Automated Music Composition with Deep Convolutional Generative Adversarial Networks

Sean Hill

for the degree of
Master of Science in Data Science
The University of Bath
September 3, 2021

Supervisor: Dr Marina De Vos

Automated Music Composition with Deep Convolutional Generative Adversarial Networks

Submitted by: Sean Hill

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

This dissertation investigates the functionality of deep convolutional generative adversarial networks (DCGAN) as a means of automated music composition of polyphonic music in the style of Bach's chorale harmonisations. The approach outlined converts the corpus of music into piano roll images, which represent the musical notes graphically with respect to their pitch, duration and timestamp. Points of failure within the models are analysed and explained, while experimentation into the tuning of hyperparameters concludes that the level of convergence in DCGAN is improved, however models suffer from mode collapse later in training. The deep learning problem is unsupervised, and no declarative rules or musical information are encoded into the system. As a result, the models are able to produce music that is much more than a collection of sounds, and music theoretic objects such as chords, melodic lines and lack of dissonance can all be observed in the outputted 8-bar musical samples. The automated composition of Bach's chorales is widely researched, and this dissertation builds on many other works in both deep learning and automated music composition, however to the best of our knowledge, is the first application of the DCGAN infrastructure to a dataset of only Bach's chorales.

Contents

List of Figures

Abbreviations

AMC	Automated Music Composition
ANN	Artificial Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
CNN	Convolutional Neural Network
GAN	Generative Adversarial Network
DCGAN	Deep Convolutional Generative Adversarial Network
Adam	Adaptive Moment Estimation
FID	Fréchet Inception Distance
SD	Standard Deviation

Acknowledgements

I would like to thank my family for their support throughout this project, and my supervisor, Marina De Vos, for valuable guidance and feedback.

Chapter 1

Introduction

Musical composition is the art of creating an original piece of music; the intention of such compositions can be to convey emotions, tell a story, or to provide an audible accompaniment to a visual concept. For many years, the idea that something with such a profound expressionism as music, could be produced computationally with anywhere near the success of the creative prowess of a musician would have seemed impossible.

Artificial intelligence is consistently pushing the boundaries of the automation of traditionally human activities. The World Economic Forum estimates that 70% of jobs currently worked will not exist in 30 years time (WEF, 2020), much of which they accredit to the development of artificial intelligence systems. The idea of automating the composition of music is one that very much suits artificial intelligence models. Models that can encompass the mathematical patterns that underlie music, combined with an understanding of the creative tools that composers equip themselves with, have great success in musical composition applications.

Deep learning is a subset of artificial intelligence which uses artificial neural network models, that take influence from the structure and function of the brain, to learn from data to imitate the way humans learn. Deep learning models are popular for their ability to generate content that is unique, but representative in characteristics to the data it has learned from. These generative models are well suited to the the task of automating musical composition.

A problem with generative music composition models is often the quantification of success, since the quality of music is subjective. Generative adversarial networks (GANs) (Goodfellow et al., 2014) employ an adversarial learning process between two artificial neural networks, in which the output of a generative deep learning network can be classified using a discriminative network as a tool to classify generated outputs as real or fake, to quantify the similarity of outputs to the dataset it is learning from. This combines the generation of samples with a form of success evaluation, which is ideal to combat the subjectivity of music.

A lot of the most exciting GAN applications exist in the realm of image gener-

ation (Radford, Metz and Chintala, 2015; Brock, Donahue and Simonyan, 2018; Zhang et al., 2017). One image generation GAN is Radford, Metz and Chintala's deep convolutional GAN (DCGAN), which uses a convolutional neural network architecture by Springenberg et al. (2014) to generate images typical of a training dataset. Using the piano roll representation of music (Nankipu, 2010) to form a training dataset of Bach chorales, this dissertation investigates the suitability and performance of deep convolutional generative adversarial networks in producing stylistic imitations of JS Bach's chorale harmonisations (Bach and Riemenschneider, 1941). As far as can be seen, the research in this project is the first application of a DCGAN model to a dataset of graphical musical representation made up of only Bach chorales, however many related works exist and are thoroughly reviewed in Chapter 2.

This dissertation aims to apply the DCGAN infrastructure to an automated music composition application, and as such has two key research focuses. Firstly, automated music composition using a DCGAN trained on a graphical representation of music with concentration on stylistic imitation, and secondly, the development and difficulties involved in training GANs, with particular focus on the DCGAN model (Radford, Metz and Chintala, 2015). The overall aim of this research is to produce unique imitations of Bach chorales through a DCGAN. In addition, research into how tuning the hyperparameters of the DCGAN model can affect convergence between networks and overall model success will be extended (Salimans et al., 2016), to include the ability of a network to learn musical features, in a model without any declarative assistance, and is a key research focus. This problem provides an interesting canvas to showcase the capabilities and limitations of GANs.

In the research of related work and different methodologies, and the implementation of the DCGAN model, this model makes the following contributions to the relevant fields:

- Thorough literature review of automated music composition using GANs.
- Analysis into the ability of DCGAN to compose music using piano roll images as training data, and with no human assistance.
- Discussion into the effect of hyperparameter tuning in GANs and the difficulties associated with it.

This dissertation is structured as follows:

Chapter 2: *Literature Review & Technical Background*

Relevant literature regarding automated music composition, deep learning and music theory is explained and compared. Technical elements about deep learning algorithms, data preparation and technological requirements are also listed.

Chapter 3: *Data Processing*

The way in which the dataset of Bach chorales is acquired, validated and converted into a format useable with the DCGAN model is explained.

Chapter 4: *DCGAN Model Implementation*

The overall architecture of the DCGAN model which has been implemented for this project is described. Model design decisions are justified with explanations of why they are taken and examples of various stages in the modelling pipeline are given. A baseline DCGAN model with hyperparameters from Radford, Metz and Chintala (2015) is evaluated and results explained.

Chapter 5: *Experiment Design*

The experimentation procedure is explained, with varying dataset sizes, model architecture, and model parameter values. An overview of the effect of each of the hyperparameters is explained.

Chapter 6: *Experiment Results*

Performance of the DCGAN with three different sets of hyperparameters is shown and analysed.

Chapter 7: *Critical Analysis*

An analysis of the results of the experiment is given, including statistical and qualitative analysis of the musical output, explanations for where the models didn't work, and a summary of the success of the models is provided.

Chapter 8: *Conclusion*

This provides a high-level analysis of the project as a whole, including the overall success of the models and how significant the results are. Several areas in which the research in this dissertation can be extended are explained as further work.

Chapter 2

Literature Review & Technical Background

This chapter provides a survey of relevant literature and theory relating to the generation of Bach chorales using deep learning, with particular focus on the DCGAN model (Radford, Metz and Chintala, 2015). To achieve this, initially a summary of the history of automated music composition is given, with concentration on applications using machine learning techniques. The design of various machine learning models in related fields, and detailed explanations into the fundamentals of deep learning and model architectures is followed. An overview is provided of the music theory that is necessary to be understood for this dissertation, and the ways in which musical features can be extracted in AMC is analysed, by investigating dataset conditions and musical data representations. Finally, a review of various deep learning concepts and algorithms is carried out to provide context for the use of the DCGAN deep learning infrastructure.

2.1 History of Automated Music Composition

Automated Music Composition (AMC) is not a new idea, however the extent to its automation, and the methods used to produce it, have changed over time. In ancient Greek times, Plato and Ptolemy both believed that mathematical laws were the basis of systems of musical intervals (Barker, 1994), and 15^{th} century canonic composition relied on musicians given systematic instructions to play in reference to a core melody. John Cage was known for applying the randomness of systems such as the movement of chess pieces in a game, and his *Atlas Eclipticalis* was composed by laying musical score paper on top of astronomical charts, and playing notes where the stars lay.

Despite these early ideas, it was the invention of the computer that kick started the notion of AMC. Ada Lovelace, widely regarded as the inventor of the first computer algorithm, acknowledged the potential of computational techniques outside of mathematics (Taylor, 1837), noting musical composition as an example (Charman-

Anderson, 2015). This concept is known as 'algorithmic composition', and can be defined as 'a set of rules or a sequence of operations designed to [compose music]' (Simoni, 2003). The first published example of implementing Lovelace's vision of computer-generated music is Hiller and Isaacson's *Illiac Suite* (Hiller Jr and Isaacson, 1957), which saw probability distributions and Markov chains being applied to accept or reject random pitches and rhythms and generate a piece for string quartet. There are countless techniques that have been carried out to achieve algorithmic composition since this early work, such as knowledge-based systems, grammar sets and answer-set solvers, and deep learning.

2.1.1 Types of Automated Composition

In a review of artificial intelligence methods in music generation, Fernandez and Vico (2013) break down the different types of algorithmic composition into the following two categories:

1. **Imitation**: Generating music that imitates a dataset, which can be made up of songs from a particular style or corpus.
2. **Composition**: Automatically composing music, which can range from designing an interface with which a user can use as a tool for producing music, to composing music without human intervention.

Although it should be noted that there is not always a clear distinction between the two, and imitation of particular styles of music are much more dominant throughout the history of AMC (Nierhaus, 2009), and Fernandez and Vico (2013) accredit this to creators of AMC models being largely from computer science backgrounds, thereby missing the "artistic influence" required in compositional AMC. Within these two categories, the nature of the music that is composed varies also, from generating musical melodies, to rhythmic patterns, harmonisation (adding more notes to an existing melody to add polyphonic complexity), orchestration (converting existing single-instrument music to several parts for an ensemble of instruments) amongst others. This dissertation focuses on imitation-based AMC.

2.1.2 Machine Learning-Based AMC

Machine learning (ML) is the name given to algorithms that gain knowledge from data in order to improve their performance on a particular process. In the context of AMC, ML uses a dataset of musical samples, and generates a unique composition based on any inherent characteristics that can be learned from the data. As a result of this, machine learning techniques on their own are restricted to imitation models.

Fernandez and Vico (2013) define the two most commonly used machine learning models for AMC as Markov chains and artificial neural networks. They state how the application of Markov chains to music composition generally either relies on a training dataset of pre-existing music, or derivations from musical theory to form the Markov probability matrices, although the former is much more common.

The survey also indicates how Markov chains can be used both generatively and analytically, by either generating a sequence of states (musical notes), or evaluating the probability of a sequence of states. Examples of this given in the survey have applications in melodic composition and jazz improvisation.

Deep learning refers to machine learning methods that use multilayered artificial neural networks to learn from data. Fernandez and Vico explain how deep learning methods for AMC prominently use a training dataset of example musical patterns to set the neural weights, and generate similar patterns. A thorough review of deep learning methods in AMC is given in Section 2.3, and in general, the problem that a majority of early deep learning AMC models struggle to combat is detecting and establishing the global structure of the music. This was first outlined by Chen and Miikkulainen (2001), who state that the issue is not the ability to add a constraint to the network to measure the overall form, but rather detecting what is a 'good' structure from the dataset alone.

2.1.3 Bach Chorales

Johann Sebastian Bach (1685-1750) was a German baroque composer who is widely regarded as one of the greatest composers of Western music. Chorales are a type of hymn that were traditionally sung in the Lutheran churches of Germany, and Bach's collection of 371 four-part chorale harmonisations are renowned as the superior sources of study for learning the basis of musical harmony and counterpoint (Marshall, 1970). These chorale compositions have subsequently been a very popular choice in AMC models due to their consistency in always comprising four musical parts, regular patterns in harmonic convention, and having homogeneity in characteristics over a large corpus.

Early roots of chorale-based AMC lies mainly in harmonisation models, wherein a model provides the accompanying harmonic music to an input melody. The first example of this is the rule-based model by Ebcioğlu (1988), which was successful, but with the technology at the time, Ebcioğlu considered full algorithmic representation of Bach chorale characteristics as "intractable". Ebcioğlu's rule-based implementation, while functional, is made of a model of over 300 rules written using musical expertise, which implies that while the harmonisation is algorithmic, it is not truly automated. The work of Hild, Feulner and Menzel (1992) is the first known application of deep learning to the chorale harmonisation problem in their three-staged model. The first stage is a neural network that encodes musical information and extracts harmonisation, followed by a rule-based model to generate chords, and a final neural network which adds ornamental musical information to the outputted chords. Exclusively deep learning models have been increasingly common in developing Bach chorales (Liu, Ramakrishnan et al., 2014; Liang et al., 2017; Johnson, 2017), and this is explored further in Section 2.3

2.2 Music Theory

On an elementary level, music is any sequence of sounds that is arranged with the intention of audible interest or expression, usually melodically and harmonically. A piece of music can be broken down into musical notes, which are characterised by their pitch (frequency of the note) and duration (how long the note is played). The following section aims to provide insight into the rules that define music, both generally and in specific styles, and use this to justify a choice of musical data structure to train the DCGAN model on.

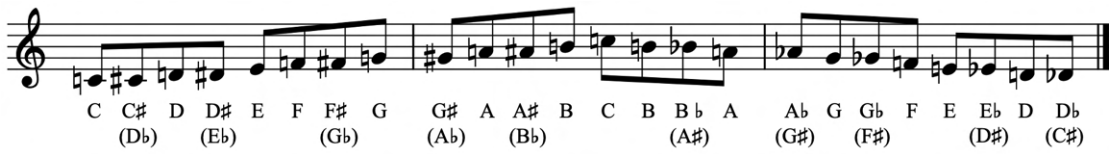


Figure 2.1: Musical score showing all of the notes in an octave of the twelve-tone equal temperament musical system that is commonplace in western music. Note that equivalent notes can sometimes be referred to by multiple names, which is known as enharmonic equivalence.

Bach's chorales are based on the 440 Hz twelve-tone equal temperament musical system, that is the standard and principal tuning system in Western music since the 18^{th} Century, which maps the non-linear relationship between musical notes and their audio frequency (in Hz) to linear spacings (see Figure C.3). The musical notes in this system are shown in Figure 2.1, and are made up of repetitions of octaves, which are sets of the twelve notes shown, with each sub-division between the notes known as semi-tones, and notes that are two semi-tones apart a tone apart. Rhythmically, music is split into bars (also known as measures), which are segments of time that correspond to a number of beats within the music. In terms of the characteristics of the music that will be read and subsequently composed in this dissertation, it will be important to split each bar into a consistent number of sub-divisions in time, a technique known as quantisation.

2.2.1 Elements of Music

Music is widely known to be made up of the following seven "elements" (Schmidt-Jones, 2012):

- **Texture**: How much is going on in the music at any given moment. Monophonic music refers to music with only one line, whilst polyphonic represents many melodic and harmonic musical lines.
- **Timbre**: The nature of the sound that is being produced, for example, the timbre of a violin is what makes it sound like a violin and not a flute.
- **Harmony**: The result of more than one pitch being played at the same time, can be classified generally as dissonant (harmonically unstable) or consonant (harmonically stable).

- **Melody**: A series of musical notes one after the other that form the principal part of harmonic music.
- **Dynamics**: How loud the music is.
- **Rhythm**: A pattern of music in terms of the notes' space in time and duration.
- **Form**: The shape or structure of a sequence of music, on varying scales. The 'global' scale refers to how an entire musical piece is structured, whilst 'local' structure implies a smaller scale, over a subset of notes.

In *Nature, Music and Algorithmic Composition*, Leach and Fitch (1995) explain how these elements should be taken into account in the context of AMC. Leach highlights the importance of the repetition of note sequences, and ensuring that this can be recognised with varying pitch between repetitions. A musical interval is the spatial distance in pitch between two musical notes, and Leach explains how it is important for an algorithmic composition model to distinguish music not by the particular notes of the piece, but the patterns between notes. For example, a 'perfect fifth' refers to the interval between notes that are seven semi-tones apart, so an AMC model will need to be able to recognise that the relationship between 'C' and 'G' is the same as the relationship between 'E' and 'B' (see Figure 2.1). This is known as the transpositional invariance of music, and is something that can be exploited in data pre-processing and model design to emphasise that the relative, as opposed to absolute, positions of musical notes are what characterises music.

2.2.2 Musical Characteristics of Bach Chorales

Bach chorales are four-part harmonisations, meaning that they consist of four lines of music that merge together to form harmonic patterns. These parts are often known as voices, and will be referred to as soprano (S), alto (A), tenor (T) and bass (B) (ordered by descending pitch). Marshall (1970) concludes that Bach most likely wrote his chorale harmonisations by first composing the soprano line, which almost exclusively holds the most melodic qualities, followed this by the bass voice, and then filled out the alto and tenor voices in such as way that the characteristic harmonic patterns are developed.

A phrase is a sequence of chords (chord progression) that can be seen as a coherent unit, and end with a cadence. Cadences are one of the most defining features of chorales, and the presence of perfect, imperfect and interrupted cadences are elementary marking criteria for the quality of an attempted chorale harmonisation task (Oxford, n.d.).

Alongside ensuring cadential order in chorale harmonisations, there are many musical patterns that are atypical in chorales, and musical harmony in general. Pankhurst (2017) establishes the most necessarily avoidable of these as the inclusion of parallel fifths and octaves, using second inversion chords, and not using the same inversion of a chord twice in a row.

2.3 Deep Learning

Deep learning refers to a subset of machine learning in which algorithms are developed as artificial neural networks (ANN), that a computer can 'learn from experience' without knowledge about the semantics of an environment being inputted by a human (Goodfellow et al., 2016). Artificial neural networks are eponymously inspired by the biological neural networks that constitute human brains, but it is mainly just by inspiration that the two are linked. Whilst the brain makes decisions asynchronously, neural networks structure neurons into layers, which perform different transformations on the data, beginning with an input layer, passing the data through several layers of non-linear mappings known as the hidden layers, and an output layer produces the result of the overall network transformation. This section will introduce the fundamentals of deep learning, and explore applications of AMC in several different neural network architectures, to establish how these can be incorporated into a GAN.

2.3.1 Generative and Discriminative Models

One of the ways in which machine learning models can be split is into generative and discriminative models. Discriminative models learn from a dataset such that they can make statistically-driven predictions about an unseen data sample, whilst generative models learn the distribution of the data to generate unique outputs that retain the essence of the dataset. In general, but not exclusively, generative models estimate $P(X, Y)$ (joint probability), where discriminative models capture $P(X|Y)$ (conditional probability), for data X and labels Y.

2.3.2 Fundamentals of Deep Learning

Feedforward Networks

Feedforward networks are the simplest form of an ANN, composed of an input layer, one or more hidden layers, and an output layer sequentially. The layers contain one or more nodes (neurons), which connect to adjacent layers through connections triggered by inputs from neurons activated in the previous layer (Goodfellow et al., 2016). The strength of the connections between neurons is determined by the weight of each neuron. The weights are tweaked in the training stage of the learning process, with the ambition of mapping an input to a desired output. These networks are known as feedforward because there are no cyclic connections, meaning neurons only send information to subsequent layers. An example of a fully-connected feedforward neural network is given in Figure 2.2.

Forward Propagation

Forward propagation is the process in which data is fed through the network in the forward direction, and is processed by each input, hidden and output layer, to generate an output (Luhaniwal, 2019). This calculates and stores all of the intermediate variables from the input to the output layer. Each neuron has its

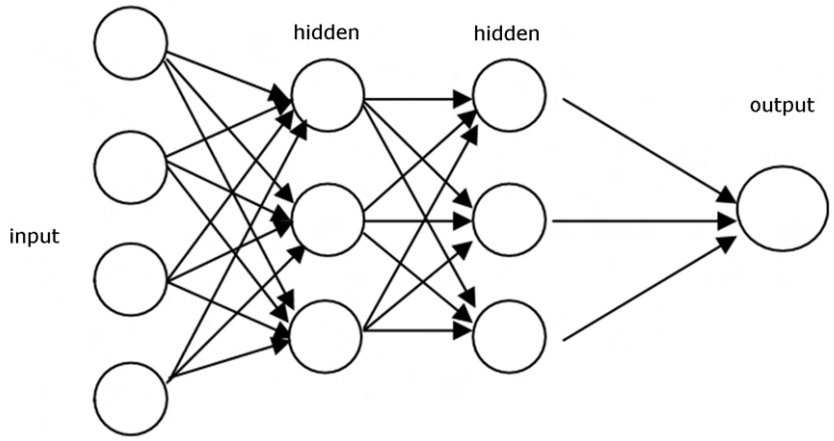


Figure 2.2: An example of a feedforward neural network with fully-connected layers. This is the most basic structure of an ANN, made up of input, hidden and output layers, and with connections only going forward between layers. *Figure obtained from Ciaburro and Venkateswaran (2017).*

own activation function which is used to convert the input data into the weighted connection z to the next layer, and the value of the j^{th} neuron in the l^{th} layer is given in Equation 2.1, for a layer of size m, weighted by w_{jk}^{l}, and an activation function a_k^{l-1}.

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} \tag{2.1}$$

Three commonly used activation functions are sigmoid (ϕ), hyperbolic tangent (tanh) and rectified linear unit (ReLU), and these are defined in Equation 2.2. These are all non-linear, which allows networks to learn more complex problems, and ReLU is widely used since it can help counteract the vanishing gradient problem (see Section 2.3.9).

$$f_{\text{ReLU}}(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \qquad f_{\text{sigmoid}}(x) = \frac{1}{1 + \mathrm{e}^{-x}} \qquad f_{\tanh}(x) = \tanh(x) \tag{2.2}$$

This forward pass through a neural network generates an output, which can be used in the training process to set the neural weights. Generative models can use a forward pass of the neural network with the evaluated neural weights from training to generate a 'fake' output.

Loss Functions

The training of neural networks requires implementation of optimisation algorithms in order to update the network weights to continually improve the output. Achieving

this requires a quantification of the performance of the model, which is measured using a loss function; the difference between the output produced and the label. By minimising this loss function, the difference between the output and the label is decreased, implying higher model performance. For an AMC model, we want to establish the probability that a given musical output is from the training dataset or has been generated by the model, and a common loss function for such a task is binary cross-entropy (BCE) loss, shown in Equation 2.3 to evaluate the loss between two vectors $\mathbf{x}$ and $\mathbf{y}$ (Gomez, 2018), and the mean of this loss vector can be minimised to train the network.

$$J_{\text{BCE}}(\mathbf{x}, \mathbf{y}) \qquad l_n = y_n \cdot \log x_n + (1 - y_n) \cdot \log (1 - x_n) \tag{2.3}$$

Gradient Descent

Gradient descent is a form of gradient-based optimisation that facilitates the minimisation of loss functions (Goodfellow et al., 2016). The gradient (derivative) of the loss function in a network is the rate of change of the loss function for each weight, and gradient descent should update the weights in the direction of the steepest negative gradient. A simple representation of gradient descent to update a set of weights w at time t is given in Equation 2.4. η represents the learning rate and ∇J is the direction of greatest gradient of the loss function.

$$w_t = w_{t-1} - \eta \nabla J(w_{t-1}) \tag{2.4}$$

Three common forms of gradient descent are explained as follows (Goodfellow et al., 2016), with each using differing subsets of the dataset to perform the optimisation:

- **Deterministic**: Evaluating the gradient of the loss over the entire dataset before updating the neural weights. This is deterministic so will find the true local minima, but at a high computational cost.
- **Online**: This only uses a single sample of the dataset to estimate the gradient of the loss function. Has low computation but can lead to high levels of noise which will not find an accurate minimum.
- **Stochastic**: Performs gradient descent on a subset of the dataset to reduce computation, while capturing more information about the loss function than online gradient descent. The batch size is the number of samples that is taken from the dataset in estimation.

Goodfellow et al. (2016) explain how stochastic gradient descent is the most commonly used optimisation algorithm in machine learning problems, and particularly deep learning.

Backpropagation

Whilst gradient descent on its own provides an optimisation tool that is sufficient for simple networks, the hidden layers of an ANN requires the help of backpropagation

to calculate the gradients of the loss function (Rumelhart, Hinton and Williams, 1986). With hidden layers in the model, the effect that each weight has on the loss function depends not only on themselves, but also on the weights in succeeding layers. Backpropagation combines gradient descent with reverse auto-differentiation to evaluate the loss between the network output and the target, and use this to propagate backwards through the network and find the effect that each individual weight has on this loss.

For a layer l in a network, made up of m neurons, the gradient of the loss function (J) between the j^{th} neuron in that layer to the k^{th} neuron in the $(l-1)^{\text{th}}$ layer is evaluated using the chain rule:

$$\frac{\partial J}{\partial w_{jk}^{l}} = \frac{\partial J}{\partial z_{j}^{l}} \frac{\partial z_{j}^{l}}{\partial w_{jk}^{l}} \tag{2.5}$$

By differentiating Equation 2.1, and substituting into 2.5 we get:

$$\frac{\partial z_{j}^{l}}{\partial w_{jk}^{l}} = a_{k}^{l-1} \implies \frac{\partial J}{\partial w_{jk}^{l}} = \frac{\partial J}{\partial z_{j}^{l}} a_{k}^{l-1} \tag{2.6}$$

This is the result of backpropagation, and the resultant value can be applied using gradient descent to update every neural weight in the network as:

$$w_{jk}^{l} = w_{jk}^{l} - \eta \frac{\partial J}{\partial w_{jk}^{l}} \tag{2.7}$$

Adam

Gradient descent is highly reliant on parameters, in particular the learning rate - how much the model weights should be tweaked in response to the loss of an iteration. Kingma and Ba (2014) introduce the adaptive moment estimation (Adam) optimisation algorithm with the aim of reducing computational and memory cost, with intuitive, less sensitive hyperparameters. Instead of relying on a single learning rate (η) for all weights that remains constant throughout learning, each network weight has its own learning rate, which are adapted throughout.

The learning rates for each weight are tuned based on the average first and second moments of the gradients (the gradient mean and squared gradient). The algorithm uses an exponential moving average of these moments, with the parameters β_1 and β_2 controlling their respective decay rates. The aspect of Adam that separates it from preceding algorithms is its bias correction. The first and second moments of the gradients are initialised at zero, so Adam divides each moment by $(1-\beta_i)$ respectively to correct this. Implementations of Adam are available in all deep learning python libraries so an understanding of the parameters and their influence is all that is necessary for this dissertation, but the full pseudo code given by Kingma and Ba is shown in Figure B.1 for further comprehension.

2.3.3 Recurrent Neural Networks

The first application of deep learning to an AMC problem was by Todd (1989), who developed a recurrent neural network (RNN), which implements a sequential 'note-by-note' approach to AMC, in which the note at time step $t + 1$ is generated as a result of the note at t. RNNs are networks that store information taken from the data within the network by backpropagation between layers, and can therefore use this to inform upcoming events. The network is designed with the intention of learning the musical structure of the input, and Todd chooses to represent the time of each note played as relative to its neighbouring notes, rather than in fixed subdivisions of time. In a development of Todd's model to imitate the compositions of Bach, Mozer (1994) explains why the 'note-by-note' approach is unsuitable, stating that the outputted compositions lacked any correct global musical structure. Eck and Schmidhuber (2002) attribute the failure of these models to the problem of vanishing gradients, which is a characteristic failure point of neural networks, meaning RNNs cannot detect long-term dependencies well, since backpropagation means the gradients can tend to zero or infinity. In the context of music, long-term dependencies are the global musical structure and form, which are some of the key elements that differentiate between musical styles (Scaringella, Zoia and Mlynek, 2006).

2.3.4 Long Short-Term Memory Networks

The solution that Eck and Schmidhuber (2002) provide to the vanishing gradient problem is the then-novel long short-term memory (LSTM) algorithm, in which not all of the gradients are altered in backpropagation, with the use of input, output and forget gates. The forget gates decide what information previously learned should be erased from memory, the input gate controls what information learnt in each current iteration is to be stored in memory, and the output gate determines the value of the next state form the memory. The 'long term memory' of the network is stored in the 'cell state'. Eck and Schmidhuber's model successfully deduced both the local structure and the global structure of a corpus of blues music, which is where RNNs had previously been unsuccessful. Their approach involved initially generating a blues melody and chord patterns in isolation, and finally combining the two in parallel. The 12 bar blues lends itself well to deep learning-based AMC, due to its regimented and uniform structure, in a similar way to Bach chorales, which were the subject of another LSTM AMC model developed more recently by Liang et al. (2017). They do not provide their model "BachBot" with any prior information regarding music theory concepts, however under inspection discover that certain neurons in their model align to music-theoretic objects, in particular chords. This is a very interesting finding, and poses the question of whether individual neurons can be altered within an AMC model to change it's musical output as desired.

2.3.5 Convolutional Neural Networks

A convolutional neural network (CNN) is a deep learning algorithm that makes use of a combination of convolutional, fully-connected, pooling and activation layers in its architecture, first introduced by LeCun et al. (1998). Neurons in the convolution layer iterate through sub-sets of the input data and apply a kernel function to each sub-set sequentially. They are usually applied when processing images, due to their ability to capture spatial dependencies within an object. For example, if you were to embed an image (at least 2-dimensional) to a 1-dimensional data structure for use in an RNN, you may lose information regarding what pixels are situated near to each other. The CNN architecture focused on in this dissertation regards the model proposed by Springenberg et al. (2014), wherein it is found that a simple convolutional network made up of just convolution layers and batch normalisation can outperform models such as LeCun's, and an adaptation of this architecture is the basis of DCGAN (Radford, Metz and Chintala, 2015).

A convolutional layer is made up of weights in a two-dimensional kernel K, and any input image $\mathbf{x}$ will undergo a linear transformation with the kernel to generate a output, known as the feature map (Goodfellow et al., 2016). The all-convolutional model proposed by Springenberg et al. replaces pooling layers with convolutional layers with stride of two to aid with dimensionality reduction. The stride of a convolution is how much the convolutional filter moves across the image, so a stride of two moves the filter by two pixels for each new convolution instead of one (as default). The transformation applied to a single-channel image for a single pixel at (i, j) with stride s using a convolutional filter (kernel) size of $(m \times n)$ is shown in Equation 2.8.

$$C(i,j) = \sum_m \sum_n \mathbf{x}(si - m, sj - n)K(m,n) \tag{2.8}$$

Another parameter of convolutional layers is zero-padding, which allows convolutional kernels to preserve the input size of the data, especially with strides greater than one. Convolutions go across the image from left to right until the rightmost pixel in the kernel reaches the end of the image. This means that, without zero-padding, pixels on the border of images are never in the centre of the kernel so are underrepresented. The data is 'padded' by a layer of zeros on the border, and this means that convolutions on the extremities of the image retain the desired dimensions whilst having equal kernel coverage across the image.

On an elementary level, music can be regarded as two-dimensional in pitch and time, and this is the basic structure of the piano roll musical representation seen in many music production interfaces (see Figure C.1). This makes the task of a deep learning model learning musical features from a graphical, rather than sequential, representation of music interesting. Briot, Hadjeres and Pachet (2017) explains how CNN architectures are less common in deep learning music generation models, citing that they are not as effective as recurrent sequential models, however success has been seen in (Liang et al., 2017; Johnson, 2017; Yang, Chou and Yang, 2017),

so the problem is not impossible.

2.3.6 Generative Adversarial Networks

A generative adversarial network (GAN) is a neural network system made up of two distinct networks: a generator and a discriminator; and uses an adversarial learning procedure between the two networks to train generative models (Goodfellow et al., 2014). The two networks are trained in parallel, with the generator generating samples to try and imitate the training data, whilst the discriminator evaluates the authenticity of the output of the generator, to distinguish it from the real data. Ideally, the generator can be trained through the discriminator to generate an output, in this case music, that is indistinguishable from the data it is trained upon, whilst not overfitting the dataset. This is achieved by the generator using the discriminator as a loss function, which is learned in training and forms a minimax game between the two networks.

A random noise vector $\mathbf{z}$ is fed into the generator G, which is then passed through the network to output a sample $G(\mathbf{z})$. The discriminator D is trained on real samples from the training data ($\mathbf{x}$), and fake samples from the generator. $D(\mathbf{x})$ represents the labels that classify the real data, and $D(G(\mathbf{z}))$ is the classification of the fake generator output. A diagram showing the GAN framework is shown in Figure 2.3. When using labels for the fake and real images of 0 and 1 respectively, the success of the discriminator can be described as the success in classifying samples from the dataset $D(\mathbf{x})$ as 1 and the generator output $G(D(\mathbf{z}))$ as 0. The generator's success can be determined from how much it can 'deceive' the discriminator into outputting $D(G(\mathbf{z}))$ as 1.

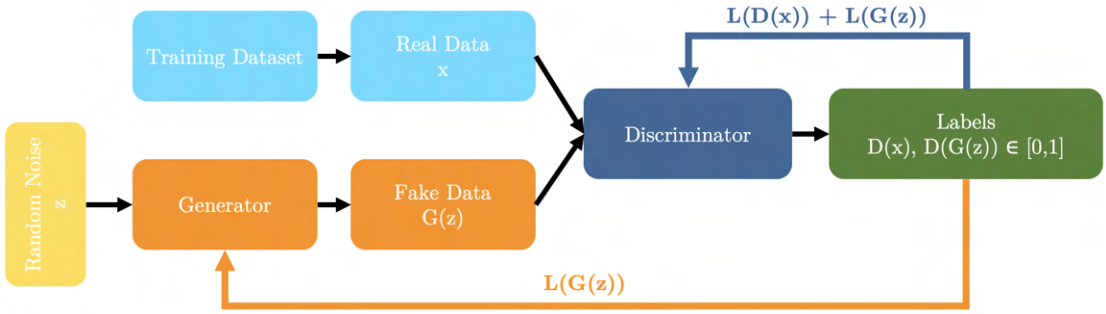


Figure 2.3: Diagram showing the basic structure of a generative adversarial network (GAN) model. This shows the adversarial relationship between the generator and discriminator and how the generator and discriminator losses are used to train the networks through backpropagation.

Minimax Game

Goodfellow et al. (2014) explains how the training process in GANs can be represented as a two player minimax game, with value function $V(D, G)$. The discriminator is trained to maximise the probability of assiging correct labels the real and

fake data, whilst the generator minimises the log probability of the discriminator correctly identifying the generator output $\log(1 - D(G(z)))$. Equation 2.9 explains the mathematical formulation of this game.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \quad (2.9)$$

Convergence is found in the GAN setting when a Nash equilibrium is found between the generator and discriminator. Since both networks are trying to outperform and 'fool' eachother, training GANs to such an equilibrium can be difficult.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 2.4: Pseudo code showing the adversarial minimax game between the generator and discriminator as proposed by Goodfellow et al. (2014). *Figure obtaned from Goodfellow et al. (2014).*

The application of GANs in AMC is relatively new, with an early model called C-RNN-GAN developed by Mogren (2016) using two LSTMs as the generator and discriminator. The C-RNN-GAN model was implemented using continuous musical data for model training, but an example of a GAN which generates music imitating the style of a symbolic dataset is MuseGAN as developed by Dong et al. (2018). The GAN is trained on a dataset of rock music multi-track MIDI files, of which the tracks are analysed both individually and collectively, resulting in a complex GAN, the architecture of which is displayed in Figure 2.5. This network is made up of a multi-track model, that allows the network to understand relationships between different tracks in the MIDI files, and a temporal model to understand musical phrases instead of note-by-note.

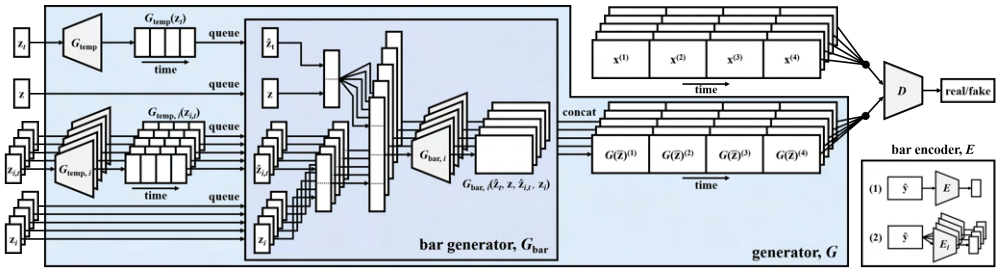


Figure 2.5: System diagram displaying the neural network architecture that makes up the model implemented by Dong et al. for multi-track MIDI file sequential music generation, showing how extensive the generator network can be. *Image obtained from Dong et al. (2018, Figure 5).*

2.3.7 Deep Convolutional Generative Adversarial Networks

The focus of this dissertation lies in deep convolutional generative adversarial networks (DCGAN), which is a GAN in which the discriminator is a CNN, and the generator is a convolutional-transpose neural network. With the use of CNNs, this is a model that is primarily used for image generation, and first introduced by Radford, Metz and Chintala (2015). They found that using traditional CNN architectures from supervised learning problems had little success in a GAN setting. To combat this, three main changes to CNNs are made and explained in detail below.

Strided Convolutions

The networks that make up the generator and discriminator are of the form introduced by Springenberg et al. (2014), wherein they replace pooling layers seen in traditional CNNs with zero-padded and strided convolutions (Equation 2.8).

Eliminating Fully Connected Layers

The DCGAN model does not make use of fully connected layers such as global average pooling, and instead directly connects the highest features of the convolutional layers to the input and output of both GAN networks. This was found to increase convergence speed in the model. Replacing the full-connected hidden layers of the generator with upsampling convolutional layers is a key component of DCGANs.

Batch Normalisation

DCGAN uses batch normalisation in its networks, which stabilises the convergence in the minimax game between generator and discriminator by normalising batch inputs to zero mean and unit variance. Ioffe and Szegedy (2015) introduced this concept, which is said to allow much higher learning rates in training, eliminate the need for dropout, and is said to reduce the probability of mode collapse, which is a common point of failure in GANs where the generator over-optimises on the

discriminator to produce the same output for all inputs (see Section 2.3.9). Radford, Metz and Chintala (2015) apply batch normalisation to all layers apart from the output of the generator and the input to the discriminator in DCGAN.

Activation Functions

The generator network uses ReLU activation in input and hidden layers, and tanh for the output. The tanh activation requires images to be normalised between -1 and 1. In the discriminator, leaky ReLU activation functions are found to work best. Leaky ReLU is used when normal ReLU functions are found to 'die', which is when the activation output gets stuck in the negative domain, where $f_{ReLU}(x) = 0$, and the introduction of a slight gradient of parameter α in leaky ReLU aims to counteract this. To produce the scalar discriminator output, the final convolutional layer flattens the data and is passed through a sigmoid activation.

$$f_{LeakyReLU}(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.10}$$

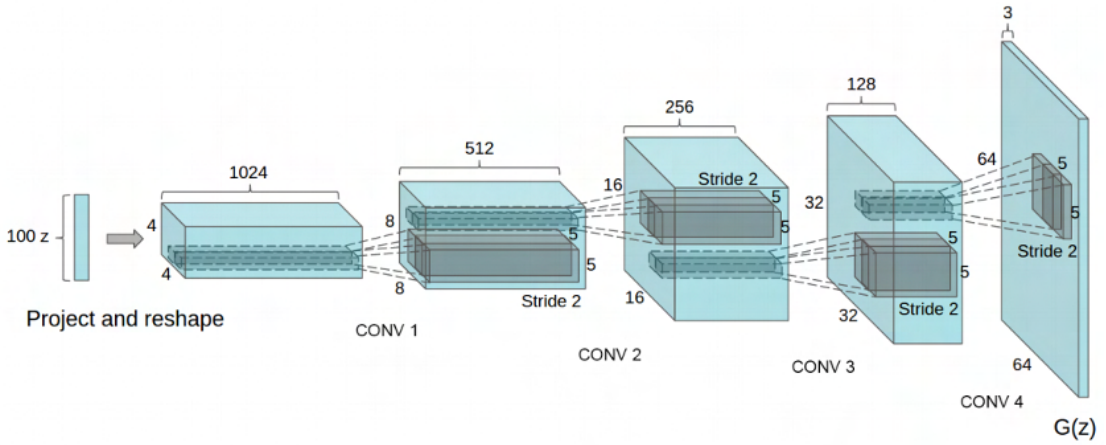


Figure 2.6: Architecture of the generator network used in Radford, Metz and Chintala's DCGAN implementation. This shows how only convolutional layers are used, with strided convolutions scaling up the data. *Figure obtained from Radford, Metz and Chintala (2015).*

A model by Liu, Schneider and Yuan (n.d.) implements a DCGAN model in a comparison of GAN methods of generating music, but uses a spectrogram representation of the MIDI files as opposed to piano roll. Liu cites Donahue, McAuley and Puckette (2018) as a good source of initial DCGAN parameter values, and well documents the issues with exploding gradients and the discriminator constantly outperforming the generator were encountered. It is concluded that the DCGAN infrastructure did not work well with their dataset, with repeated modal collapse.

Yang, Chou and Yang (2017) use a modified DCGAN architecture in their symbolic AMC model to generate pop music, without eliminating all of the fully-connected layers in the generator architecture. This work is the state-of-the-art in regards to DCGAN AMC models using symbolic musical data. The dataset used in their work is of a similar size to the Bach chorale corpus, and this size is said to be restricting, so inspiration in the data augmentation techniques used can be taken. The dataset is transposed to all 12 major and minor keys, which bolsters the training sample over 10-fold. Since the reasons explained for transposition in Section 2.5.1 are to reduce variability in the data, it will be interesting to see whether this data augmentation will counteract that, or improve model performance. A difference between this dissertation's work and that of Yang, Chou and Yang is that they are generating chords, which reduces the need for harmonic patterns between individual notes to be learned.

As far as can be ascertained, this dissertation provides the first model that aims to compose Bach chorales using an image-based piano roll dataset to train a DCGAN with two-dimensional convolutional layers. A blog post in the same application with limited detail is the only example found on the internet (Galluzzo, 2020). Given chorales' popularity in AMC literature, and success of various other DCGAN models, it will be interesting to see if musical rules and stylistic characteristics can be learned in this way.

2.3.8 Quantifying Success in GANs

Judging the performance of a GAN is not always as transparent as other machine learning algorithms. For a traditional classification model with a training and test dataset, the accuracy in performance on hidden test data is sufficient to explain how well the model is achieving it's goals. Salimans et al. (2016) provide an analysis on ways to improve GANs, and explain how they do not have an objective loss function to assess the progress of the model. GANs are notoriously difficult to train, and oftentimes a Nash equilibrium cannot be found between the generator and discriminator. In the original GAN paper, Goodfellow et al. (2014) fits a Gaussian Parzen window (a type of kernel density estimation) to form a probability distribution around the output of the generator, and use the log-likelihood under this distribution as an evaluation technique, however they note that the high variance in this metric does not fare well in high-dimensional problems, so open the door to further evaluation measures.

Borji (2019) compares various evaluation measures of GANs, and even though different candidates are explained, it is concluded that there is no clear metric that unresolvedly "best captures the strengths and limitations of models". For many problems, a simple way to assess an image-generating convolutional GAN's performance is by visual inspection of the results. In early stages of training, this is one of the easiest ways to ensure the model performance is somewhat successful, however this is the extent of its adequacy as a means of performance analysis, to avoid bias. For the music generation model in this dissertation, visual analysis is very limiting, but aural judgement of the images once converted to audio can be

used. Once a model is producing convincing outputs, Borji suggests the use of a nearest neighbours model to check for overfitting on certain samples from the dataset. This is a technique using in the initial DCGAN implementation (Radford, Metz and Chintala, 2015).

In term of quantitative measures, Borji suggests inception score and Fréchet inception distance as the most robust ways of evaluating the quality of images generated. Inception score is a metric introduced by Salimans et al. (2016), which aims to imitate human judgement of images produced by a GAN. Generated images are fed into *inceptionv3* (Szegedy et al., 2015), a pre-trained image classification model, and the score is defined as the label entropy produced by images, with a low score implying a more realistic image. The problem with inception score is that it relies on the generated images being similar to that which the *inceptionv3* model is trained on. Heusel et al. (2017) propose a development on inception score with Fréchet Inception Distance (FID), which uses the inception model on both the 'fake' generated outputs, and the 'real' images from the training data. Rather than evaluating the entropy of the generated images, FID captures the activations from the final pooling layer of the model, instead of the output, in an attempt to grasp "computer-vision-specific-features" of the image. The real and fake images can be approximated by multivariate Gaussian distributions, and the Fréchet distance (Har-Peled et al., 2002) between the distributions is the FID. The formulation of the FID between distributions X_R and X_F, formed for the real and fake data respectively, is shown in Equation 2.11.

$$\begin{gathered} X_R \sim \mathcal{N}(\mu_R, \Sigma_R) \qquad X_F \sim \mathcal{N}(\mu_F, \Sigma_F) \\ FID\,(X_R, X_F) = \|\mu_R - \mu_F\|^2 + Tr\left(\Sigma_R + \Sigma_F - 2\sqrt{\Sigma_R \Sigma_F}\right) \end{gathered} \tag{2.11}$$

Salimans et al. (2016) introduce inception score as a way to improve GAN performance, and it classifies the generated images using a classification model trained on the training data. The Fréchet inception distance was introduced in a successful GAN model by Heusel et al. (2017), which they say captures image similarity better than just inception score for supervised problems. The musical data in this dissertation will not be explicitly labelled which makes this somewhat limited.

In the context of the Bach chorale problem, known musical features can be quantified from both the training dataset and the 'fake' generated images. A comparison between the statistics of both can quantify how well the GAN is actually learning musical features. For example, avoidance of parallel fifths/octaves, and generation of characteristic cadential harmony.

2.3.9 How to Improve GANs

Even though GANs are powerful tools for generating data, the process of having two networks simultaneously competing to outperform each other is fundamentally troublesome. Salimans et al. (2016) highlight common difficulties in training GANs and ways to overcome them in a development on the original GAN paper

(Goodfellow et al., 2014). It is explained how finding a Nash equilibrium in the minimax game between the discriminator and generator is difficult, with "non-convex cost functions, continuous parameters and high-dimensional parameter space". This section investigates common problems that have been documented in literature, and ways in which they can be remedied in the training process, using the three challenges explained by Ayari (2020).

Mode Collapse

It is common for GANs fall into a scenario where the generator repeatedly outputs very similar or identical outputs for different inputs to the generator, this is known as mode collapse. Complete mode collapse is often simply a case of the training dataset not being vast enough for the generator output to be truly generalised, while partial mode collapse can be a complex problem that requires careful analysis and choice of loss functions. Modifications to the learning rate of the Adam optimisers for each network can help the generator and discriminator learn at a rate at which they are less prone to mode collapse.

Failure to Converge

Convergence of two competing gradient-based optimised networks is difficult, and with GANs still in their infancy there is no one-size-fits-all technique which solves this. In the case of Heusel et al. (2017), having distinct learning rates for the generator and discriminator, known as a two time-scale update rule, with the generator learning at a faster rate resulted in convergence. Label smoothing can also help with convergence, where using labels such as 0 and 0.9 as opposed to 0 and 1 for fake and real labels in the discriminator can reduce extrapolation to extreme values. Usually, label smoothing is only applied to the 'true' label, so as to not to encourage the generator to reproduce erroneous samples that it has previously generated (Goodfellow, 2016). Equation 2.12 explains the beneifit of label smoothing, for real and fake labels α and β respectively (Salimans et al., 2016). 'Smoothing' the labels encourages p_{model} to move nearer the data in areas where p_{data} is near to zero and p_{model} is large. This encourages adversarial learning between the networks and helps convergence.

$$D(\mathbf{x}) = \frac{\alpha p_{data}(\mathbf{x}) + \beta p_{model}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_{model}(\mathbf{x})} \tag{2.12}$$

Vanishing Gradients

A reason for non-convergence is often down to the discriminator being 'too good', with $D(x) \approx 1$ and $D(G(z)) \approx 0$, and therefore not giving enough for the generator to learn from and improve. Ayari (2020) explains how backpropagation diminishes the gradient as it propagates through the network in this case, providing no feedback to the generator, and this is known as the vanishing gradient problem. In the original GAN paper, vanishing gradients provide the rationale behind maximising

$\log D(G(z))$ instead of minimising $\log(1 - D(G(z)))$, to provide greater gradients when the generator is weak (Goodfellow et al., 2014). The DCGAN implementation (Radford, Metz and Chintala, 2015) uses ReLU, tanh and sigmoid activation functions to limit inputs to network layers between -1 and 1 to avoid vanishing gradients.

2.4 Musical Data Representations

A key aspect of designing a machine learning model is based around the availability and structure of the data. The data found in AMC literature can be split into three categories: symbolic, spectrographic, and raw audio data. Marolt, Kavcic and Privosnik (2002) used spectrograms to train a multilayer perceptron neural network to detect the onset between musical notes to distinguish them for musical transcription (producing written musical scores from an audio recording). Dieleman and Schrauwen (2014) produced a convolutional neural network that learned to retrieve information from musical waveform recordings of music. For this project, however, symbolic musical data will be used to train the model, since all necessary musical information is contained in symbolic musical data for Bach chorales.

Symbolic musical representations explain the pitch, duration and dynamics of the music, but may use differing methods to quantify those elements. Many symbolic musical representations exist, including MIDI Files (Back, 1999), MuseData stage2 Files (Hewlett, 1997) and MusicXML Files (MusicXML, 2017).

For the scope and methodology proposed in this project, MIDI files will be used. This is due to the vast availability of Bach chorale datasets online, simplicity of representation, and a consideration of frameworks already in place for parsing, extracting features, and generating audio files from each file type. As harmonic compositions, the key information that will need to be extracted from the Bach chorales in pre-processing is the relationship between successive and simultaneous pitches. This means that there will be no need to obtain information about the dynamics, timbre or articulation in the music, as explained in Section 2.2.1. For this reason, MIDI files will be sufficient sources of data.

2.5 Data Preparation

When reading musical data from MIDI files as mentioned in Section 2.4, it will be crucial to the success of the model to make sure that the key musical information of the input files can be extracted, and then encoded as the input of a neural network correctly.

2.5.1 Transposition

In developing a recurrent neural network for AMC, Johnson transposed all of the pieces in the training dataset into C major or C minor, and this is a good way of reducing the complexity of the model (Johnson, 2017). The network will only have

to learn the musical rules for that key domain, rather than generalising over all keys. While Johnson chose to transpose the music in his dataset to the parallel keys[1] of C major and C minor, it will be interesting to see whether more information is retained and the transposition invariance between notes can be emphasised with the relative keys[2] of C major and A minor instead, which both have the same key signature. Another way in which transposition can be used to bolster model performance is through data augmentation. By transposing the data to all 12 keys, model performance can be improved, since limited datasets have been known to cause modal collapse in GANs (Yang, Chou and Yang, 2017).

2.5.2 Quantisation

In music, quantisation refers to defining a precision in the time domain to which all musical notes are considered to. It will be important that music is split into equal sub-divisions in time for use in a DCGAN model since this constrains the times at which musical notes can be generated. For Bach's chorales, the choice is whether to quantise the music to quavers (1/8 notes) or semi-quavers (1/16 notes), and both have been seen in previous works. Liang et al. (2017) criticises the decision of works such as Boulanger-Lewandowski, Bengio and Vincent (2012) and Eck and Schmidhuber (2002) in quantising the chorales to 1/8 notes, claiming a "non-negligible level of distortion". Given that over 97% of the notes in Bach's chorales are of at least quaver length, and that instances of semi-quavers and beyond are almost exclusively ornamental (Marshall, 1970), quantisation to the quaver level will not detract from any harmonic patterns in the music and is therefore a suitable pre-processing step.

Johnson also explains the importance of distinguishing between a note being played twice on the beat, and a note being held for two beats, since this is something that can be easily overlooked when working with MIDI files. Dong et al. (2018) establish the likely range of note pitches necessary for their model, and restrict any notes outside of this range from being read into their GAN. This limits the dimensions of their input/outputs to the network, and therefore avoids unnecessarily sparse arrays in their model, improving computation.

2.5.3 Graphical Musical Representation

The inspiration in a deep learning AMC model that uses a graphical representation of music originates from the piano roll music storage medium. Originally, piano rolls refer to paper scrolls that are fed into a self playing piano developed in the late 19th Century (Nankipu, 2010). Music is encoded axially in two dimensions corresponding to the pitch and time of the notes, and this is often seen as the antecedent to MIDI files. Figure C.1 can be used for further understanding of this format, and displays how the pitch and time scales have to be both limited and quantised. It is important to limit the pitch sufficiently to reduce the sparsity of

[1]Parallel keys have the same tonic note (C for C Major/Minor).

[2]Relative keys have the same key signature (C Major and A Minor are relative major/minor).

arrays, whilst encompassing all typical notes seen in the chorales, and this is a design choice that will be made as a result of dataset exploration.

2.6 Technological Requirements

This section explains the technological requirements necessary for this project, and the software and hardware chosen to facilitate them. All of the code developed in this project was programmed in Python 3.7, chosen due to its collection of suitable deep learning, data processing and musical libraries.

The `music21` Python library (Cuthbert and Ariza, 2010) is a tool designed for analysis of symbolic musical data, where symbolic musical data such as MIDI files can be converted into a 'stream', which are containers that represent instruments or parts within the music (see Figure C.2). These musical streams will be converted to a graphical image-based representation using `imageio`. Outputted images from the models can then be converted back into MIDI files using `music21`, and further into audio files using the `midi2audio` library over a sound font from *FluidSynth* (Zámečník, 2016).

Python has various powerful open source libraries for deep learning. The work in this project is carried out using PyTorch (Paszke et al., 2019), due to its functionality for accelerated processing using graphical processing units (GPUs) through the CUDA interface, and compatibility with image-based data using PyTorch's `torchvision` package.

Deep learning is computationally expensive, and whilst all of the pre-processing and analysis scripts written will comfortably run on a locally available CPU, all deep learning algorithms in this dissertation are ran through a GPU to speed up operations. A GPU is a specially designed cimputational processor, originally designed for use in video games, which dedicates part of its memory to performing floating point operations. Deep learning requires vast matrix multiplications in training, so GPUs greatly speeds up this process (Dsouza, 2020). *Google Colab* (Bisong, 2019) is used in the design of models, debugging and testing, before full code runs are executed through SSH (secure shell) connection to access an Nvidia GeForce RTX 2080 GPU made available through the University of Bath.

2.7 Research Proposal

The remainder of this dissertation will be dedicated to the implementation of a DCGAN model with the ambition of producing stylistic imitations of Bach chorales using an image-based representation of the music, in the piano roll format. DCGAN is a deep learning model which uses adversarial learning between two convolutional neural networks for the generation of images (Radford, Metz and Chintala, 2015). The model uses a dataset of images as training data, so it will be important to the success of the music outputted that this conversion from symbolic musical data to images is valid. Initially, a suitable dataset needs to be found, that

both encompasses the entire Bach chorale corpus, and provides sufficient musical information to learn from. A dataset of MIDI files will be acquired to satisfy this criterion. The DCGAN infrastructure and training process will be implemented using *PyTorch*, using the model proposed by Radford, Metz and Chintala (2015) as a baseline.

This dissertation will investigate the effect of modifying hyperparameters on the performance of models in this music composition task, by making comparisons with the baseline model. Research in Section 2.3.9 has shown that training GANs can be difficult, so the experimentation will focus on quantifying model success, diagnosing downfalls, and overcoming problems by tweaking of hyperparameters. Model performance will be quantified using the binary-cross entropy loss for each network to quantify the adversarial relationship between the networks, and the Fréchet inception distance to evaluate the quality of images produced by the DCGAN. These values, along with qualitative and quantitative analysis of the music generated, will be used to determine the success of the models.

Chapter 3

Data Processing

This chapter describes the acquisition of a dataset of Bach chorales, and the process in producing graphical representations of the music for use in deep learning algorithms. Firstly, an explanation and justification into the chosen dataset is given. This is followed by design choices and implementation details about how the data is pre-processed and converted into graphical representations, with reference to research in Chapter 2.

Proper dataset analysis and preparation is an essential aspect of a machine learning project, and this dissertation requires images to be formed that encompass all the features and characteristics of Bach chorales, so that the DCGAN model designed in Chapter 4 has the best chance of composing music that has musical qualities, and imitates the corpus' style.

3.1 Data Acquisition

The exact number of Bach chorales is not precisely defined, with 419 individual BWV numbers assigned to chorales, however once accounting for duplicates and revisions, this number can be reduced to 371 (Bach, 1832). In line with the previous research in the automated music composition of these settings, a dataset devised by Margaret Greentree containing MIDI and MusicXML files of 371 Bach chorales is commonly used, and has seen success in other AMC models (Johnson, 2017; Lichtenwalter, 2009). A revised archive of this dataset produced by Daniel Bump (Bump, 2014) has been acquired that corrects errors in these chorales based on the Riemenschneider chorale publication (Bach and Riemenschneider, 1941). These are given in the transcription software *MuseScore* format (Watson, 2018), and subsequently exported as MIDI files using the 'Batch Convert' MuseScore plugin [1].

[1] `https://musescore.org/en/project/batch-convert`

3.2 Data Analysis

The MIDI files obtained were parsed using the `music21` library into streams (see Figure C.2). These streams facilitate quantitative analysis and application of musical transformations such as transposition and quantisation. In total, 364 out of 371 of the chorales in all 12 transposed keys have been read, accounting for invalid data, and data regarding the pitch and time stamps of all notes in the music are stored in .json files for use in data preparation. Initial analysis of the dataset indicates that the range of pitches is the 54 notes between G1 and C6 for the entire dataset. The chorales range from 6 to 48.25 bars in length, with a mean duration of 13.5 bars.

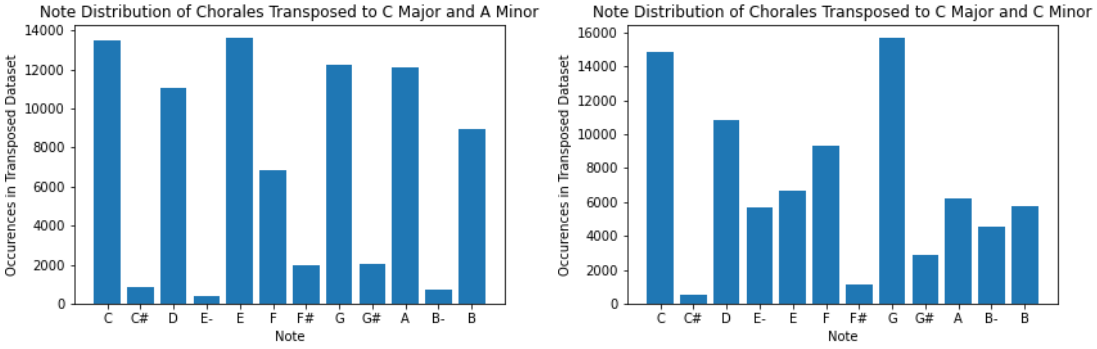
Transposition

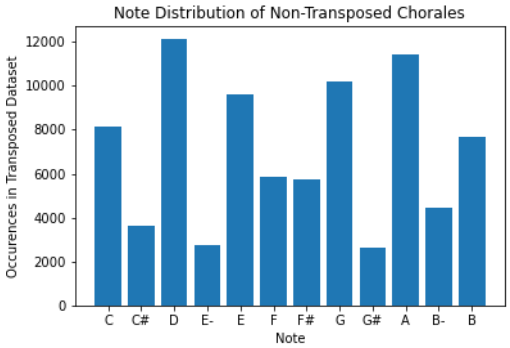
As per Section 2.5, for some of the models to be experimented with in Chapter 5, all of the chorales will be transposed to the same key to emphasise the transpositional invariance of music, and help the network to learn these harmonic relationships generally rather than separately for each note. Since Bach's chorales exist in both the major and minor keys, so music has been transposed to both the parallel and relative minor to C Major, code shown in Figure A.1. Figure 3.1 shows the distribution of all notes within the dataset for the untransposed music (3.1c), conversion to the parallel keys of C Major and C Minor (3.1b) and to the relative keys of C Major and A Minor (3.1a). Transposition to C Major and C Minor has been chosen for this dissertation, since this places much greater emphasis on the tonic[2] and dominant[3], which are the most harmonically significant notes in Bach chorales due to frequent modulations between the tonic and dominant keys (Oxford, n.d.).

It is clear that transposition limits the spread of notes within the dataset, and this will hypothetically reduce the output of atypical notes for different key signatures, since the model will only have to learn the characteristics of major and minor music, instead of the 24 individual key settings. One potentially restricting factor on the dataset used in this dissertation is it's limited size of only 364 samples. In an attempt to combat this restriction, the musical data is also transposed to all 12 major and minor keys, to investigate whether data augmentation in this regard has similar success to what is seen in Yang, Chou and Yang (2017). This produces a dataset of 4,368 chorales which is a substantial level of augmentation.

For the remainder of this dissertation, the two datasets to be produced and subsequently used in modelling are the original dataset transposed to C Major and C Minor, and an augmented dataset which is transposed to all 12 major/minor keys. These will be referred to as the 'C Major/Minor' and 'augmented' datasets respectively.

[2]The tonic of a musical key is the first note of it's scale (C for C Major, A for A Minor).
[3]The dominant of a key is the fifth note in the scale (G for C Major, E for A Minor).





(c)

Figure 3.1: Figures explaining the decision made to transpose all major MIDI files to the key of C Major and all minor chorales to C Minor. This tranposition, shown in 3.1b, has the leats variation out of the two options, with the large majority of notes being either C or G; the tonic and dominant of C Major and C Minor.

3.3 Converting to Images

The MIDI files are converted to images in the piano roll format (Figure C.1), and the dimensions and resolution of these images has been considered with regard to the dataset and what is deemed requisite.

The y-axis of the images generated has been limited to the 64 notes between D1 and F#6, since this is equidistant from the minimum and maximum pitches seen in the dataset, to give dimensionality in a power of two to assist with regularised quantisation of the time domain. If the range of pitches in the graphical representation was to significantly exceed the typical range of pitches, this would lead to unnecessary errors in early training stages, as well as sparse arrays implicating higher computation.

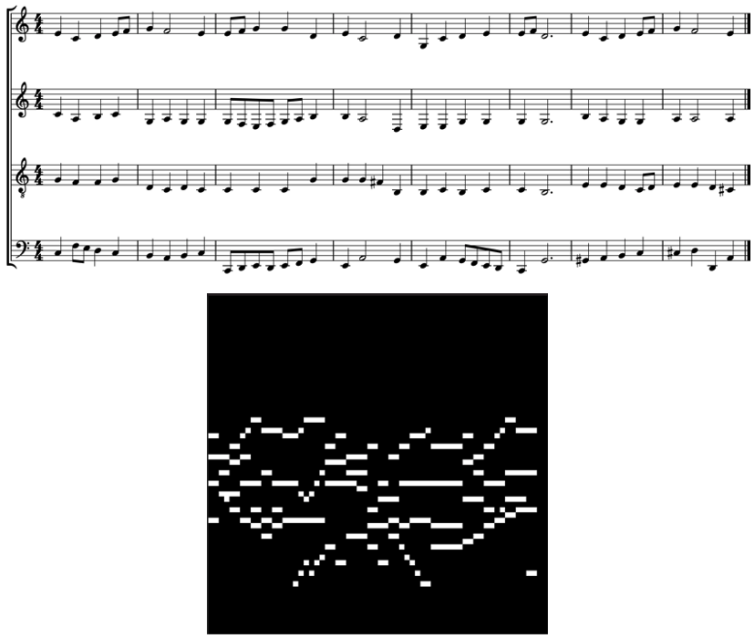


Figure 3.2: Example of how 8 bars of the chorale Nun ruhen alle Wälder (BWV 13/6) are represented as a binary grayscale image representing the piano roll of the music.

Quantisation procedures have been performed in the conversion of musical data to images, and is controlled by the resolution of the image generated. Figure A.2 shows the full implementation of the image generation code, which is a modified version of an implementation by Gatti (2021). The images represnenting the Bach chorales have been generated as 64x64 binarised grayscale images that constitute 8 bar phrases of chorales. An example of this is shown in Figure 3.2.

3.4 Image Pre-Processing

The deep learning models used in this dissertaion are all coded in PyTorch, and the associate `torchvision` library can be used to faciliate the importing, pre-processing, and composition of the image dataset. The data loader function `torchvision.datasets.ImageFolder` is used to import the data from file, and several transformation functions from `torchvision.transforms` applied, with code displayed in Figure A.3.

- `Grayscale`: Converts images to grayscale: two dimensional images with pixel ranges from 0 (black) to 255 (white).
- `ToTensor`: Converts images from the PIL (Python Imaging Library) format to PyTorch tensor data structures. In PyTorch, tensors are the data sturcture used to represent matrices or arrays. They are differentiated from other computational arrays, such as those seen in *NumPy*, due to their compatibility with GPUs.

- `Normalize`: Normalises the images from a range of $[0, 255]$ to $[-1, 1]$, which is required for use with DCGAN.

Finally, `torch.utils.data.DataLoader` is used as to iterate through the generated dataset and collate individual data samples into batches for use with PyTorch models. The size of these batches is a hyperparameter of the models used in this dissertation, and will need to be tuned accordingly. Radford, Metz and Chintala (2015) use a batch size of 128 in their model, but given the discrepancies in size of dataset this is unlikely to be suitable for the piano roll images.

3.5 Converting Images to Music

For an image-based music generation task, visual inspection of images throughout the training process to aid model design is not sufficient to monitor performance, and the generator outputs will need to be analysed aurally. The images are converted into music files using `midi2audio` Zámečník (2016), which is a Python library to be used as an API to the FluidSynth sound synthesiser. The images are first converted into MIDI files using `music21`, by scanning across the images left to right to obtain details of all notes produced by the DCGAN model. All of the images are converted as 80 beats per minute, according to Sherman (2000). The sound files can be produced from the MIDI files using `midi2audio`, and using a soundfont[4] downloaded from MuseScore Watson (2018), and converted into Waveform Audio Format (.wav) files. This process has been validated on the training samples, both visually though the MIDI files, and through the ensuring audio files are identical. Full implementation of this process is shown in Figure A.10.

3.6 MNIST

In machine learning, a toy dataset is used to validate that a given algorithm is functioning the way that you intend it to. In the context of this dissertation, a dataset is required that is similar in characteristics to the images produced in Chapter 3, and has garnered success with DCGANs before, so that any model implementations can be validated on this dataset. The MNIST dataset (LeCun and Cortes, 2010), is a set of handwritten digits stored as 64x64 grayscale images. These images are very similar in features to the piano roll datasets produced, so are perfect to be used as comparison within our models. The images produced from the Bach chorales are not a standardised dataset, which gives no guarantee of good performance in a generative deep learning model. MNIST is known as a standard in these problems, and is one of the datasets used in Radford, Metz and Chintala (2015), making this comparison essential to assess model performance. Samples of the dataset are given in Figure 3.3.

[4]Soundfont (.sf2) is a file type that contains samples of musical instruments and other musical information, and are used to generate sounds from MIDI files.

Figure 3.3: A sample 8x8 grid showing a selection of the MNIST handwritten digit images used in training as a toy dataset for the baseline model. (LeCun and Cortes, 2010)

Chapter 4

DCGAN Model Implementation

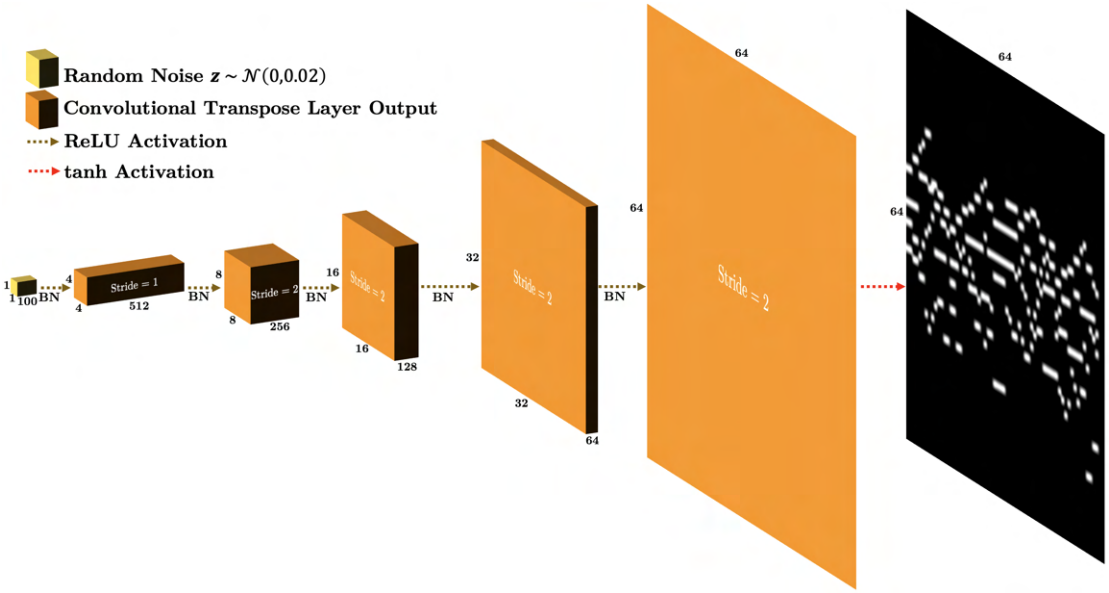
This chapter explains the implementation details and model architecture used in developing a DCGAN-based model to generate imitations of the Bach chorale piano roll images produced in Chapter 3. This will be achieved by using the chorale dataset to train the discriminator network in how to classify the images as real or fake, using its loss to train the generator network produce increasingly convincing images, to compose Bach-like compositions. Firstly, the model architecture will be established, followed by the implementation of the training process, in which the minimax game between the generator and discriminator is formed. Decisions for set model parameters will be justified with reference to literature and experience from models. A baseline model, hyperparameters from the original DCGAN paper Radford, Metz and Chintala (2015), is used, and results of the baseline on MNIST and the training data produced in Chapter 3 are given.

4.1 Network Architectures

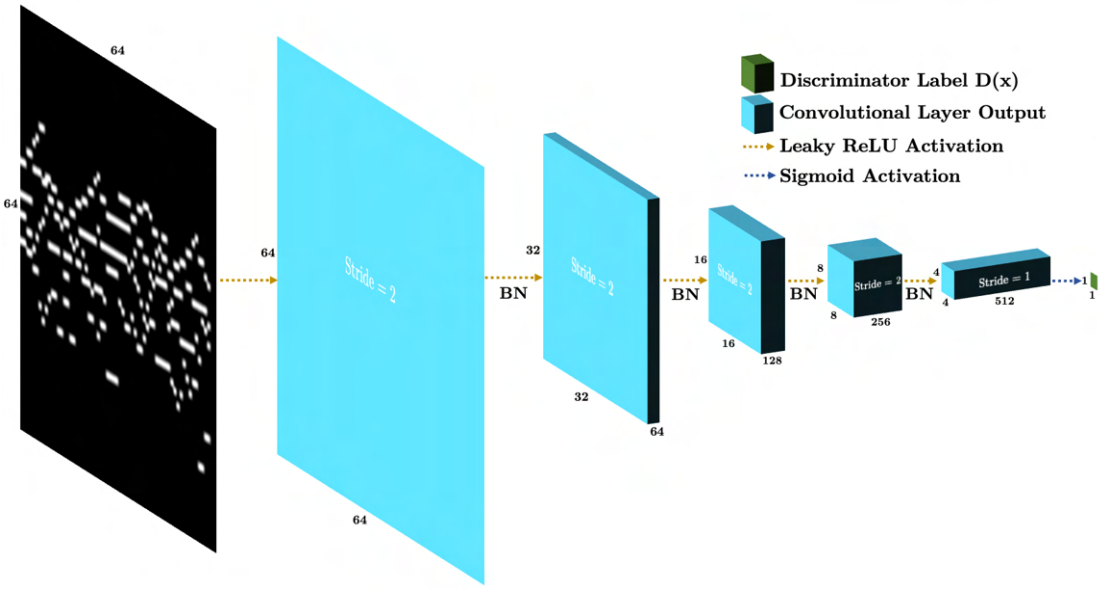
The model used in this dissertation is an adaptation of the DCGAN architecture used in Radford, Metz and Chintala (2015). The key differences between the datasets used by Radford and the piano roll images are as follows:

- **Size of Dataset:** Radford uses a training corpus of over 3 million training examples, but the Bach chorale dataset has only 364 images, and 4368 images in the augmented dataset.
- **Type of Images:** Instead of 32x32x3 pixel RGB images, the piano roll images shown in Figure 3.2 are 64x64x1 pixel grayscale.

These differences are accounted for by varying the input and output of the discriminator and generator networks respectively, and using one less convolutional layer than what is shown in Figure 2.6. Both networks are all-convolutional CNNs as introduced by Springenberg et al. (2014), and network diagrams are shown in Figures 4.1a and 4.1b for the generator and discriminator network respectively.



(a) Generator network, made up of strided convolutional transpose layers, which aid in upscaling the dimensionality of the random noise input to the outputted image. The convolutions are followed by ReLU activation functions, and the output layer performs a tanh activation. One notable layer in this network is the initial convolution, which expands the 100x1 random noise vector considerably, into a 4x4x512 output.



(b) Discriminator network, made up of strided convolutional layers, batch normalisation, leaky ReLU activations subsequent to convolutions, and a sigmoid activation in the output layer.

Figure 4.1: The architectures of the networks used in the music generation models. They modified versions of Radford, Metz and Chintala (2015)'s, using all-convolutional CNNs that eliminate pooling and fully-connected layers from Springenberg et al. (2014). In order to scale up and down in dimensionality without pooling layers, strided convolutions are used, and all layers with a stride value of two also include a single layer of zero padding. 'BN' in the diagrams indicates the points at which batch normalisation of the convolutional outputs occurs, and the colour of arrows corresponds to the activation function that is applied to the output data.

4.1.1 Network Parameters

Kernel Size

The kernel size determines the size of the convolutional windows that are used to extract features from the images. In the DCGAN paper, a 4x4 kernel is used, however one of the paper's authors, Chintala, claims this is purely to match their code implementation[1]. Despite this claim, subsequent work by Odena, Dumoulin and Olah (2016) suggests always using a kernel size which is divisible by the stride in deconvolutional layers, to avoid 'checkerboard' patterns in generator image output. For this reason, kernel sizes of 4x4 and 8x8 have been investigated, but for a 64x64 images, an 8x8 kernel proves too large a convolution, so 4x4 kernels are used.

Number of Feature Maps

The number of feature mappings essentially corresponds to the number of convolutions that occur in each layer, and determines how the data is scaled up and down in the networks. The models implemented use 64 feature maps.

Leaky ReLU Gradient

In the leaky ReLU activation for the discriminator, 0.2 is the default value of α (Equation 2.10), and used in the most DCGAN applications in literature. Small changes to this parameter had little difference to model stability, and Gustafsson and Linberg (2021) found this hyperparameter to be the least significant in their investigation into GAN hyperparameter tuning, so $\alpha = 0.2$ is retained throughout.

4.2 Implementation Details

The following section describes the steps taken to implement a DCGAN model in Python with the PyTorch library. This includes design of the neural networks and programming the training loop which forms the GAN. All code produced for this section of the dissertation uses modular programming; details of the repository structure can be seen via the *GitHub* repository[2]. The modularity of the code can be seen in the `main` function, which compiles all of the code at once, shown in Figure 4.2. All of the early stages of development were carried out on Google colab, which provides access to GPUs to run code on, however the runtime is capped to approximately 2 hours, which is not sufficient to train the models, so the code is made in such a way as to be able to run through the command line with modular .py files, and easily copied into and run through a Python notebook when needed. For reproducibility and sake of comparison, a random seed is used in both PyTorch and the Python `random` library throughout the training process.

[1] From answer given in PyTorch forum by DCGAN co-author Soumith Chintala:
`https://discuss.pytorch.org/t/in-dcgan-why-the-kernel-size-of-4-is-used/20616`

[2] `https://github.com/sean-p-hill/DCGANBach`

```python
# Import functions, networks, and training loop classes
from utils       import (import_data, mk_output_dir, save_generator_output, plot_losses,
                         save_models, LSLoss, plot_fake_grid, generate_params)
from DCGAN_Model import  Generator64, Discriminator64, weights_init
from train_model import  TrainDCGAN

def main():
    # Certain Parameters are Passed through the Command Line
    if not google_colab: _ , gpu_num, input_data, num_epochs, parameter_file = sys.argv[:5]

    # Choosing whether to use a GPU or not - GPU pointer passed through command line
    try:
        gpu_num = [0] if google_colab else list(map(int,gpu_num.strip('[]').split(',')))
    except(ValueError):
        gpu_num = None

    # Create output folder labelled with the time that the code is ran at for identification
    output_dir = mk_output_dir(google_colab)

    # Deciding which dataset tp use, manually add in google colab but command line otherwise
    fn = "drive/MyDrive/NewGANData/12keys/Polyphonic" if google_colab else '../Data/'+ input_data
    print('Dataset: {}'.format(fn.split('/')[-1]))

    # Sets of Hyperparameters are pre-saved on file, so parameter_file determines which set to use
    params, device = generate_params(parameter_file, google_colab, gpu_num, input_data, fn, output_dir)

    # Import the dataset requested
    dataloader = import_data(fn,params)

    # Initialise and train the DCGAN model using the parameters and dataset specified
    model = TrainDCGAN(dataloader,device,params,output_dir)
    model.train(int(num_epochs))

    # Generates graphs, statistical metrics and generates images for the better stages of training
    run_analysis(output_dir)
```

Figure 4.2: The `main` function in *main.py*, which is used to import data, initialise models, train the DCGAN, and evaluate all performance metrics from a single file. This is an evaluation of the modular program designed to facilitate experimentation into modifications made to the standard DCGAN model.

4.2.1 Neural Networks

The generator and discriminator networks that compose the DCGAN model with architectures as described in Figure 4.1 are implemented in PyTorch, using object-oriented programming in Python with a `Generator64` and `Discriminator64` class to represent the networks. The networks are both designed as `torch.nn` modules, and the `Conv2d`, `ConvTranspose2d`, `BatchNorm2d`, `tanh` and `sigmoid` objects of this module provide the layers of the network. These are added to a `nn.Sequential` container, which allows forward and backpropagation in the given order through the layers. The network architecture is contained within `__init__`, and the only object method is `forward()`, which is a standard of all `torch.nn` modules, and therefore facilitates compatibility with the PyTorch methods used. The code used for these networks is given in Figures A.4 and A.5.

4.2.2 Training Process

This section explains the implementation of adversarial training between the generator and discriminator in the form of a minimax game, to form the generative

adversarial network. The weights of the convolutional and batch normalisation layers in the networks are initialised from a normal distribution of zero mean and standard deviation of 0.02 as in Radford, Metz and Chintala (2015).

```
def weights_init(m):
    '''
    Radford et al specify intiailising model weights to normal distribution
    with mean of 0 and stdev of 0.02.
    '''
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

netG = Generator64(params).to(device).apply(weights_init)
netD = Discriminator64(params).to(device).apply(weights_init)
```

Figure 4.3: Initialisation of weights in the convolutional and batch normalisation layers of the networks to values from $\mathcal{N} \sim (0, 0.02)$.

In the baseline model, binary cross entropy loss is used to evaluate the performance of the networks and influence model updates, and this is evaluated using `torch.nn.BCELoss()`. The optimiser which is used to evaluate gradients for backpropagation and update the network weights is the Adam optimisation algorithm, using `torch.optim.Adam()`. A separate optimiser is used for each network, and the function takes arguments of learning rate η and the Adam coefficients β_1 and β_2 (Figure B.1).

The training loop is where the adversarial training occurs, and for each epoch (full pass through the training data), the networks form a minimax game, where the generator is trained by minimising $\log(1 - D(G(z)))$, and the discriminator maximises $\log(D(x)) + \log(1 - D(G(z)))$. Details of the process carried out in each iteration; that is, for a single batch in the training dataset in each epoch, are given below.

Discriminator Training

The discriminator is trained using real images from the dataset, and fake images produced by the generator. The objective is to maximise $\log(D(x)) + \log(1 - D(G(z)))$, meaning maximising occurrences of correctly identifying real images x as real, and fake images $G(z)$ as fake. The training process is as follows, and PyTorch implementation is shown in Figure A.8:

1. **Reset Discriminator Gradients**: PyTorch retains gradients from previous backpropagations, so they need to be reset to zero.
2. **Generate Labels for Real Images**: Forward pass through the discriminator network to generate labels for images from the training data.
3. **Evaluate Discriminator Performance on Real Images**: Use the loss function to evaluate $L(D(x))$, against an array of real labels.

4. **Back Propagate Real Image Loss through Discriminator**: Calculate the discriminator gradients from generating the real labels.
5. **Calculate D(x)**: Evaluated as the mean of the discriminator output for real images.
6. **Generate Labels for Fake Images**: Generate an array of fake images using a random noise vector input, and pass these through the discriminator to determine whether they are real of fake.
7. **Evaluate Discriminator Performance on Fake Images**: Calculate the loss $L\left(D(G(z))\right)$ using an array of fake labels.
8. **Back Propagate Fake Image Loss through Discriminator**: Evaluate gradients from classifying the generator output.
9. **Calculate D(G(z))**: Mean of the labels generated by the discriminator for the generator output. The total loss from the discriminator for this iteration is evaluated as $D(x) + D(G(z))$.
10. **Update Discriminator Weights**: Use the Adam optimisation algorithm to evaluate and update the network weights after this iteration.

Generator Training

The training of the generator network occurs through the minimisation of $\log\left(1 - D(G(z))\right)$ - minimising the classification of generator outputs as fake images. To achieve this the generator loss is evaluated as if the fake images are real. Implementation steps are as follows, with code shown in Figure A.9

1. **Reset Generator Gradients**: As in the discriminator, gradients need to be reset to zero for each iteration.
2. **Generate New Set of Labels for the Fake Images**: Using the images produced by the generator, but with the updated discriminator network, classify the fake images by a forward pass through the discriminator.
3. **Evaluate Discriminator Performance**: Calculate the loss function of the discriminator, but using real labels instead of fake labels to form the requisite minimisation.
4. **Back Propagate through Generator**: Backpropagate the loss evaluated through the generator to calculate gradients for this iteration.
5. **Calculate D(G(z))**: Mean of labels given to the generator output.
6. **Update Generator Weights**: Use the generator's Adam optimiser to update the network weights.

4.2.3 Evaluation Metrics

Throughout the training process, the loss, accuracy and FID score of the networks' performance is evaluated and saved. Since this work has been carried out on shared GPU servers, which can be vulnerable to crashing, it was vital to save training information at certain points in the model. This can be achieved using PyTorch's `torch.save` functionality, to save all of the neural weights from `state_dict()`. These weights can be loaded later to either continue training and updating the weights on the dataset, or to generate further samples.

```python
from pytorch_fid import FrechetInceptionDistance

# Import pre-trained Frechet Inception Distance Model
FIDModel = FrechetInceptionDistance.get_inception_model()
FIDModel = FIDModel.to(device)

def FIDScore(real_batch,fake_batch,model,device,params):

    print('Calculating Frechet Inception Distance...')

    fid = FrechetInceptionDistance(model, dims=64,batch_size=params.batch_size)
    stats1 = fid.get_activation_statistics([gray2rgb(fake_batch)])
    stats2 = fid.get_activation_statistics([gray2rgb(real_batch)])
    score = fid.calculate_frechet_distance(*stats1, *stats2)

    return score

###### In training loop ##########
# Generate Sample from the Trained Generator
with torch.no_grad():
    fake_batch = self.G(self.fixed_noise).detach().cpu()

# Every 2000th Iteration Evaluate the Frechet Inception Distance
if i % 2000 == 0:
    fid_i = FIDScore(real_batch,fake_batch,self.FIDModel,self.device,self.params)
    FIDScores.append([fid_i,epoch])

    # If this is the best frechet seen so far, save the model
    if fid_i < best_frechet:
        print('New Best Model: Epoch {} FID: {}'.format(epoch,fid_i))
        save_models(self.G,self.D,self.output_dir,fn=f'BestEpoch{epoch}')
        best_frechet = fid_i
```

Figure 4.4: Demonstration of how the Fréchet Inception Distance is implemented using `fid_score` (Seitzer, 2020). This contains excerpts from two different areas in the code to summarise how the pre-trained model is initialised and added to the GPU initially, and a helper function to evaluate the FID from this mdoel is written.

4.2.4 Fréchet Inception Distance

Unless there is clear convergence in the adversarial learning process, choosing a point at which to stop training a GAN is complicated, and the more advanced generator may not be the best one. Although loss graphs are a good way to check for failure points in training, or to ensure convergence, it is hard to discern whether or not the GAN has 'peaked' or not. Borji (2019) explains how the Fréchet inception distance (FID) is a good metric for detecting the success of GANs and can be used to analyse at what point the generator produces the 'best' output. For this reason, the FID is evaluated between the generator output and the real images periodically. If the FID value is the best seen so far, the networks weights

are saved, so that the best networks from training can be used to generate more images once training is complete. The pre-trained *inceptionv3* model is required to evaluate the FID between samples, and a PyTorch API[3] containing trained neural weights for `torchvision.models.inception_v3` is used to evaluate the FID score (Seitzer, 2020). Due to computational restrictions, the FID score is only carried out once every 2000 iterations.

4.3 Model Hyperparameters

The hyperparameters to be tuned in the DCGAN model are shown in Table 4.1, alongside the values used by Radford, Metz and Chintala (2015). These parameter values will be used to form a baseline model.

Hyperparameter	**Original DCGAN Value**
Generator Learning Rate (η_G)	0.0002
Discriminator Learning Rate (η_D)	0.0002
Batch Size (B)	128
Adam Optimiser Coefficient (β_1)	0.5
Fake Image Label (y_F)	0
Real Image Label (y_R)	1

Table 4.1: Hyperparameters to be tuned for the models in this dissertation, and the values that were implemented in the DCGAN model by Radford, Metz and Chintala (2015).

Tuning hyperparameters in GANs is not as simple as other machine learning models, where a hyperparameter grid search can evaluate the performance of all possible combinations of sets of parameters, and find near optimal sets of values as such. The computational complexity, difficulties in quantifying success, and time constraints that come with training GANs mean this is not an option. Hyperparameter tuning in GANs instead often requires informed changes to parameters from performance feedback, and trial and error where that fails.

The parameters in this model are stored in an `AttrDict` dictionary called `params` and shown in Figure A.6, which allows all parameters to be passed to methods when needed. The `AttrDict` library allows dictionary values to be called as an object of the dictionary itself. Storing the hyperparameters in the dictionary that is passed through the training process is necessary when experimenting with hyperparameter tuning, for simplicity and thoroughness.

4.4 Implementation on MNIST Dataset

The DCGAN implementation with hyperparameters from Table 4.1 showed good performance on the MNIST dataset as explained in Section 3.6, and was able

[3] `https://github.com/kklemon/pytorch-fid`

to generate realistic looking handwritten digits as shown in Figure 4.5, without the generator collapsing to a single mode for all inputs. This was important to ensure that further modelling with the piano roll images can be carried out in the knowledge that there is nothing intrinsically wrong with the DCGAN implementation, and faults can be attributed to problem-specific issues.

Figure 4.5: Images generated by the DCGAN model when trained on the MNIST dataset for 200 epochs, showing good performance with no mode collapse, but with a small number of erroneous samples.

4.5 Baseline Model Results

As a baseline model, the hyperparameters shown in Table 4.1 are used in DCGAN, as proposed in Radford, Metz and Chintala (2015), and results evaluated on the C Major/C Minor and augmented dataset.

4.5.1 C Major/C Minor Dataset

With just 364 images in the training data, the minimax game formed when learning from this dataset suffered from mode collapse and failure to converge, when used with parameters from Radford, Metz and Chintala (2015), collapsing to an erroneous mode. Figure 4.8 shows how the network's loss functions progressed throughout training, and Figure 4.7 shows the mode to which the model collapses to for an arbitrary random input. For the discriminator loss to stay at zero and the generator to stay extremely high, this indicates that the generator has been unable to learn how to imitate the dataset before the discriminator became too strong and overtook it.

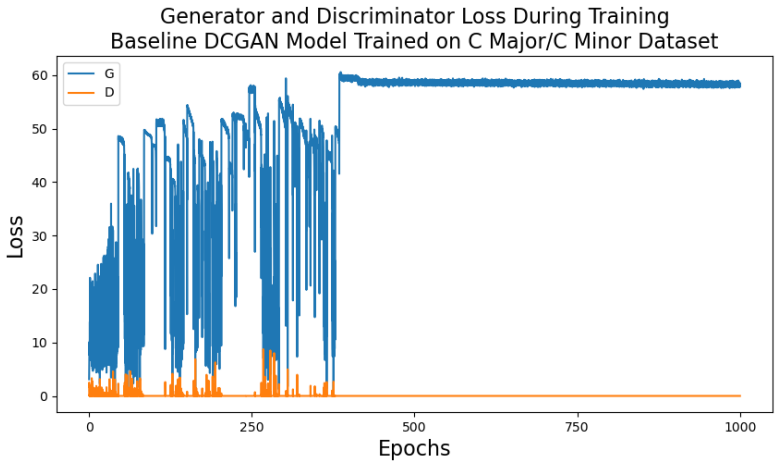


Figure 4.6: Generator and discriminator loss throughout training for the baseline DCGAN model when trained on the piano roll dataset for chorales transposed to C Major and C Minor. This shows severe mode collapse after approximately 350 epochs, and very little sign of adversarial training.

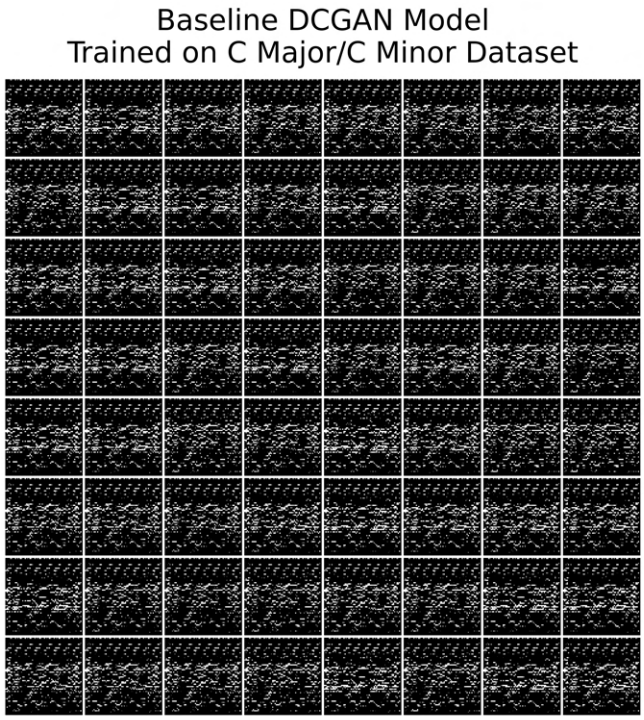


Figure 4.7: The images generated by the baseline DCGAN on the non-augmented dataset, showing complete mode collapse failure, with every one of the 64 inputs to the generator producing very similar outputs.

4.5.2 Data Augmentation to All Twelve Keys

The augmented dataset fell to mode collapse towards the end of training, however there is much more promising signs of performance in the early epochs, and at epoch 483, the DCGAN returned a FID value of 72.1, which is considerably lower than what was possible with the more limited dataset. Figure 4.8 shows the learning curve for this model, as well as the sample outputs taken from epoch 483, where the GAN learning peaks.

Baseline Model	C Major/Minor Dataset				Augmented Dataset			
	Min	**Max**	**Mean**	**SD**	**Min**	**Max**	**Mean**	**SD**
FID	266	457	370	41.6	72.1	479	251	140
Generator Loss	3.66	57.1	39.2	20.2	0	71.3	36.3	21.4
Discriminator Loss	0	5.78	0.07	0.28	0	16.6	0.04	0.45

Table 4.2: Statistics for the baseline model's performance on the piano roll data.

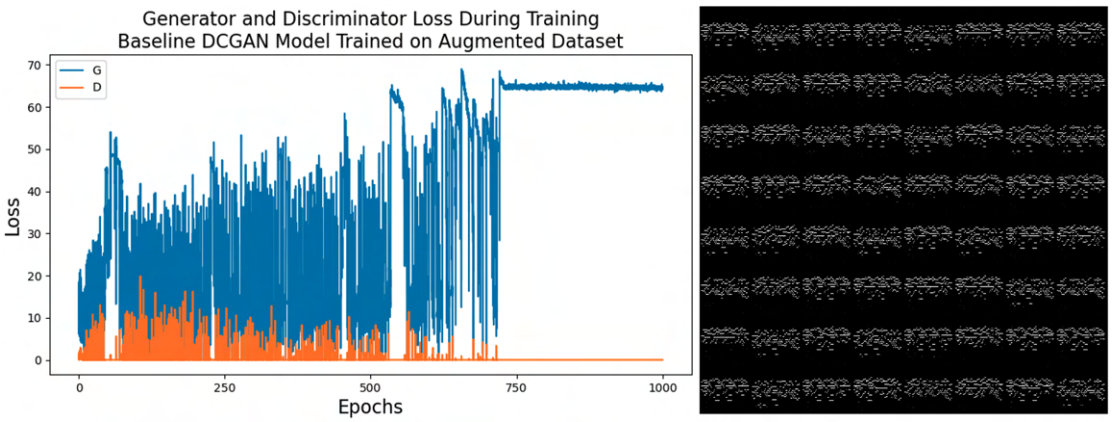
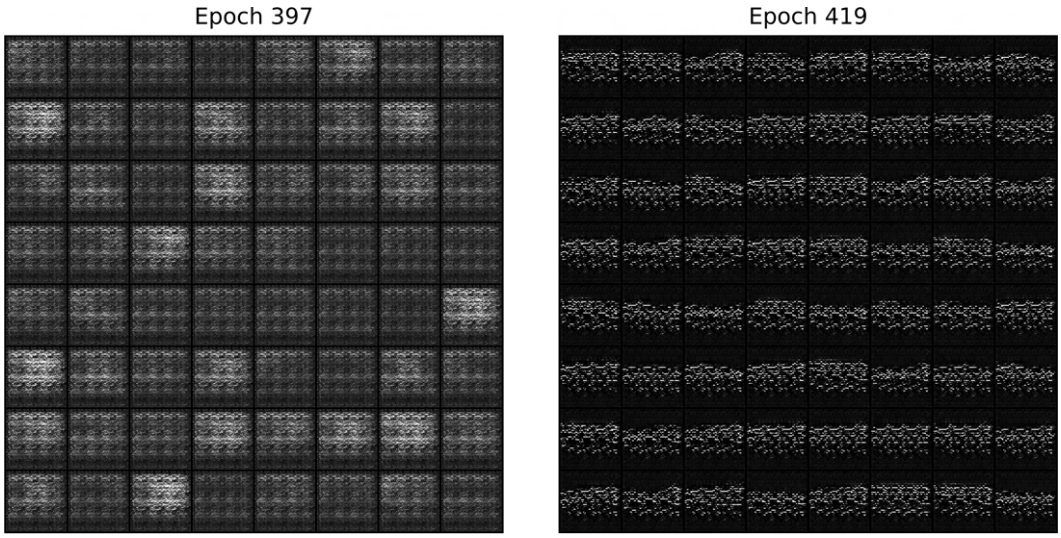


Figure 4.8: The baseline model still suffers from mode collapse on the augmented dataset, but it does not occur until much later in training than with the C Major/C Minor dataset. The images on the right shows the generator output at epoch 483 in training, where the FID score is minimised at 72.1, there is a lot of variety in the images outputted, and they produce music with some recognisable musical features. This suggests that the GAN is training well initially.

Figure 4.9c shows how the FID score for this model changes throughout training, and how there is a sharp drop just before the 400 epoch point, which is shortly before the best generator is formed. This indicates how unpredictable the training of GANs can be, in just 20 epochs, after 400 epochs of training, the model manages to seemingly learn the majority of the learning that it carries out over the entire 1000 epochs.



(a) (b)

(c)

Figure 4.9: Progression of the Fréchet inception distance (FID) between the generator output and the real images from the dataset for the baseline model trained on the augmented dataset. This shows how the optimal epoch in this scenario was not at the end of training. The generator output before and ofter this drop is shown in 4.9a and 4.9b, and indicates a large improvement in the images produced over just 22 epochs, or 2.2% of the overall training period.

This chapter has shown that the adversarial learning process between the two networks could use some improvement. Despite this, the music outputted by the baseline model shows evidence of learning musical features, and has a melodic feel to it. The results from this model show a level of unpredictability, and a somewhat lack of intuition. The two datasets behave differently when used to train the DCGAN, and in the baseline model, the augmented dataset outperforms C Major/Minor. The parameters used by Radford, Metz and Chintala (2015) are

evidently sub-optimal for use with this dataset, so some hyperparameter tuning is needed to produce better results that reduce the problems of mode collapse and non-convergence within training. The next chapter will propose three models with variations on the hyperparameters used and the training process itself, in an attempt to improve performance on the dataset, while evidencing the effect that different parameters have on the training process.

Chapter 5

Experiment Design

This chapter explains the process of tuning hyperparameters and making training adjustments with the aim of improving the baseline DCGAN model. Three models with varying hyperparameters are proposed, accompanied by justifications for their investigation. Initially, a qualitative analysis of the effect of changing hyperparameters, learned from research and experience in modelling, is explained to give context to the choice of models to be experimented with. The choices for the fixed hyperparamters are justified also, since the extent to which modifications to the DCGAN hyperparameters change its behaviour cannot be entirely explained by the three models alone.

5.1 Qualitative Analysis of Hyperparameters

In experimentation with DCGAN, lots of manual hyperparameter tuning and other modifications have been investigated to determine their effect. Listing all iterations and versions of models tested is not useful nor practical, so initially a heuristic understanding of how each modification individually effects the generation of images is given, before specific model iterations are explained. Overall, the hyperparameters that had the greatest effect were batch size, learning rates and real/fake image labels.

Learning Rates

The learning rate passed to Adam optimiser for the networks seems to be the parameter that DCGAN is most sensitive to. It seems that any increment to any one of the other parameters in the model requires extensive search to find a learning rate that matches them to produce good GAN performance. In general, however, higher learning rates have been found to be preferable when incrementing either way. This sensitivity to learning rates is helped by introducing a two time-scale update rule (Heusel et al., 2017), where the generator's learning rate is scaled up from the discriminator's, in an attempt to 'weaken' the discriminator and reduce vanishing gradients.

Batch Size

Using a large value of batch size, as in the baseline model, had an adverse effect on convergence between networks. This can been attributed to an over training of the discriminator too early on in the process with an overload of data. There is no benefit in having the discriminator far stronger than the generator early in training as the chances of convergence are greatly reduced. Another hindrance caused by the batch size used in training is the increase in training time. Ghosh et al. (2020) believes that there is an optimal batch size for each problem, however the methodology in finding it is still a source of deliberation in GAN research.

Adam Optimiser Coefficients

Modifying the β_1 and β_2 parameter of the Adam optimisation algorithm from the values given by Radford, Metz and Chintala didn't seem to have too much of an effect on the output of models in this research. Since Adam takes β_1, β_2 and the learning rate η as inputs to itself, and the learning rate has such a drastic effect on model output, it was ineffective to change these parameters. In *Deep Learning*, Goodfellow et al. (2016) describe Adam as 'robust' to hyperparameter tuning, and as such, the parameters were retained as $\beta_1 = 0.5$ and $\beta_2 = 0.9$ throughout.

Image Labels

The introduction of label smoothing in the models had a large effect on their performance, where 'fake' and 'real' labels of 0 and 1 are replaced by values such as 0.1 or 0.9 with the hope to reduce adversarial outputs from the generator. Goodfellow (2016) states how only the real labels should be altered when implementing one-sided label smoothing into the model, and explains how it is important to ensure the discriminator is not extrapolating excessively to generate unrealistic losses.

Number of Generator Updates Per Iteration

A method to encourage GAN convergence proposed by Robinson (2017), and which has seen success in the related work of Almeida and Pinho (2018), is to update the generator network twice for each update of the discriminator. This is a tool that worked in a similar way to applying the two time-scale update rule learning rates, and will be used in experimentation to encourage convergence in the minimax game between networks.

5.2 Proposed Models

The hyperparameters for the three models exhibited in this dissertation are given in Table 5.1. They have been deigned in ways to attempt to combat the three most common failure points of GANs: mode collapse, vanishing gradients and non-convergence, and these are three of the most revealing models based on extensive

experimentation with different hyperparameters. A conclusion from the baseline DCGAN parameters is that 128 is much too large a batch size for this dataset, and a batch size of 16 is found to work better, this also coincides with an increase in network learning rate, as per Ghosh et al. (2020). All three models use one-sided label smoothing to labels of 0 and 0.9 (Salimans et al., 2016), to attempt to encourage convergence of the networks. In the baseline model and model 1, the discriminator seems to be much too 'good' at discerning image labels for the generator to learn anything from it. Model 2 introduces a two time-scale update rule, in an attempt to weaken the discriminator and lessen the training gap between the two networks, and Model 3 attempts to reinforce the generator network further, by updating its neural weights twice at each iteration.

Hyperparameter	**Model 1**	**Model 2**	**Model 3**
Generator Learning Rate (η_G)	0.0002	0.001	0.001
Discriminator Learning Rate (η_D)	0.0002	0.0001	0.0001
Batch Size (B)	16	16	16
Fake Image Label (y_F)	0	0	0
Real Image Label (y_R)	0.9	0.9	0.9
Generator Updates per Iteration (N_G)	1	1	2

Table 5.1: Hyperparameters for the three models to be used in the experimentation. Model 1 reduces batch size and introduces one-sided label smoothing and attempt to weaken the discriminator. Model 2 applies a two-time scale update rule in the network learning rates to encourage convergence, and Model 3 investigates the effectiveness of updating the generator weights twice in each iteration, to further support convergence between the networks.

There are limitations to the experimentation procedure in this dissertation, which restrict the extent to which hyperparameters can be tuned. The time required to run the DCGAN for 1,000 epochs through the dataset varies with parameters and dataset, but the code can take up to 9 hours on a GPU for the augmented dataset of piano roll images, and considerably more on the entire MNIST dataset, so thoroughness to the extent of Ghosh et al. (2020) is not feasible within the time frame of the dissertation. Despite this, all models have been run to a stage where the pattern of learning is clear, or convergence has been reached, so this will not be a limiting factor on the analysis of results possible.

Chapter 6

Experiment Results

This chapter describes how the varying hyperparameter models have been implemented, and provide the results seen from simulating the DCGAN with said parameters. Three models with varying hyperparameter sets are investigated, in order to summarise the experimentation process that has formed this research; provide insight into the difficulties that come with training GANs; and apply hyperparameter remedies to these problems as found from literature. The results from each of the models when trained on both the C Major/Minor and augmented datasets are shown, and explained in terms of the adversarial learning between the networks and loss generated, with example outputs given.

6.1 Model 1

This model makes two changes to the baseline DCGAN: reducing batch size to 16, and introducing one-sided label smoothing. The results of this model are given in Table 6.1, and the lowest the FID score reached in training is 93.1 for C Major/Minor, and 95.7 for the augmented dataset, which is a sign that the model isn't imitating the augmented dataset quite as well as the baseline model, but exhibits much better results on C Major/Minor. The model suffers from non-convergence between the discriminator and generator, and it is clear from the loss curve in Figure 6.1 that the discriminator is much too strong for there to be sufficient early adversarial learning between the networks. It should be noted that batch size and learning rates rely on each other heavily, and Ghosh et al. (2020) states that the lack of consensus in research for an acceptable relationship between the two in the training of stable GANs mean that evaluating optimal values for these hyperparameters has to be problem-specific "trial and error". It is difficult for the model to adapt to a new batch size without a simultaneous change in the learning rates.

In contrast to the baseline model, there is very little discrepancy between the performance of the two datasets, with both learning at a very similar rate, and having similar metric values, as shown in Table 6.1 and Figures 6.2a and 6.2b. One key difference is a much higher degree of mode collapse in the C Major/C Minor

dataset than the augmented dataset, and this could be an indication that a smaller dataset can encourage mode collapse in the GAN.

Model 1	**C Major/Minor Dataset**				**Augmented Dataset**			
	Min	**Max**	**Mean**	**SD**	**Min**	**Max**	**Mean**	**SD**
FID	93.1	482	170	60.1	95.7	478	171	48.2
Generator Loss	0.69	21.6	8.63	2.44	0.37	31.0	11.3	2.52
Discriminator Loss	0.33	2.64	0.35	0.074	0.35	3.53	0.33	0.035

Table 6.1: Results from training Model 1 on both datasets, showing much better results on the C Major/Minor dataset, but reduced FID score on the augmented dataset.

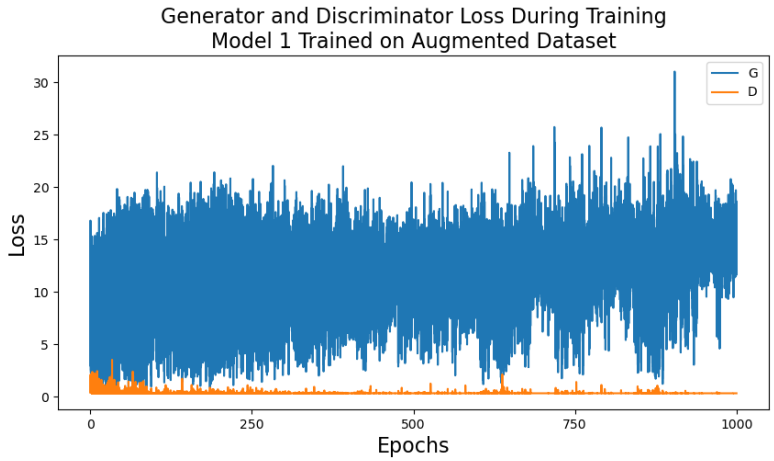
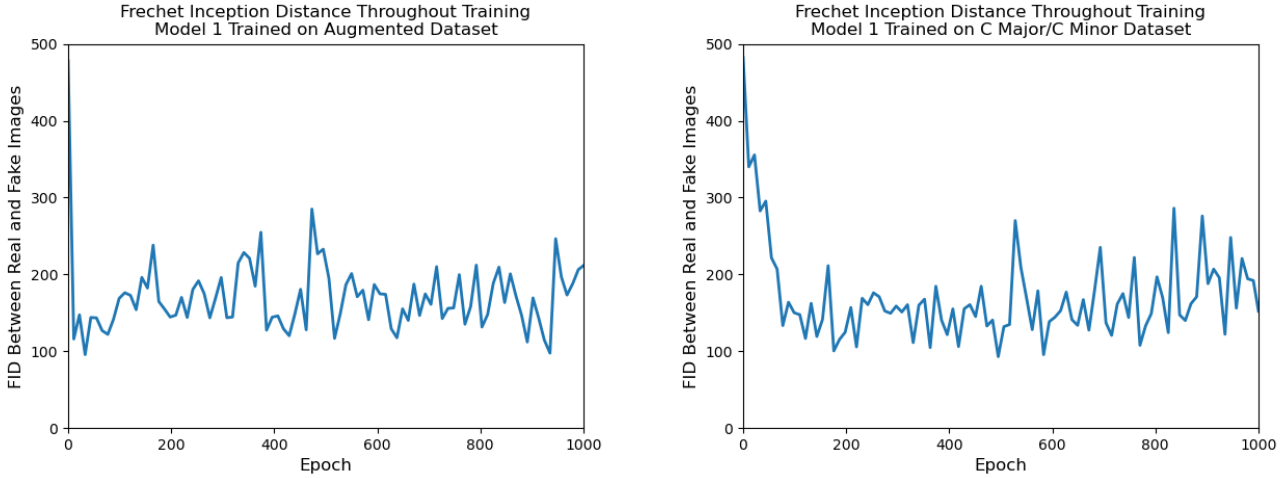


Figure 6.1: Loss curve for model 1 on the augmented dataset, this shows how this model improves on the baseline model, with no complete convergence failure, however the presence of adversarial learning between the networks is quite weak, with the discriminator overpowering the generator significantly.



(a) (b)

Figure 6.2: Plots showing how the progression of the FID score for model 1 on both datasets is very similar, with no significant improvement made after approximately 200 epochs for both. Interestingly, the models both peak very early into training.

Despite the slight increase in FID score for the images produced by the generator of DCGAN with these hyperparameters, when listening back to the images converted to music, the C Major/C Minor output exhibited signs that it had learned some musical features from the dataset. Figure 6.3 shows an example of three bars of the output provided by model 1, when trained on he C Major/C Minor dataset. The music shows signs that the model is learning musical features from the data, and a C major chord can be seen in the extract shown. The rhythmic structure seems to be syncopated off the beat, but there is evidence of the music showing chords, notes in the correct range, and roughly four notes at all times. Unfortunately, the output which this image corresponds to is from a batch with mode collapse, so the model has overfit to produce this exact melody, rather than to compose music in general.

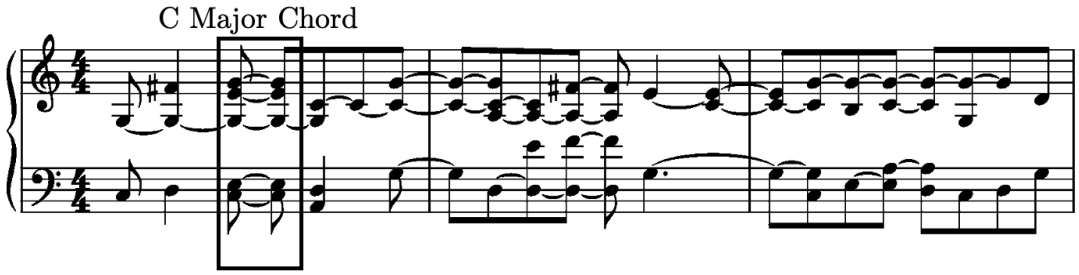


Figure 6.3: Example of a three bar extract from the optimal output of model 1 when trained on the C Major/C Minor dataset, evidencing clear learning of musical features, most notably with the repeated use of the C major chord.

6.2 Model 2

The primary point of failure for the DCGAN using the model 1 hyperparameters is that it cannot avoid the vanishing gradient problem, where the discriminator is too 'good' a classifier in the early stages of training. This means the generator is not provided with enough information from discriminator loss, so each update to its weights through backpropagation provides little gain. Model two implements a two time-scale update rule (Heusel et al., 2017), setting the generator learning rate to 0.001 and the discriminator to 0.0001. This strengthens the generator, with the aim of inducing a level of convergence between the networks.

Model 2	**C Major/Minor Dataset**				**Augmented Dataset**			
	Min	**Max**	**Mean**	**SD**	**Min**	**Max**	**Mean**	**SD**
FID	73.4	459	142	55.8	103	459	163	69.2
Generator Loss	0.73	16.3	7.39	1.84	0.43	22.0	9.38	2.70
Discriminator Loss	0.32	7.06	0.37	0.23	0.33	7.90	0.35	0.17

Table 6.2: Results from model 2 when trained on both datasets, this shows a much better FID score for the C Major/Minor dataset than the augmented data.

When applied to the piano roll images, the two time-scale update rule didn't have the desired effect, and the discriminator still seems to overpower the generator. There is the beginnings of mode collapse with both datasets from approximately 100 epochs, and the amount that the data is prone to modal collapsing will not help with convergence between the networks. Similarly to model 1, the best generator under these hyperparameters does produce music that has melodic characteristics, and recognisable chords. Interestingly, the passage of music that is generated by training model 2 on the C Major/Minor dataset for this model is in the minor key as opposed to the major key in model 1, and Figure 6.4 shows this in a musical score.

Intriguingly, while the presence of a minor chord in the music indicates learning of musical concepts in the model, close inspection reveals this to be a second inversion chord. The presence of this second inversion chord indicates that it is highly unlikely that the model is learning musical characteristics of Bach chorales, and rather the chord itself to fit in with the key signature, since second inversion chords are highly uncharacteristic of the corpus (Pankhurst, 2017; Oxford, n.d.). Despite this, it is interesting to see that the models are capable of producing exclusively minor or major music.

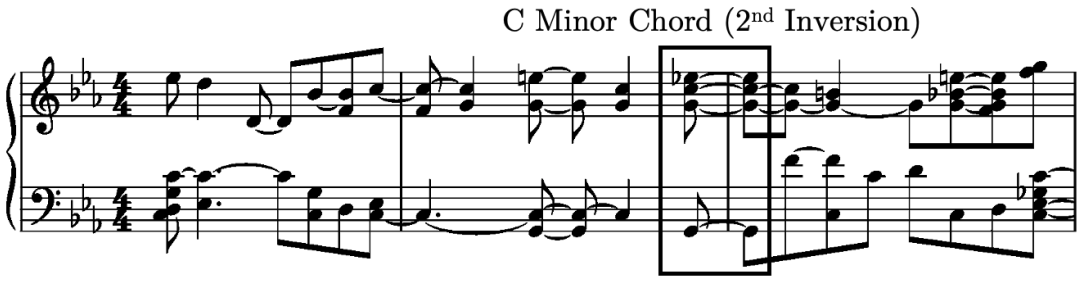


Figure 6.4: Sample generated using the highest performing generator through the training process of model 2 according to FID score, trained on the C Major/Minor dataset. This shows clear musical features, such as chords, melodic lines, and distinct 'voices' in the music, which are all features of Bach chorales. The presence of a 2nd inversion minor chord is rare in chorales (Marshall, 1970), so it is unlikely that this has been learned directly from the dataset.

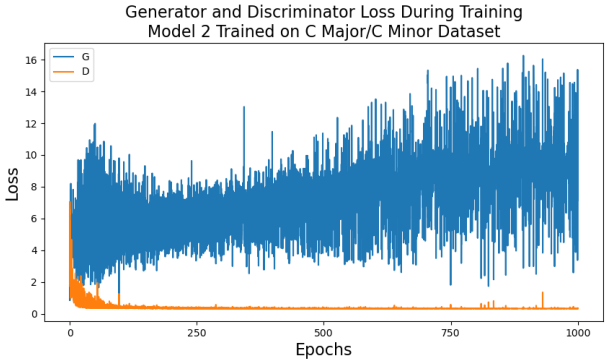


Figure 6.5: Loss evaluated over the training process for the generator and discriminator of model 2.

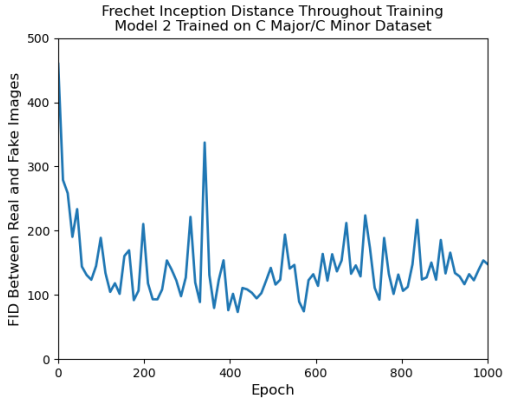


Figure 6.6: Plot of the Fréchet inception distance throughout training in model 2 when trained on the C Major/Minor dataset. Unlike the results in model 1, the DCGAN continues improving its output in terms of its FID score until over 400 epochs.

6.3 Model 3

The non-convergence between networks was not entirely remedied by applying a two time-scale update rule for the learning rate parameters of each network. Another method proposed by Robinson (2017), where the generator is updated twice at each iteration in the training process is attempted in model 3 in combination with the two time-scale update rule. This aims to enhance the extent to which the loss from the discriminator is backpropagated through the generator network, and increase their adversarial relationship.

Model 3	**C Major/Minor Dataset**				**Augmented Dataset**			
	Min	**Max**	**Mean**	**SD**	**Min**	**Max**	**Mean**	**SD**
FID	75.8	492	143	54.0	100	479	174	61.9
Generator Loss	1.16	19.0	5.67	1.98	0.56	22.0	8.13	2.12
Discriminator Loss	0.33	6.58	0.38	0.28	0.33	7.20	0.34	0.15

Table 6.3: Statistics of the network losses and Fréchet Inception Distance score for the result of model 3 when trained on each dataset.

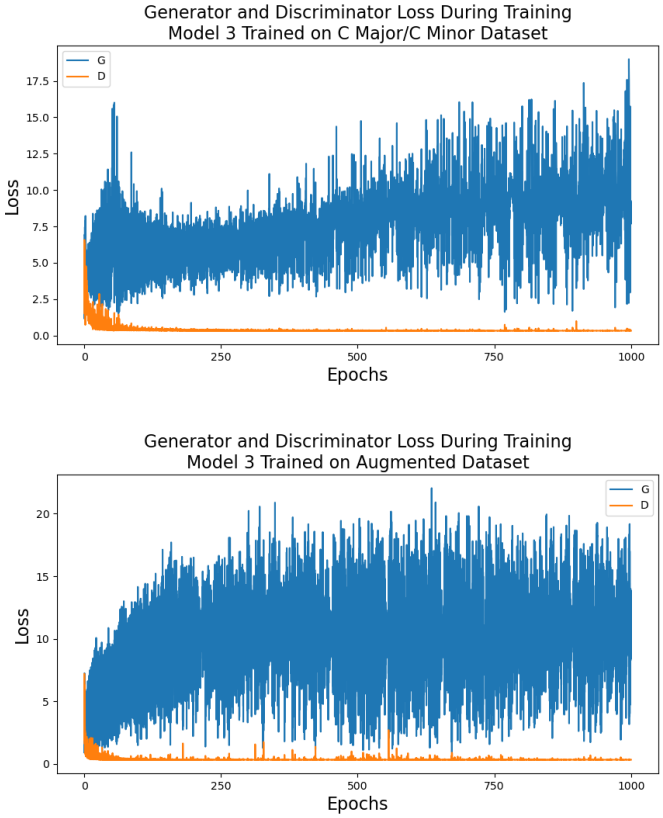
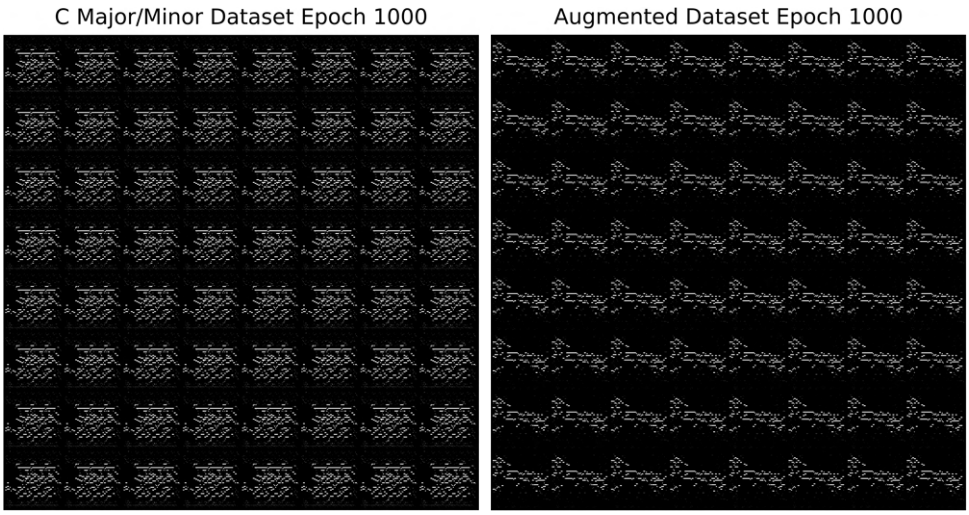


Figure 6.7: Loss for model 3, with two generator updates per iteration. This does not have the intended effect of resulting in convergence between the two networks.

Figure 6.7 shows the network losses over the training period for this model on both datasets, and although there isn't much improvement, there is a step in the direction of GAN convergence. There is evidence of the generator learning being improved with the extra update per iteration, with the mean generator loss decreasing from 7.39 to 5.67 from model 2 to 3 for the C Major/Minor dataset, however this has no effect on the discriminator, so strengthening the generator did not help convergence between the networks. Once again the models seem to struggle with mode collapse, as can be seen in Figure 6.8, which undoubtedly contributes to the non-convergence.



(a) (b)

Figure 6.8: Generator output for the model 3 at the end of training, showing how both models suffer from mode collapse. This can be seen since each of the images in each batch correspond to a different input, yet all are similar or identical.

6.4 Experiment Summary

Experimentation with these models over the two datasets has demonstrated the effect of hyperparameter tuning in the DCGAN model, and allowed interpretation of the benefits and drawbacks of the two datasets.

In terms of the datasets, data augmentation has been validated as required for this work, since the size of the C Major/C Minor dataset has made the DCGAN, with only 364 samples, very prone to collapsing to specific modes. If there wasn't a restriction on dataset size, the C Major/Minor dataset would likely be preferable to the augmented dataset, and this is reflected in the music generated by the DCGAN. Having all of the pieces transposed to the same key exploits the transpositional invariance of music, and therefore requires the network to only learn harmonic rules for two domains: the minor key and the major key. Evidence of this being learned has been proven, with Models 2 & 3 producing a musical output with recognisable chords, and outputs were generally in the major or minor key, rather

than a mixture. This is stylistically representative of the dataset, and evidence that some features are being learned.

The models investigated, while not exhibiting complete convergence to a Nash equilibrium, have given a good insight into the difficulties that come with training a GAN. Although none of the models massively outperformed the baseline model from an adversarial learning perspective, the music produced as a result of this learning exhibits promise that DCGAN is able to learn musical features from the dataset, and will be explored further in Section 7.1. This experimentation has shown that when a dataset is prone to mode collapse, tuning of the hyperparameters alone is not sufficient to prevent this recurring, and other methods that can be implemented to remedy this as an extension of this dissertation will be investigated as part of the critical analysis.

Chapter 7

Critical Analysis

This dissertation set out to compose music in the style of Bach chorales using a DCGAN model which is trained on the music represented as images. The music generated as an output of this model feels 'musical', and is much more than a collection of sounds. The experimentation in hyperparameter tuning the DCGAN model, to enable convergence, and proper adversarial learning between the generator and discriminator networks, had less success, and the vulnerability to modal collapse proved intangible with these methods alone. The used of GANs to produce music is still a relatively new area, so it is pleasing that the research into common points of failure in GANs, difficulties in hyperparameter estimation, and evaluation methodology can further advance and be valuable to any future work to be carried out of this kind.

The nature of the problem at hand is intrinsically troublesome, and whilst there has been success in developing GANs and other deep learning algorithms to imitate the style of music in a training corpus, the majority of these successful models are trained on many thousands of artefacts, and represent the data sequentially to capture the temporal relationships between musical notes. Using convolutional networks to grasp the same musical features and characteristics of a dataset is found to be much more difficult, however signs of promise are there, with the DCGAN learning musical features such as chords. Further research will be worthwhile to develop more problem specific models, which is likely what will be be required to garner success. This chapter analyses the work carried out in this dissertation, and critiques any areas that need particular improvement.

7.1 Analysis of Generated Music

The quality of the music generated as outputs to the various iterations of the DCGAN varies, from complete lack of musical identity in cases such as the mode collapse at the end of training the baseline model, to having clear identifiable musical features, such as familiar rhythmic melodic structure, occurrences of major and minor chords, and even some signs of cadential beginnings. Despite this, for all

four of the models, the DCGAN generator at the epoch with the lowest FID score throughout training all produce sounds that can definitely be called 'musical'. This section will provide a qualitative analysis and comparison of the music generated by the models. For the sake of comparison, this section of the dissertation compares the music produced by images generated by the generator at the epoch with best FID score between DCGAN output and the training data. Details of these outputs are given in Table 7.1, and examples of the music produced by the models in dissertation can be listened to on soundcloud[1].

Dataset	**Model**	**Best Epoch**	**FID**	**From Generator with Mode Collapse?**
C Major/Minor	Baseline	286	266.1	Yes
	Model 1	495	93.0	Yes
	Model 2	418	73.4	Yes
	Model 3	264	75.8	Yes
Augmented	Baseline	484	72.1	Yes
	Model 1	33	95.7	No
	Model 2	11	102	No
	Model 3	11	100.5	No

Table 7.1: Table explaining the epoch and FID scores of the best images generated by each model, and whether they are a result of modal collapse or not. All models were trained to 1000 epochs, yet the best outputs all came in the first half of training, mainly due to modal collapse in the latter stages.

It is difficult to quantify the quality of music generated by all of the models, but musical concepts can be extracted from the scores and analysed. Figure 7.1 displays the first four bars of the best output from each iteration of the models. The key signatures of each of the outputs has been determined using `music21.analyze.key`, and shows how, when trained on the C Major/Minor dataset, all of the generated outputs are in said keys, whilst the augmented dataset generates music from four different keys for each model. There is also clear evidence in every output of recognisable chords.

One way in which the music is distinguishably different to the Bach chorale harmonisations is rhythmically. The majority of the notes and chords in all of the pieces seem to be played off the beat, whilst in Bach's chorales, off beat notes are usually reserved for ornamentation and passing notes between chords. The music is also lacking from a sense of form or global structure over the eight bar phrases, which is a familiar point of failure in many AMC models (Fernandez and Vico, 2013). There is definitely evidence of 2-3 bar phrases within the music, and notably a clear 5 bar melodic phrase, with consonant harmony, in the beginning of the output for model 2 trained on the augmented dataset.

[1] `https://soundcloud.com/user-9329809-979208538/sets`

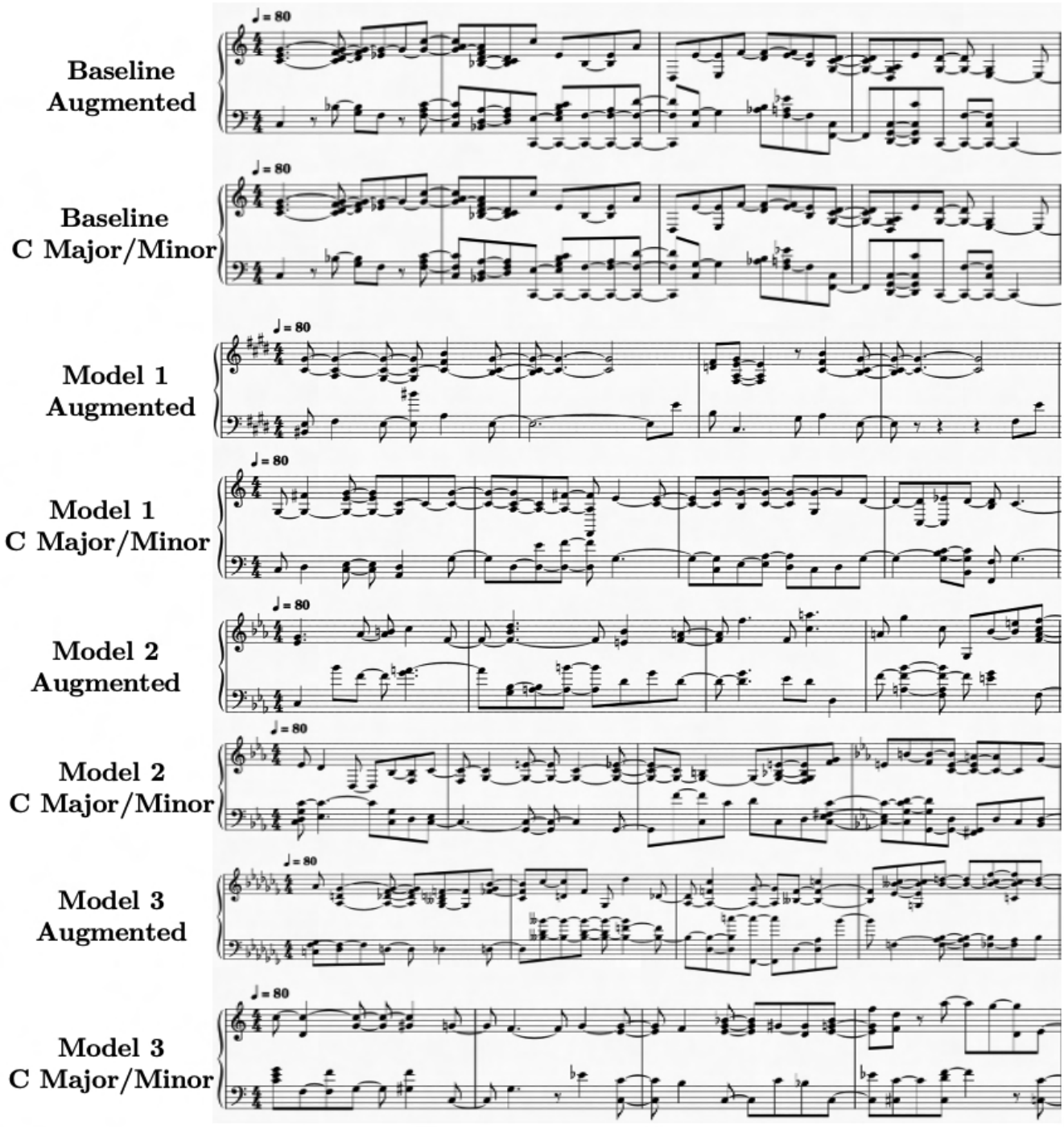


Figure 7.1: Music score excerpts of the best output of each model according to the FID score against the training data. This shows evidence of the C Major/Minor models restricting the output to the input key signatures, more syncopated rhythms than would be expected in Bach's chorales, and clear evidence of recognisable chords.

7.2 Features Learned from Images

While the output of the DCGAN models is not at the stage of being classed as musical imitations of Bach chorales, their outputs still show evidence of learning from the dataset, and ways in which this has occured are analysed further in this section.

Low Grayscale Noise Levels

The training dataset is made up of piano roll images, made up of black and white pixels only; their grayscale pixels are only of the values 0 and 255. In the planning stage, it was assumed that a filter may be necessary in the generator network to enforce this, however this aspect of the images was learned very well, and experimentation with adding a `torch.nn.threshold` layer as the output to the generator actually reduced the models' performance. Figure 7.2 shows histograms for each of the models output pixels.

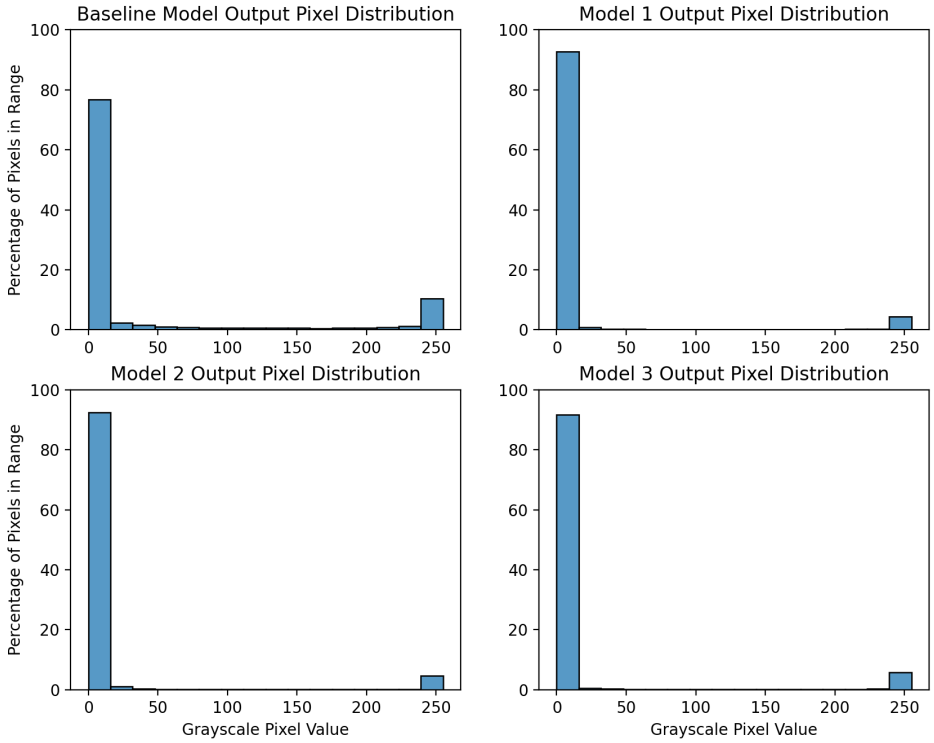


Figure 7.2: Distribution of pixels outputted by a batch of the best generator by FID score for each model explained in this dissertation. This shows that the models are, by large, learning to only output black or white pixels and nothing in between.

7.3 Mode Collapse

The research in this dissertation investigates the extent to which the tuning of hyperparameters can improve the DCGAN model (Radford, Metz and Chintala, 2015). It has been concluded that the main obstruction between complete convergence between the two networks is susceptibility to modal collapse in the generator, and that the tuning of hyperparameters alone is not sufficient to prevent this. The aim for the DCGAN in this research was to be able to produce varying musical outputs, all of which retain characteristics of Bach chorales, whilst instead the generator optimises to produce a single output to appease the generator, although evidence of learning from the chorales is clear. This section investigates ways that extend simply tuning hyperparameters, as methods in which mode collapse could have been avoided given more time.

Using Multiple GANs

Although mode collapse indicates a failure point in the intended learning outcome of a GAN, it doesn't mean that the output is nonsensical. From Table 7.1, it can be seen how the top four FID scores out of the eight models came from outputs that the generator had mode collapsed towards. It is also evident that the generated output, although consistent regardless of input, continues to improve it's FID score after mode collapse has occured. This is exhibited in the baseline model trained on the augmented dataset, Figure 4.9 shows how mode collapse occurs between the 397^{th} and 419^{th} epochs, yet the lowest FID score occurs in the generator at epoch 484. In theory, if you were to simulate the GAN training from scratch enough times, then enough valid modes will be produced as to fix this problem. This is a trivial solution to the issue of mode collapse, and the time it takes to train GANs makes this an unrealistic solution for most GAN settings in mode collapse.

Minibatch Discrimination

Traditional training of GANs requires the discriminator to produce a label for each image within a batch that is passed to it, and to classify it as either real or fake. In the case of mode collapse, there is no way for the discriminator to identify if all of the image within the batch are the same, and the issue goes unnoticed. Salimans et al. (2016) introduce minibatch discrimination as a way to alleviate modal collapse, where the output to the penultimate layer in the discriminator $(\mathbf{x}_i)$ is concatenated with a measure of the similarity $o(\mathbf{x}_i)$ between the image that forms $\mathbf{x}_i$, and the rest of the images in the batch. The intention of this so that mode collapse is detected instantly, and this information is propagated through the two networks, to prompt the generator to change it's tactic.

Feature Matching

The problem of mode collapse in GANs stems from when the generator is overtraining on the discriminator to produce a single valid output, rather than setting itself up to produce multiple valid outputs. Another method propose by Salimans

et al. (2016) is feature matching, which modifies the objective of the generator network, so that the generator is aiming to match the statistical features of the real data, instead of trying to maximise the output of the discriminator. The role of the discriminator instead becomes to decide what statistical features of the data discriminate the most between real and fake data; hence 'feature matching'. To achieve this, a new loss function for the generator is developed. This is shown in Equation 7.1, defined as the difference between the expected value of the value of an intermediate activation function f on the real data and the generator output. This is a way to incorporate adversarial learning between the two networks without being reliant on the output classification of the discriminator.

$$L_G(\mathbf{x}) = \left\| \mathbb{E}_{x \sim p_{data}} f(x) - \mathbb{E}_{z \sim p_z(z)} f(G(z)) \right\|^2 \tag{7.1}$$

Unrolled GANs

In traditional GANs, the generator learns only from the discriminator at the current iteration in the training process. Metz et al. (2016) present 'Unrolled GANs', which instead train the generator to also learn from several future iterations of the discriminator. When a GAN goes into mode collapse, the generator becomes focused on local optima that satisfy the current discriminator. 'Unrolling' the GAN prevents this by giving the generator knowledge of how the discriminator will classify a given output as the discriminator gets more advanced, making it less likely to converge to a single output. When using k future discriminators, this means that the generator's updated neural weights will be as a result of k full backpropagations through the various networks.

7.4 Dataset Restrictions

The dataset used in this dissertaion; Margaret Greentree's Bach chorales as revised by Bump (2014), is well researched in AMC literature, and has seen success in models such as Hadjeres, Pachet and Nielsen (2017); Johnson (2017) owing to its homogeneity. Despite this, it is possible that the size of the dataset may well have been a restricting factor on the success of this work. Ng (2016) explains how one of the primary advantages of deep learning over traditional machine learning is that the performance of neural networks increases with the size of available data (Figure 7.3). Given success on the much larger MNIST dataset without the network falling to modal collapse, this is a reasonable hypothesis to make, but one that cannot be validated without a much larger corpus.

It would be interesting to see the performance of the DCGAN architecture explained in this dissertation on a larger musical dataset. It must be noted, however, that while in this problem will reduce the similarity between data samples, and it can be seen how the generalisation of the augmented dataset reduced the level of musical features that the music was able to learn. Liang et al. (2017) describes the Bach chorales as the largest corpus of music with such homogeneity, so an increase in

dataset size will have to sacrifice similarity between samples.

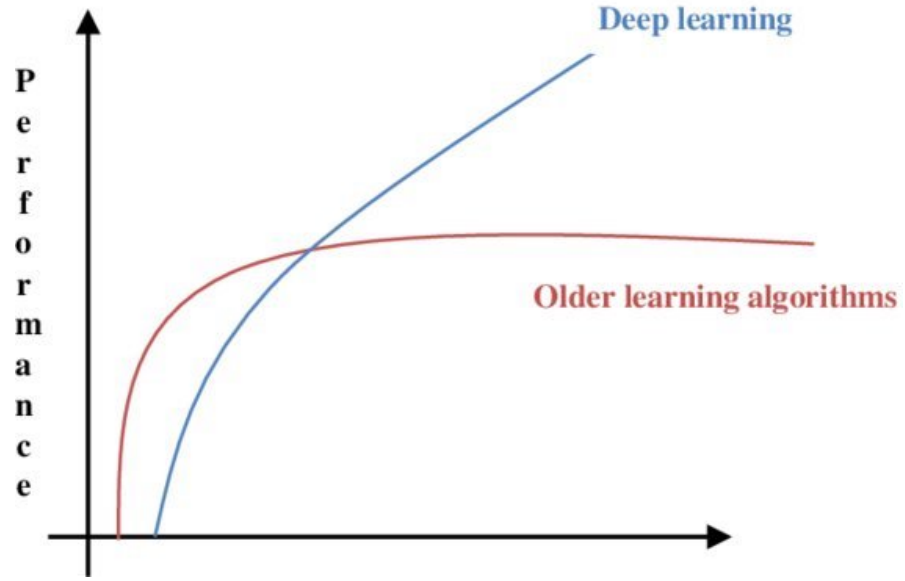


Figure 7.3: Graph showing how deep learning perfomance tends to improve as the amount of training data available increases. *Figure obtained from Kumar and Manash (2019).*

Chapter 8

Conclusion

This chapter provides a high level analysis to evaluate the success of the work carried out in this dissertation, establishing the contributions made to literature, whether the research aims have been met, and explaining what further work can be taken to develop on this research.

8.1 Contributions and Achievements

This dissertation set out to develop a dataset of graphical representations of Bach chorales, and use a deep convolutional generative adversarial network (DCGAN) to generate imitations of these images. The objective of these images is that they can be converted into music that is unique, and in the style of Bach's corpus of chorale harmonisations. The research and programming carried out in this work has produced models that, whilst struggle with collapsing modes in their output, generate music that is much more than a collection of sounds, and display evident learning of musical features, but lacked the form and harmonic consonance of Bach's compositions. The development of these models enabled extensive research into the difficulties that are commonplace in the training of GANs, and hyperparameters of the DCGAN infrastructure are established, to allow analysis of their effects, and tuning to optimise outputs. The success of two datasets has been examined, with comparisons made between a dataset using chorales transposed to C Major and C Minor to assist in the network's ability to learn the basis of musical harmony; and an augmented dataset where chorales are transposed to all 12 musical keys, which aims to bolster the performance of the DCGAN with a larger dataset.

From experimentation with varying hyperparameters of the DCGAN model, it can be seen that variation of the network learning rates with two-sided time update rules, implementation of one-sided label smoothing, and a reduction in batch size shifted the model towards convergence in the minimax game between the networks. The problem of mode collapse; where the generator converges to a local optima for all inputs to satisfy the discriminator, has proven difficult to overcome in a model such as this. It has been concluded that the augmented dataset succeeds further

into training, while maintaining an adversarial relationship between networks, but the smaller C Major/Minor dataset shows much more evidence of learning musical features. Ultimately, the models produced in this research have success in that they produce musical samples which retain music theoretic concepts, and show signs of imitation of the dataset in terms of statistical features. This is a success given the unsupervised nature of the learning, and without any declarative assistance provided to the model.

This dissertation contributes to the research of automated music composition by showing that a deep learning music generation model which is trained on only piano roll images, and without any declarative inputs or musical knowledge encoded into the training process, can generate music that sounds 'musical'. With modifications to the design of the piano roll-based image, and developing the DCGAN training process to encourage learning of musical aspects, this work has shown that DCGAN is a good competitor as a music generation deep learning model, and further experimentation ideas are proposed in Section 8.2 to extend this research.

Music composition requires the following of systemic rules with added embellishment from human creativity. The research in this dissertation has further reaffirmed how automating the composition of music is a an great tool to both showcase the abilities of these powerful algorithms with the use of GANs, and to influence the development of new ideas in the area.

8.2 Further Work

8.2.1 Participant Survey

Although the analysis in Section 7.1 provides a good understanding of the success in the models in terms of their music produced, given more time, a participant survey can be produced to quantify the quality of the music generated in this dissertation. As explained in Section 2.3.8, a challenging aspect of GANs is a lack of universal metrics for assessing the performance of models. This is further amplified in a music generation model, due to the subjectivity of music. Analysis of results shows how the Fréchet inception distance and loss functions are limited in regards to analysing musical success, so a participant survey would improve the level of analysis possible.

Given the quality of the output produced isn't representative of Bach chorales, a good way to incorporate a participant survey into analysis would be to evaluate each of the pieces on different musical concepts. A good set of questions for this would be:

To what extent does the musical sample satisfy the following points? Give a score from 1 to 10, with 1 meaning 'not at all' and 10 meaning 'entirely'.

- Definitively in either a major or minor key.
- Shows characteristics of Bach chorales.
- Music is free from uncharacteristic dissonance from erroneous pitches.
- Cadential harmony exists within the passage.
- The music is made up of four voices at all times.

Given the nature of these questions requiring a fairly high pre-requisite knowledge of music theory, the sample participants can be taken from an online music related forum, or a university music society. For validation, a preceding question defining the participants' musical experience will ensure of this.

8.2.2 Comparison of Model Adaptations

As has been explained in Section 7.3, one of the main problems experienced in the training of GANs in this project was that of mode collapse, where the generator overfits to the discriminator by producing the same output for all inputs, and is difficult to overcome with hyperparameter tuning alone. With the knowledge and experience obtained from this research, a better experimentation design for this problem would be to compare the effects of all of the proposed solutions to mode collapse, which are brought forward in Section 7.3. Since the models have all shown good promise in the learning of musical features, it is believed that overcoming mode collapse will produce considerably better results.

8.2.3 Neuron Activation Analysis

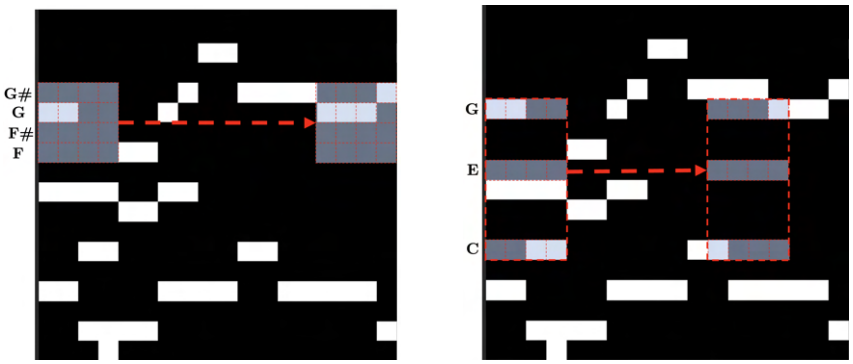
The problem that neural network-based deep learning algorithms often encounter, is that they are difficult to interpret, and as such sometimes their behaviour cannot be explained intuitively. To an extent, this is what makes hyperparameter tuning of GANs so difficult, since any change you make is only influencing the behaviour of the model, instead of determining its outcome. For example, a tweak of the learning rate of the discriminator network is modifying the amount to which your discriminator updates it's weights at each iteration in the learning process, in the hope that this will modify the loss that the network produces when classifying an image produced by the generator. This results in the generator training itself slightly differently, which in turn will be reflected in the backpropagation of the generator neural weights, which are used to generate a desired output. The connection between an increment in this learning rate to the output itself is so far detached, that it makes it difficult to understand what is actually happening within the network.

The way in which Yang, Chou and Yang (2017) looks into "opening the black box" in their RNN-based chorale generation model '*MIDInet*' would be a useful

extension on this dissertation to diagnose where the issues originate from in the DCGAN model to inform improvements to the model. Yang, Chou and Yang encounter several neurons that pick out I^6 chords, and another that correlates with perfect cadences. Given the presence of the C major chord in model 1, there will likely be neural activations in the generator model which produce this.

8.2.4 Model Specific Training

One of the aspects of the learning process in this dissertation that may have hindered success is that DCGAN was designed by Radford, Metz and Chintala (2015) to produce images which are visually recognisable, such as the MNIST dataset. The work in this dissertation focuses on producing images that can only be interpreted properly when converted a musical score, or to music and listened back to. The kernels that form convolutional filters are good at detecting features in an image based on the pixels that are close or near to it. In the piano roll representation of music (Figure C.1), vertically neighbouring pixels are notes that are a semitone apart, which are dissonant, and hold almost no harmonic relationship. It would be interesting to develop a kernel for this problem that is specifically designed to identify common harmonic relationships, such as major and minor chords. A example of a way in which this could be achieved is shown in Figure 8.1, where 8.1a shows the kernel in use by DCGAN. Figure 8.1b gives a suggestion of a 3x3 sparse-row kernel, that is vertically spaced so as to identify major chords. In the piano roll images, a major chord is spaced with three pixels separating the lowest and second note, and two pixels separating the second and highest note.



(a) 4x4 Kernel in use currently in this dissertation's models.

(b) Modified kernel designed to identify major chords.

Figure 8.1: Comparison between DCGAN's original 4x4 kernel, and an idea for a new kernel to be used in the convolutions, which is vertically spaced in such a way as to identify major chords in the music. For clarity, a cropped version of a training sample has been used as an example.

8.2.5 Improved Piano Roll Representation

The way that the music was represented graphically in this dissertation may well have restricted the extent to which the models were able to learn the harmonic features of Bach chorales. It will be interesting to investigate differing representations.

The images produced in Chapter 3 certainly encode all of the musical data into the model, giving the pitch and timestamp of all notes clearly, but it isn't designed to specifically encourage chorale-specific training. Bach's chorales are harmonisations produced as a sum of four individual lines of music, known as the voices soprano (S), alto (A), tenor (T) and bass (B). An interesting way to advance this research would be to represent the chorales as shown in Figure 8.2, with a four channel image, with each channel showing a voice. This would enable the DCGAN to learn the lines of music individually, and then relate them to eachother.

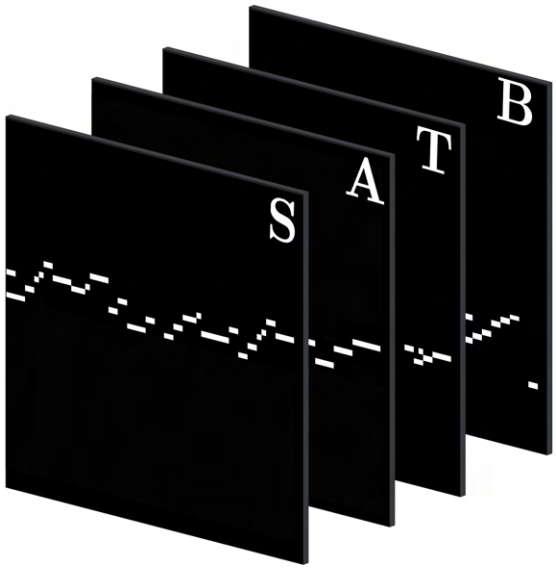


Figure 8.2: An idea of a way for further work to modify the chorale graphical representation, using images with four channels, each representing a harmonic voice in the chorales.

8.2.6 Incorporating Prior Knowledge into the Model

A common issue with ML AMC models is that they will be starting with zero knowledge about music theory or the musical domain, and as such are expected to learn these rules from patterns in the data. Other computational methods such as declarative programming do not have this issue with AMC, such as the Anton model (Boenn et al., 2008), which uses answer set programming to produce both harmonic and melodic composition by defining sets of rules about a certain musical genre. Without going into too much detail about answer-set programming, Anton aims to compose music in the style of 'Renaissance Counterpoint'. As such Boenn et al. establish melodic and harmonic rules including ensuring that the highest

and lowest notes of a melody are not dissonant and limiting the tonal distance (pitch) between harmonic parts to the stylistic norms of renaissance counterpoint. The model successfully generated unique compositions in the style, without the need to characterise a corpus of music.

The existence of both the machine learning-based and declarative programming methods poses the question of whether it would be possible to design a hybrid system between the two. Such a model could use a neural network's ability to produce imitations of a musical dataset with pre-declared musical rules built into the model, to ensure there are less erroneous compositions produced, particularly in the early stages of training. Hu et al. (2016) explore the idea of integrating human knowledge and intentions into deep learning models by developing a neural network architecture that uses iterative distillation to encode first-order logic rules about a problem into the learning process. The network is structured using a "teacher-student" framework. When applied to a sentiment analysis model, where contrastive sense is encoded as a logic rule into the network; this means that the presence of "but" in a sentence indicates that the succeeding statement has more weight in the sentiment of the sentence than the preceding phrase. It is found that a CNN with logic rules outperforms a regular CNN. An interesting extension on this dissertation would be to investigate whether this "teacher-student" CNN architecture could be incorporated into the DCGAN model and subsequently increase its performance. In theory, this may provide higher success in the generator in the earlier stages of training, making the network less reliant on the training data to learn the characteristics of the dataset, and potentially avoid modal collapse as has been seen in this paper's models.

Bibliography

Almeida, D. and Pinho, M., 2018. Music generation using generative adversarial networks.

Ayari, R., 2020. Generative adversarial networks. *Towards data science*.

Bach, J.S., 1832. *371 vierstimmige choralgesänge*. Breitkopf & Härtel.

Bach, J.S. and Riemenschneider, A., 1941. *371 harmonized chorales: and 69 chorale melodies with figured bass*. Schirmer.

Back, D., 1999. Standard midi-file format spec. 1.1, updated.

Barker, A., 1994. Ptolemy's pythagoreans, archytas, and plato's conception of mathematics. *Phronesis*, 39(2), pp.113–135.

Bisong, E., 2019. *Google colaboratory* [Online], Berkeley, CA: Apress, pp.59–64. Available from: `https://doi.org/10.1007/978-1-4842-4470-8_7`.

Boenn, G., Brain, M., De Vos, M. and Fitch, J., 2008. Anton: Answer set programming in the service of music. *Non-monotonic reasoning*. p.85.

Borji, A., 2019. Pros and cons of gan evaluation measures. *Computer vision and image understanding*, 179, pp.41–65.

Boulanger-Lewandowski, N., Bengio, Y. and Vincent, P., 2012. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arxiv preprint arxiv:1206.6392*.

Briot, J.P., Hadjeres, G. and Pachet, F.D., 2017. Deep learning techniques for music generation–a survey. *arxiv preprint arxiv:1709.01620*.

Brock, A., Donahue, J. and Simonyan, K., 2018. Large scale gan training for high fidelity natural image synthesis. *arxiv preprint arxiv:1809.11096*.

Bump, D., 2014. Available from: `http://sporadic.stanford.edu/Chorales/links.html`.

Charman-Anderson, S., 2015. Ada lovelace: Victorian computing visionary. *Ada user journal*, 36(1).

Chen, C.C. and Miikkulainen, R., 2001. Creating melodies with evolving recurrent

neural networks. *Ijcnn'01. international joint conference on neural networks. proceedings (cat. no. 01ch37222)*. IEEE, vol. 3, pp.2241–2246.

Ciaburro, G. and Venkateswaran, B., 2017. Feed-forward and feedback networks. *Neural networks in r*. Packt, p.15.

Cuthbert, M.S. and Ariza, C., 2010. music21: A toolkit for computer-aided musicology and symbolic music data. *International society for music information retrieval*.

Dieleman, S. and Schrauwen, B., 2014. End-to-end learning for music audio. *2014 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE, pp.6964–6968.

Donahue, C., McAuley, J. and Puckette, M., 2018. Adversarial audio synthesis. *arxiv preprint arxiv:1802.04208*.

Dong, H.W., Hsiao, W.Y., Yang, L.C. and Yang, Y.H., 2018. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. *Proceedings of the aaai conference on artificial intelligence*. vol. 32.

Dsouza, J., 2020. What is a gpu and do you need one in deep learning? *Towards data science*.

Ebcioğlu, K., 1988. An expert system for harmonizing four-part chorales. *Computer music journal*, 12(3), pp.43–51.

Eck, D. and Schmidhuber, J., 2002. A first look at music composition using LSTM recurrent neural networks. *Istituto dalle molle di studi sull intelligenza artificiale*, 103, p.48.

Fernandez, J.D. and Vico, F., 2013. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of artificial intelligence research*, 48, pp.513–582.

Galluzzo, E., 2020. Working in bars: Generating music through deep learning. Available from: `https://centricconsulting.com/blog/working-in-bars-generating-music-through-deep-learning/` [Accessed 18 July 2021].

Gatti, M., 2021. MIDI to image conversion. `https://github.com/mathigatti/midi2img`.

Ghosh, B., Dutta, I.K., Carlson, A., Totaro, M. and Bayoumi, M., 2020. An empirical analysis of generative adversarial network training times with varying batch sizes. *2020 11th ieee annual ubiquitous computing, electronics & mobile communication conference (uemcon)*. IEEE, pp.0643–0648.

Gomez, R., 2018. Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing names. *Gombru*.

Goodfellow, I., 2016. Nips 2016 tutorial: Generative adversarial networks. *arxiv preprint arxiv:1701.00160*.

Goodfellow, I., Bengio, Y., Courville, A. and Bengio, Y., 2016. *Deep learning*, vol. 1. MIT press Cambridge.

Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial networks. *arxiv preprint arxiv:1406.2661*.

Gustafsson, A. and Linberg, J., 2021. Investigation of generative adversarial network training : The effect of hyperparameters on training time and stability.

Hadjeres, G., Pachet, F. and Nielsen, F., 2017. Deepbach: a steerable model for bach chorales generation. *International conference on machine learning*. PMLR, pp.1362–1371.

Har-Peled, S. et al., 2002. New similarity measures between polylines with applications to morphing and polygon sweeping. *Discrete & computational geometry*, 28(4), pp.535–569.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B. and Hochreiter, S., 2017. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30.

Hewlett, W.B., 1997. The musedata representation of musical information [Online]. Available from: `http://www.ccarh.org/publications/books/beyondmidi/online/musedata/` [Accessed 16 March 2021].

Hild, H., Feulner, J. and Menzel, W., 1992. Harmonet: A neural net for harmonizing chorales in the style of js bach. *Advances in neural information processing systems*. pp.267–274.

Hiller Jr, L.A. and Isaacson, L.M., 1957. Musical composition with a high speed digital computer. *Audio engineering society convention 9*. Audio Engineering Society.

Hu, Z., Ma, X., Liu, Z., Hovy, E. and Xing, E., 2016. Harnessing deep neural networks with logic rules. *arxiv preprint arxiv:1603.06318*.

Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*. PMLR, pp.448–456.

Johnson, D.D., 2017. Generating polyphonic music using tied parallel networks. *International conference on evolutionary and biologically inspired music and art*. Springer, pp.128–143.

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arxiv preprint arxiv:1412.6980*.

Kumar, P.R. and Manash, E., 2019. Deep learning: a branch of machine learning. *Journal of physics: Conference series*. IOP Publishing, vol. 1228, p.012045.

Leach, J. and Fitch, J., 1995. Nature, music, and algorithmic composition. *Computer music journal*, 19(2), pp.23–33.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. *Proceedings of the ieee*, 86(11), pp.2278–2324.

LeCun, Y. and Cortes, C., 2010. MNIST handwritten digit database.

Liang, F.T., Gotham, M., Johnson, M. and Shotton, J., 2017. Automatic stylistic composition of bach chorales with deep LSTM. *Ismir*. pp.449–456.

Lichtenwalter, R., 2009. Applying learning algorithms to music generation. *Iicai*. Citeseer, pp.483–502.

Liu, G., Schneider, L.F. and Yuan, S., n.d. Pianogan-generating piano music from magnitude matrices (generative modeling). *Stanford university*.

Liu, I., Ramakrishnan, B. et al., 2014. Bach in 2014: Music composition with recurrent neural network. *arxiv preprint arxiv:1412.3191*.

Luhaniwal, V., 2019. Forward propagation in neural networks. *Towards data science*.

Marolt, M., Kavcic, A. and Privosnik, M., 2002. Neural networks for note onset detection in piano music. *Proceedings of the 2002 international computer music conference*.

Marshall, R.L., 1970. How js bach composed four-part chorales. *The musical quarterly*, 56(2), pp.198–220.

Metz, L., Poole, B., Pfau, D. and Sohl-Dickstein, J., 2016. Unrolled generative adversarial networks. *arxiv preprint arxiv:1611.02163*.

Mogren, O., 2016. C-RNN-GAN: Continuous recurrent neural networks with adversarial training. *arxiv preprint arxiv:1611.09904*.

Mozer, M.C., 1994. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection science*, 6(2-3), pp.247–280.

MusicXML, 2017. *Musicxml - for developers* [Online]. Available from: `https://www.musicxml.com/for-developers/` [Accessed 27 April 2021].

Nankipu, 2010. Piano roll. Available from: `http://www.bbc.co.uk/ahistoryoftheworld/objects/fl0uHkdfQkOX31MkZwKJqQ` [Accessed 16/08/2021].

Ng, A., 2016. Nuts and bolts of building ai applications using deep learning. *Nips keynote talk*.

Nierhaus, G., 2009. *Algorithmic composition: paradigms of automated music generation*. Springer Science & Business Media.

Odena, A., Dumoulin, V. and Olah, C., 2016. Deconvolution and checkerboard artifacts. *Distill* [Online]. Available from: `https://doi.org/10.23915/distill.00003`.

Oxford, n.d. Chorale harmonisation instructions [Online]. Available from: `https://www.music.ox.ac.uk/assets/Faculty-of-Music-Chorale-harmonisation-instructions.pdf` [Accessed 08 August 2021].

Pankhurst, T., 2017. Choraleguide: harmonising bach chorales [Online]. Available from: `http://choraleguide.com/harmonicdosanddonts.php` [Accessed 08 August 2021].

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S., 2019. PyTorch: An imperative style, high-performance deep learning library [Online]. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, eds. *Advances in neural information processing systems 32*. Curran Associates, Inc., pp.8024–8035. Available from: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Radford, A., Metz, L. and Chintala, S., 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arxiv preprint arxiv:1511.06434*.

Robinson, R., 2017. Generative adversarial network (gan) in tensorflow [Online]. Available from: `https://mlnotebook.github.io/post/GAN4/#train`.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J., 1986. Learning representations by back-propagating errors. *nature*, 323(6088), pp.533–536.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A. and Chen, X., 2016. Improved techniques for training gans. *Advances in neural information processing systems*, 29, pp.2234–2242.

Scaringella, N., Zoia, G. and Mlynek, D., 2006. Automatic genre classification of music content: a survey. *Ieee signal processing magazine*, 23(2), pp.133–141.

Schmidt-Jones, C., 2012. *The basic elements of music*. Connexions.

Seitzer, M., 2020. pytorch-fid: FID Score for PyTorch. `https://github.com/mseitzer/pytorch-fid`. Version 0.1.1.

Sherman, B.D., 2000. Bach's notation of tempo and early music performance:

Some reconsiderations. *Early music* [Online], 28(3), pp.455–466. Available from: `http://www.jstor.org/stable/3519061`.

Simoni, M., 2003. *Algorithmic composition: a gentle introduction to music composition using common lisp and common music*. Michigan Publishing, University of Michigan Library.

Springenberg, J.T., Dosovitskiy, A., Brox, T. and Riedmiller, M., 2014. Striving for simplicity: The all convolutional net. *arxiv preprint arxiv:1412.6806*.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2015. Rethinking the inception architecture for computer vision. *Corr* [Online], abs/1512.00567. `1512.00567`, Available from: `http://arxiv.org/abs/1512.00567`.

Taylor, R., 1837. *Scientific memoirs: Selected from the transactions of foreign academics of science and from foreign journals*. Taylor.

The University New South Wales, 2010. *Note names, midi numbers and frequencies* [Online]. Available from: `https://newt.phys.unsw.edu.au/jw/notes.html` [Accessed 27 April 2021].

Todd, P.M., 1989. A connectionist approach to algorithmic composition. *Computer music journal*, 13(4), pp.27–43.

Watson, M., 2018. Musescore. *Journal of the musical arts in africa*, 15(1-2), pp.143–147.

WEF, W.E.F., 2020. The future of jobs report 2020. World Economic Forum Geneva.

Yang, L.C., Chou, S.Y. and Yang, Y.H., 2017. Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *arxiv preprint arxiv:1703.10847*.

Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X. and Metaxas, D.N., 2017. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. *Proceedings of the ieee international conference on computer vision*. pp.5907–5915.

Zámečník, B., 2016. midi2audio [Online]. Available from: `https://github.com/bzamecnik/midi2audio` [Accessed 10/07/2021].

Appendix A

Code Listings

A.1 Dataset & Pre-Processing

A.1.1 Transposition

```
parallels = ['CMCm','C#MC#m','DMDm','D#MD#m','EMEm','FMFm','F#MF#m','GMGm','G#MG#m','AMAm','A#MA#m','BMBm']
relatives = ['CMAm','C#MA#m','DMBm','D#MCm','EMC#m','FMDm','F#MD#m','GMEm','G#MFm','AMF#m','A#MGm','BMG#m']

def transpose_midi(key_tp,mode):

    in_dir = 'JSB-Midi'
    out_dir = 'JSB-Midi-TP/{}/JSB-Midi-TP-{}'.format(mode,key_tp)
    if not os.path.exists('JSB-Midi-TP'): os.makedirs('JSB-Midi-TP')
    if not os.path.exists(out_dir): os.makedirs(out_dir)

    major_key = key_tp[:2]    if '#' in key_tp[:2]  else key_tp[0]
    minor_key = key_tp[-3:-1] if '#' in key_tp[-2:] else key_tp[-2]
    tp_keys = {'major':major_key,'minor':minor_key}

    print('Transposing to {} Major and {} Minor'.format(major_key,minor_key))

    # Loop thorugh all MIDI files
    for file in os.listdir(in_dir):
        # Read MIDI file with music21
        score = converter.parse(os.path.join(in_dir,file))

        # Extract the key signature of the music and find the transposition interval
        key = score.analyze('key')
        i = interval.Interval(key.tonic, pitch.Pitch(tp_keys[key.mode]))

        # Write the transposed MIDI file, and validate correct transposition
        newscore = score.transpose(i)
        newkey = newscore.analyze('key')
        assert newkey.upper().split(' ') == [tp_keys[key.mode],key.mode.upper()]

        # Save the transposed MIDI file
        newscore.write('midi',os.path.join(out_dir,'tp_'+file)

for k in parallels:
    transpose_midi(k,'Parallel')
for k in relatives:
    transpose_midi(k,'Parallel')
```

Figure A.1: Code showing how music is transposed to relative and parallel keys for use in the deep learning models, using the Python `music21` library. This is used for the augmentation of the dataset to all 12 keys.

A.1.2 Quantisation and Generating Images

```
def array2image(midi_path,image_width,monophonic,tp,mode,q,square=True,save=True):

    # Load Arrays
    with open(os.path.join('ProcessedData',mode,tp,'full_chorales_{}.json'.format(tp)),'r') as fn:
        data = json.load(fn)[midi_path]

    min_pitch, max_pitch = 26, 90 # D1, F#6 in MIDI numbers
    image_arr = np.zeros((max_pitch-min_pitch,image_width)) # Empty array to be filled where notes occur
    out_dir = os.path.join('ProcessedData',mode,tp,'Images',str(image_width))

    for voice, values in data.items():
        for i,note in enumerate(values["pitch"]): # Loop for all occuring pitches

            # Normalising by q and converting start and duration of notes to int quantises them to chosen level
            dur = int(values["dur"][i]/q)
            start = int(values["start"][i]/q)

            # For each pitch that occurs in the music, populate all occurences across the image
            if dur+start < image_width:
                for j in range(start,start+dur):
                    if j >= 0:
                        image_arr[(max_pitch-min_pitch)-int(note-min_pitch),j] = 255
            else: break

        if save and monophonic:
            # Saving Monophonic Images part-by-part (S, A, T, B)
            out_fn = midi_path.replace(".mid",f"_{voice}.png")
            save_image(image_arr, out_fn, os.path.join(out_dir,'Monophonic',voice))
            image_arr = np.zeros((max_pitch-min_pitch,image_width))

    if save and not monophonic:
        # Saving Polyphonic Images with All Parts
        out_fn = midi_path.replace(".mid",".png")
        plt.imshow(image_arr.astype(np.uint8),cmap='gray')
        plt.show()
        sys.exit()
        save_image(image_arr, out_fn, os.path.join(out_dir,'Polyphonic'))

with open(os.path.join('ProcessedData','Relative','CMAm','full_chorales_CMAm.json'),'r') as fn:
    files = json.load(fn).keys()

q = 0.5             # Quantisation Level
img_width = int(64) # Width of Image
for file in files:
    print(file,end=' ')
    print('Relatives...',end=' ')
    for k in relatives:
        array2image(file,img_width,False,k,'Relative',q)
        array2image(file,img_width,True,k,'Relative',q)
    print('Parallels...')
    for k in parallels:
        array2image(file,img_width,False,k,'Parallel',q)
        array2image(file,img_width,True,k,'Parallel',q)
```

Figure A.2: Code used to generate the images which used in training the deep learning models. The dimensions and resolution of the image are controlled by the parameters `image_width` and `q` respectively, and the image is generated using the `imageio` library in binary grayscale. Methodology is inspired by `https://github.com/mathigatti/midi2img`.

A.2 DCGAN Model

A.2.1 Pre-Processing

```
def import_data(fn,params):
    """
    Loads the dataset and applies proproccesing steps to it.
    Returns a PyTorch DataLoader.
    """
    dataset = ImageFolder(root=fn,
                          transform=transforms.Compose([
                                    transforms.Grayscale(num_output_channels=1),
                                    transforms.ToTensor(),#])),
                                    transforms.Normalize((0.5), (0.5)),]))

    dataloader = DataLoader(dataset,
                            batch_size=params['batch_size'],
                            shuffle=True)

    return dataloader
```

Figure A.3: Code used to import the images produced in Figure A.2, apply pre-processing steps to it, and form into batches.

A.2.2 Generator Network

```
class Generator64(nn.Module):
    def __init__(self, params):
        super(Generator64, self).__init__()
        layers = []

        # 1 x 1 x 100 - Convolutional Transpose Layer & ReLU Activation with Batch Norm
        layers.append(nn.ConvTranspose2d(100, 512, params.kernel_size, stride=1, padding=0, bias=False))
        layers.append(nn.BatchNorm2d(512))
        layers.append(nn.ReLU(True))

        # 4 x 4 x 512 - Strided Convolutional Transpose Layer & ReLU Activation with Batch Norm
        layers.append(nn.ConvTranspose2d(512, 256, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(256))
        layers.append(nn.ReLU(True))

        # 8 x 8 x 256 - Strided Convolutional Transpose Layer & ReLU Activation with Batch Norm
        layers.append(nn.ConvTranspose2d(256, 128, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(128))
        layers.append(nn.ReLU(True))

        # 16 x 16 x 128 - Strided Convolutional Transpose Layer & ReLU Activation with Batch Norm
        layers.append(nn.ConvTranspose2d( 128, 64, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(64))
        layers.append(nn.ReLU(True))

        # 32 x 32 x 64 - Strided Convolutional Transpose Layer & tanh Activation
        layers.append(nn.ConvTranspose2d( 64, 1, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.Tanh())

        # 64 x 64 x 1 - Output
        self.main = nn.Sequential(*layers)

    def forward(self, input):
        return self.main(input)
```

Figure A.4: Implementation of the generator network in PyTorch using `ConvTranspose2d`, `BatchNorm2d`, `ReLU` and `tanh`, wrapped in a `Sequential` container.

A.2.3 Discriminator Network

```
class Discriminator64(nn.Module):
    def __init__(self, params):
        super(Discriminator64, self).__init__()
        layers = []

        # 64 x 64 x 1 - Convolutional Layer & ReLU Activation with Batch Norm4
        layers.append(nn.Conv2d(1, 64, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.LeakyReLU(params.leaky_alpha, inplace=True))

        # 32 x 32 x 64 - Convolutional Layer & ReLU Activation with Batch Norm
        layers.append(nn.Conv2d(64, 128, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(128))
        layers.append(nn.LeakyReLU(params.leaky_alpha, inplace=True))

        # 16 x 16 x 128 - Convolutional Layer & ReLU Activation with Batch Norm
        layers.append(nn.Conv2d(128, 256, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(256))
        layers.append(nn.LeakyReLU(params.leaky_alpha, inplace=True))

        # 8 x 8 x 256 - Convolutional Layer & ReLU Activation with Batch Norm
        layers.append(nn.Conv2d(256, 512, params.kernel_size, stride=2, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(512))
        layers.append(nn.LeakyReLU(params.leaky_alpha, inplace=True))

        # 4 x 4 x 512 - Convolutional Layer & Sigmoid Activation
        layers.append(nn.Conv2d(512, 1, params.kernel_size, stride=1, padding=0, bias=False))

        layers.append(nn.Sigmoid())
        # 1 x 1 - Output

        self.main = nn.Sequential(*layers)

    def forward(self, input):
        return self.main(input)
```

Figure A.5: Implementation of the generator network in PyTorch using `Conv2d`, `BatchNorm2d`, `LeakyReLU` and `Sigmoid`, wrapped in a `Sequential` container.

A.2.4 Training Process

```
# Parameters Dictionary:
# 'input_data'      Reference to the Dataset used
# 'batch_size'      Batch size during training
# 'kernel_size'     Kernel size of the convolutional kernel
# 'G_learning_rate' Size of learning rate in generator training
# 'G_learning_rate' Size of learning rate in discriminator training
# 'beta1'           Beta1 hyperparam for Adam optimizers
# 'beta2'           Beta2 hyperparam for Adam optimizers
# 'leaky_alpha'     Size of α parameter in LeakyReLU activation
# 'real_label'      Image label that the real images will be trained to
# 'fake_label'      Image label that the fake images will be trained to
# 'gen_updates'     Number of times generator is updated per iteration
# 'loss'            Loss function to be used in training

params = {'input_data'      : '12keys/Polyphonic',
          'batch_size'      : 128,
          'kernel_size'     : 4,
          'G_learning_rate' : 0.0002,
          'D_learning_rate' : 0.0002,
          'beta1'           : 0.5,
          'beta2'           : 0.9,
          'leaky_alpha'     : 0.2,
          'real_label'      : 1,
          'fake_label'      : 0,
          'loss'            : 'BCE',
          'gen_updates'     : 1}
```

Figure A.6: Parameters used in the code are stored in a dictionary, and each of the parameters sets used in experimentation are saved to a .pkl file using `pickle`. This assists with the ease of playing with parameters, especially early in the model designs.

```
class TrainDCGAN():

    def __init__(self,dataloader,device,params,output_dir):

        self.dataloader = dataloader
        self.device = device
        self.params = params
        self.output_dir = output_dir

        # Create the generator and discriminator and initialize
        # all weights to mean=0.0, stddev=0.2
        self.G = Generator64(params).to(device)
        self.D = Discriminator64(params).to(device)
        self.G.apply(weights_init)
        self.D.apply(weights_init)

        # Create batch of random noise vectors (z) that is used to
        # visualise the progression of the generator
        self.fixed_noise = torch.randn(64, 100, 1, 1, device=device)

        # Choosing loss function
        loss_functions = {'BCE':nn.BCELoss(),'LS':LSLoss}
        self.lossfunction = loss_functions[params.loss]
        print('Loss Function:',self.lossfunction)

        # Setup Adam optimizers for both G and D
        self.D_Adam = optim.Adam(self.D.parameters(),
                                 lr=params.D_learning_rate, betas=(params.beta1, params.beta2))
        self.G_Adam = optim.Adam(self.G.parameters(),
                                 lr=params.G_learning_rate, betas=(params.beta1, params.beta2))

        # Import pre-trained Frechet Inception Distance Model
        FIDModel = FrechetInceptionDistance.get_inception_model()
        self.FIDModel = FIDModel.to(device)
```

Figure A.7: Weights, network classes, fixed noise and Adam optimiser are all initialised in teh `__init__` method of the `Train_Model` class.

```
###### Training Discriminator: Maximise log(D(x)) + log(1-D(G(z)))
# Resetting discriminator gradients to zero
self.D.zero_grad()

# Assigning the real images the real labels
real_batch = data[0].to(self.device)
labels = torch.full((real_batch.shape[0],), self.params.real_label,
                    dtype=torch.float, device=self.device)

# Generate labels for real images with discriminator
output = self.D(real_batch).view(-1)

# Calculate discriminator loss on real images using BCE Loss
Dloss_real = self.lossfunction(output, labels)
Dacc_real.append((output == labels).float().sum() / output.shape[0])

# Back Propagate through Discriminator to Calculate Gradients
Dloss_real.backward()
D_x = output.mean().item()

# Generate Fake Images Using Noise Vector as Input to Generator
noise = torch.randn(real_batch.shape[0], 100, 1, 1, device=self.device)
fake = self.G(noise)

# Use Fake Labels for the Fake Images
labels = torch.full((real_batch.shape[0],), self.params.fake_label,
                    dtype=torch.float, device=self.device)

# Generate labels for real images with discriminator
output = self.D(fake.detach()).view(-1)

# Calculate discriminator loss on fake images using BCE Loss
Dloss_fake = self.lossfunction(output, labels)
Dacc_fake.append((output == labels).float().sum() / output.shape[0])

# Back Propagate through Discriminator to Calculate Gradients
Dloss_fake.backward()
D_G_z_disc = output.mean().item() # D(G(z)) for first pass in loop

# Total discriminator loss is sum over real and fake images
Dloss = Dloss_real + Dloss_fake

# Update Discriminator Neural Weights using Adam Optimiser
self.D_Adam.step()
```

Figure A.8: Implementation in PyTorch of the discriminator training at each iteration in each epoch.

```
###### Training Generator: minimise log(1-D(G(z)))
# Resetting generator gradients to zero
self.G.zero_grad()

# Use real labels for the fake images when training generator
# Generator tries to fool the discriminator...
labels = torch.full((real_batch.shape[0],), self.params.real_label,
                    dtype=torch.float, device=self.device)

# Generate labels for fake images again
output = self.D(fake).view(-1)
D_G_z_gen = output.mean().item() # D(G(z)) for second time

# Calculate Generator Loss
# How well it can cause discriminator to generate wrong labels
Gloss = self.lossfunction(output, labels)

# Choosing how many times to backpropagate the generator weights each iteration
for g_loop in range(self.params.gen_updates):
    # Back Propagate Through Generator to Calculate Gradients
    Gloss.backward(retain_graph=True)

# Update Generator Neural Weights using Adam Optimiser
self.G_Adam.step()
```

Figure A.9: Implementation in PyTorch of the generator training at each iteration in each epoch.

A.3 Model Output Post-Processing (Image to .wav file)

```
def image2audio(image_path,min_pitch,max_pitch, play=False):


    # Read the image from file...
    with Image.open(image_path) as image:
        im_arr = np.fromstring(image.convert('L').tobytes(), dtype=np.uint8)
        im_arr = im_arr.reshape((image.size[1], image.size[0]))

    # Produces a dictionary of {pitch:duration_left} for all notes in column
    notes_tm1 = updateNotes(im_arr[0,:],{})

    # Iterate through the image from left to right column by column
    time_t = 0
    output_notes = []
    for column in im_arr.T[1:,:]:

        # Convert the notes in this column to MIDI numbers according to position
        notes_t = column2notes(column,min_pitch,max_pitch)
        pitches_tm1 = notes_tm1.keys()

        # Loop through the notes from the previous timestep
        for pitch in pitches_tm1:
            if not pitch in notes_t:

                # If it is a new note, first convert to music21 'Note' dtype
                new_note = note.Note(pitch,quarterLength=notes_tm1[pitch])
                new_note.storedInstrument = instrument.Piano()

                # Note is added to the list of notes in the music
                if time_t - notes_tm1[pitch] >= 0:
                    new_note.offset = time_t - notes_tm1[pitch]
                    output_notes.append(new_note)
                else:
                    new_note.offset = time_t
                    output_notes.append(new_note)

        # Add a time step to all of the note durations
        # So that they can be used in the next column
        notes_tm1 = updateNotes(notes_t,notes_tm1)

        # Resolution is the quantisation level
        # Add resolution to time_t to move to the next column
        time_t += resolution

    # output_notes: List of music21.note.Note items which contain
    #               information about the pitch, time and duration
    #               of all notes read from the image.

    # Convert output_notes to music21 Stream, and add metronome mark (80 BPM)
    midi_stream = stream.Stream(output_notes)
    mm1 = tempo.MetronomeMark(number=80)
    midi_stream.insert(0,mm1)

    # Save the MIDI file to path...
    path_list = image_path.split("/")
    midi_path = os.path.join(reduce(os.path.join,path_list[:-1]),
                             path_list[-1].replace(".png","_PP.mid"))
    midi_stream.write('midi', fp=midi_path)

    # Convert the previously saved MIDI file into .wav sound file
    # midi2audio and FluidSynth
    fs = FluidSynth('../Data/MuseScore_General.sf2')
    fs.midi_to_audio(midi_path, midi_path.replace('.mid','.wav'))
    if play: fs.play_midi(midi_path)
```

Figure A.10: Converting the images back to MIDI files and audio for analysis and to enable playback. The `midi2audio`, `FluidSynth` and `music21` libraries are used. Methodology is inspired by `https://github.com/mathigatti/midi2img`.

Appendix B

Deep Learning Theory

B.1 Adam Optimiser

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Figure B.1: Pseudo code explaining the Adam optimisation algorithm, showing the role played by parameters β_1, β_2 and α. *Figure taken from Kingma and Ba (2014).*

Appendix C

Musical Data Representation

C.1 Piano Roll

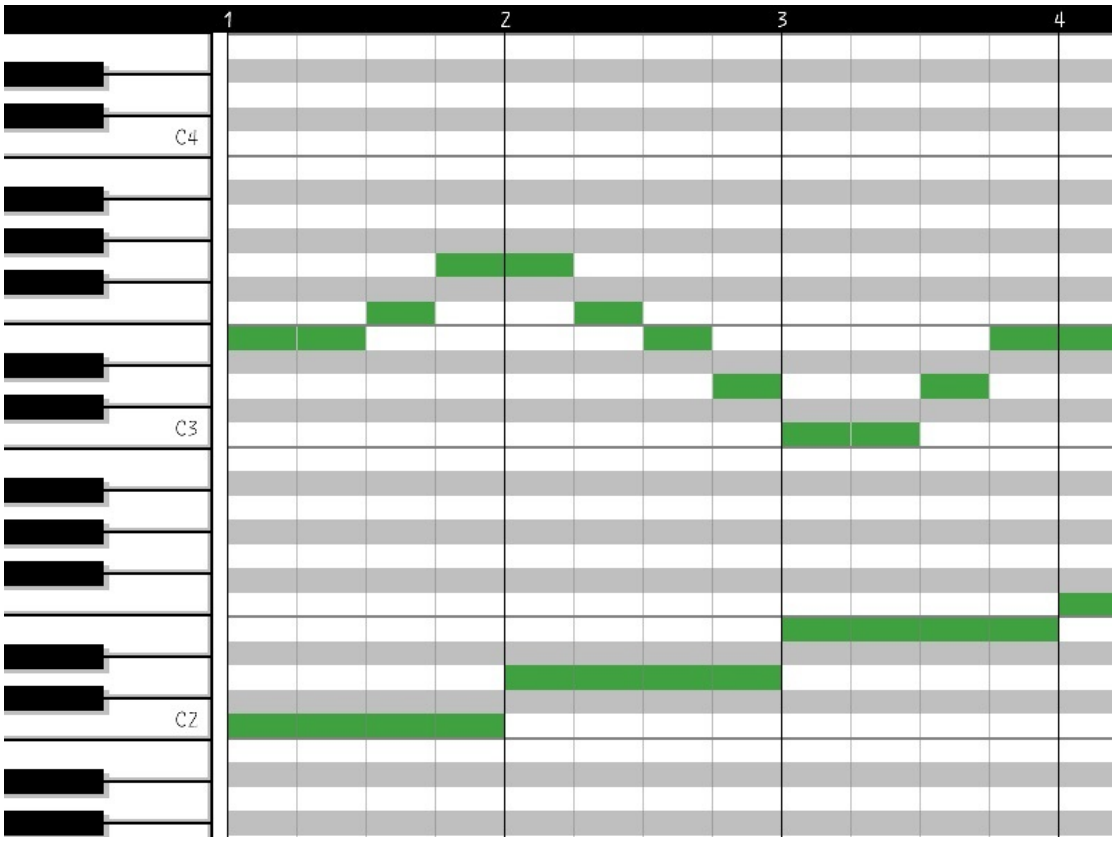
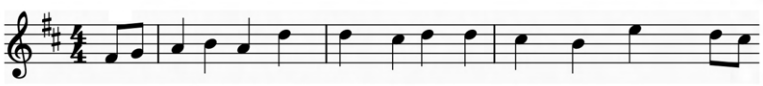


Figure C.1: Example of the piano roll musical representation used as the basis for MIDI files and in many music production interfaces such as LogicPro. *Figure taken from Briot, Hadjeres and Pachet (2017).*

C.2 MIDI Files



```
Chorale349.mid
D major
F-sharp in octave 4 Eighth Note
G in octave 4 Eighth Note
<music21.meter.TimeSignature 4/4>
A in octave 4 Quarter Note
B in octave 4 Quarter Note
A in octave 4 Quarter Note
D in octave 5 Quarter Note
D in octave 5 Quarter Note
C-sharp in octave 5 Quarter Note
D in octave 5 Quarter Note
D in octave 5 Quarter Note
C-sharp in octave 5 Quarter Note
B in octave 4 Quarter Note
E in octave 5 Quarter Note
D in octave 5 Eighth Note
C-sharp in octave 5 Eighth Note
```

Figure C.2: An example of the soprano voices of one of the chorales in the dataset as acquired in MuseScore file format, and once converted to MIDI and parsed using `music21`'s MIDI parsing functionality into a 'stream' data structure. Information about each musical note is displayed using the `fullName` function.

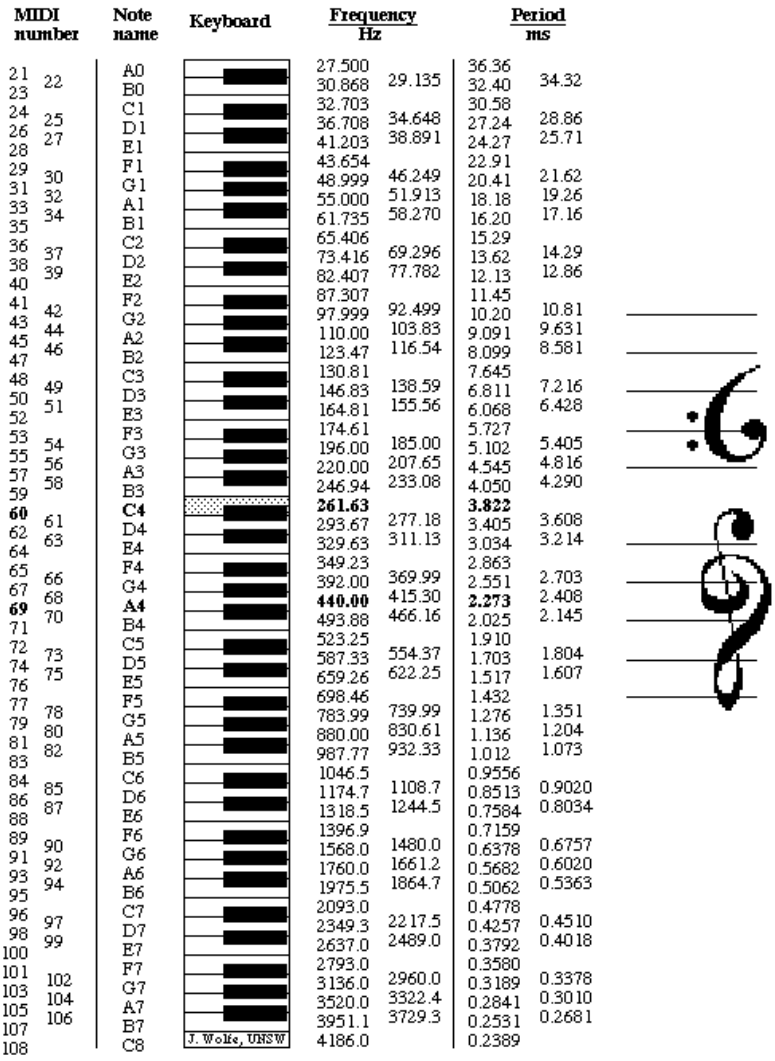


Figure C.3: In MIDI files, notes are encoded as 'MIDI numbers'. This diagram, obtained from The University New South Wales (2010), explains tbe relationship between note name, MIDI number, and frequency of the sound produced. For note name, the naming convention is 'name of note' followed by 'octave', so 'C4' refers to the note of C in the 4^{th} octave (261.63 Hz).

Appendix D

Ethics Checklist

Department of Computer Science
12-Point Ethics Checklist for UG and MSc Projects

Student Sean Hill

Academic Year or Project Title Automated Music Composition using GANs

Supervisor Marina De Vos

Does your project involve people for the collection of data other than you and your supervisor(s)? NO

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Will you prepare a Participant Information Sheet for volunteers?* N/A
   This means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* N/A
   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Will there be any intentional deception of the participants?* N/A
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* N/A
   The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. *Will participants voluntarily give informed consent?* N/A

Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete. If your data collection is entirely anonymous and does not include collection of personal data you do not need to collect a signature. Instead, you should include a checkbox, which must be checked by the participant to indicate that informed consent has been given.

6. *Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?* N/A
Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. *Will you be offering any incentive to the participants?* N/A
The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. *Will you be in a position of authority or influence over any of your participants?* N/A
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. *Will any of your participants be under the age of 16?* N/A
Parental consent is required for participants under the age of 16.

10. *Will any of your participants have an impairment that will limit Their understanding or communication?* N/A
Additional consent is required for participants with impairments.

11. *Will the participants be informed of your contact details?* N/A
All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. *Will you have a data management plan for all recorded data?* N/A
Personal data is anything which could be used to identify a person, or which can be related to an identifiable person. All personal data (hard copy and/or soft copy) should be anonymized (with the exception of consent forms) and stored securely on university servers (not the cloud).