatsoever, and three professional musicians. The deviation was sufficiently high enough to regard the value of the mean as unindicative of the average skill of a participant.

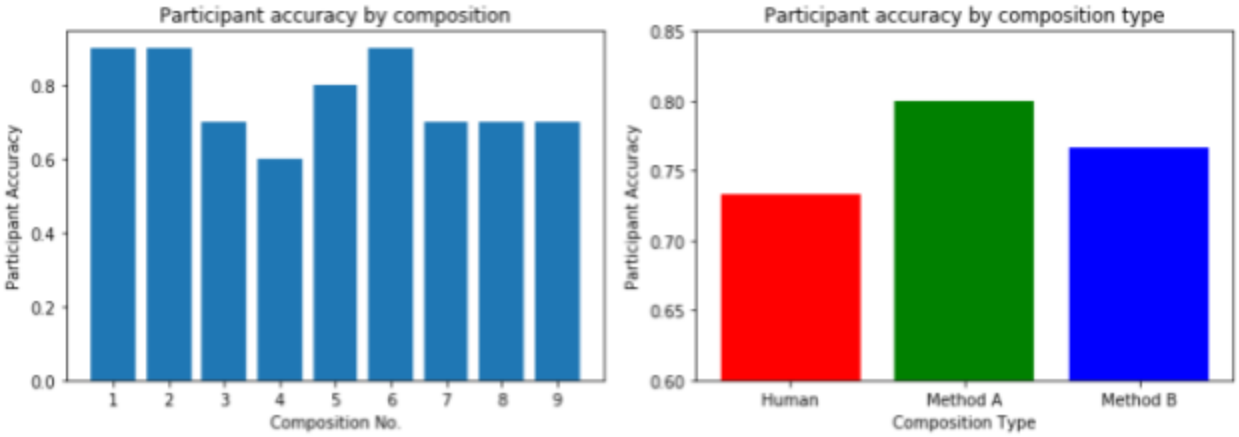


Figure 5.3.3 : Participant Accuracy

From Figure 5.3.3, participants accuracy in identifying the composition source is shown. On the left is the mean accuracy by question, with the right hand figure showing the mean accuracy of predictions by composition type. Unsurprisingly, Method A was the easiest to identify, due to the extremely simple nature of the “compositions”, with Method B following and human compositions the most difficult to identify for test participants.

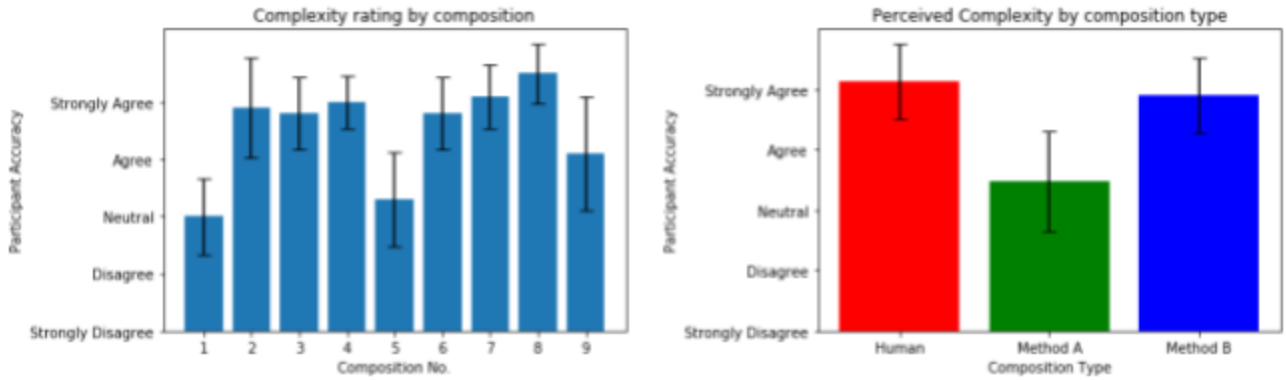


Figure 5.3.4 : Participants Perceived Complexity

Figure 5.3.4 shows participants responses to the statement “This composition was musically complex”. Method B was clearly considered to produce more complex music, and almost matches that of the human compositions, where Method A can be considered significantly less complex. It is strange that the Method A results are as high as they are, given the indicated quality of the compositions from symbolic analysis.

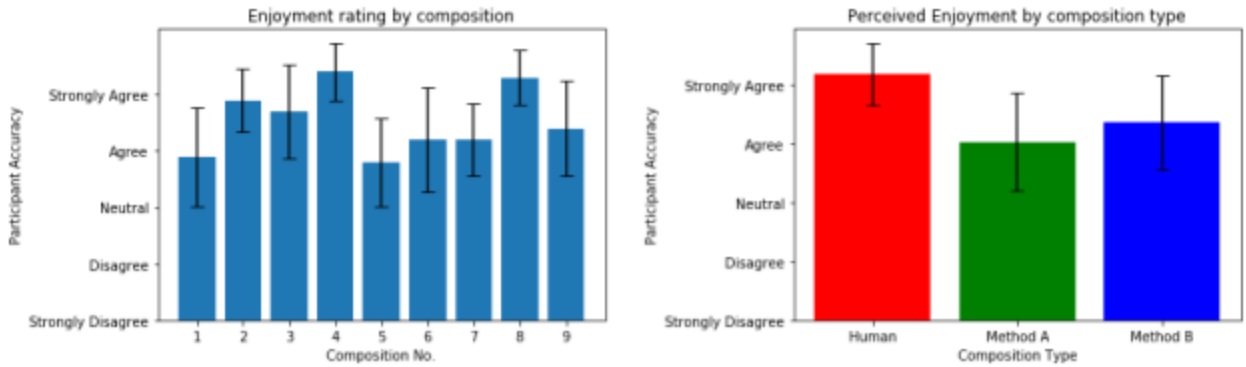


Figure 5.3.5 : Participants Perceived Enjoyment

Figure 5.3.5 shows participants perceived enjoyment of the compositions across all compositions, and by their source. Participants read the statement “I enjoyed this composition” and were asked to respond using a standard likert scale. Unfortunately, compositions from both Method A & B were significantly less enjoyable to participants than the human-written compositions. Both methods also have a slightly larger variance than that of the human written music, implying that the human compositions are more consistently enjoyable than either of the networks.

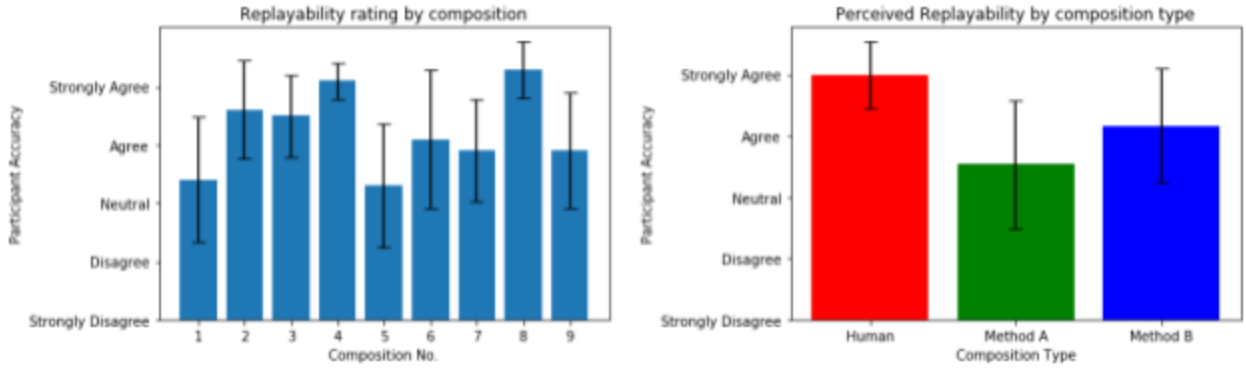


Figure 5.3.6 : Participant Replayability

Finally, Figure 5.3.6 shows participants likelihood to relisten to a composition, with participants reading the statement “I would listen to this composition again”, again responding using a likert scale. Method B was again, slightly closer to matching the human compositions and had a moderate advantage compared to Method A. With this being said, the human compositions again had a significant advantage over both Methods of generation and a significantly lower variance, meaning that participants wanted to listen to human compositions more consistently than either generative method.

Findings from Subjective Analysis

Reflecting on the results of the listening questionnaire, it is clear that participants preferred the human compositions in every measured way. Surprisingly, test participants were least likely to correctly identify the human compositions out of all of the presented methods. Method A had the highest rate of true identification, and this also shows from participants responses to these compositions: Method A was consistently the least preferred method of composition, scoring lowest in complexity, replayability and enjoyability. Method B had more success falling directly in between Method A and the human compositions, in terms of participants ability to correctly identify the compositions source. Specifically, Method B exhibited a nearly identical perceptual complexity to that of the human compositions. This is surprising as it almost directly contradicts the findings from the symbolic analysis stage; while the network was proficient in certain aspects (notably, variability of note duration), it failed to capture many other temporal aspects, such as motion and ornamentation. Despite the perceived similar complexity, participants still preferred the human compositions by a significant margin, and were much more likely to relisten to a human composition than any Method B composition.

6. Conclusion

This project has assessed the feasibility of using Recurrent Neural Networks as a method of generating music by scoping existing solutions, identifying the principal problems currently facing RNN music systems and provided two potential solutions for evaluation. The overall goal was to produce “human sounding” music, with a goal of achieving a long-term musical structure in the compositions. Unfortunately, neither Method A or B were able to fully achieve this goal. Despite this, the generations from the network were not wholly unpleasant, and while on average human participants preferred the human compositions, some of the participants also enjoyed the generated compositions.

To increase the number of pleasant compositions the network produces, some combination of a selection algorithm (such as a GA) and jSymbolic analysis data could be used to identify the best compositions, in terms of similarity to human compositions. This would also save a lot of man-hours, as instead of having to listen to each composition individually, the system could identify the most interesting compositions autonomously. Furthermore, continued development of GTTM and its computational applications could provide a much more granular and structure-focussed analysis, as opposed to purely empirical means.

One potential issue is the relatively small size of the training dataset. This imposed a catch-22 situation: whilst a larger dataset could well have increased the performance, the existing dataset and network architecture had to be carefully tweaked and refined to get running on a relatively performant PC, though the low amount of GPU memory (4GB) may well have imposed a greater restriction on the network than initially thought. If further development were to be continued with this project, a better GPU and larger dataset could reasonably aid the performance of the system, as it is a well known axiom that machine learning generally scales well with large amounts of data. Alternatively, a cloud hosting solution (such as Google Cloud Platform) could be used to remove these

restrictions on the network, though this requires a significant investment of time to setup GPU accelerated machine learning.

Based on the findings from the analysis stage, this project would recommend the use of a sequence as the fundamental representation, in which an arbitrary number of notes may be active (Method B). Though this method produces more data and takes longer to train, the results are infinitely more musical than that of Method A's representation, which sees each note occupy its own timestep. One potential reason Method A's representation failed to capture the structure of the data was the fact that the timing information was encoded as a feature of the data, rather than being represented explicitly by the sequence (of data). One method highlighted in the literature proposes a fixed interval between notes to solve this problem, however this directly contradicts the aim of replicating human sounding music, as notes are often placed in extremely irregular patterns. There is the potential that timing and pitch features are simply too distant in terms of relation to be grouped together in one datapoint, which is indicated by the fact that our network - which did not encode timing as a feature - performed significantly better.

Looking to the future, RNNs and specifically LSTM / GRU networks still have much research to undertake when aiming to identify the best techniques and practices when dealing with generative music. Some other network architectures have had massive successes well above what has been seen using RNNs. MuseGAN (Dong et al., 2019) shows extremely promising results, being able to simultaneously compose five interwoven tracks of music for bass, guitar, piano, drums and strings. The transformer discussed in the literature review also exhibited impressive performance, besting a traditional RNN and LSTM network by a substantial margin when attempting to recreate Bach chorales.

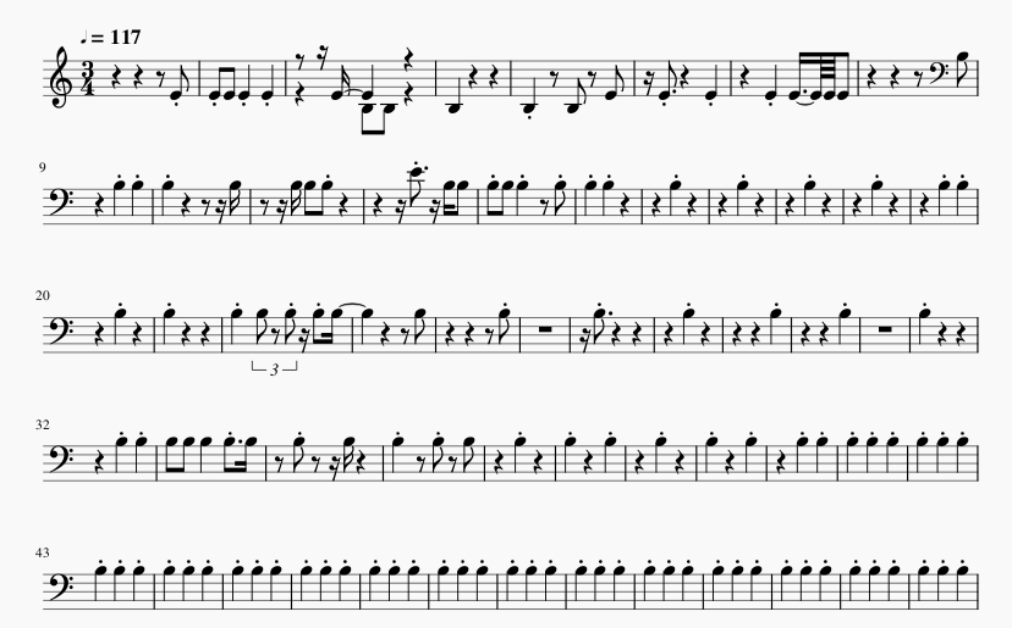


Figure 6.1 : Example Method A composition in Music Notation

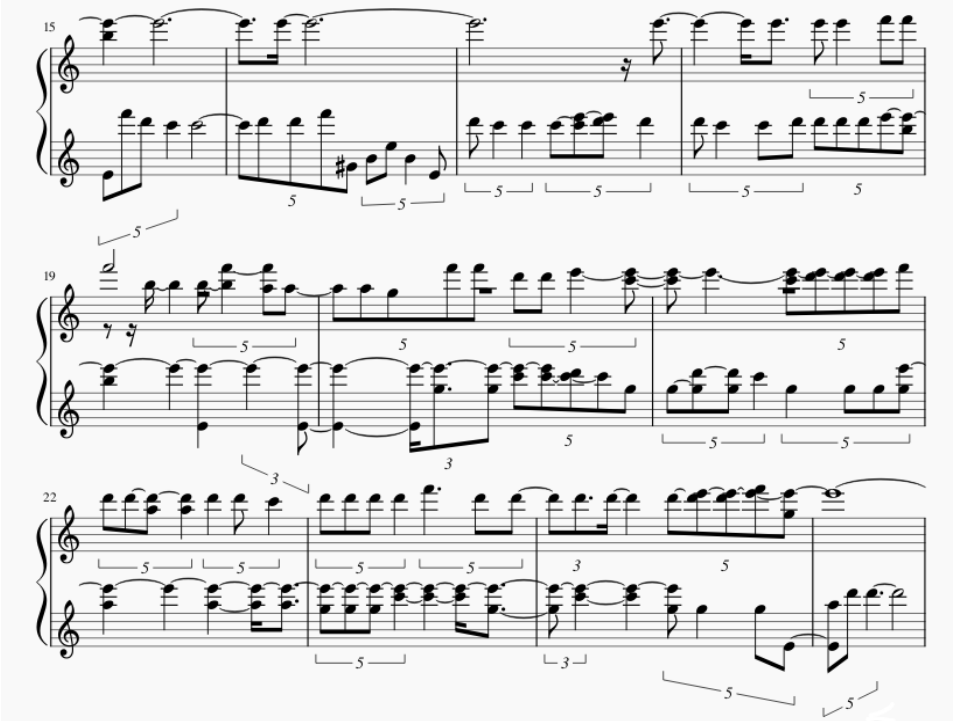


Figure 6.2 Example Method B composition in Music Notation

The code for this project is available at:

https://github.com/liamdx/RnnMu

7. Appendices

Anssi Klapuri and Davy, M. (2011). *Signal processing methods for music transcription*. New York; London: Springer.

Antti. Eronen (2017). *CHORUS DETECTION WITH COMBINED USE OF MFCC AND CHROMA FEATURES AND IMAGE PROCESSING FILTERS*. [online] semanticscholar.org. Available at: https://www.semanticscholar.org/paper/CHORUS-DETECTION-WITH-COMBINED-USE-OF-MFCC-AND-AND-Eronen/1a279740f210c77869a7f0fcf890446e1a2e5174 [Accessed 16 Mar. 2019].

Ariza, C. (2009). The Interrogator as Critic: The Turing Test and the Evaluation of Generative Music Systems. *Computer Music Journal*, 33(2), pp.48–70.

Bengio, Y., Simard, P. and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), pp.157–166.

Berezovsky, J. (2019). The structure of musical harmony as an ordered phase of sound: A statistical mechanics approach to music theory. *Science Advances*, [online] 5(5), p.eaav8490. Available at: https://advances.sciencemag.org/content/5/5/eaav8490/tab-pdf [Accessed 5 Jul. 2019].

Bogdanov, D., Wack, N., Gómez, E., Gulati, S., Herrera, P., Mayor, O., Roma, G., Salamon, J., Zapata, J. and Serra, X. (2013). *ESSENTIA: AN AUDIO ANALYSIS LIBRARY FOR MUSIC INFORMATION RETRIEVAL*. [online] Available at: https://repositori.upf.edu/bitstream/handle/10230/32252/essentia_ismir_2013.pdf?sequence=1&isAllowed=y [Accessed 11 Jul. 2019].

Boone, H.N. and Boone, D.N. (2012). Analyzing Likert Data. *Journal of Extension*, 50(2).

Bown, O. and Lexer, S. (2006). Continuous-Time Recurrent Neural Networks for Generative and Interactive Musical Performance. *Lecture Notes in Computer Science*, pp.652–663.

Cataltepe, Z., Yaslan, Y. and Sonmez, A. (2007). Music Genre Classification Using MIDI and Audio Features. *EURASIP Journal on Advances in Signal Processing*, 2007(1).

Chen, C.J. (2001). Creating melodies with evolving recurrent neural networks - IEEE Conference Publication. In: *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*. [online] IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/938515 [Accessed 16 Mar. 2019].

Cheng-Zhi, A., Huang, Vaswani, A., Uszkoreit, J., Shazeer, N., Simon, I., Hawthorne, C., Dai, A., Hoffman, M., Dinculescu, M., Eck, D. and Brain, G. (2018). *MUSIC TRANSFORMER: GENERATING MUSIC WITH LONG-TERM STRUCTURE*. [online] Available at: https://arxiv.org/pdf/1809.04281.pdf [Accessed 11 Jul. 2019].

Chung, J., Gulcehre, C., Cho, K. and Bengio, Y. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. [online] arXiv.org. Available at: https://arxiv.org/abs/1412.3555 [Accessed 17 Jul. 2019].

Collins, K. (2009). An Introduction to Procedural Music in Video Games. *Contemporary Music Review*, 28(1), pp.5–15.

Cuéllar, M.P., Delgado, M. and Pegalajar, M.C. (2006). AN APPLICATION OF NON-LINEAR PROGRAMMING TO TRAIN RECURRENT NEURAL NETWORKS IN TIME SERIES PREDICTION PROBLEMS. *Enterprise Information Systems VII*, 7, pp.95–102.

Cuthbert, M.S. and Ariza, C. (2010). *music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data*. [online] Available at:

https://dspace.mit.edu/bitstream/handle/1721.1/84963/Cuthbert_Ariza_ISMIR_2010.pdf?sequence=1&isAllowed=y [Accessed 11 Jul. 2019].

Dannenberg, R. (2006). *The Interpretation of MIDI Velocity*. [online] pp.193–196. Available at: https://pdfs.semanticscholar.org/92a7/dc5007d770e0c5a3a637f66ee128ba107a92.pdf [Accessed 19 Jul. 2019].

Darko, J. (2009). *Artscape - Brian Eno In Conversation 2009*. [online] Vimeo. Available at: https://vimeo.com/5763066 [Accessed 5 Jul. 2019].

Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R. and Makhoul, J. (2014). *Fast and Robust Neural Network Joint Models for Statistical Machine Translation*. [online] Available at: https://www.aclweb.org/anthology/P14-1129.

Dodd, A. (2015). *Assessing the Suitability of the GTTM as the Basis of a Generative Music System*. [online] Academia.edu. Available at: https://www.academia.edu/31668156/Assessing_the_Suitability_of_the_GTTM_as_the_Basis_of_a_Generative_Music_System?source=swp_share [Accessed 25 Jun. 2019].

Doll, C. (2011). Rockin' Out: Expressive Modulation in Verse-Chorus Form. *Science for Music Theory*, [online] 17(3). Available at: http://www.mtosmt.org/issues/mto.11.17.3/mto.11.17.3.doll.pdf [Accessed 16 Mar. 2019].

Dong, H.W., Hsiao, W.-Y., Yang, L.-C. and Yang, Y.-H. (2019). *MuseGAN Results*. [online] MuseGAN. Available at: https://salu133445.github.io/musegan/results [Accessed 15 Aug. 2019].

Eck, D. and Schmidhuber, J. (2002a). *A First Look at Music Composition using LSTM Recurrent Neural Networks*. [online] Available at: http://people.idsia.ch/~juergen/blues/IDSIA-07-02.pdf [Accessed 16 Mar. 2019].

Eck, D. and Schmidhuber, J. (2002b). Finding temporal structure in music: blues improvisation with LSTM recurrent networks - IEEE Conference Publication. In: *Ieee.org*. [online] IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/1030094 [Accessed 16 Mar. 2019].

Edwards, M. (2011). Algorithmic composition. *Communications of the ACM*, [online] 54(7), p.58. Available at: http://people.cs.vt.edu/~kafura/CS6604/Papers/Algorithmic-Composition-CT-Music.pdf [Accessed 16 Mar. 2019].

Fazenda, B. (2014). *How to Design and Conduct Listening Tests for Audio and Acoustics*. [online] Available at: https://usir.salford.ac.uk/id/eprint/34338/1/Fazenda-Uni%20of%20Salford%202015%20-%20How%20to%20Design%20and%20Conduct%20Listening%20Tests.pdf [Accessed 25 Jun. 2019].

Franklin, J.A. (2006). Recurrent Neural Networks for Music Computation. *INFORMS Journal on Computing*, 18(3), pp.321–338.

Gers, F.A., Schmidhuber, J. and Cummins, F. (2000). Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10), pp.2451–2471.

Gingras, B., Marin, M.M. and Fitch, W.T. (2014). Beyond Intensity: Spectral Features Effectively Predict Music-Induced Subjective Arousal. *Quarterly Journal of Experimental Psychology*, 67(7), pp.1428–1446.

Graves, A., Mohamed, A. and Hinton, G. (2013). *Speech recognition with deep recurrent neural networks - IEEE Conference Publication*. [online] Ieee.org. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6638947 [Accessed 16 Mar. 2019].

Hamanaka, M., Hirata, K. and Tojo, S. (2007). *ATTA: IMPLEMENTING GTTM ON A COMPUTER*. [online] Available at: https://pdfs.semanticscholar.org/a217/804953b8fee9dee0635a5a50def5a2c1c38e.pdf [Accessed 11 Jul. 2019].

Hamanaka, M., Hirata, K. and Tojo, S. (2014). *MUSICAL STRUCTURAL ANALYSIS DATABASE BASED ON GTTM*. [online] ISMIR. Available at: http://gttm.jp/hamanaka/wp-content/uploads/2015/12/ISMIR2014-hamanaka.pdf [Accessed 25 Jun. 2019].

Hamanaka, M., Hirata, K. and Tojo, S. (2015). Implementing Methods for Analysing Music Based on Lerdahl and Jackendoff's Generative Theory of Tonal Music. *Computational Music Analysis*, pp.221–249.

Hutchings, P. (2017). *Talking Drums: Generating drum grooves with neural networks*. [online] arXiv.org. Available at: https://arxiv.org/abs/1706.09558 [Accessed 16 Mar. 2019].

Jackendoff, R. (1994). *Consciousness and the computational mind*. Cambridge (Mass.) ; London: Mit Press.

Jackendoff, R. (2009). Parallels and Nonparallels between Language and Music. *Music Perception: An Interdisciplinary Journal*, 26(3), pp.195–204.

Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *NIPS 2012*, [online] pp.1097–1105. Available at: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networ [Accessed 15 Aug. 2019].

Krumhansl, C.L. (2002). Music: A Link Between Cognition and Emotion. *Current Directions in Psychological Science*, 11(2), pp.45–50.

Langston, P.S. (1989). *Six Techniques for Algorithmic Music Composition*. [online] Available at: http://peterlangston.com/Papers/amc.pdf [Accessed 16 Mar. 2019].

Lerdahl, F. and Jackendoff, R. (1983). *A generative theory of tonal music*. Cambridge ; London: The Mit Press.

Li, H., Tang, Z., Fei, X., Chao, K.-M., Yang, M. and He, C. (2017). A Survey of Audio MIR Systems, Symbolic MIR Systems and a Music Definition Language Demo-System. *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*.

Liu, I.-T. and Ramakrishnan, B. (2014). *BACH IN 2014: MUSIC COMPOSITION WITH RECUR- RENT NEURAL NETWORK*. [online] Available at: https://arxiv.org/pdf/1412.3191.pdf [Accessed 27 Jun. 2019].

Lu, Q., Chen, X., Yang, D. and Wang, J. (2010). BOOSTING FOR MULTI-MODAL MUSIC EMOTION CLASSIFICATION. *ISMIR 2010*. [online] Available at: https://www.researchgate.net/profile/Chen_Xiaoou/publication/220723554_Boosting_for_Multi-Modal_Music_Emotion_Classification/links/558a857f08ae1110021d34d8.pdf.

Luong, M.-T., Sutskever, I., Le, Q. and Zaremba, W. (2015). *Addressing the Rare Word Problem in Neural Machine Translation*. [online] Available at: https://nlp.stanford.edu/pubs/acl15_nmt.pdf [Accessed 19 Jul. 2019].

Maas, A., Daly, R., Pham, P., Huang, D., Ng, A. and Potts, C. (2011). *Learning Word Vectors for Sentiment Analysis*. [online] Available at: https://www.aclweb.org/anthology/P11-1015 [Accessed 19 Jul. 2019].

Mckay, C., Cumming, J. and Fujinaga, I. (2018). *JSYMBOLIC 2.2: EXTRACTING FEATURES FROM SYMBOLIC MUSIC FOR USE IN MUSICOLOGICAL AND MIR RESEARCH*. [online] Available at: https://archives.ismir.net/ismir2018/paper/000026.pdf [Accessed 19 Jul. 2019].

Melicher, W., Ur, B., Segreti, S.M., Komanduri, S., Bauer, L., Christin, N. and Lorrie Faith Cranor (2016). *Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks*. [online] Available at: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_melicher.pdf [Accessed 19 Jul. 2019].

Morrison, S.J., Demorest, S.M. and Stambaugh, L.A. (2008). Enculturation Effects in Music Cognition. *Journal of Research in Music Education*, 56(2), pp.118–129.

Odekerken, D. (2018). Audio-Symbolic Alignment of Popular Music with application to Automatic Chord Estimation. *Library.uu.nl*. [online] Available at: https://dspace.library.uu.nl/handle/1874/372620 [Accessed 19 Jul. 2019].

Olah, C. (2015). *Understanding LSTM Networks*. [online] Github.io. Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Paine, T., Jin, H., Yang, J., Lin, Z. and Huang, T. (2013). *GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training*. [online] arXiv.org. Available at: https://arxiv.org/abs/1312.6186 [Accessed 16 Mar. 2019].

Parizet, E., Hamzaoui, N. and Sabatié, G. (2005). Comparison of Some Listening Test Methods: A Case Study: Ingenta Connect. *Acta Acustica united with Acustica*, [online] 91(2), pp.356–364. Available at: https://www.ingentaconnect.com/content/dav/aaua/2005/00000091/00000002/art00018#expand/collapse [Accessed 19 Jul. 2019].

Pascanu, R., Mikolov, T. and Bengio, Y. (2013). *On the difficulty of training recurrent neural networks*. [online] Available at: http://proceedings.mlr.press/v28/pascanu13.pdf [Accessed 16 Mar. 2019].

Raffel, C. (2016). Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching. Columbia University.

Saon, G., Soltau, H., Emami, A. and Picheny, M. (2014). *Unfolded Recurrent Neural Networks for Speech Recognition*. [online] Available at: https://www.isca-speech.org/archive/archive_papers/interspeech_2014/i14_0343.pdf [Accessed 5 Jul. 2019].

Schellenberg, E.G., Bigand, E., Poulin-Charronnat, B., Garnier, C. and Stevens, C. (2005). Children's implicit knowledge of harmony in Western music. *Developmental Science*, 8(6), pp.551–566.

Skuli, S. (2017). *How to Generate Music using a LSTM Neural Network in Keras*. [online] Towards Data Science. Available at: https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5 [Accessed 21 Jun. 2019].

Sloboda, J.A. (1991). Music Structure and Emotional Response: Some Empirical Findings. *Psychology of Music*, 19(2), pp.110–120.

Volk, A., Wiering, F. and van Kranenburg, P. (2011). Unfolding the Potential of Computational Musicology. *ICISO 2011*.

Wickland, D.D., Calvert, D.A. and Harley, J. (2018). Evaluating symbolic representations in melodic similarity. *Proceedings of the 5th International Conference on Digital Libraries for Musicology - DLfM '18*.

Yu, L., Wang, S. and Lai, K.K. (2006). An integrated data preparation scheme for neural network data analysis. *IEEE Transactions on Knowledge and Data Engineering*, 18(2), pp.217–230.