CHORAL MUSIC GENERATION:

A DEEP HYBRID LEARNING APPROACH

by

Daniel James Szelogowski

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy in Artificial Intelligence

Capitol Technology University

March 2024

CHORAL MUSIC GENERATION:

A DEEP HYBRID LEARNING APPROACH

Approved:

Dr. Susan Goodwin, Chair

Dr. Sophia Noir, External Examiner

Dr. Paul Bodily, Committee Member

Dr. Ian McAndrew, Dean of Doctoral Programs

Accepted and Signed:

_____

Susan Goodwin, Ph.D. Date
Chair
Capitol Technology University

_____

Sophia Noir, Ph.D. Date
External Examiner

_____

Paul Bodily, Ph.D. Date
Committee Member
Idaho State University

_____

Ian McAndrew, Ph.D. Date
Dean, Doctoral Programs
Capitol Technology University

ABSTRACT

Despite advancements in AI and machine learning, the creation of AI-generated music that captures the complexity and nuance of classical choral arrangements has remained largely unexplored. This gap is significant, considering the intricate compositional techniques and sophisticated music theory knowledge required for classical choral music. My research is motivated by the quest to bridge this gap, aiming to develop an AI model capable of producing realistic choral compositions that adhere to the rich traditions of classical music. Additionally, I explore how AI-generated music is perceived across different listener groups, contributing insights into the intersection of AI and human perception in the arts. This study seeks to address three primary research questions: the feasibility of current machine learning architectures in generating SATB choral music, the ability of a hybrid AI model to produce compositions indistinguishable from human-created music by the general public, and the varying perception of AI-generated classical music among individuals with varying musical backgrounds. This dissertation presents the "Choral-GTN" system: a novel architecture combining a Generative Transformer Network with a rule-based post-processing system, alongside the curated "CHORAL" dataset to address the challenge of generating realistic four-part (SATB) choral music. My comprehensive approach integrates advanced deep learning techniques with sophisticated musical theory insights to produce compositions that closely mimic those created by human composers. Through a meticulously designed survey and rigorous statistical analysis, I evaluate the realism and indistinguishability of my AI-generated music across varied listener groups. The findings demonstrate that this model successfully created music that was largely indistinguishable from human compositions, achieving a significant milestone in AI-assisted music composition and music perception.

Keywords: Artificial Intelligence, Transformer Networks, Music Generation, Choral Music, Music Perception, Music Theory, Deep Learning, Rule-Based Post-Processing, Hybrid AI

DEDICATION

I dedicate this dissertation to my family, friends, educators, and students, whose unremitting support has motivated and encouraged me to pursue such a meaningful education.

Thank you, truly, for everything you do and the inspiration you provide.

ACKNOWLEDGEMENTS

I would first like to thank my supervisor Dr. Susan Goodwin, who has been exactly the advisor I needed – one who encourages, inspires, motivates, and reassures every step of the way. Your guidance and expertise provided me with the support and opportunities I needed throughout this journey to continuously work toward success. Finally, your faith in my ability to complete such a rigorous body of work has been truly motivating; I have done my part to the best of my capability.

I owe additional gratitude to my subject matter expert Dr. Paul Bodily of Idaho State University. Your interest in my research passions, willingness to advise as necessary, and additional expertise was beyond helpful as I began my research. I am exceedingly thankful for the research methodologies you shared with me in beginning the literature review, and your knowledge of various generative music architectures laid the foundation for my entire project.

I would also like to thank Dr. Lopamudra Mukherjee, my graduate and thesis advisor during my time at the University of Wisconsin-Whitewater, who introduced me to machine learning as I began my master's coursework in my senior year of undergrad. Your diligence, attention to detail, and expert knowledge prepared me with the confidence to tackle any project, along with the skills needed to succeed. Thank you again for all your support, especially in publishing my first conference paper following the completion of my thesis.

I would like to thank all my music and computer science professors from the University of Wisconsin-Whitewater – especially Drs. Jiazhen Zhou, Hien Nguyen, Zach Oster, Benjamin Whitcomb, and Robert Gehrenbeck – who taught me nearly everything I know about music analysis, choral music, and scientific writing in computer science. The many classes I had with each of you were some of the most enjoyable I have ever studied and prepared me with the prerequisite knowledge necessary to be successful in my field and research especially.

I would like to especially thank Mr. Robert Getka, who not only sparked my passion for computer science, but has continued to support and inspire me during my time as a teacher. Not only that, but you also still teach me new things and new methods of teaching to this day – skills I hold dearly among those I gained from the many classes with you during my high school career.

Most importantly, I would like to thank all the people closest to me, who have raised, educated, and encouraged me throughout my life – my family, especially my parents David and Diane, and two of my best friends for life, Avery Tomlin and Casey Sharp. You all have provided me with so much support, guidance, and motivation to continue learning every day, alongside my students who have been graciously supportive as they spectate my academic journey as well. Finally – to another of my best friends – I would like to thank Joseph Clancy, for being the one to truly push me to pursue a college education. I hope I have made all of you proud.

"So long as the human spirit thrives on this planet, music in some living form will accompany and sustain it and give it expressive meaning."

–Aaron Copland

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
CA	Cellular Automata
CNN	Convolutional Neural Network
CSV	Comma-Separated Values
DAW	Digital Audio Workstation
DL	Deep Learning
DT	Decision Tree
GAN	Generative Adversarial Network
GPT	Generative Pretrained Transformer
GTN	Generative Transformer Network
HMM	Hidden Markov Model
LSTM	Long Short-Term Memory
MC	Markov Chain
MIDI	Musical Instrument Digital Interface
ML	Machine Learning
MLP	Multilayer Perceptron
MRF	Markov Random Field
NLP	Natural Language Processing
QC	Quantum Computing
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SCE	Sparse [Categorical] Cross-Entropy

CHAPTER 1
INTRODUCTION

1.1 Motivation and Problem Statement

A notable gap exists in the field of AI-generated classical music, particularly in the realm of multi-voiced compositions such as choral music. While advancements in AI and Machine Learning have led to significant progress in various forms of music generation (Huang et al., 2019), the creation of realistic, AI-generated classical music involving complex, multi-voice arrangements remains largely unexplored (Liu et al., 2022). This gap is especially evident in the context of generative choral music, where the challenge extends beyond mere note generation to the creation of a complete, harmonically and structurally coherent score. The problem, therefore, lies in the limited attempts and success in using AI to replicate the intricacies and nuances of classical choral compositions (including the sophisticated compositional rules/techniques found in music theory), a genre that demands a high level of musical complexity and understanding.

Current neural network architectures, including the widely acclaimed **Transformers** (see *Chapter 2.1.3.3*), face inherent bottlenecks in generating realistic AI-generated classical music due to their limitations in processing and replicating the long-term structural dependencies and nuanced expressions fundamental to classical compositions (Huang et al., 2019). While transformers excel in pattern recognition and sequence prediction, they often struggle to capture the extended musical narratives and intricate emotional nuances that define classical music, leading to outputs that lack coherence over longer compositions (see *Chapter 6.2*). The complexity

and subtlety of classical music dynamics – such as tempo variations and expressive timing – also pose significant challenges for these architectures, limiting their ability to produce compositions that truly resonate with the depth and richness characteristic of human-created classical music.

The motivation for this research stems from a desire to bridge this gap by developing an AI model capable of generating realistic choral music that adheres to the rich traditions and complex rules of classical composition (see *Appendix C*). This endeavor is driven by the potential of AI to not only mimic human creativity in music composition but to also expand the boundaries of musical expression through technological innovation. As well, this dissertation seeks to better understand how such AI-generated music is perceived by different listener groups. By examining whether listeners (with varying degrees of musical training) can distinguish between AI-generated and human-composed choral music, this dissertation aims to contribute valuable insights into the evolving relationship between AI and human perception in the arts. This exploration is particularly pertinent in an era where AI's role in creative processes is expanding, raising questions about the nature of creativity, originality, and the human experience in art.

In addressing this research gap, this study aims to make a significant contribution to both the field of AI in music generation and the understanding of music perception. It seeks to offer new perspectives, datasets, and tools for composers, musicians, and technologists, opening possibilities for collaborative human-AI creation in music. Additionally, by examining the public's response to AI-generated choral music, this research provides insights into the potential acceptance and integration of AI in classical music – offering a foresight into the future of how music is composed, experienced, and appreciated in a world where technology and art converge. The research presented in this dissertation is thus intended for a reader with a strong background in computer science and statistical analysis, and at least elementary knowledge of music theory.

1.2 Research Questions

In response to the identified gap in the realm of AI-generated classical music, particularly in (multi-voiced) choral compositions, this study is poised to explore several critical questions. These questions are conceived to dissect the core aspects of AI's capability in choral music generation and the perceptual impact of such music on diverse listener groups. This inquiry is driven by the need to understand not only the technical feasibility of AI in accurately emulating the complexity of classical choral music but also to gauge the authenticity and appeal of these AI compositions from the perspective of the audience. This investigation delves into the intricacies of advanced AI architectures for music generation, examining their effectiveness in creating compositions that resonate with the depth and nuance of traditional choral music. Additionally, it seeks to uncover the nuances in the perception of AI-generated music across varied musical backgrounds, aiming to reveal how different levels of musical expertise influence the discernment and appreciation of such compositions. Thus, this study seeks to answer the following questions:

1. Is there a current Machine Learning or Deep Learning architecture that is feasible for the generation of musically realistic four-part (SATB) choral music, and does such a training dataset currently exist?
2. Can a hybrid Artificial Intelligence model generate realistic choral music to the degree that the general public would struggle to differentiate from genuine human compositions?
3. Do individuals of varying musical backgrounds perceive AI-generated classical music differently than non-musicians?

These research questions are integral in navigating the intersection of technology, creativity, and human experience, providing a comprehensive understanding of the potential and limitations of Artificial Intelligence in the domain of classical music composition.

1.3 Purpose of the Study

1.3.1 Goals

The following list represents the goals of this dissertation:

- Explore and establish the effectiveness of machine learning and deep learning architectures in generating musically realistic four-part (SATB) choral music, addressing the technological gap in current AI music generation systems.
- Contribute to the advancement of AI technology in the field of music composition, particularly in creating complex choral pieces that adhere to the intricacies of music theory, thereby bridging the intersection of AI innovation and artistic creativity.
- Deepen the understanding of how AI can be employed as a tool for artistic expression, specifically in the domain of music, and explore its potential for innovation and creative enhancement in the arts (see *Chapter 2.2.9*).

1.3.2 Objectives

The following list represents the objectives of this dissertation:

- Examine prior research in the field through extensive background investigation.
- Establish the feasibility of existing AI architectures in creating realistic SATB choral music and curate a suitable dataset for this purpose.
- Assess the ability of the AI model to generate compositions indistinguishable from human-created music through a comprehensive public survey, thereby gauging its success in mimicking human compositional techniques.
- Explore and analyze how individuals with varying degrees of musical expertise perceive AI-generated classical music, aiming to identify perceptual differences between musicians and non-musicians.

1.3.3 Significance

The significance of this dissertation lies in its pioneering exploration into the realm of AI-generated choral music, addressing a notable gap in the current landscape of music generation technology.[1] This research thus stands at the forefront of investigating and developing a hybrid machine learning/deep learning (and rule-based) architecture capable of producing musically realistic four-part (SATB) choral compositions. As such, the project's success in creating a system that adheres to the nuanced voice leading rules of music theory represents a significant advancement in the field of AI music generation (see *Chapter 6.1.2*). Likewise, this research responds to the critical question of whether such an AI system, supported by an appropriate training dataset, can exist and function effectively (AIContentfy, 2023; Stanek, 2017).

Furthermore, the study ventures beyond technical capability, probing into the perceptual impact of this technology – i.e., **Music Perception and Cognition (MPC)**. By examining whether a hybrid AI model can generate choral music that challenges the discernment of the general public, including both musicians and non-musicians, this research provides invaluable insights into human-AI interaction in the creative domain. I not only assess the feasibility of AI in replicating human-like compositions but also explore the diverse perceptual nuances among different listener groups (see *Chapter 5.3*). This exploration is crucial in understanding the broader implications of AI in the arts, particularly in how AI-generated music is received by varied audiences. Hence, the outcomes of this dissertation have the potential to shape future developments in AI music generation, setting a precedent for future studies and technological advancements in creating AI systems that not only imitate but also innovate in the field of music composition (see *Chapter 7.2*).

[1] It is worth noting that a similar work has recently been published as a preprint during the duration of my study (Zhou et al., 2023), but the methodology and evaluation metrics appear to be tailored toward Bach chorales specifically – rather than more traditional (polyphonic) four-part choral music. As well, it also lacks a corrective post-processing system for voice leading and other compositional errors, leading to less musically realistic results than mine.

1.4 Research Scope and Boundaries

1.4.1 Scope

The scope of this dissertation encompasses the development and evaluation of a hybrid Artificial Intelligence model capable of generating (plausibly) realistic SATB choral music. It involves the curation of a novel dataset – "CHORAL" – consisting of a wide range of classical choral pieces and the subsequent training of a deep learning model using this dataset. The evaluation aspect extends to conducting a comprehensive survey to assess the perception of the AI-generated music by a diverse audience, including both musicians and non-musicians. This study aims to explore the technical feasibility of AI in classical choral music generation and to gain insights into the public's ability to distinguish between AI-generated and human-composed music. This research thus covers both the technological development of AI in music composition and the perceptual aspects of its reception within a varied listener demographic (see *Chapter 6.1.2*).

1.4.2 Limitations

This study's limitations are primarily related to the inherent challenges in AI music generation and the subjective nature of music perception. While this research endeavors to develop a sophisticated AI model, the intricacies of choral music – such as the adherence to Voice Leading rules and the emulation of human-like creativity – present significant challenges (see *Chapter 2.2.9*). Furthermore, the perceptual evaluation of music is inherently subjective, with individual preferences and backgrounds playing a crucial role in how music is received and interpreted (Droit-Volet et al., 2013). While a strength, the diversity of the survey respondents also introduces variability in the data that may affect the generalizability of the findings. Additionally, the study is limited by the scope of the dataset and the technical capabilities of current AI architectures in capturing the full range of musical expressions found in choral compositions.

1.4.3 Delimitations

Delimitations in this study are defined by the intentional choices made in the research design to focus and streamline the investigation. The research is delimited to the genre of classical choral music (roughly in the range of years 1500-1950) and does not encompass other musical genres or forms (see *Appendix B*). The choice of four-part (SATB) choral music as the primary focus is a delimitation that concentrates the study on a specific and traditional form of choral arrangement. Additionally, the study delimits itself to the use of machine learning and deep learning techniques for the generative model, excluding other potential AI or computational methods (e.g., Reinforcement Learning and probabilistic models like Markov Chains) for music generation. The survey-based evaluation is also a delimitation, focusing on perceptual aspects rather than technical or compositional analysis of the AI-generated music (i.e., disregarding harmonic and form analyses typically performed on classical music compositions). These delimitations thus provide a clear and manageable framework for the study, ensuring a focused and in-depth exploration of the research questions within the defined boundaries.

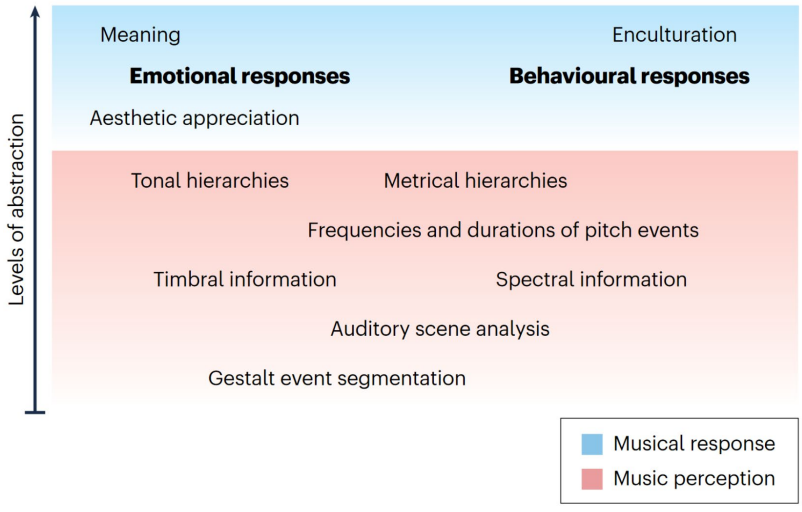


Figure 1.1: Music perception and cognitive responses by abstraction (Singh & Mehr, 2023)

1.5 Structure of the Report

- **Chapter 1: Introduction –** This chapter presents the dissertation's research questions, motivation, goals, objectives, significance, and scope, as well as the structure of the report.
- **Chapter 2: Background and Related Work –** This chapter presents the background of the dissertation, a definition of terms, and an extensive literature review (as of July 2023).
- **Chapter 3: Methodology –** This chapter outlines the methods used to develop the new dataset, system elements, model architecture, research design, and data collection/validity.
- **Chapter 4: Implementation –** This chapter details the implementation of the numerous system components, their constraints, and training data preparation.
- **Chapter 5: Evaluation –** This chapter discusses the system's evaluation metrics to examine the model performance and the analysis of the data collected from the main study.
- **Chapter 6: Discussion –** This chapter provides a discussion on the implications discovered through the observational study experiments and identifies areas of potential improvement.
- **Chapter 7: Conclusion –** This chapter summarizes the dissertation and discusses potential future research to expand on this field.
- **Appendix A: Example Training Data and Attention Plots –** This appendix includes examples of the training data used for the system and other model data visualizations.
- **Appendix B: Compositions Used for Training –** This appendix provides a categorical listing of each piece used in the proposed novel dataset.
- **Appendix C: Related Definitions –** This appendix supplements music composition rules.
- **Appendix D: Survey Design –** This appendix contains the original evaluation survey.
- **Appendix E: Implementation of Choral-GTN System –** This appendix provides the implementation of the proposed deep hybrid learning model and system architecture.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter, I will give an overview of the necessary technical background and relevant theoretical foundations of artificial intelligence/machine learning and music theory (including methods of automatic music composition), as well as provide a comprehensive literature review discussing the historical development of the field and the current state of the art.

2.1 Background

This section will provide a comprehensive overview of the requisite principles and terminology of machine learning and music needed to comprehend the content of this dissertation.

2.1.1 Artificial Intelligence and Machine Learning

The field of **Artificial Intelligence (AI)** represents an effort to simulate or replicate human intelligence in machines through the creation and application of specialized algorithms (Burnham, 2020; Turing, 1950). This replication encompasses activities such as learning, planning, and problem-solving, accomplished by analyzing data to discern patterns and subsequently emulating the desired behaviors. AI is being utilized in numerous industries (e.g., assistive devices, automotive technology, economics, education, healthcare, video games, etc.) to enhance their services and products, though the number of applications continues to grow daily as other fields research its potential (DeepAI, 2019a). Although AI is loosely defined, it is generally classified into two subcategories: "weak" **Artificial Narrow Intelligence (ANI)**, where systems can perform very narrow tasks at human-level competence or beyond (e.g., diagnosing medical conditions); or

"strong" **Artificial General Intelligence (AGI)**, a theoretical form of AI where systems could outperform humans across a wide range of tasks and possibly exhibit consciousness (IBM Cloud Education, 2023).

Machine Learning (ML) is an interdisciplinary branch of computer science and data science, as well as a subfield of AI, based on statistical learning theory (Mohri et al., 2018). It utilizes algorithms and statistical models to analyze and draw inferences from data to enable computers to learn and generate output without the need for explicit programming. ML models typically undergo training on massive amounts of data, allowing them to predict (e.g., regression) and/or classify unseen test data or generate new, realistic data that mirrors the training set – among various other possible functions (IBM Cloud Education, 2020). While AI constitutes a general term for any "smart" technology or intelligent system, ML represents a specific branch of AI that emphasizes information and data processing (DeepAI, 2019c). As such, ML tasks are typically classified into one of five different categories (Nassif et al., 2019; Szelogowski, 2022):

- **Supervised Learning (Task-Driven)** – Classification, regression, similarity learning.
- **Self/Semi-Supervised Learning (Self-Driven)** – Generative networks, self-training, low-density separation, graph-based algorithms.
- **Unsupervised Learning (Data-Driven)** – Dimensionality reduction, clustering.
- **Reinforcement Learning (Experience/Reward-Driven)** – Q-learning, heuristic methods, dynamic programming, Monte Carlo methods, gaming, navigation, decision-making.
- **Deep Learning (Knowledge-Driven)** – Artificial neural networks.

Common ML algorithms include AdaBoost, Apriori, **Decision Trees**, Dimensionality Reduction (e.g., PCA and SVD), Gradient Boosting, k-Means Clustering, k-Nearest Neighbors, Linear and Logistic Regression, Naïve Bayes, Random Forests, and Support Vector Machines (Ray, 2017).

2.1.1.1 Decision Trees and Ensemble Learning

Decision Trees (DTs) are a non-parametric supervised learning methodology utilized for classification and regression tasks, partitioning the input space into distinct output regions (Cravit, 2023). The structure of the model is intrinsically interpretable, similar to a flowchart (see Figure 2.1 (a)) – each **decision node** signifies a specific criterion, each **branch** represents the outcome of a criterion, and each **leaf node** stands for a prediction (i.e., a specific output class or value). DTs are also highly robust, given their insensitivity to missing values, capacity to handle both numerical and categorical data, and prevention of overfitting via pruning strategies. Likewise, they serve as the building blocks of several ensemble methods (e.g., forests) via combination by **boosting** (in sequence) or **bagging** (in parallel). **Random Forests**, for instance, employ a multitude of DTs (see Figure 2.1 (b)) – each trained on a different data subset – the collective decision of which (through majority voting or averaging) leads to more robust and generalizable predictions (E R, 2021). In music generation, DTs can be utilized effectively to post-process the output of a deep learning model, ensuring the generated results adhere to the guidelines of music theory (Szelogowski, 2022). For example, when a neural network generates a new piece, the DT could review and modify its output, correcting any violations of music theory to ensure the final composition maintains both the creativity of the model and the consistency of established musical structures.

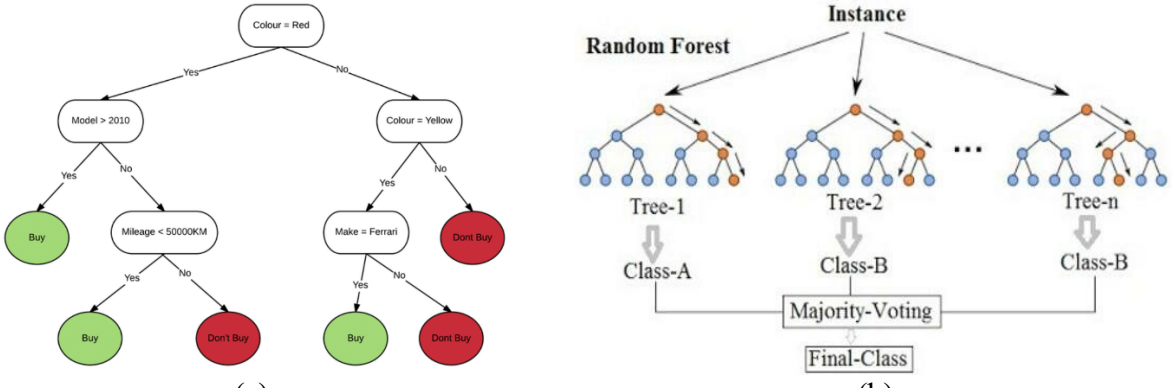


Figure 2.1: Basic decision tree model (a; Kumar, 2022), a random forest (b)

2.1.1.2 Generative Algorithms

Generative machine learning algorithms have been instrumental in the field of music generation, providing a mathematical and computational framework to create novel musical compositions. These algorithms – such as **Hidden Markov Models (HMMs)**, **Bayesian Networks**, and **Markov Random Fields (MRFs)** – operate by learning the underlying distribution of a given music dataset and subsequently generating new music samples from this learned distribution. HMMs are particularly suited to capturing temporal dependencies in music; the generation of a musical note depends on a hidden state, which in turn depends on the previous hidden state (Van Der Merwe & Schulze, 2011). This allows the model to capture patterns and structures that occur over time, such as the progression of chords in a piece of music (Yanchenko & Mukherjee, 2018). **Bayesian Networks** (or [Bayesian] **Belief Networks**), which represent dependencies among variables in a probabilistic graphical model, can similarly be used to model the complex interrelationships between different musical elements, such as pitch, rhythm, and harmony (Fang et al., 2021). These dependency models can be structured to create highly sophisticated chord-based harmonic generation networks, including four-part harmonizations, and chord voicing/prediction (Kitahara, 2017).

MRFs extend the capability of HMMs to capture temporal dependencies in music by allowing for dependencies between all pairs of notes, not just those that are temporally adjacent (Freedline, 2021). These models are particularly suited to modeling polyphonic music, where multiple notes can be played simultaneously and the relationships between these notes are crucial to the overall harmony of the piece (Lavrenko & Pickens, 2003). By representing the musical piece as a graph, where each note or chord is a node and the edges represent dependencies between notes, MRFs can capture both the melodic progression of individual voices and the harmonic

relationships between different voices (Arya et al., 2022). One currently unexplored approach includes **Gaussian Mixture Models (GMM)** – a probabilistic model that presumes all data points originate from a combination of a limited number of Gaussian distributions, where the probability density function is unknown (Fernando et al., 2012). In the context of music generation, each distribution could represent a different musical pattern or motif, allowing the model to sample from these distributions to create a mix of patterns in the generated music.

While these algorithms have proven effective in generating music that adheres to the statistical properties of the training data, they also face limitations. For example, they often struggle to capture higher-level musical structures and long-range dependencies, which are crucial for creating musically coherent and satisfying compositions (Yang et al., 2023). Furthermore, these models typically operate on a note-by-note basis, which may not adequately capture the holistic nature of musical composition (Carnovalini & Rodà, 2020). Despite these challenges, generative machine learning algorithms represent a promising approach to music generation, providing a foundation upon which more sophisticated models and techniques can be developed.

2.1.1.3 Natural Language Processing

Natural Language Processing (NLP) is an interdisciplinary branch of AI with applications spanning from text analysis to speech recognition. The structural similarities between language and music, coupled with the ability of NLP to manipulate both structure and meaning, provide a robust framework for the generation of complex and emotionally resonant music (Ma et al., 2020). Music can be treated as a form of communication with its own syntax, semantics, and structure, providing a plausible basis to apply NLP techniques to music generation given the language-like nature of musical composition.

Applying NLP to music generation involves the translation of musical elements into a format that can be processed as a language. This process, often referred to as **tokenization**, involves converting musical notes and rhythms into a series of tokens that can be manipulated using NLP techniques (Tham, 2021). The tokens are then processed using various algorithms to generate new sequences of tokens, which are subsequently translated back into musical notation. This approach allows for the generation of music that adheres to the structural rules of the original input while also introducing novel elements. Likewise, notes and chords carry emotional and thematic connotations in music like the semantic meanings of words in a language (Douek, 2013), enabling NLP models to perform semantic manipulation to generate structured music that also conveys specific emotional or thematic content.

2.1.1.4 Diffusion Models

Diffusion models, a class of generative models in ML built upon Markov chains, have recently gained attention for their potential in various creative domains, including music generation (Huang et al., 2023). These models operate by simulating a stochastic process, where data is gradually transformed from a simple prior distribution to a complex target distribution. In the context of music generation, this target could represent a specific style or genre of music, and the diffusion process would involve transforming random noise into a coherent musical piece that adheres to the characteristics of the target (Mariani et al., 2023). Once the model is trained on a dataset of music from the desired style or genre, it can generate new music by sampling from the learned distribution. This involves initiating the diffusion process with a sample from the prior distribution (typically random noise), and then gradually transforming this sample over a series of steps until it resembles one from the target distribution (Mittal et al., 2021). The result is a piece of music that is both novel and consistent with the style or genre of the training data.

One key advantage of diffusion models is their ability to capture long-range dependencies in the music data (Yang et al., 2023). This capability is crucial for generating music that is not only melodically and harmonically coherent but also structurally consistent over longer time scales. Furthermore, diffusion models can generate a diverse range of outputs, making them suitable for creative tasks where novelty and variation are desired, including text or image-based generation prompts (Forsgren & Martiros, 2022). However, like all generative models, diffusion models face challenges in ensuring the quality and coherence of the output music, though future improvements to the architecture and training methods may lessen these issues (Huang et al., 2023).

2.1.1.5 Cellular Automata

Cellular Automata (CA), a discrete computational system found in Automata Theory, has also been recently used to generate music (Berto & Tagliabue, 2017). While not inherently part of AI, the concepts of CA share similarities with evolutionary/genetic algorithms in principle and usage. In this model, a grid of cells evolves according to a set of rules based on the states of neighboring cells. To generate music, each cell state can be mapped to a specific musical note or a set of musical parameters such as pitch, duration, or volume (Murchison, 2003). As the cellular automaton evolves, it creates a sequence of cell states, which in turn generates a sequence of musical notes or changes in musical parameters, creating a complete piece of music.

The rules of the automaton, the initial configuration of the cells, and the mapping from cell states to musical elements can all be varied to create different types of music (French, 2013). This method can generate complex and interesting musical patterns from simple rules and initial conditions (Bilotta et al., 2000). However, Parent (2023) notes that while CA are useful in creating rhythmic components in music due to their ability to produce recurring patterns, they may not be sophisticated enough to create effective melodies without adding complexity to the models.

2.1.2 Quantum Computing

The rapid emergence of **Quantum Computing (QC)** has given rise to the novel field of **Quantum [Computer] Music**, building upon the foundation of algorithmic computer music and introducing quantum adaptations of traditional methods such as Markov chains and random walks (Miranda, 2022). One such example is the three-dimensional musical quantum walk (based on classical random walks), which generates musical sequences by running a quantum walk circuit twice for each note, once for pitch, and once for rhythm (Miranda & Basak, 2021, pp. 7-13). The measurements from these walks are then used to determine the pitch and rhythm of the next note in the sequence, creating a dynamic, probabilistic composition process influenced by the principles of quantum mechanics. Due to the statistical nature of QC, multiple executions of the algorithm, or "shots," are necessary to achieve statistically plausible results and mitigate errors. The novel Basak-Miranda algorithm exemplifies this approach by utilizing quantum mechanical properties of constructive and destructive interference (similar to Grover's algorithm) to operate a musical Markov chain (Miranda & Basak, 2021, pp. 14-24). Few attempts have been made to modify classical AI models into **Quantum Machine Learning (QML)** models – as well as Quantum Cellular Automata (Miranda & Shaji, 2023) – for use in music generation, though potential benefits of this approach have yet to outperform or be evaluated against the classical systems (Miranda et al., 2021; Oshiro, 2022).

2.1.2.1 Quantum Annealing

Recent studies on QC for music generation have utilized a process known as **Quantum Annealing (QA)**, a heuristic optimization technique emerging from the Adiabatic Quantum Computing (AQC) framework (Arya et al., 2022). The process facilitates the discovery of an objective function's minimum by leveraging properties intrinsic to quantum physics (such as

quantum tunneling, entanglement, and superposition) to return low-energy solutions (D-Wave, 2020). Freedline (2021) utilized QA and MRFs to translate the principles of music theory into a probabilistic field of chord progressions. A D-wave Leap Sampler was used to sample the distribution for the lowest energy state, deciding the subsequent chord in a music sequence. This algorithmic approach was incorporated into a custom CLI that converts the raw, quantum-generated music into playable MIDI files. Similarly, Arya et al. (2022) demonstrated QA's application in generating various aspects of music like melody, rhythm, and harmony using D-Wave quantum annealers. In melody generation, for example, the authors model the process of producing notes from a set of pitches as a QUBO (Quadratic Unconstrained Binary Optimization) problem, deploying binary variables to represent notes at specific positions.

2.1.3 Deep Learning and Neural Networks

Deep Learning (DL) is a specialized subset of ML (see Figure 2.2) that focuses on the application and refinement of [Artificial] **Neural Networks (NNs)** to replicate the function and/or structure of the human brain for highly performant big data processing (DeepAI, 2019b). These networks are composed of interconnected computational units, or "**neurons**," which mimic the basic function of biological neurons (such as pattern and relationship recognition) by receiving, processing, and transmitting information. Each neuron computes a weighted sum of its inputs (i.e., the **dot product** of the weights and input data) and applies a non-linear (**activation**) function to this sum to produce its output (DeepAI, 2019d). This output can then serve as input to neurons in the next layer, creating a network of neurons. While both ML and DL aim to learn from data and make predictions or decisions, DL is distinguished by its capacity for automated feature extraction (decreasing the need for human input) and scalability with data size, often leading to superior performance on tasks with large, complex datasets (Pai, 2020).

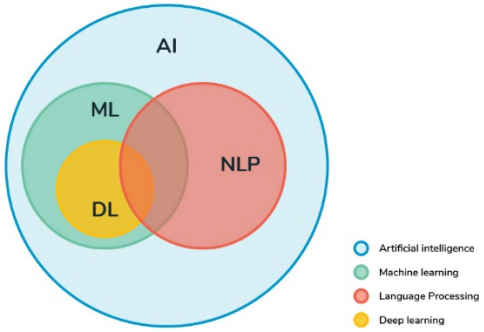


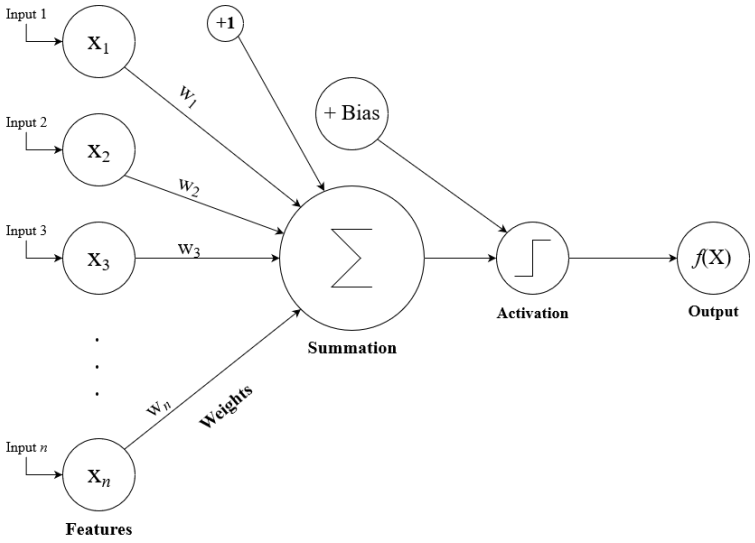
Figure 2.2: Relationship between AI, ML, NLP, and DL (Mehra & Hasanuzzaman, 2020)

Neural Networks are comprised of three different classes of layers that mimic the behavior of biological neurons, each with its own respective purpose (DeepAI, 2019d):

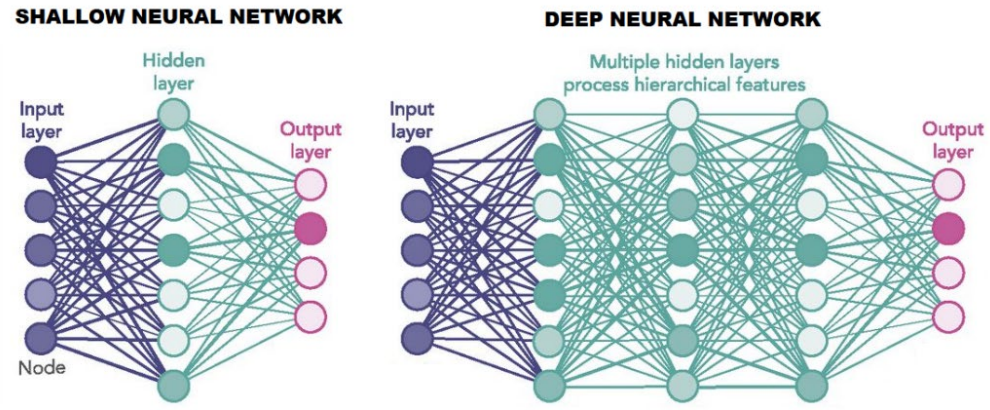
- An **input layer** (neural dendrites) – the first NN layer where raw data is fed into the system, with a one-to-one correspondence of neurons to features from the input data;
- One or more **hidden layers** (cell body) – any layers between the input and output that perform the data processing of the network, using neurons to perform weighted summation and apply transformations (i.e., **activation functions**) to the input data to extract intermediate features. Note that the number of hidden layers and their respective quantity of neurons (which define the complexity of the model) is not fixed and requires engineering for optimality; and
- An **output layer** (axon) – the final layer in the network that produces the model's output using an activation function specific to the desired task (e.g., sigmoid for **binary classification**, softmax for **multi-class classification**, or no function/identity for **regression**) where each neuron represents a possible output value or class.

These models consist of interconnected computational units, referred to as Artificial Neurons or **Perceptrons** (see Figure 2.3 (a)), with the simplest form being the Single-Layer Perceptron which makes binary decisions by linearly combining its input features (Sethi, 2019).

The transition from **shallow** architectures (with one or two hidden layers) to **deep** architectures (with many hidden layers; see Figure 2.3 (b)) facilitates the extraction of more complex, higher-level features (Singh, 2020). Training NNs involves multiple **epochs** (full passes through the training data) and **iterations** (updates per **batch** of training examples), typically relying on the **Gradient Descent** optimization algorithm to minimize the discrepancy between the network's predictions and the actual outcomes. In a **Multilayer Perceptron (MLP)** – an elementary **Deep NN** – information flows forward from the input layer to the output layer through multiple hidden layers (i.e., **feedforward**), and the error is then propagated backward (the **backpropagation** algorithm) to adjust the weights, aiming to reduce the overall prediction error (Sethi, 2019).



(a)



(b)

Figure 2.3: A Perceptron (a; Szelogowski, 2022), shallow vs. deep NNs (b; Waldrop, 2019)

2.1.3.1 Generative Neural Architectures

The following list presents several of the most common neural architectures for generative models:

- **Convolutional Neural Networks (CNNs)** – a specialized neural architecture that excels at processing grid-like data, including images and time-series data (Sethi, 2019). **Convolutional** layers apply a series of filters to local features of the input data while **Pooling** (i.e., downsampling) layers reduce dimensionality and create an abstraction of the learned features (see Figure 2.4). This layered approach facilitates the extraction of complex, high-level features and enables CNNs to provide superior performance on tasks like image and video recognition.

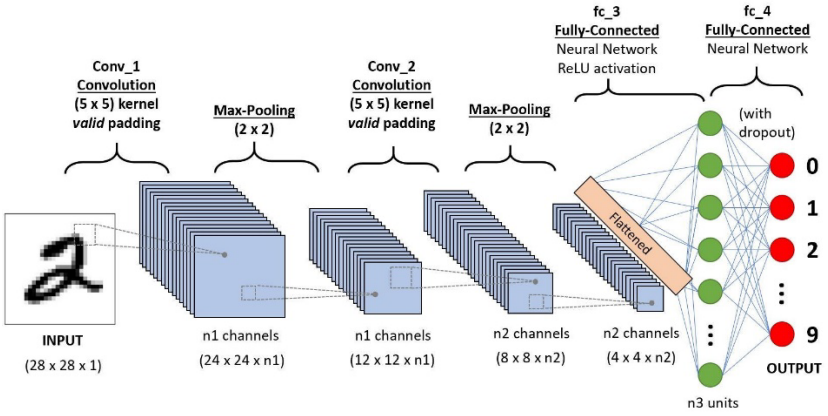


Figure 2.4: A simple Convolutional Neural Network architecture (Suha, 2018)

- **Generative Adversarial Networks (GANs)** – a class of generative models consisting of two components: a **Generator**, which produces synthetic data, and a **Discriminator**, which evaluates the authenticity of the generated data (Szelogowski, 2021). The interplay between the two components leads to the generator producing increasingly realistic data, allowing for use in various tasks such as audio generation, image synthesis, image-to-image translation, and data augmentation (see Figure 2.5).

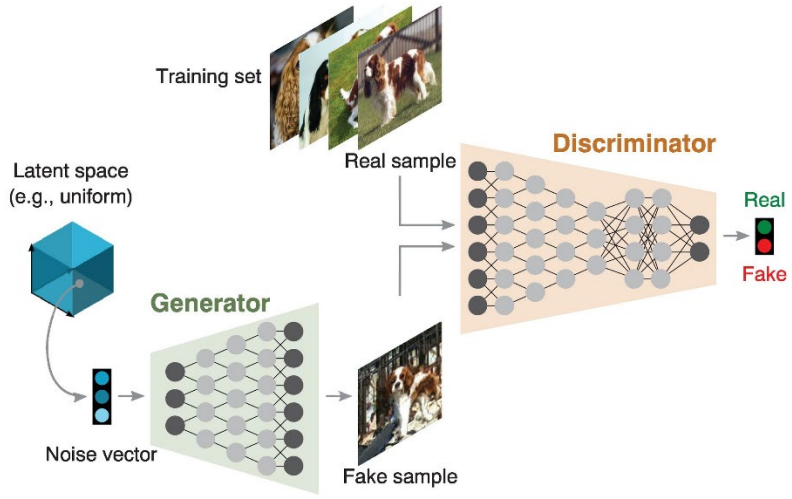


Figure 2.5: A typical GAN architecture (Goetschalckx et al., 2021)

- **Variational Autoencoders (VAEs)** – generative models that use principles of probability theory and statistics to generate new data (Rocca, 2021). Unlike traditional autoencoders, VAEs impose a probabilistic structure on the encoded latent space, enabling the generation of new data instances by sampling from this space (Kumar, 2023). They are primarily used for tasks like image generation, anomaly detection, and neural style transfer.
- **Restricted Boltzmann Machines (RBMs)** – a class of generative stochastic NNs capable of learning a probability distribution over their input set, as well as performing dimensionality reduction (Karakus & Kose, 2020). They are composed of visible and hidden units with symmetric connections between them, but no connections within a layer.
- **Auto-Regressive Models (ARMs)** – statistical models used for predicting future values based on past values (e.g., forecasting; Hyndman & Athanasopoulos, 2018). In the context of DL, ARMs such as PixelRNN and PixelCNN generate data one component at a time, with each component conditioned on the previously generated ones (Ho, 2019). This approach is especially effective for sequential data generation tasks, such as text and music.

2.1.3.2 Natural Language Generation Architectures

The following list discusses several of the most common neural architectures used for **Natural Language Generation (NLG)** frequently applied to generative music tasks:

- **Recurrent Neural Networks (RNNs)** – a category of NNs designed specifically for handling sequence data, characterized by their inherent ability to process input sequences of arbitrary length through **hidden states** (see Figure 2.6) that propagate information over time (Sethi, 2019). Given their temporal dynamic behavior, they are widely employed in tasks such as NLP, speech recognition, and time series prediction. Their key strength, maintaining temporal dependencies, is often leveraged through variants like **Long Short-Term Memory (LSTM)** units and **Gated Recurrent Units (GRUs)** to enable longer sequence generation.

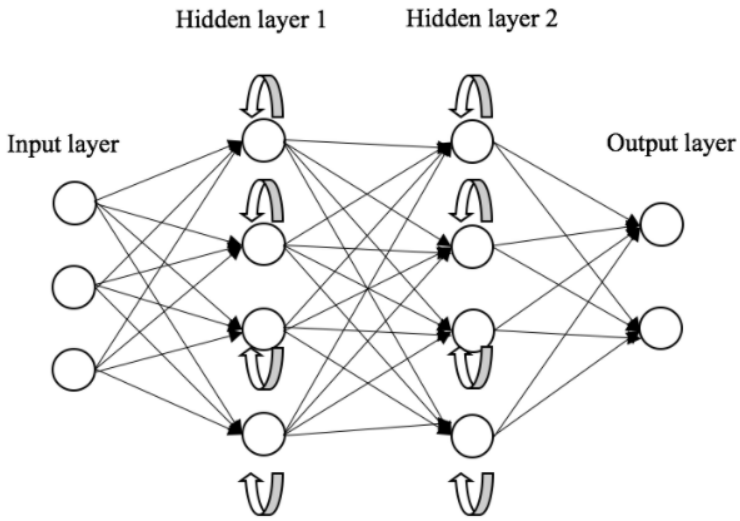


Figure 2.6: A simple Recurrent Neural Network architecture (ODSC Community, 2020)

- **LSTM Networks** – a special type of RNN designed to mitigate the vanishing gradient problem (a prevalent issue in training standard RNNs) by maintaining a more constant error through specialized units called **gates** (Choubey, 2020). These gates – specifically, the

input, **forget**, and **output** gates – control the flow of information into, within, and out of each LSTM cell, enabling the network to learn longer sequences and dependencies without losing important information (see Figure 2.7). This feature makes LSTMs particularly well-suited for tasks requiring the understanding of long-term dependencies such as machine translation and various NLP applications.

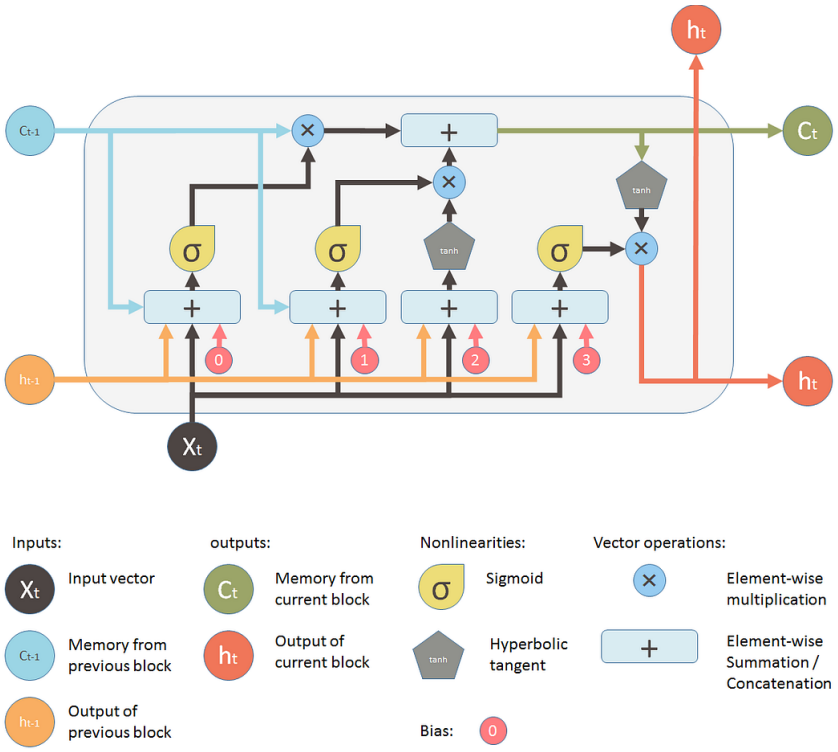


Figure 2.7: Complete LSTM unit (Choubey, 2020)

- **Seq2Seq/Encoder-Decoder Models (EDMs)** – a robust framework for various sequence-to-sequence prediction tasks (e.g., neural machine translation and summarization) consisting of two primary components (Moses, 2021): an **Encoder**, which converts the input sequence into a fixed-length vector, and a **Decoder**, which processes this vector to produce the output sequence. The encoder and decoder are typically implemented with RNNs or LSTMs, allowing the model to handle variable-length input and output sequences while capturing temporal dependencies in sequence data.

2.1.3.3 Transformers

Transformer models – a class of EDMs where both the encoder and decoder are composed of several identical layers that work in parallel – have recently established themselves as a significant architecture in the field of NLP due to their scalability and performance (Vaswani et al., 2017). These models rely on a specialized "**[self-]attention mechanism**," a computational process that assigns different weights to different parts of the input, based on their relevance to the task at hand (see Figure 2.8). Unlike RNN and LSTM systems, transformers do not process the input sequentially; instead, they leverage the attention mechanism to process all parts of the input simultaneously. This shift from sequential to parallel processing enables these models to handle longer sequences more efficiently and effectively, also providing a strong basis for use in music generation (Huang et al., 2019). The architecture has also gained popular recognition for its use in **[Auto-Regressive] Large-Language Models (LLMs)** such as BERT (Bidirectional Encoder Representations from Transformers; see Figure 2.9) and **GPT (Generative Pretrained Transformer)**, which have shown exceptional performance in a range of tasks including text generation, translation, and question-answering (Merritt, 2022).

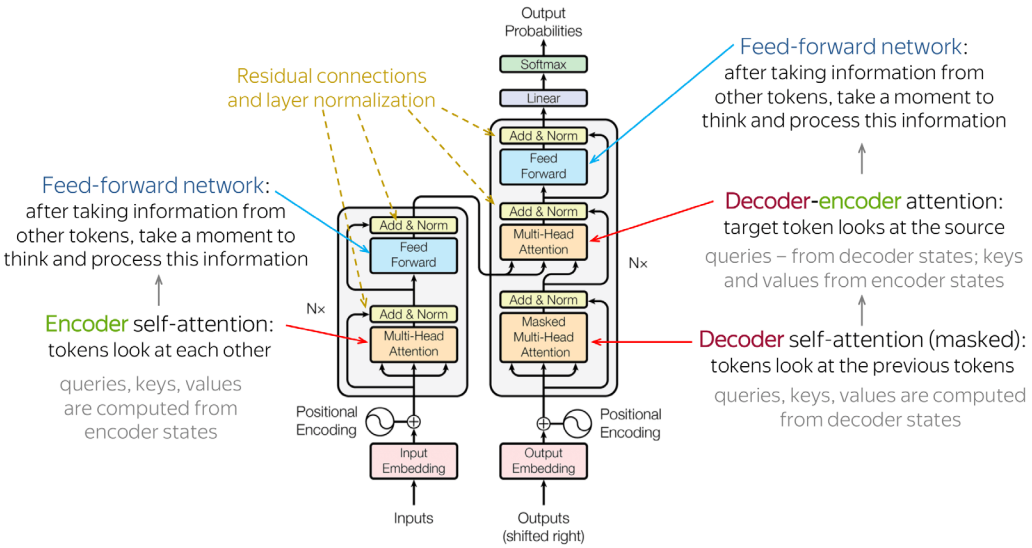


Figure 2.8: Transformer architecture (Voita, 2023)

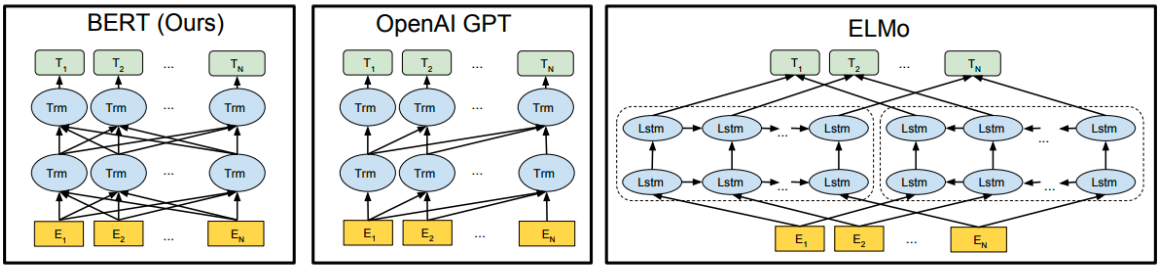


Figure 2.9: BERT, GPT, and ELMo architectures (Lei et al., 2022)

The self-attention mechanism allows the model to compute a representation of each word in the context of all other words in the input sequence (Cristina, 2022). Specifically, each word's representation is computed as a weighted sum of all word representations in the sequence, where the weights are determined by the attention mechanism. This enables the system to capture both short-term and long-term dependencies in the input data regardless of their positions in the sequence. To enhance this functionality, transformers employ "**multi-head self-attention**" wherein the self-attention process is repeated multiple times in parallel but with different learned linear transformations of the input (see Figure 2.10), allowing the model to focus on different features and capture various aspects of the input sequence simultaneously (Vaswani et al., 2017).

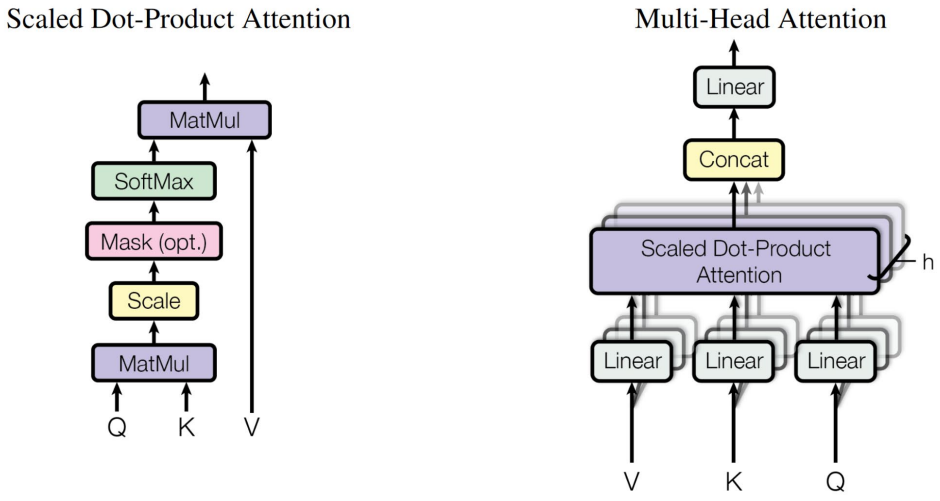


Figure 2.10: Self-Attention and Multi-Head Attention mechanisms (Vaswani et al., 2017)

2.1.4 Music Theory and Composition

Music Theory, as a field of study, provides a foundational framework for understanding, analyzing, and creating music. It encompasses a range of concepts including pitch, scales, modes, rhythm, melody, harmony, analysis, and the structure or form of musical works (Kostka et al., 2017). This theoretical knowledge serves as a critical tool for composers, enabling them to make informed decisions about the development and organization of musical ideas. **Composition**, the creative process of producing a new piece of music, leverages these theoretical concepts to construct musical structures that communicate specific emotions or ideas (Benward & Saker, 2020). Thus, while music theory provides the "rules" or conventions of music (which current generative AI models lack the inference of), composition applies and often innovatively interprets these rules to create new musical works.

The **Common Practice Period** (or Era), spanning roughly from 1600 to 1900, is a term used to denote an era in Western art music in which composers largely adhered to a shared set of rules or conventions for harmonic progression and resolution, tonality, and musical form (Kostka et al., 2017). This period encompasses the Baroque, Classical, and Romantic eras, each with its unique stylistic characteristics yet underpinned by a common theoretical framework. It was during this period that many fundamental principles of Western music theory were codified and rigorously applied, resulting in a coherent musical language that continues to influence composition today. Many of the works produced during this period have come to represent the "canon" of Western classical music, and the principles formalized during this era continue to form a significant part of music education and analysis (Jean, 2021). Understanding the conventions of the period provides significant insights into the evolution of Western music, serving as a critical context for both the interpretation of classical works and the exploration of contemporary compositional techniques.

2.1.4.1 Musical Form

Musical form refers to the structural organization of a piece of music – the blueprint that outlines the arrangement and development of musical materials over time (Szelogowski, 2022). This structure is commonly delineated through distinct **parts** (labeled with uppercase letters e.g., **A, A′**, **B, C**, etc. with prime marks (′) or superscripts (^{n}) denoting **varied repetition** of an existing structure) or **sections** (labeled with numbers, e.g., **Sec. 1, Sec. 2**), indicating material that is perceived as a unified entity or idea. These sections comprise smaller structural units known as **phrases** (labeled with lowercase letters, e.g., **a′, a′′, b, c**), which in turn consist of one or more **motives** (or motifs), the smallest recognizable musical ideas. The end of a phrase is typically marked by a **cadence**, a musical punctuation mark that provides a sense of closure or continuation (see *Appendix C.1*). Just as a literary work is divided into chapters, paragraphs, and sentences, a musical composition is structured into parts, phrases (or **themes**, i.e., melodies/subjects), and motives, each contributing to the overall narrative of the piece (Kostka et al., 2017).

Cadences play a crucial role in signaling the end (or resting point) of a phrase or section. They are typically classified as one of four key types, each with distinct tonal effects (Green, 1979):

- **Authentic Cadence (AC)** – the most conclusive type; consists of a dominant to tonic progression (V to I in major keys, V to i in minor keys);
- **Deceptive Cadence (DC)** – diverts expectations by resolving the dominant to a chord other than the tonic, typically the submediant (VI or vi), providing a sense of surprise;
- **Half Cadence (HC)** – also known as an imperfect cadence, is less conclusive, ending on the dominant (V) and thus providing a sense of anticipation or suspension; or
- **Plagal Cadence (PC)** – or "Amen cadence" from its common usage in hymns, involves a subdominant to tonic progression (IV to I or iv to i), imparting a sense of gentle resolution.

These cadences provide composers with essential tools for shaping the musical narrative of a composition; however, generative AI models have yet to display evidence of high-level cadential awareness, often leading to the "chaotic" and unrestful-sounding music created by most systems and contributing to the lack of overall form/structure in AI compositions (Szelogowski, 2022).

There are numerous types of (large) musical forms that composers employ to organize their musical ideas (Green, 1979). Among the simplest is **Binary** form (AB), in which two contrasting sections are presented (or **Rounded Binary** (ABA′) where A partially returns). **Ternary** form (ABA) consists of two similar sections separated by a contrasting one, often providing a sense of balance and symmetry. **Rondo** form (ABACA, or ABACABA, etc.) alternates between a recurring section (**refrain**/A) and contrasting sections (**episodes**). **Sonata[-allegro]** form (a composite, rounded binary form), a complex structure commonly used in the first movement of classical symphonies, comprises **Exposition**, **Development**, and **Recapitulation** sections with various themes for introduction, exploration, and resolution. Through musical forms, composers articulate musical narratives, leading listeners through a series of musical ideas and their transformations.

Analyzing musical form involves identifying and understanding the structural components and their relationships within a piece of music (see *Appendix C.2*; Green, 1979). The process encompasses recognizing repeating and contrasting parts, motives, and phrases (by cadences), and observing how they evolve throughout the composition. One may also consider elements like tonality, rhythm, texture, and dynamics in the analysis, as these aspects can significantly influence the perception of form. Through this systematic examination, form analysis offers critical insights into the composer's intention, the piece's narrative arc, and its aesthetic impact. This is also very difficult for AI models given the objective nature (i.e., **fuzzy logic**) of music analysis, making the compositional rules difficult to impose upon generative AI as well (see Szelogowski, 2022).

2.1.4.2 Voice Leading and Counterpoint

In the realm of (mixed voice) choral music, voices are typically divided into the following four groups (in order as they appear in musical scores from top to bottom; Kostka et al., 2017):

- **Soprano** (top voice) – usually the highest female (or treble) voice, often carries the melody in choral compositions due to its prominence in the tonal spectrum (see *Appendix C*).
- **Alto** (middle voice) – typically the lower female voice, frequently provides harmonic support, filling the gap between the Soprano and male voices.
- **Tenor** (middle voice) – generally the highest male voice, often bridges the sonic space between the Alto and Bass harmonically; and
- **Bass** – the lowest of the four voices, provides the harmonic foundation, often delineating the root notes of the chord progressions.

Each of these voices plays a critical role in shaping the overall texture and harmonic richness of the choral ensemble, although these roles are frequently rearranged or doubled by the composer.

Voice leading is a fundamental concept in music theory that refers to the practice of arranging smooth, coherent lines for each voice in a **polyphonic** composition (Hutchinson, 2023). The primary goal is to maintain the independence and integrity of individual voices while ensuring their harmonious interaction. The rules of voice leading – which evolved predominantly during the **Common Practice Period** – guide the resolution of dissonances, the movement between chord tones, and the avoidance of certain voice overlaps or gaps (Kostka et al., 2017). While these rules are not hard constraints, they provide a systematic approach to arranging voices in a way that ensures musical coherence and aesthetic satisfaction. Proper voice leading is integral to choral writing, where each section of the SATB ensemble must maintain a distinct and navigable melodic line, contributing to the overall harmony without obscuring the clarity of individual voices.

Counterpoint, a sophisticated technique in music composition, involves the weaving together of independent melodic lines to create a complex, polyphonic texture (Gotham et al., 2021). Originating in the vocal music of the Medieval period and reaching a high level of development during the Common Practice Period, counterpoint is a central feature of Western music that continues to be used in various forms in contemporary composition (Jackson, 2020). At its core, counterpoint is **a set of voice leading rules** covering the interplay between consonance and dissonance, independence and interdependence, tension and resolution. These rules (most comprehensively articulated in the study of **Species Counterpoint**; Hutchinson, 2023), ensure the independence of each melodic line while also guiding their harmonious combination. As such, understanding and applying the principles of counterpoint enables the creation of intricate, multi-layered compositions that fully exploit the potential of polyphonic music ensembles (Chase, 2020).

The rules of counterpoint revolve around managing the relationships between simultaneous melodic lines to maintain their independence while achieving harmonic coherence (Gotham et al., 2021). One fundamental principle is the maintenance of **melodic interest** (ensuring that each voice follows its own distinctive and comprehensible melodic contour), often ensured by avoiding **parallel perfect intervals** (unison, octave, and fifth), which can cause two voices to lose their individuality (Chase, 2020). **Contrary motion**, where one voice ascends while the other descends, is often encouraged to maximize the independence of the voices. Similar attention is paid to **dissonances** (which are usually approached and left by step), resolving to a **consonance** to provide a sense of **tension and release**. Another critical guideline is the **avoidance of voice crossings**, where a lower voice moves above a higher one, or vice versa, as this can compromise the clarity and individuality of the voices (Gotham et al., 2021). Lastly, range considerations are essential, and each voice must remain within its appropriate **tessitura** to maintain effective projection.

2.1.4.3 Composing Versus Arranging

In the realm of music theory, "**composition**" and "**arranging**" represent distinct yet interconnected facets of music creation. Composition refers to the process of creating a new piece of music from scratch, encompassing the development of melody, harmony, rhythm, and form (Haefele, 2022). It involves crafting original musical elements to express an artistic concept or emotion, often resulting in the creation of a novel piece of music. In contrast, arranging (sometimes "**reorchestration**"; Cambridge, 2013) is the process of adapting or modifying an existing piece of music for a specific ensemble, instrument, or performance context, often involving changes in texture, voicing, or even style (Adler, 2016).

While both processes require a deep understanding of musical principles, the key distinction lies in their starting point: compositions begin as an entirely new piece, whereas arrangements take an existing piece of music and reshape it to fit a specific context or objective. Generative AI presents intriguing possibilities in both cases; in composition, models can be used to generate novel musical ideas, utilizing large datasets of music to produce original pieces that echo certain styles or genres – thus automating and expanding the creative process (Dey, 2023). Conversely, in arranging, models can use their learned musical knowledge to reinterpret existing works for different instruments or ensembles, potentially streamlining the arrangement process and offering novel reinterpretations (Band Pioneer, 2023). Current generative AI models have been trained primarily for composition (mostly popular music), rather than for arranging, however.

2.2 Literature Review

This section presents a chronological overview of significant literature related to the usage of generative algorithms, ML/DL models, and hybrid systems in generating or aiding in the composition of realistic classical music.

2.2.1 Holtzman, 1981

While the earliest notable research projects in AI music generation date back to as early as 1957 with Lejaren Hiller's composition, the Illiac Suite, and 1960 with Rudolf Zaripov's algorithm for the Ural-1 computer (Li, 2019), Holtzman (1981) presents a more novel approach to computational music – using **generative grammars**. The author discusses the difficulties of early algorithmic music generation, including the issue of attempting to formalize compositional rules; simulating the logic of a composer requires explicit definitions for informal rules, creating a barrier of difficulty for researchers seeking to generate realistic compositions. Holtzman previously implemented a program known as the **Generative Grammar Definition Language (GGDL) compiler**, allowing for the description of language grammars using sets of formal and unambiguous rules – being especially tailored toward the description of music languages as well. By defining music languages through generative grammars, the compiler allows for generating such a language's utterances (i.e., compositions). As such, a researcher could define formalized composition rules over a conceptual framework through formal grammars and language theory rather than explicitly programming the rules.

The GGDL language is written as a series of rewrite rules, which describe how characters in a string are replaced or rewritten, including allowing for alternate mappings and more complex replacement rules such as serial selection (i.e., requiring all objects to have been selected before any can be selected a second time – a common technique in atonal music). More applicable to tonal music, however, is the use of finite-state transition matrices for production rules, where "successive productions are dependent on previous productions [from the same left-hand side (LHS) nonterminal]" (Holtzman, 1981, p. 52). This creates a row of transition probabilities indicating the likelihood of the production itself, followed by the other possible string generations.

Other generative rules in GGDL include selection functions to retrieve the index of one of the right-hand side (RHS) strings and meta-production rules, which indicate where single generations from a rule are substituted for all occurrences in the LHS.

GGDL's generative process is broken into three stages: abstract relationship structure generation (generating utterances in a language defined by the rewrite rules and selection functions), a transformation process, and a process where "objects (terminals) are mapped […] from the abstract domain (as morphemes, e.g., any musical objects, individual sound samples, etc.) to defined representations (as sounds)" (Holtzman, 1981, p. 53). The transformation process uses transformational change markers to initiate musical transformations characteristic of atonal music, including inversion, retrograde, and transposition on strings of objects. Given the simpler, logic-based compositional techniques found in twelve-tone row (serial) atonal music, these transformations can be used to replicate existing atonal compositions solely through generative utterances. By adding additional rules through functions and finite-state matrices, the output can be constrained to specific group durations, repetitions (such as a canon), and the sound envelope itself (i.e., attack, steady-state, and decay) for dynamic control.

The abstract tokens manipulated by the GGDL grammar system can take the form of micro-components of notes/objects, notes themselves, or complete sections, allowing a great degree of control over the generated output. Also, by manipulating the sound envelope, the FM representation of the sound can be controlled to create different representations, including replicating the sounds of different instruments. While the fine degree of control and simple rulesets are useful for serial melody generation, for authentic compositions, these techniques lack the informality of real decision-making (especially) for tonal classical pieces. Music generation research has typically been concerned with generating notes, durations, dynamics, octave/pitch-

class distribution, and other lower-level concepts like rhythm. However, this does not consider the prerequisite knowledge and musicality of the composer: understanding when, where, and why certain instruments are used in a piece (e.g., musical "clichés"), phrasing, performance techniques (bowing styles, tonguing, shifts, etc.), performer limitations (breath support in phrase lengths, range, tessitura), as well as segregating the melodic writing-styles used for various instruments (i.e., involving either more leaping, chromatic, stepwise, or neighboring motion).

Although the grammar system may be constrained to instrument-specific rulesets to better moderate the output to a more performance-realistic composition, the program leaves much work to be rearranged by hand to remove unplayable parts and other unwanted structures. As such, the author proposes that GGDL may be likened to a generative tool where the composer acts as a **selector**, filtering out and combining the most appealing utterances. While the usage of generative grammars is efficient in that it can be bound by logic to create more appealing or appropriate output related to the intended ruleset, music is not intended to be bound to or evaluated by correctness – the listener must simply decide if the music sounds good.

2.2.2 Van Der Merwe and Schulze, 2011

This research by Van Der Merwe and Schulze (2011) utilizes a **Hidden Markov Model (HMM)** approach to extend and modify the rule-based music composition system Willow (Högberg, 2005), referred to as SuperWillow. The system separates the composition process into multiple steps, "[including] chord progression, melodic curve, cadence, and so forth" (Van Der Merwe & Schulze, 2011, p. 78). At each step, the parameters for **probabilistic automata** are calculated from music data, which are then used to generate the new composition. By generating chords for a given melody or vice versa, simple two-voice harmony can be achieved, and prediction-suffix trees are used to model the melody and rhythms.

To build the automata, the system must first filter, extract, and analyze the music data – initially filtering out pieces the system cannot analyze, e.g., a piece where the time signature changes. During extraction, the musical data are transposed to the key of C while preserving the mode of the scale, then the list of note durations and classified chords from the data are returned. Finally, the analyzer component of the system builds analysis objects, including **Markov Chains (MCs)** and HMMs, which are then grouped into styles for use in generating or imitating compositions in a particular style. The MCs are used to model chord duration and chord and rhythm progressions, then model sequences of events using states with transitional probabilities between them.

Similarly, **Prediction Suffix Automatons (PSAs)**, equivalent to mixed-order MCs, allow transitions of varying memory lengths without the exponential growth found in higher-order MCs and can be built from prediction suffix trees through statistically significant string (motif) discovery during training. HMMs are also used in modeling relations between hidden and observed sequences, which can be used to determine the MC transition probabilities. These varying models are used to represent chord and rhythm progression and duration, and melodic arc, which are then analyzed using Roman numeral chord analysis and voice separation. After compiling the analyses, the models can generate or imitate compositions by "sampling from the distributions obtained from the analysis" (Van Der Merwe & Schulze, 2011, p. 82). Generation styles can thus be chosen by the user or randomly, followed by the tempo, time signature, scale, and instrumentation being randomly selected in relation to the specified style. The chord and rhythm MCs are then used to generate note value sequences of a specified total duration.

The finite-state transducer package Carmel provides SuperWillow with generalized automata that produce output symbols as input symbols are consumed, sampling from the

distributions created by the compositional analyses. Rhythm generation and chord progression are also controlled through Carmel, providing a probabilistic rhythm automaton for string transformation to the desired chord progression length. In a survey prompting users to select whether or not a piece was human or computer generated where the human composers were given the same constraints as the Markov models, 38% of respondents (out of 263) incorrectly identified the computer-generated music, and 59% indicated a preference for one or both computer compositions over the human compositions. However, the authors note that the generation steps do not account for other steps, leading to failed composition attempts. Although HMMs show promise in modeling melody/chord relationships, the SuperWillow system lacks proper analysis of rests and rhythms, places too many restrictions on the training data, and can only generate two-voice compositions.

2.2.3 Buys, 2011

Buys' (2011) research presents a survey of generative music models for various contexts and historical developments, including composer and style imitation. While focused initially on Markov Processes and models, Buys also notes the limitations of Markov models in generating extended musical pieces with a desirable overall structure, and as such proposes a novel approach "**BACCHUS**" (Bach-inspired Algorithmic Computer Composer and Harmonizer, University of Stellenbosch) that leverages large datasets of musical pieces for training and evaluation. The approach utilizes the finite-state transducer package Carmel for processing Markov models, which offers capabilities including training, composition, sequence sampling, and the computation of sequence probabilities from transducers. Similarly, the tree transducer package Tiburon, which offers similar functionalities as Carmel, is employed to implement context-free grammar and tree

transducer models. Using Tiburon, context-free grammars can be converted to an equivalent regular tree grammar or tree transducer, facilitating parsing and training operations.

Markov models and their variants have remained dominant in the field of statistical music generation since 1950 due to their simplicity and ease of use. The main challenge in using Markov models is the sparse data problem, which requires a balance between low-order Markov chains that do not constrain the structure of the generated music enough and high-order Markov chains that reproduce large fragments of the music used for training. Researchers have explored various approaches to address this challenge, including Multiple Viewpoint Systems, Probabilistic Suffix Automata, mixed-order Markov models, and higher-order Markov models. Other researchers have used machine learning algorithms (such as the MDI learning algorithm) to learn Stochastic Finite Automata for musical style recognition and jazz grammars, which involves abstracting the melody into note categories, durations, and melodic contour and constructing a Markov chain of clusters. However, some constraints may have to be relaxed to allow for the generation of pitch sequences that satisfy the note category and interval range restrictions over a given sequence of chords.

Buys (2011, p. 16) also notes the prevalence of Hidden Markov Models (HMMs) in music generation, which can effectively model the discrete sequences present in music, such as harmony. Allan and Williams (2004) utilized HMMs to harmonize given melodies, where the visible states represent the melody notes and the hidden states represent the possible harmonizations for the melody line, accounting for different configurations of the same chord-pitch combinations. The MySong automatic accompaniment system developed by Microsoft also utilizes HMMs for music generation, where the model generates chords for a user-sung melody and includes parameters for musical styles, including the “jazz factor” and “happy factor,” which are adjustable by the user. The system transcribes the user-sung melody into a key, generates chords with the HMM (using

chords as hidden states and melody notes as observations), and transposes the chords back to the original key.

In the field of music generation, various grammar-based systems have been proposed to generate musical sequences, such as the linear-time algorithm SEQUITUR which infers structure from a sequence of symbols to form a **context-free grammar** and was also found to identify cadences correctly (Nevill-Manning & Witten, 1997). Keller and Morrison (2007) used a hand-crafted context-free grammar in their program ImproVisor to generate jazz improvisations based on given chords, representing pitches as note categories with assigned probabilities for generating terminal symbols and durations. Similar research has proposed probabilistic context-free grammar to generate melodies in terms of melodic reductions, where the grammar is trained by an expectation-maximization algorithm and probabilistic tree grammars that model phrase structure in music and language uniformly. Stochastic *k*-testable tree models, represented by probabilistic tree automata, have also been used to classify given melodies based on style, where the pitches and rhythm of a melody are represented by a tree structure and trained with music pieces of similar styles.

Buys (2011, p. 18) discusses that beyond context-free grammars, Lerdahl and Jackendoff (1983) put forth a **generative (unrestricted) grammar** approach to understanding the psychological phenomenon of music cognition with the belief that the obvious forms of organization in music provide the foundation for understanding its complexity. The hierarchical structures of musical intuition, including grouping structure, metric structure, prolongational structure, and time-span reduction, are induced through generative models with a distinction between well-formedness rules and preference rules (i.e., analyzing grouping and meter independently). Time-span reduction describes the interaction of grouping and rhythm, while

prolongational structure models the perception of tension and relaxation in music. Likewise, transformational rules are used to describe non-hierarchical events. Similar systems also implement complex strategies for recombining musical material through augmented transition networks using unrestricted grammars by categorizing melodic fragments into numerous semantic classes with possible successors specified for each category.

When generating music pieces with any generative statistical model, the dominant approach of random walk sampling may result in lower probabilities of the generated sequences than the paths with the highest probabilities in the model. However, the restriction to only a few pieces with the highest probabilities may not be appropriate in music generation. To overcome this challenge, Conklin (2003) suggests using **Gibbs Sampling**, which involves starting with a given music piece, iteratively choosing a random position in the music, modifying a note or a sequence of notes, and accepting the change if the modified piece has a sufficient probability. Regardless, such a model must also be capable of preserving similarities within the music piece.

2.2.4 Hadjeres, Pachet, and Nielsen, 2017

This work by Hadjeres et al. (2017) presents a novel system – the **DeepBach** generative model, trained on Johann Sebastian Bach's chorale harmonizations – designed to model polyphonic music (e.g., hymns) and generate realistic four-voice chorale pieces in the same style as the composer. The system represents data using MIDI pitches to encode notes and model voices separately, ensuring that each voice only has one note notated at a time, and discretizes time into sixteenth notes (i.e., four equal subdivisions per beat) while modeling rhythms using a symbol to represent the hold duration. A chorale is modeled as a tuple of 4 voices and a set of 2 metadata lists – one for storing the subdivision indices of a beat (acting as the metronome), while a separate Boolean list records whether an instance of a note has a fermata (indicating cadences or phrase

ends). Likewise, expert musicological information, such as scales and harmony, are discarded in favor of generalized data representations.

A dependency network defined on a finite set of variables forms the time-invariant model, allowing for the prediction of notes (constrained to the corresponding voice ranges) based on knowledge of neighboring notes, the presence of fermatas, and the subdivision of the beat it is on. The authors fit these conditional probability distributions to the data by maximizing the log-likelihood, forming four classification problems. Four neural networks are thus employed in an ensemble: two **Deep Recurrent Neural Networks (RNNs)** where one sums past information and one sums future information with a non-recurrent neural network for notes occurring at the same time, and one final non-recurrent neural network whose output contains the merged output of the prior three. As such, the architectural design is intended to replicate the real compositional practice of chorale writing, where starting from the cadence and writing music "backward" is often simpler when reharmonizing. By not sampling the musical data from left to right, users can define unary, positional constraints such as for cadences, chords, notes, parts, and rhythms. Utilizing these constraints, the model can quickly and efficiently generate coherent musical phrases and varied reharmonizations without plagiarizing from the training set.

The dependency network generation is performed using the pseudo-Gibbs sampling procedure – a **Markov Chain Monte Carlo (MCMC)** algorithm similar to Gibbs sampling but may have potentially incompatible conditional distributions and violates Kolmogorov's criterion (i.e., it is not reversible and converges to another stationary distribution rather than to a joint distribution whose conditional distribution matches the ones used for sampling). Nonreversible MCMC algorithms have been found to perform better at sampling than reversible Markov Chains in some cases, and the algorithm can be parallelized to provide significant speedups during

generation using the GPU. However, Gibbs sampling fails to sample joint distributions correctly when variables are highly correlated, leading to regions of high probability states that may trap the MCMC chain.

The DeepBach model was implemented using Keras with the TensorFlow backend, using a database of 352 chorale harmonizations by J.S. Bach. After removing chorales with instrumental parts and simultaneous notes, the dataset was augmented by adding all chorale transpositions within the vocal ranges defined by the initial corpus (rather than transposing all pieces to the same key), resulting in a corpus of 2503 chorales split between a training set (80%) and a validation set (20%). The model used local interactions between a note and its context, with only elements within a certain time scope as inputs. A neural network with one hidden layer of size 200 and ReLU nonlinearity was chosen as the "neural network brick" and two stacked LSTMs, each of size 200, as the "Deep RNN brick." An "embedding brick" was also used to apply the same neural network at each time slice, and dropout was applied at the input and after each layer. The authors also found that sharing weights between the left and right embedding layers did not improve the accuracy or musical quality of the generated chorales.

The quality of the DeepBach model was evaluated through an online discrimination test conducted with human listeners, including comparison against a **Maximum Entropy (MaxEnt)** model and a **Multilayer Perceptron (MLP)** model. The trial included 1272 participants who were asked to identify whether a musical extract was composed by Bach or the computer-generated DeepBach, MaxEnt, or MLP models. Musical extracts were obtained by reharmonizing 50 chorales, creating 400 extracts in total. Experimental results indicated that the percentage of "Bach" votes increased as the complexity of the model increased. The accuracy of distinguishing computer-generated extracts from Bach's extracts was higher for individuals with higher musical

expertise. DeepBach was found to have produced the most extracts that were considered to be composed by Bach by the participants, and the MLP model produced the fewest. Following the evaluation, a MuseScore plugin was also developed using the model (with some changes), enabling users to generate chorales, reharmonize melodies, and regenerate chords, measures, or parts. The quality of solutions produced by the plugin was evaluated using examples of chorales generated without manual input. Despite some compositional errors, the analysis showed that the DeepBach compositions reproduced typical Bach-like patterns.

2.2.5 Mao, Shin, and Cottrell, 2018

Mao et al. (2018) present a novel deep learning model, DeepJ that generates polyphonic music tailored to the style of various composers. The research highlights the importance of tunable parameters in generating music for the purpose of assisting artists, composers, and filmmakers in creating new pieces and other creative works. As such, DeepJ incorporates methods for learning musical dynamics and enforcing specific music styles; while this research focuses specifically on classical composers of various eras due to dataset limitations, the approaches used in developing the model can be generalized to other forms of music as well as being extended to other tunable parameters such as emotion or mood. To this end, the authors compare their novel architecture and a similar Biaxial LSTM (Long Short-Term Memory) network, which produced less preferable compositions than the DeepJ model.

The Biaxial LSTM architecture is a powerful model that generates high-quality music and has been used in various applications, such as generating novel music and accompaniments to existing melodies. The model generates polyphonic music using a piano roll-like representation of notes, where notes played at each time step are represented as a binary vector (with 1 representing the note being played or 0 otherwise). The model thus learns the conditional probability of a note

being played based on all prior time steps and all notes within the current time step that have been generated already. The architecture supports biaxial probability conditioning using LSTMs that take inputs along two axes: a Time-axis module and a Note-axis module.

The time-axis module, inspired by Convolutional Neural Networks, consists of two-layer stacked LSTM units recurrent in time connected to each note octave in the input (reminiscent of a convolution kernel) and takes note octaves and recurrent states from the previous time step to output higher level features for each note. Likewise, the note-axis consists of another two-layer stacked LSTM units recurrent in note, sweeping from the lowest note feature to the highest to predict each note based on the predicted lower notes. Each note's features are first concatenated with the lower chosen note, enabling the note-axis LSTM to learn probability estimations based on lower chosen notes. By sharing LSTM weights in the time-axis module, the architecture provides the advantage of transposition invariance, which allows the model to generalize to different transpositions. The biaxial architecture also feeds contextual inputs to each LSTM in the time-axis, which includes a representation of the current beat in the bar and aids the model in producing outputs that are consistent in tempo.

The DeepJ model improves the Biaxial architecture for generating musically plausible results, utilizing the same note representation but augmented with the music dynamics (i.e., the relative volume of a note, provided here by MIDI input). DeepJ uses a method of representing beats and encoding several contextual inputs to guide the model to create music with better long-term structure. The model was trained on a dataset of 23 different music composers (across the Baroque, Classical, and Romantic eras), each corresponding to a particular artistic style. As such, various styles can be mixed during music generation by changing the vector representation to include the desired styles and normalizing the vector sum. The model also introduces global

conditioning to the Biaxial architecture, allowing the style embedding to be connected to another fully connected hidden layer with tanh activation.

For each time step at each note, the DeepJ model produces three outputs per note: play probability, replay probability, and dynamics, each with their respective loss functions. The system trains all three outputs simultaneously and play and replay are treated as logistic regression problems trained using binary cross entropy (as defined in the Biaxial LSTM architecture). In contrast to the biaxial model, dynamics are trained using mean squared error, and style conditioning is applied at every layer. Given that musical styles are not necessarily orthogonal to each other (although pieces from adjacent periods likely share many characteristics), the authors believe representing style using a learned distributed representation is more appropriate than a one-hot representation provided by the input.

During training, the pitch range of the MIDI input is truncated to four octaves, and the resolution is quantized to 16 timesteps per bar. Training is performed using stochastic gradient descent with the Nesterov Adam optimizer, with specific hyperparameters set for the LSTMs, style embedding space, and note octave convolution layer, alongside truncated backpropagation. Dropout is used as a regularizer during training to help the model recover from mistakes. After training, the model generates music by sampling from its probability distribution, with a coin flip determining whether to play a note. An adaptive temperature adjustment method is also implemented to prevent the model from generating long periods of silence, which is achieved by increasing the temperature of the output sigmoid functions proportional to the number of consecutive prior time steps of silent outputs the model has produced.

A subjective experiment was conducted using a user survey to evaluate the quality of DeepJ's music generation compared to the Biaxial architecture, which found that DeepJ produced

better-sounding music than Biaxial, with users preferring DeepJ's compositions 70% ± 5.16% of the time. Additionally, the study evaluates the style diversity of the model by surveying 20 individuals with musical backgrounds to classify music generated by the system as baroque, classical, or romantic. The results reveal that DeepJ produces music with styles approximately as distinguishable as those composed by humans, with participants correctly classifying the styles 59% ± 13.36% of the time. A t-SNE visualization also showed that composers from similar classical periods tend to cluster together, indicating that the model has learned the similarities and differences between composers and the dynamics of particular styles.

The authors found that the addition of dynamics made the output sound more like that of a human composer and improved the qualitative output of the model, highlighting the impact of dynamics on the quality of generated music. Aside from resulting in improved music quality, enhancing the Biaxial model through the additions of volume and style also solved the issues of style consistency in the unaugmented model. However, the authors note that the generated music lacks long-term structure and central themes, suggesting that future exploration with Reinforcement Learning (RL) methods from models such as Sequence Tutor or the use of adversarial methods may help train models with better long-term structure. The representation used by the Biaxial architecture is also expensive to train, so developing a sparse representation of music may be preferable.

2.2.6 Yachenko and Mukherjee, 2018

Yanchenko and Mukherjee's (2018) research delves into the efficacy of state space models – specifically Hidden Markov Models (HMMs) and their variants – for composing classical piano pieces reminiscent of the Romantic era, with a focus on the models' capacity to generate human-like compositions. While the models demonstrate considerable success in producing new pieces

with predominantly consonant harmonies, particularly when trained on original compositions with simple harmonic structures, the principal limitation lies in the absence of melodic progression. The study explores probabilistic time series models, focusing on piano pieces from the Romantic era to develop automated metrics assessing originality, musicality, and temporal structure. While HMMs have been widely applied to musical analysis and algorithmic composition tasks, their use in composing melodies remains limited.

Automated generation of compositions necessitates a music representation that can be generated by time series models, specifically variations on HMMs. The study employed two standard representations of compositions – sheet music and MIDI format –treating compositions as a symbolic sequence of note pitches over discrete fixed time steps. In addition to the standard HMM, the study considered various statistically richer Markov models, such as Autoregressive HMMs, hidden semi-Markov models, higher-order Markov models, Left-Right HMMs, and more. **Time-Varying Autoregression (TVAR)** models were explored to model the non-stationarity and periodicity of the note pitches for the original pieces. A total of 15 models were considered to generate compositions, with observable variables given by the pitch representation and parameter inference executed by the Baum-Welch algorithm. All models were trained on ten piano pieces from the Romantic period (either initially composed for piano or arranged for piano) and selected to have a range of keys, forms, meters, tempos, and composers.

The authors propose a set of evaluation metrics to assess generated compositions based on their similarity to original compositions in terms of musicality, originality, and temporal structure, focusing on the concepts of melody and harmony in music theory (specifically, Romantic era music). Musicality metrics focus on harmonic aspects, accounting for dissonant melodic and harmonic intervals and the distribution of note pitches throughout the composition. Likewise,

originality metrics, including empirical entropy, mutual information, and minimum edit distance, quantify the complexity and dissimilarity of generated pieces from the original training pieces. Temporal structure metrics – such as **Autocorrelation Function (ACF)** and **Partial Autocorrelation Function (PACF)** – measure decay correlations in a time series to capture a sense of melody. Lastly, Root Mean Squared Error (RMSE) is utilized to rank generated pieces for evaluation by human listeners and analyze general trends except for mutual information and edit distance metrics.

The generated compositions' quality was assessed through learned model parameters, quantitative metrics, and human listening evaluations. A case study of "Twinkle, Twinkle, Little Star" was initially conducted to examine the learned model parameters and general traits, followed by validation with the more complex "Fugue No. 2 in C Minor" by J.S. Bach (BMV 871). The authors note that the simpler piece allowed for a more straightforward evaluation. The HMM with five hidden states also captured the predominant harmonies even though the observed states were treated as a univariate series. Similar trends were observed for Bach's Fugue, with each hidden state emitting observed notes with the highest emission probabilities corresponding to predominant harmonies in the original piece. The primary goal was to generate new compositions that encapsulated Romantic music's harmonic and melodic characteristics, answering questions about which time series models and original compositions were best suited for generating realistic Romantic-era pieces. Generated compositions were numerically evaluated for originality, harmony, and melody, with the top three pieces evaluated by sixteen human listeners of varying musical backgrounds. These pieces included Chopin's "Piano Sonata No. 2", Movement 3 (Marche funebre), and Mendelssohn's "Hark! The Herald Angels Sing," both modeled by layered HMMs and Beethoven's "Ode to Joy" modeled by a first-order HMM.

A human evaluation consisting of two groups – one with a musical background and the other without – revealed several insights into the generated pieces based on various metrics. Notably, the pieces were perceived as having a more modern or contemporary composition style rather than evoking the Romantic era due to features like atonal chords and sustained base chords. The absence of phrasing or structure and the presence of repetitive elements were also highlighted as limitations. Chopin's "Marche funebre," built on chords of fifths, proved to be the most well-received among the generated pieces, likely due to its harmonic simplicity and easier resolution of dissonance. In contrast, pieces built on major chords composed of thirds, such as Mendelssohn's "Hark! The Herald Angels Sing" and Beethoven's "Ode to Joy," exhibited higher levels of dissonance. Although not entirely replicating the training pieces, the generated compositions displayed similarities in overall themes and structure. Two hypotheses emerged from these observations: generated pieces lacked overall or melodic progression, and models trained on harmonically simple pieces tended to produce more consonant pieces, which listeners preferred. To validate these hypotheses, an additional training piece from the Baroque era, Pachelbel's "Canon in D," was introduced, and a TVAR model was employed to capture more structure over time. The new compositions generated from this canon displayed fewer dissonant melodic intervals, while the TVAR model showed a slower decay in correlations, indicating better temporal structure. However, the human listeners did not consistently rate the TVAR-generated piece as superior in melodic qualities compared to the layered HMM-generated pieces.

The authors found that the state space models trained on Romantic-era pieces showed potential in generating music realistic enough to confuse human evaluators. The models demonstrated more significant success in modeling harmony than melodic progression, with simple harmonies and perfect intervals yielding more consonant pieces. Hierarchical structures,

such as layered HMMs, outperformed other models in generating music with reduced dissonance and slightly improved melodic progression. However, the generated pieces were perceived as more characteristic of the Modern era and lacked global structure or long-term melodic progression. To address these shortcomings, the authors suggest that future work explore natural language models for music and models with greater hierarchical structure to enhance generated pieces' harmonic and melodic aspects. Models capable of capturing longer-term structures – such as RNNs and Long Short-Term Memory units, which have demonstrated success in music composition-related tasks – represent promising avenues for improving the global structure of generated music.

2.2.7 Huang, 2019

Huang (2019) examines the multifaceted contributions of deep learning to music composition, focusing on generative models, recommendation tools, and model control at various musically meaningful levels, as well as non-musical aspects such as subjective qualities. The research encompasses four distinct projects – **AdaptiveKnobs**, **ChordRipple**, **Music Transformer**, and **Coconet** – that demonstrate the potential of deep learning in rendering music composition more accessible. AdaptiveKnobs employs Gaussian Processes to capture the complex relationships between sound synthesis parameters and perceived qualities, utilizing active learning to assist sound designers in refining intuitive controls. ChordRipple incorporates Chord2Vec to ascertain chord embeddings for creative substitutions and employs a Ripple mechanism to disseminate changes, thereby facilitating novice composers in devising adventurous chord progressions. The third project, Music Transformer, capitalizes on self-attention mechanisms to capture the self-similarity structure inherent in music, generating expressive piano compositions from scratch and allowing improvisers to develop coherent motives. Lastly, the fourth project, Coconet, harnesses convolutions to capture pitch and temporal invariance, generating a model

capable of completing partial musical scores and performing various musical tasks. Drawing inspiration from human composers' rewriting techniques, this model (recently used in Google's Bach Doodle) relies on Gibbs sampling to harmonize millions of user-composed melodies.

The creative process typically encompasses divergent and convergent thinking, with the former focusing on generating diverse novel ideas and the latter evaluating the appropriateness of ideas and adapting them to serve the creative goal. The interpretation of whether computers assist composers in the convergent or divergent thinking phase often hinges on the composer's certainty and the extent of exploration undertaken. Huang suggests that more effective tools can be designed by decomposing how machine learning aids the compositional process into three components – generation, recommendation, and control. By developing deep learning models to match the structure of music better, these models can **generate**, **recommend**, and provide **control** in the creative process, ultimately making music composition more accessible. Generative models that mimic the self-similarity structure in music can generate minute-long coherent musical sequences by reusing previous motives. Likewise, the quality of the generated music can be improved by adopting generative models that allow for rewriting. Recommendation practices, such as substitutions, can encourage novice composers to be more adventurous in selecting chords and assist machine learning models in automatically learning mappings akin to macro knobs.

Generation necessitates the development of a rich generative model of the domain to facilitate the production of novel ideas. By learning the data distribution and sampling from it, a model can approximate the data distribution more closely and thus produce output that exhibits the characteristics of the data. As the model acquires useful domain features, it can generate novel yet stylistically consistent examples with the data, potentially aiding in the divergent thinking phase by sparking new ideas and chance discoveries. Contrasting with unconditioned generation,

which ideally generates based on the data distribution, conditioned generation enables users to supply context, leading to samples drawn from a distribution that considers the context. The level of abstraction when specifying context may vary depending on the user's progression in the compositional process and the degree to which the idea is conceptualized. Context could range from a broad genre specification to a more specific partial score or even an existing melody for which the model can assist in determining accompanying chords.

Recommendation aims to ensure that novel and relevant material emerges by design rather than chance. While collaborative filtering is commonly employed in recommendation systems, the creative context often requires working with novel content and accommodating highly personal preferences. Conditioned generation enables recommendation systems to generate candidates typical of a specific music genre and relevant to the user's composition. For example, when harmonizing a melody, the system could provide harmonizations that complement the user's melody, solving a structured prediction problem involving inferring a sequence of chords that work well with the given context. The recommendation system could also suggest altering parts of the melody to facilitate more adventurous harmonizations. Moreover, generative models can be utilized to assess the typicality of a harmonization, ranking candidates from average to atypical based on the composer's preference, depending on whether they are in the divergent or convergent thinking phase. The system can rank candidates based on similarity to the composer's current solution, leveraging embeddings learned by generative models for computing similarity. However, Huang notes that it is crucial to consider that pre-existing ranking schemes may not always align with the composer's goals, who may want to rank output based on controllable attributes.

Control, the third component, seeks to enable users to influence generative outcomes based on attributes they find important. Conditioned generation on partial scores allows composers

to affect the outcome by modifying notes, which subsequently prompts the model to adjust other notes to preserve the solution's internal consistency – although this control's efficacy depends on users grasping the causal relationship. Additionally, users may prefer expressing their requirements at a higher level of abstraction, such as specifying the desired mood of a musical passage rather than individual notes. High-level controls empower users to brainstorm music based on emotional trajectories, and generative models conditioned on these trajectories can prototype sketches. In the divergent thinking phase, users can quickly explore different trajectories and their realizations, while in the convergent thinking phase, high-level controls facilitate quick adjustments to non-trivial attributes with minimal manual effort. To accomplish this, an adaptive mapping from high-level attributes to music reflecting those attributes must be established, with the ability to condition on additional musical context to ensure generated outcomes remain relevant.

2.2.7.1 AdaptiveKnobs, 2014

Huang et al. (2014) present "AdaptiveKnobs," an active learning-based approach for learning intuitive, personalized control knobs for sound synthesis, enabling users to bypass low-level controls and directly adjust high-level attributes of a sound such as its perceived "scariness" or "steadiness." The approach employs Gaussian Processes to model high-level features, mapping low-level controls to perceived levels of that attribute in the sound they create. However, the resulting functions are often multimodal. A new formulation for high-level control knobs is thus proposed to address this challenge, where the mapping is not fixed but dynamically constructed for each sound as a path in the low-level control parameter space that follows the gradient of the learned function. This formulation provides a novel objective function that measures the probability mass placed on the multiple paths that adjust a set of starting sounds. By tracking the

uncertainty on these dynamically constructed paths, active learning can be used to quickly calibrate the knobs by querying the user about the sounds the system expects to improve its performance. Empirical results demonstrate that the active learning approach learns high-level knobs faster than several baselines. As a result, the new formulation of knobs results in automatically constructed knobs capable of directly adjusting non-linear, high-level concepts.

2.2.7.2 ChordRipple, 2016

Huang et al. (2016) introduce the novel creativity support tool "ChordRipple" to assist novice composers in exploring a wider range of relevant chords and overcoming the challenge of going beyond common chord progressions. ChordRipple generates chord recommendations that aim to be both diverse and appropriate to the current context, allowing composers to select the next chord or replace sequences of chords in an internally consistent manner. The word2vec NLP neural network architecture is adapted to the music domain as chord2vec to create these recommendations. This model learns chord embeddings from a corpus of chord sequences, placing chords nearby when they are used in similar contexts and supporting creative substitutions between chords.

ChordRipple introduces a novel user interaction called Ripples to address the issue of maintaining internal consistency when changing one part of a sequence. This interaction automatically propagates necessary changes and presents the entire Ripple as a recommendation – thus making it easier for users to adopt and explore recommendations further from their own. The learned embeddings in chord2vec exhibit topological properties corresponding to theories in music, such as the arrangement of major and minor chords in the latent space in shapes corresponding to the **Circle of Fifths**. Empirical results from structured observations and controlled studies with music students demonstrate that ChordRipple helps them explore a broader

palette of chords, providing new ideas for moving forward in their compositions and leading to the adoption of more adventurous chords.

2.2.7.3 Coconet, 2017

Machine learning models of music typically approach composition chronologically, composing a piece of music from beginning to end, which can be challenging when supporting the human compositional process due to its nonlinear nature. Inpainting models are often trained to address this issue by masking out parts of the input and learning to fill in through reconstruction, allowing the model to learn different ways of factorizing music. Huang et al. (2017) propose the ensemble model "Coconet," which embodies all possible orderings, achieved by masking out the input with uniformly distributed mask-out sizes – later identified as an instance of **Orderless NADE (Neural Autoregressive Distribution Estimator)**. This approach makes it possible to fill in arbitrary partial scores with a generative model not tied to any causal composition ordering.

Gibbs sampling is employed as an analogy to rewriting, allowing composers to revisit and update earlier decisions based on new information. Orderless NADE models contain all possible conditionals (making them particularly suitable for Gibbs sampling) and can turn unconditioned generation into a series of simpler conditioned generation problems. A Convolutional Neural Network is used to parameterize the NADE model, resulting in a generative model that fills in gaps in incomplete scores. Empirical results show that both Orderless NADE and Gibbs sampling improve sample quality, with Coconet being applicable in a wide range of musical tasks such as unconditioned generation, melody harmonization, and partial score completion.

Music utilizes repetition and other self-referential techniques to establish long-term structure. However, Recurrent Neural Networks struggle to repeat motives that are not from the recent past due to their reliance on compressing history into a fixed-size hidden state. The

transformer, a sequence model based on self-attention, demonstrates improved results in generation tasks requiring long-range coherence, suggesting its suitability for modeling music. However, the original transformer architecture relies on absolute timing signals, making learning regularity based on relative distances, event orderings, and periodicity challenging. Existing approaches to represent relative positional information in the transformer modulate attention based on pairwise distance, which proves impractical for long sequences such as musical compositions due to high memory complexity.

2.2.7.4 Music Transformer, 2019

Huang et al. (2019) propose a new memory-efficient formulation of relative self-attention to tackle this challenge, aiming to model minute-long music while reducing intermediate memory requirements through a sequence of tensor re-indexing operations. Experimental results show that the transformer with this relative attention formulation can generate coherently at lengths longer than it is trained on, as well as minute-long compositions with compelling structure and coherent elaboration on given motives, while the original transformer deteriorates beyond its training length. The "Music Transformer" model achieves a state-of-the-art likelihood on language modeling of performed piano MIDI tracks in the Maestro dataset and performed significantly higher in listening tests than in the original transformer architecture.

A key challenge in generative modeling is capturing long-term structure, as these dependencies manifest themselves differently across domains, such as surface repetitions of motives in music and paintings, verse and chorus in lyrics, and latent context persisting and evolving at various levels of abstraction. Music Transformer significantly improved long-term coherence in music samples generated by neural networks, suggesting that self-attention could be well-suited for music. Huang notes that self-attention could capture repetition by functioning as a

differentiable self-similarity, and relative attention may also improve performance on language-based tasks such as translation and language modeling. The proposed formulation can also scale to longer sequences, improving image generation in Image Transformer, and has the potential to be used in tasks requiring longer contexts, such as document generation and dialogue generation where variable lengths are common. However, Music Transformer-generated samples do not yet exhibit structure on the musical form level (such as binary form), suggesting the need for hierarchical models that first generate a sequence of latents and then elaborate to produce details.

One approach to generating coherent music on the waveform level is to factorize the generative process into two stages. First, a music language model (such as Music Transformer) generates the notes, and then an audio synthesis model turns the notes into waveforms. The first stage requires high-quality music encodings in symbolic forms, such as in the MIDI protocol, and the second stage requires aligned sequences of notes and high-quality audio. The overall approach, called Wave2midi2wave, consists of training a suite of models capable of transcribing, composing, and synthesizing audio waveforms to generate coherent musical structure on timescales ranging six orders of magnitude (0.1 ms to 100 s). Similarly, Coconet's modeling approach demonstrates the potential for mixed-initiative music composition, allowing composers and the model to rewrite the score collaboratively. The flexible conditioning capability of Coconet enables human-in-the-loop interactions, such as mixed-initiative composition, where the composer explicitly requests the model to regenerate specific parts or the model proactively propagates the necessary changes to maintain the internal consistency of the musical score. This approach has also been extended to Music Transformer to support infilling.

Music Transformer and Coconet demonstrate the potential of building state-of-the-art generative models by matching the inductive bias of the models to the structure and generation

process of music. Huang suggests that a wider range of controls is needed to describe music at different levels of abstraction to design more effective human-computer interactions in composers' creative workflows, such as providing "knobs" in deep generative models for controlling the generation of music on multiple levels of abstraction. Furthermore, current systems are often designed implicitly for one-shot interactions without learning from past interactions or iteratively adapting recommendations to user preferences. Change propagation between multiple levels of abstraction may also enhance the overall coherence of a piece when introducing edits. Addressing these limitations could allow composers to communicate more easily with generative models and provide a more seamless creative experience.

2.2.8 Herremans and Chew, 2019

This research by Herremans and Chew (2019) addresses the issue of automatic music generation systems grappling with the critical challenge of long-term structure – an instrumental task in conveying a sense of musical coherence. Long-term structure generates coherence over larger time scales, spanning phrases to the entire piece. It encompasses the modulation of features like loudness, tension, and the utilization of recurrent patterns and motivic transformations over time. In response, the authors present the MorpheuS music generation system: a novel framework capable of generating polyphonic pieces with specified tension profiles and long- and short-term repeated pattern structures. This system employs a mathematical model for tonal tension to quantify tension profiles and state-of-the-art pattern detection and optimization algorithms to extract repeated patterns within a template piece.

MorpheuS utilizes a **Variable Neighborhood Search (VNS)** algorithm, an efficient optimization metaheuristic, generating music by assigning pitches that optimally fit the prescribed tension profile to the template rhythm, simultaneously hard constraining the long-term structure

through the detected patterns. While combinatorial optimization problems related to music generation are computationally complex, practical applications often use metaheuristics to find good solutions within a limited computing time. These techniques encompass population-based, constructive, and search-based algorithms. Population-based algorithms (such as evolutionary algorithms) have gained popularity in recent literature, while constructive metaheuristics build solutions from constituent parts, like ant colony optimization. Likewise, local search-based heuristics make iterative improvements to a single solution, with algorithms including iterated local search, simulated annealing, and variable neighborhood search.

The authors discuss two critical aspects of long-term structure that MorpheuS incorporates: tension profiles and recurring patterns. The tension profile is significant for generating program music, providing musical cues for the "story" of a piece. Research on musical tension has uncovered correlations between tension and emotion, with tonal tension being a powerful structural influence on emotions. Structural patterns in music contribute similarly to the listening experience. Various approaches have been employed to generate music with long-term structure, including concatenating rhythmic and melodic patterns using complex statistical learning methods such as recursive neural networks and integrating Markov models in optimization algorithms. MorpheuS expands upon these ideas by automatically detecting more complex long-term patterns in the template piece and applying the tension model to create music with long-term structure.

In previous research, Herremans and Chew (2016) developed a model for tonal tension based on the 3D spiral array model for tonality (Chew, 2014). The model consists of an outermost helix representing pitch classes and inner helices representing higher-level tonal constructs such as keys and chords, major or minor. These constructs are generated from their lower-level components, and any set of points in the spiral array can be weighted and summed to generate a

Center of Effect (CE), which represents the aggregate tonal effect of its components. The tension model in MorpheuS uses the pitch class and major and minor key helices, representing tonal tension as captured by the geometry in the spiral array.

A piece of music is divided into equal-length segments that can be mapped to clouds of points in the spiral array, allowing for the calculation of tonal tension within each musical fragment. The segment length, expressed in beats, can be set by the user (or an eighth note by default). Based on these clouds, three measures of tonal tension can be computed: **cloud diameter**, which captures the dispersion of the cloud in tonal space; **cloud momentum**, reflecting the movement in tonal space between consecutive clouds of notes by quantifying the distance between their CEs; and **tensile strain**, measuring the distance between the CE of a cloud and the position of the global key in the array. MorpheuS uses these three tension characteristics to evaluate musical output and match it to given template tension profiles. Users can also set the weights for each characteristic to reflect the most crucial aspect of tension.

Detecting recurring patterns is a critical aspect of musical engagement, where listeners recognize structure through repetition and relationships between different parts of a piece. MorpheuS thus employs recurring patterns from a template piece to establish structural elements in a new composition, focusing on symbolic music. The system also utilizes two state-of-the-art greedy compression-based classification algorithms – **COSIATEC** and **SIATECCompress** – both derived from the translation-invariant pattern discovery algorithms SIA and SIATEC. When applied to polyphonic MIDI files, the algorithms utilize a point-set representation of the score in pitch/time space and compute a compressed encoding in the form of **Translational Equivalence Cases (TECs)** of maximal-length patterns. SIATECCompress selects a subset of TECs covering the input dataset, while COSIATEC iteratively finds the best TEC and removes it from the input

dataset. Both algorithms produce TECs with high compression ratios covering a point-set, with COSIATEC generating more compressed encodings and SIATECCompress producing intersecting patterns that may be more relevant to music analysis.

The authors model music generation as an optimization problem to allow for the constraining of global structure and optimizing the music to fit a tension profile. MorpheuS begins with a template piece, maintaining the dynamics and rhythm as constants, and works to find a new set of pitches to minimize the objective function and satisfy the repeated pattern constraints. The objective is to find a solution that closely matches a given tension profile, either calculated from the template piece or manually input by the user. The Euclidean distance between the tension profiles is also calculated, and the sum of the distances between each of the tension measures forms the objective function (which is minimized).

The VNS optimization algorithm features two systems of constraints: soft constraints, which enable users to fix certain pitches in the solution with an additional term added to the objective function imposing a high penalty if a note is not set to the required pitch, and hard constraints, which are applied to enforce the patterns detected in the template piece, ensuring the recurrence of themes and motives in the output. A data structure stores the solution so that only the pitches of the original pattern occurrence need to be determined, with other occurrences automatically set based on the pitches for the original pattern and the set of translational equivalence vectors. An additional hard constraint is imposed on the pitch range for each track, set based on the lowest and highest occurring pitch in the template piece.

VNS has been successfully applied to various combinatorial optimization problems, including generating contrapuntal music and being modified to work with complex polyphonic piano music. The algorithm is driven by a local search strategy, which starts from an initial solution

and iteratively makes minor changes to find a better solution. MorpheuS' VNS system implements a first descent strategy, accepting a new solution as soon as it improves the value of the current solution. Three types of moves are implemented: **change1**, **swap**, and **changeSlice**. A change1 move changes a single note to all permutations in the allowed range to form a neighborhood, swap moves swap the pitch of any two notes in the piece, and changeSlice (an extension of change1) modifies two randomly selected notes in a vertical time slice. The VNS algorithm starts from a random solution and performs a local search using the neighborhood formed by change1, switching to a different neighborhood type when no improving feasible solution can be found. A perturbation strategy is utilized to escape local optima and continues the search for the global optimum, iterating until the stopping criterion is reached. The order of the different move types is also based on the increasing computational complexity of calculating the entire neighborhood.

The authors evaluated the system to examine the effect of the pattern detection algorithm on the musical outcome, along with the generated musical output compared to the original template piece and the efficiency of the optimization algorithm. Experimental results found that the length of the patterns and the pattern detection algorithm (either COSIATEC or SIATECCompress) significantly influence the resulting pieces. Infrequently repeated patterns (including longer ones) may lose sight of constraining the long-term structure. Short, frequent patterns can also overly constrict the generation process, causing it to converge quickly to the original piece. Likewise, the degree to which the chosen pattern detection algorithm restricts pitch patterns also significantly affects the musical outcome, especially the originality of the output pattern set (although both algorithms support settings for pattern length constraints).

Herremans and Chew investigated the effectiveness of the Morpheus architecture in the context of its tension-based objective function with pattern constraints, running the VNS algorithm

100 times on Kabalevsky's "Clowns" and Rachmaninov's "Étude Tableau Op. 39, No. 6" (both with SIATECCompress patterns). The results demonstrated a steep improvement in solution quality during the initial seconds of the algorithm's run, with slight improvements continuing after 2 minutes. This outcome was further evidenced by the high correlation coefficients between the tension profiles of the optimized and template pieces (indicating that the optimization algorithm indeed minimizes the objective function). The authors note that the optimization process significantly improved the observed musical quality between the random and optimized pieces – the optimized pieces were more tonal and exhibited long-term recurrent structures. However, they suggest that there is still room for improvement, such as using a machine learning approach to integrate transition probabilities in the current objective function and incorporating playability constraints.

2.2.9 Carnovalini and Rodà, 2020

Carnovalini and Rodà's (2020) research seeks to provide a comprehensive introduction to **Computational Creativity (CC)** and **Music Generation Systems (MGSs)**. CC is a multifaceted domain that aspires to endow computers with creative abilities, with music generation being one of its most prominent subfields. This field draws from diverse disciplines, making it challenging to establish clear objectives and determine the extent to which existing systems have addressed specific problems. CC also encompasses an array of goals, from understanding creativity in artificial intelligence to enhancing human creativity through computers; however, the multidisciplinary nature of the field results in various perspectives and objectives that sometimes make it difficult to discern the overall research direction. MGSs are particularly appealing within CC, offering many mathematical and computational representations without requiring explicit semantics like other art forms.

Understanding creativity and incorporating its definitions into generative processes is crucial (especially in CC) since observers will apply similar evaluative criteria to both human and machine-generated products. In recent decades, the evaluation of creativity has emerged as an important area of study, with various methodologies proposed for assessing creativity in computer systems. Turing Test-like approaches, which gauge creativity through human judgment, have given rise to methods such as the Consensual Assessment Technique. This approach (although not initially designed for CC) involves expert evaluation of artifacts compared to human-made ones. Likewise, other novel evaluation techniques and self-assessment frameworks such as the Lovelace Test, Creative Tripod, the FACE and IDEA models, and quantitative metrics continue to be criticized, revised, and expanded upon to promote practicality and realistic unbiased expectations.

Various algorithms and techniques have been applied to music generation, which the authors group into **seven main categories**: Markov Chains, Formal Grammars, Rule/Constraint-based systems, Neural Networks/Deep Learning, Evolutionary/Genetic algorithms, Chaos/Self-Similarity, and Agent-based systems. While some of these techniques – such as Markov Chains and Formal Grammars – have limitations regarding their creative potential, others, like Neural Networks and Deep Learning, have demonstrated impressive results in music generation. The authors also note that the creative process in music generation systems can vary significantly, from simple combinatorial creativity to higher levels of transformational creativity. Evaluation of MGSs also varies widely, from quantitative and qualitative assessments to live concert performances.

Rule/Constraint-Based Systems in music generation have been utilized since the early days of algorithmic composition, as music theory traditionally provides guidelines for the compositional process. These systems typically require some input material or use other methods (such as random generation) as a starting point refined through rules. Various rule implementations have been

explored, including Constraint Programming, which is well-suited to describe music theory rules. Challenges in designing these systems arise from the complexity of explicitly programming a sufficient number of rules, many of which do not have a formal definition in musicology literature. Moreover, there is a tradeoff between adding more rules for style accuracy and allowing for flexibility in accommodating different music styles. Rules and constraints can be perceived as either bounding or reshaping the conceptual space or as guidance in exploring the conceptual space. Hence, the use of rules can result in more efficient exploratory creativity, but it may also limit the output variety by reducing the size of the conceptual space or restricting the explored areas.

Deep Learning (i.e., neural network) techniques have gained popularity in music generation due to increased computational power and the widespread use of general-purpose GPU programming. Machine learning has significant implications for creativity, as learning machines have the potential to expand and change conceptual space and exploration methods, possibly reaching transformational creativity. However, critics argue that connectionist systems cannot achieve human levels of creativity, and the learning involved in a neural network may not be sufficient to pass the Lovelace Test. Although deep learning approaches can be seen as the closest implementation of transformational creativity to date, the black-box nature of these systems makes it challenging to understand the generation process, how the corpus information is used, and how to control the system's output (a fundamental capability by some definitions of creativity).

Genetic/Evolutionary Algorithms have also been utilized for music generation – i.e., any algorithm that iteratively generates and refines a population of solutions to a problem (inspired by natural selection in biological evolution). The authors postulate that to utilize Genetic Algorithms for music generation, one must first establish a way to generate random but suitable initial

solutions, evaluate the "fitness" of a solution, and mutate and recombine solutions. However, evaluating musical fitness is inherently challenging due to the subjective nature of music aesthetics. One of the most well-known Genetic MGSs, GenJam, illustrates various approaches to fitness function design through a jazz improvisation model that "evolves" human improvisation input. This evolutionary approach can be considered an exploratory creative process, wherein the fitness function directs exploration towards promising areas within the defined conceptual space. Still, a weak or nonexistent fitness function might result in purely combinational creativity, limiting the algorithm's effectiveness.

Musical compositions exhibit a degree of self-similarity in structure and spectral density, generally adhering to a $1/f$ distribution for pleasant-sounding pieces. Consequently, fractals and other self-similar systems have been employed to generate musical material, though their output is typically considered a source of inspiration for human composers rather than a final product. Various approaches include generating self-similar structures, utilizing tree structures produced by chaotic systems, and employing **Cellular Automata** (dynamic systems consisting of cells governed by a set of transition rules). However, melodies generated by these systems often require further human intervention due to their lack of aesthetic value. Despite their limitations, these models offer valuable insights into unconventional melodies.

Agent-Based Systems (software agents with autonomous perception and action capabilities) are advantageous when multiple agents cooperate within a single software or **Multi-Agent System (MAS)**. MAS architectures can effectively model certain musical behaviors, such as in George E. Lewis' computer music composition "Voyager," which simulates an orchestra improvising within a set of agreed-upon parameters (Lewis, 2000). Additionally, these systems can model social interactions by assigning specific characteristics or personalities to each agent.

This approach has been used to create systems where agents have specific emotions and can express them through singing, with other agents reacting accordingly. Cognitive models like the Belief-Desire-Intention architecture can also be employed in agent-based systems. As agent-based systems are a meta-technique rather than a specific algorithm, they cannot be framed from a process perspective; however, they provide exciting research directions in CC by exploring the influence of personalities and interactions on creative output.

The authors discuss challenges for MGSs beyond creativity and evaluation, including the concepts of Control, Narrative Adaptability, and Emotion. **Narrative Adaptability** refers to the system's capability to convey a sense of development in the generated music. In contrast, **Emotion** refers to the ability to convey specific emotions. However, both may be considered a particular instance of **Control** (related to the creative dimension of "Active involvement & persistence"), where the controlled features relate to emotional aspects or specific events in the narration. These challenges are crucial for creativity as they address the creative dimensions of "Social interaction and communication," "Progression & development," and "Intention & emotional involvement," which are essential for the affective perception of the machine (Jordanous & Keller, 2016). As such, the authors suggest that future research could explore how expressive features can influence each other and how to select specific expressive techniques to convey certain emotions, leading to more "Originality" in generated pieces.

The use of **hybridization** (i.e., employing multiple techniques to address the various challenges inherent in generating music) may facilitate a more comprehensive "Domain competence" and promote "Variety, divergence & experimentation." This approach facilitates the notion that no single method has demonstrated superiority over others nor the capacity to resolve all issues MGSs must confront. Hybrid models, such as Genetic algorithms, are often coupled with

other algorithms (e.g., Neural Networks and Markov Models) to achieve desirable results. Previous approaches include the Kinetic Engine, which employs distinct algorithms for designing agents capable of generating rhythm, melodies, and harmony, and the division of melodic phrase composition into successive steps that use different algorithms.

Although some MGSs focus on symbolic music generation, the ultimate fruition of music lies in sound, making Rendering (referring to the quality of the produced audio waveform) an essential aspect of the "Generating results" creative dimension. Researchers have explored various methods of enhancing rendering, such as expressivity, orchestration, and style transfer. Real-time rendering, though less explored, presents an alternative avenue for investigation. Composition as a creative task may be distinct from rendering, but the latter plays a crucial role in influencing human evaluators and determining the ultimate value of the generated music. The notion of Playing Difficulty, which pertains to the ability of an MGS to regulate the technical difficulty of the generated music, remains relatively unexplored but holds potential for enhancing "Domain competence," "Social interaction and communication," and the overall perception of computational creativity. Likewise, the authors emphasize the need for a holistic approach to music generation that benefits from diverse techniques to address different challenges, noting that a well-studied hierarchical hybridization could tackle many open challenges and facilitate comparisons between other methods.

2.2.10 Caren, 2020

Caren (2020) introduces TRoco, a generative algorithm for music composition featuring a novel method for the abstraction and analysis of musical structures based on the use of tension and release found in jazz music theory. This approach allows users to input the desired degree of musical tension over time which is applied in a generative algorithm that produces chord sequences

with the desired tension-release contour. Jazz theory is related to but largely distinct from the classical approach to music theory, and it can be generalized and abstracted such that a few key concepts can be used to analyze and create very complex structures. Moreover, it is especially powerful when considering various possible musical and emotional directions. While tension and release in classical music have been extensively studied, the author notes that research on tension and release in non-classical music theory is relatively scarce. However, there are well-established techniques in jazz theory for creating and releasing tension which are foundational to jazz composition and improvisation.

TRoco utilizes the key elements of the jazz harmony system taught at the Grove School of Music, which allows for the creation of music using conventional jazz chord symbols that can be decomposed into two components: a "root" and a "quality." This approach ensures that chords are analyzed in terms of their functions rather than their manifestations, allowing for the creation and analysis of musical structures instead of individual notes. The algorithm aims to generate musical structures that exhibit a desired tension/release profile using a "tension/release quotient" (TRQ) to quantify the degree of tension or release imparted by a change from the current state to a possible next state. The TRQ is an integer between -10 and 10, with negative values representing release and positive values representing tension. The TRQ is influenced by both vertical and horizontal factors, such as chord quality, chromaticism, root note interval, and notes in common with the previous chord.

TRoco's input is an array or continuous stream of TRQ values that represent the desired tension/release profile of the generation. The basic structure of the algorithm involves defining a set of possible states that the generation could output, calculating the TRQs of all possible following states, then selecting the state with a TRQ that most closely matches the target profile.

This process repeats until the generation is complete, and the best overall choice is selected at the end of the generation – although one limitation of the system is that the initial state must be chosen at the beginning of the generation. As such, the algorithm can be executed with multiple threads in real-time or near-real-time, and the weights of the various considerations within the TRQ function can be altered to provide different variations of generation types. TRoco thus shows promise as a tool for musical structure generation with the desired T/R profile, and its use of jazz theory concepts and quantitative methods could have broader implications for music generation and analysis.

Caren's initial implementation of TRoco was written in JavaScript (utilizing the open-source **Tonal.js** library for basic musical structures) for use in the generation of music for a simple video game. The game consists of a 3D environment with the player's location determining the target TRQ (calculated every second), where a higher tension is generated the closer the player is to a red sphere, and being closer to a green sphere generates a lower tension. After performing the TRQ calculation, the generated chords are passed to a series of arpeggiators that trigger samples to create the music heard in the game.

The author notes that given the flexibility of the TRoco algorithm, it may be applied to the generation of any musical structure by adapting the TRQ calculation to determine the tension or release of each possible musical choice, thus allowing it to be used in various applications (such as music creation for a game, movie, or VR experience). Additionally, using sentiment analysis to calculate TRQ over time from text sources could be applied to generate musical accompaniment for messaging conversations, e-books, or social network feeds. Caren also implies that the exploration of contemporary jazz theory in composition algorithms may offer other novel contributions to machine learning-based models in future research.

2.2.11 Micchi et al., 2021

Micchi et al. (2021) describe the process and results of their **Music Information Retrieval (MIR)** research team's experimentation with musical co-creativity to compose a song for the AI Song Contest 2020, where 14 international teams competed to produce an AI-generated piece in the style of Eurovision pop. The team's approach used AI to suggest melody, chords, lyrics, and structure, aiming to promote the co-creative process between humans and AI. The group comprises MIR researchers specializing in music modeling and analysis, focusing on structure, harmony, and texture, with diverse backgrounds in classical, folk, choral, and electronic music. They have previously researched form analysis of fugues and sonata forms using a co-creative music composition approach, where AI assists the human composer rather than substituting them. The researchers express that AI can help composers in two ways: as **automation**, to liberate the composer from some compositional sub-tasks and decisions, and as **suggestion**, where the AI suggests solutions for a set of compositional sub-tasks, but the composer has the final decision.

AI was used to generate a structure for the song, consisting of a sequence of section labels such as "chorus" or "verse," which was then used to condition the generation of the remaining four layers of the song: several chord sequences, lyrics, melodies, and hooks. The co-creative approach involved two successive steps: model/generate and select/compose. During the model preparation/content generation stage, the team modeled structure, harmony, and hook melody based on corpus data and iteratively refined the models based on subjective output evaluations. During actual song composition, the team filtered out some of the AI-generated material, gave the final word to chance, and selected among the remaining alternatives by rolling dice (used to stimulate creativity for the subsequent steps). Human interventions were recorded throughout the composition process to highlight the co-creative approach.

The team generated a structure template separately with a dedicated model using the SALAMI dataset, which provides structures for a large set of songs with a diverse vocabulary. However, the dataset was simplified by ignoring section durations and merging consecutive identical labels. Likewise, training and generation were performed by a random walk on a first-order Markov model learning the succession of sections and using special "start" and "end" states to track the starting and ending points. This model generated 20 structures, though the team discarded some and kept only 11 candidates – the ones marked from S1 to S11. A dice roll selected the structure S8, [intro, chorus, verse, bridge, verse, chorus, bridge, chorus, hook], which was utilized solely as a particular creative constraint (disregarding its abnormalcy).

A small neural network trained on the Eurovision MIDI dataset (consisting of over 200 songs) was used to generate pop song chord sequences. Due to the limited size of the dataset, the authors simplified the problem by generating short chord sequences for each section of the song rather than a long sequence covering the entire piece. Chord sequences were transposed to C major or A minor after key detection with the **Krumhansl-Schmuckler algorithm**, and the chords were then encoded using two one-hot encoded vectors. Finally, the model was trained on a single LSTM layer with 40 hidden units. Thirteen chord progressions were selected from the model-generated sequences to be modified by the authors for use in song composition. The resulting song was created using a piano voicing of the generated chords alongside recorded vocals.

The authors found that their model generated both tonal and atonal chord sequences, though the team considered some of the generated sequences common. While the model and generation process could have been improved, they felt the generated sequences were sufficient for use in the composition of a song. Several human adjustments were made to the generated sequences, including swapping the Bridge and Verse sequences and keeping the "unexpected"

chords suggested by the model; likewise, the harmonic rhythm was not generated with the chords but was chosen by the authors. They also noted an unintended variation in the piano rendering of the output piece, where a G major chord was played instead of the Gm generated by the model. Still, they decided to keep it as a manifestation of the co-creative approach.

Small language models were trained on the Eurovision Lyrics dataset (also comprised of over 200 songs) to generate lyrics; however, the initial lyrics produced lacked musicality due to the models being more targeted at semantics than metrics. To address this, the authors tested generating lyrics based on bigrams, allowing them to generate longer lyrics without the risk of plagiarism. The team then composed a simple and catchy melody for the Hook instrumental section, using AI to generate melodies based on a dataset of around 10,000 melodies from the **Common Practice Period** (c. 1600-1910, from "A Dictionary of Musical Themes"). The resulting melodies were limited to diatonic tones, which were expected to play a more significant role in pop songs. The authors also noted that their approach involved a tension between conformism and creativity – for example, generating lyrics based on bi-grams and using melodies from the Common Practice Period could be considered conformist. However, this approach allowed the team to focus on more creative aspects of songwriting, such as harmonization and instrumentation.

A human approach to melody generation was also favored; one person played the chords on the piano, and the rest of the team hummed along until a simple and catchy melody appeared. While the team decided not to use AI for the music arrangement and production component, human interventions were intended to be as discreet as possible to avoid the AI-generated content being pushed into the background. The lead vocal part was given to a human singer, and the piano/pad, bass, and strings were composed by humans. The team also used the LPX Drummer tool, virtual

instruments, and voice synthesis effects that emulate human performance. A professional sound engineer did the final mix and mastering.

The AI panel for the competition judged the songs based on the practical/creative use of AI, expansion of creativity, discussion on co-creativity, diversity, and collaboration. However, the authors note that the evaluation of AI-generated music is challenging because it is based on aesthetic and subjective criteria rather than objective metrics. The decision to collectively compose something according to what the team liked also introduced risks due to differences in musical backgrounds and biases, though the resulting song was well-received and reached 4th place in the competition despite these difficulties.

The authors opine that constraints can often help an artist achieve a more creative result, both in the case that the artist is human and machine. In this case, the use of AI may be regarded as a constraint that maximizes creativity by generating a set of candidate musical objects that limit the choice of the composer; however, the use of AI as a co-creative approach is a deliberate choice that reinforces human-computer interaction. Regardless, intellectual property is a challenge in using AI methods, given that deep learning often makes it difficult to know what influenced the output. Nonetheless, these co-creative approaches may be essential in the design of ethical AI music generation methods, and much research is still needed on both **AI as automation** and **AI as suggestion** paradigms for song composition.

2.2.12 Naruse, Takahata, Mukuta, and Harada, 2022

This research on controlled pop music generation by Naruse et al. (2022) proposes a novel method for sequentially generating a piece that allows for controlling each phrase length and the length of the entire piece. The approach adds PHRASE and BAR COUNTDOWN events to existing event-based music representations (which indicate the number of bars remaining in a

phrase and reflect user input on phrase lengths) to allow for control over the lengths of various musical structures, including the piece itself. The Auto-Regressive generation model adds these events to the generated event-token sequence based on user input for use as input in the next time step, allowing for natural transitions between phrases and controlled piece length.

Recent deep learning techniques for modeling music, such as Recurrent Neural Networks (RNNs) and Transformers, require representing a piece as a sequence of tokens, with event-based representations such as MIDI-like, REMI (REvamped MIDI-derived events), and CP (Compound Word Representation) being commonly used. Studies have focused on smaller musical units, such as phrases and sections, allowing for controlling attributes like melody, rhythm, and harmonic fullness. However, phrase-by-phrase generation policies have a limitation in natural transitions between phrases, and sequential generation policies were proposed to overcome this limitation. To improve the generation process, the authors added two new events to the REMI and CP models: PHRASE (indicating which phrase a bar belongs to) and BAR COUNTDOWN (the number of bars remaining in a phrase). These events allow the model to track which phrase it is generating and adjust the generation toward the end of the phrase and the piece.

The new models were trained and evaluated using the POP909 MIDI dataset, consisting of 909 piano arrangements of pop songs divided into three parts, obtaining 763 MIDI files after preprocessing. The evaluation method measures the length of each phrase, the entire piece, and the "naturalness" of the ending. The study conducts a subjective evaluation using two listening tests: one to divide a piece into phrases and another to evaluate the naturalness of the ending. To form their novel architectures, the authors augmented the REMI and CP models to include the PHRASE and BAR COUNTDOWN events, creating the "REMI + Ph&BC" and "CP + Ph&BC" models. Their evaluative study compares these methods with the existing REMI and CP music

representations. **Ablation studies** were also conducted by comparing these methods with ones with fewer events (e.g., REMI + Ph_{fewer}&BC and CP + Ph_{fewer}&BC). Eight methods were tested in total for comparative evaluations. Listening tests were conducted using the Human Computation service **Amazon Mechanical Turk** for subjective assessment, and qualification tests were performed to select participants who understood the tasks.

The proposed method improved the quality of generated music compared to previous transformer models, as seen in an objective evaluation based on the proportion of generated pieces recognized as valid musical pieces by human evaluators. In contrast, the subjective evaluation was based on the quality of the generated pieces assessed by human evaluators. The results showed that adding these two events effectively controlled the length of each phrase and the entire piece; thus, the proposed method achieved a higher proportion of valid pieces and higher quality scores than the previous models. These findings suggest that the proposed method can effectively generate musical pieces with designated phrase lengths while reflecting user input – a significant advancement in automatic composition technology. Additionally, the evaluations found that the BAR COUNTDOWN event is more effective in controlling length than the PHRASE event. However, the authors note that creating repetitions by designating the same phrase labels in the input was impossible. Future research should also address controlling other phrase attributes such as emotions, evaluating length diversity, and combining the proposed method with music theory to achieve more natural pieces.

2.2.13 Liu et al., 2022

Liu et al. (2022) present a novel **Multi-track Multi-instrument Repeatable (MMR)** representation for use in modeling symphonic music sequences, as well as a modified **Music Byte Pair Encoding (BPE)** algorithm for music tokens – both of which are compatible with all existing

symbolic music ensembles (e.g., piano solo, quartets, and pop bands). This representation is used in a novel transformer-based Auto-Regressive language model – dubbed "**SymphonyNet**" – with 3D positional embedding that considers the properties of semi-permutation invariance in symphonic music scores and avoids the loss of local structure that may occur with a 1D text-like sequence. The SymphonyNet model generates complex, coherent, harmonious, and novel symphonic music, which can be used as a pioneer solution for multi-track multi-instrument symbolic music generation. Additionally, the Music BPE algorithm tokenizes symbolic music at the subword level, automatically aggregating notes to intervals and chords without a predefined vocabulary. The authors also provide a symphony MIDI dataset of 46,359 high-quality MIDI files with multiple instruments and tracks to advance research on symphonic music generation with deep learning.

The authors address the challenge of generating symphonic music with proper instruments for different tracks by proposing a unique linear transformer decoder architecture (the basis for SymphonyNet) for instrument classification with joint-task training. SymphonyNet uses a 3D positional embedding to represent the semi-permutation invariant features of musical events and employs the linear transformer as the backbone of the model to handle the challenge of long symbolic music sequences. The model follows a decoder-only fashion, and different feed-forward heads are designed for the four attributes of musical events: Instrument, Track, Duration, and Event tokens. To increase the diversity of training data, SymphonyNet masks instrument information for every input token and allows the model to learn instrumentation from the output side with instrument loss and automatic orchestration without relying on instrument information as a predefined input source. Merging the same instruments in symphonic music is also challenging, which can damage the intrinsic structure of the music. However, the proposed MMR representation

solves this by modeling repeated instruments separately and capturing more heuristic musical information within a single track.

The Music BPE algorithm consists of two primary functions of special tokens: to represent the musical structures of notes and to control the model output during the inference phase. Several structural and controlling tokens, such as score, measure, chord, track, and position tokens, are designed to specify the general time-spatial features of notes. The algorithm can also aggregate note-related tokens, such as pitch and duration. Unlike natural language, notes played at the same position are permutation invariant, which poses an issue for tokenization using conventional methods such as standard BPE. Music BPE is thus based on the concurrence of notes rather than adjacency of characters and defines a maximum set of two or more notes with the same duration at the same global position and within the same track as a "mulpi" (multiple pitches), equivalent to a "word" in the BPE algorithm. The vocabulary list is initialized with 128 MIDI pitches, where each token represents a pitch set containing a single pitch. Each time all concurrent pairs of tokens in the bag of mulpies are located, the most frequent pair is merged into a new symbol and replaced in each mulpi until the vocabulary size reaches the maximum limit.

The authors also present a new symphony dataset, consisting of over 46,000 MIDI files of symphonic music, containing more than 279 million notes and 567 million tokens. This dataset was used in the training process for the SymphonyNet model, which uses the Music BPE algorithm to reduce the length of the input sequence and a permutation-invariant 3D positional embedding to improve model performance and generalization ability. For the training experiments, the model used 4096 as the input sequence length, 512 as the embedding size for event tokens, durations, instruments, and 3D positional embedding (all of which derive their size from the dataset after running Music BPE), 12 self-attention layers, 16 attention heads, and a batch size of 128 with an

AdamW optimizer. Music BPE also reduces the length of the input sequence and creates a vocabulary list of length 1,000 to present the top 5 merged pairs of tokens with the highest frequency.

An ablation study conducted by the authors found that the proposed Music BPE algorithm and permutation-invariant 3D positional embedding significantly improve the model's performance and generalization ability. These results suggest that SymphonyNet can generate realistic, unique symphonies and may serve as a foundation for future work in multi-track music production. The performance of SymphonyNet was also compared with a baseline model (trained with the vanilla positional encoding of GPT-3) and with the Multi-Track Music Machine (MMM) architecture – a previous model for symphony generation (Ens & Pasquier, 2020) – using both objective metrics and a human evaluation. This evaluation showed that SymphonyNet outperforms both models regarding coherence, diversity, harmoniousness, "structureness," orchestration, and overall preference. The authors plan to model long-term musical structures for future research, including parts or movements for more complex musical works.

2.2.14 Lu et al., 2022

This research by Lu et al. (2022) proposes MeloForm, which generates melodies with musical form using expert systems and neural networks. The method leverages the precise musical form control of expert systems and the musical richness learning of neural models. The expert system generates synthetic melodies with proper musical form by developing motives to phrases and then to sections with repetitions and variations according to the pre-given musical form. At the same time, the transformer-based refinement model improves the musical richness of the generated melody without altering its musical structure by conditioning on rhythm and harmony, differentiating sections, and using a phrase-level refinement strategy. Both subjective and

objective experimental evaluations demonstrate that MeloForm generates melodies with precise musical form control and outperforms baseline systems regarding subjective evaluation scores without any labeled musical form data. Moreover, the system can support various forms, such as sonata form, rondo form, theme-and-variation, and verse-chorus form.

Early expert systems for music generation, such as the music dice game and recent rule-based algorithms (e.g., dMelodies and Computoser), fail to consider the rules for generating melodies with musical form. Some also cannot develop melodies from motives to phrases. To overcome this, MeloForm's expert system generates melodies with a hierarchical structure of motives, phrases, and sections with repetitions and variations, while the neural network module refines the generated melodies to improve musical richness. This two-module combination allows for precise musical form control and rich melodic expression without needing labeled data – forming a new unsupervised learning architecture. The system is inspired by music theory, generating motives based on chord progressions and rhythm patterns, which are then developed into phrases and arranged into sections to create the final composition. The initial pitches of notes are selected from tones in corresponding chords, and embellishing tones are added to decorate the motif melody. Throughout the process, the model is guided by meta-information, such as scale, pitch range, tempo, and meter.

MeloForm also introduces three basic categories of development strategies, including sequence, transformation, and ending, which can be combined to create compound development strategies. To build relationships between phrases in the same section, the system generates a new motif similar to already generated motives or borrows rhythmic patterns from them. Then, repeated phrases are arranged with variations in a specific order to create higher-level structural units to form the sections. Directly generating new motives when building different sections is not always

musically expressive, so the system also introduces another method for establishing similarity and contrasting motives. The model can also borrow a random fragment from phrases in other sections to develop similarity and adjust the rhythmic patterns and pitch selection in the desired section to change the tension and form contrast.

While the expert system generates melodies with precise musical form, they lack musical richness due to being generated by hand-crafted rules. To address this issue, the authors propose a transformer model that refines the melodies while maintaining their form. They outline three design principles for refining melodies: refining phrases by phrases, conditioning on rhythm and harmony, and considering differentiation between sections. Refining phrases by phrases involves refining similar phrases in an iterative process instead of refining the entire melody simultaneously, which can lead to the loss of musical form information. The second principle involves conditioning the model with rhythm patterns, chord progression, and cadence to guide the refinement process and maintain the bottom-level music structure. Lastly, differentiation between sections is considered in the refinement process by controlling rhythm patterns and pitch distribution. Using these principles, the authors found that their model was effective on synthetic melodies from the expert system, demonstrating that it improves the musical richness of the melodies while retaining their intended form.

The authors trained the transformer-based neural network on the LMD-matched MIDI dataset, selecting files in 4/4-time signature normalized to C major or A minor – yielding 30,218 MIDI samples (containing 471,058 phrases, with a distinct set of 100,948 similar phrases). The MeloForm system was compared to three other baseline music generation systems (**Music Transformer**, MELONS, and POP909_lm), and the authors found that it outperforms them in structure, thematic quality, richness, and overall quality. MeloForm is also highly controllable,

allowing precise musical form and pitch distribution control. Given that MeloForm is designed with several components, including a melody refinement neural network, development strategies from expert systems, and a fine-grained rhythm pattern condition, the authors conducted ablation studies to verify the contributions of these components to the system's overall performance. Subjective evaluation metrics were defined for the system, and 10 participants evaluated 30 samples based on these metrics, which include structure, thematic quality, richness, and overall quality.

The experimental results indicate that MeloForm achieves 97.79% accuracy in musical form control and outperforms the baseline systems in all evaluation metrics as measured through the subjective evaluation. The system is also shown to be scalable, with extensions demonstrating its ability to generate long compositions and adapt to different styles of music. While MeloForm is found to outperform other baseline systems and is shown to be highly controllable, the authors note that there still exist gaps between melodies generated by the system and those composed by humans. They suggest that their proposed model could be used in various applications, such as music composition, education, and entertainment, and could potentially improve the efficiency and creativity of music generation. However, the lack of evaluation metrics for assessing the musical quality of the generated melodies and the need for more diverse training data present shortcomings for the current system. To support future research in music form modeling, the authors plan to release the dataset of synthetic melodies generated by the expert system and the refined version from their transformer model, and also intend to explore the generation of intro, outro, and bridge parts and investigate the arrangement generation with musical form to complete the composition.

2.2.15 Neves, Fornari, and Florindo, 2022

Neves et al. (2022) present a generative music model that can synthesize songs based on values of valence and arousal corresponding to perceived sentiment. The model aims to enable users without musical backgrounds to translate their perceptions into music suitable for their artistic aspirations. The proposed approach employs a **Transformer-GAN** and complements the teacher forcing-style (or **Maximum Likelihood Estimation, MLE**) training with an additional adversarial signal from a Discriminator network. Evaluations using automatic metrics and human feedback demonstrate that the proposed model competes with the current state-of-the-art model, having fewer parameters and using a simpler symbolic representation of music.

Transformer-based models (such as Music Transformer and MuseNet) often employ self-attention, **Generative Adversarial Networks** (**GANs**, including Midinet and MuseGAN), and CNNs to generate symbolic music that exhibits long-term structure and spans multiple musical genres and artists. Some models also incorporate emotional content as a conditioning factor for generating music. Additionally, the authors note that the representation used to train these models – typically either Compound-Word or **Revamped MIDI-derived Events (REMI)** – plays a critical role in the quality of the generated music, especially for contemporary styles such as Pop or Hip-Hop.

Previous (unconditional) models are limited in their ability to engage with users and take their guidance during the generative process. Additionally, generating music that maintains coherence across longer timescales while still capturing the local nuances of human-like sound remains challenging due to computational requirements and training limitations. To address these challenges, the authors propose a generative model of symbolic music conditioned by human sentiment data. Conditioning factors such as emotions, sentiment, and mood tags associated with

each chord in a progression are applied to guide each step of the generative process and steer the generated music toward a desired affective state.

The authors propose a transformer model that employs the Compound Word representation to generate songs conditioned by affective states. The model comprises a Generator and a Discriminator, both transformers with linear versions of the Attention Mechanism. The Generator has two objectives – to predict missing items in sequences from the dataset and generate sequences similar to the training set from scratch. Likewise, the Discriminator takes each sequence as a whole and tries to determine if it is real or generated. Two datasets were used to train the model: AILABS Pop1K7 (MIDI transcriptions over 108 hours of pop song covers on piano) and EMOPIA (1,087 music clips with emotion labels from 387 pop songs). Both datasets consist of piano covers of pop songs automatically transcribed into MIDI files (Huang et al., 2021; Hsiao et al., 2021). The Generator was pre-trained until convergence through the teacher forcing method, and then both the Generator and Discriminator were trained simultaneously on both datasets.

The proposed model is designed to prioritize local structure and incorporate prior knowledge about musical structure into the generated sequences. A local loss function was used to inform if each phrase or part of a phrase is realistic. Similarly, a global prediction unit was used to encapsulate the overall quality of the sequence and its ability to convey the desired emotional state. The networks were trained via a combination of the teacher forcing objective (MLE), the **Relativistic GAN (RSGAN)** objective with gradient penalty, and 26,000 global optimization steps. After training, the model was shown to generate sequences that exhibit high-quality and diverse musical content, significantly improving over previous approaches to music generation using GANs.

The authors evaluated the proposed Transformer-GAN model by comparing it with a state-of-the-art generative model, the Compound-Word Transformer, and a Vanilla Transformer model that was not trained with the adversarial scheme. Automatic evaluation metrics, including Pitch Range, Number of Pitch Classes, and Polyphony, were calculated using the **Muspy** Python library. A human evaluation was also conducted using a 5-point Likert scale for six characteristics: Arousal, Human-likeness, Originality, Overall Quality, Structure, and Valence. The evaluations found that the Transformer-GAN model performs superiorly to the other models on two of the three automatic metrics and is competitive with the state-of-the-art model concerning four of the qualitative metrics (Human-likeness, Originality, Overall Quality, and Structure).

Based on the qualitative evaluation, the authors found that all models have more difficulty capturing valence than arousal, indicating that factors impacting valence are more difficult to model for neural networks. However, the proposed Transformer-GAN surpasses the simple transformer in situating the generated excerpts more strongly to the side to which they theoretically pertain. This result suggests that Transformer-GAN is a promising approach for music generation conditioned by emotion despite using a simpler representation than the state-of-the-art model. Given the potential of using GANs for music generation conditioned by sentiment (despite training limitations), these architectures show promise for future research in guided automatic composition and music generation based on specific sentiments.

CHAPTER 3
METHODOLOGY

This chapter outlines the design of the proposed dataset, detailing its structure and the components involved in feature extraction and data preparation. It includes the neural network and system architecture, along with the research methodology, population sampling, data collection procedures, and their rationale and validity.

3.1 Proposed Dataset

I propose the novel "**CHORAL**" dataset (**C**horal **H**armony **O**ptimized **R**epository for **A**I **L**earning), a collection of 1,000 classical choral music pieces I curated, specifically arranged in four-part (SATB) voicing without accompaniment (see *Appendix B*). This dataset – provided primarily in MIDI format and collected from free and open sources (ChoralTech, 2023; Hodgson, 2005; Alvarez, 2009) – is a comprehensive resource that also includes pre-processed tokenized versions in (Python) **Pickle** files, facilitating ease of use in AI applications. To enhance versatility and usability, the dataset features numerous augmentation sets (see *Chapter 4.1.1*) and splits by individual voices, a component designed for in-depth analysis and experimentation – although these were not utilized in the final system implementation but were retained for future research (see *Appendix B.1*). To standardize and simplify the learning process for the AI, a transposed variant of the dataset is also included, wherein all pieces are uniformly transposed to C major or A minor. This standardization addresses key variability in the original compositions and serves to streamline the training process, ensuring more consistent/focused learning outcomes from the AI.

3.2 Requirements and Specifications

All components of the Choral-GTN system were implemented in the Python (3.10) programming language (see *Appendix E*). The complete project utilizes numerous open-source frameworks for machine learning, computational musicology, and statistical analysis, including:

- **TensorFlow** – A library for machine learning and artificial intelligence tasks primarily focused on deep learning models (Yegulalp, 2024).
- **Keras** – A subset of TensorFlow that provides a simple interface for deep learning models.
- **Scikit-learn (sklearn)** – A library used for scientific and technical computing, including modules for integration, linear algebra, optimization, and statistics (Groeneveld, 2023).
- **NumPy** – A library for scientific computing that provides support for large, multi-dimensional arrays/matrices and high-level mathematical functions (Jha, 2023).
- **SciPy** – A scientific computation library that extends NumPy for scientific computing.
- **Matplotlib** – A plotting library for static, interactive, and animated visualizations.
- **Music21** – A toolkit for **computational musicology**, allowing for the analysis and manipulation of musical data (Cuthbert & Ariza, 2010).
- **Mido** – A library for working with MIDI ports, files, and messages, useful for creating, parsing, and editing MIDI files (Agnew, 2019).
- **PrettyMidi** – A library for analyzing and modifying MIDI files, with advanced functions for handling notes, instruments, and control changes (Raffel, 2023).
- **Pandas** – A data manipulation and analysis library for tabular (relational) data.
- **Pickle** – A module for serializing and deserializing object structures (Selvaraj, 2023).
- **StatsModels** – A library that provides classes and functions for many different statistical models, including for conducting statistical tests and data exploration (Pandata, 2023).

3.3 System Design

I propose the "Choral-GTN" system, a sophisticated AI-based framework designed for generating choral music, consisting of three integral components: the **Generative [Choral] Transformer Network (GTN)**, the **Music Generator Callback Interface**, and the **Rule-Based Post-Processing System** (see Figure 3.1). At its core, the GTN architecture utilizes a set of inputs, including a temperature value (typically within the range of [0, 1)), a predetermined sequence length (e.g., 200 tokens), and initial note and duration tokens for each voice part (**S**oprano, **A**lto, **T**enor, **B**ass). This input configuration allows users to either start generation from a basic template (e.g., [**S**:START, **A**:START, **T**:START, **B**:START] for notes and corresponding duration tokens, [S:0.0, A:0.0, T:0.0, B:0.0]) or use a custom generation seed for more controlled compositions.

As the GTN model generates tokens, the outputs are interfaced with the Music Generator (Callback) system. This system plays a pivotal role in refining the output by filtering out irrelevant or corruptive tokens like START tokens or notes with zero duration, ensuring the integrity of the MIDI files. Each voice's generated notes and durations are then seamlessly compiled into MIDI streams using the **music21** library and written to a MIDI file. Following generation (of an ideal piece), users can employ the post-processing script as a "music theory corrector." This script applies rule-based adjustments for voice leading and counterpoint errors in the AI-generated MIDI, offering additional options for key or tempo modifications. Users can either select these parameters manually or allow the system to assign them randomly based on the common distributions observed in choral music. The final output, a post-processed MIDI file, provides a refined version of the AI-generated piece, ready for further review, comparison, or conversion into sheet music using various **DAW** and music notation software. This multi-component system design ensures the generation of musically coherent pieces and the flexibility for user-guided refinement.

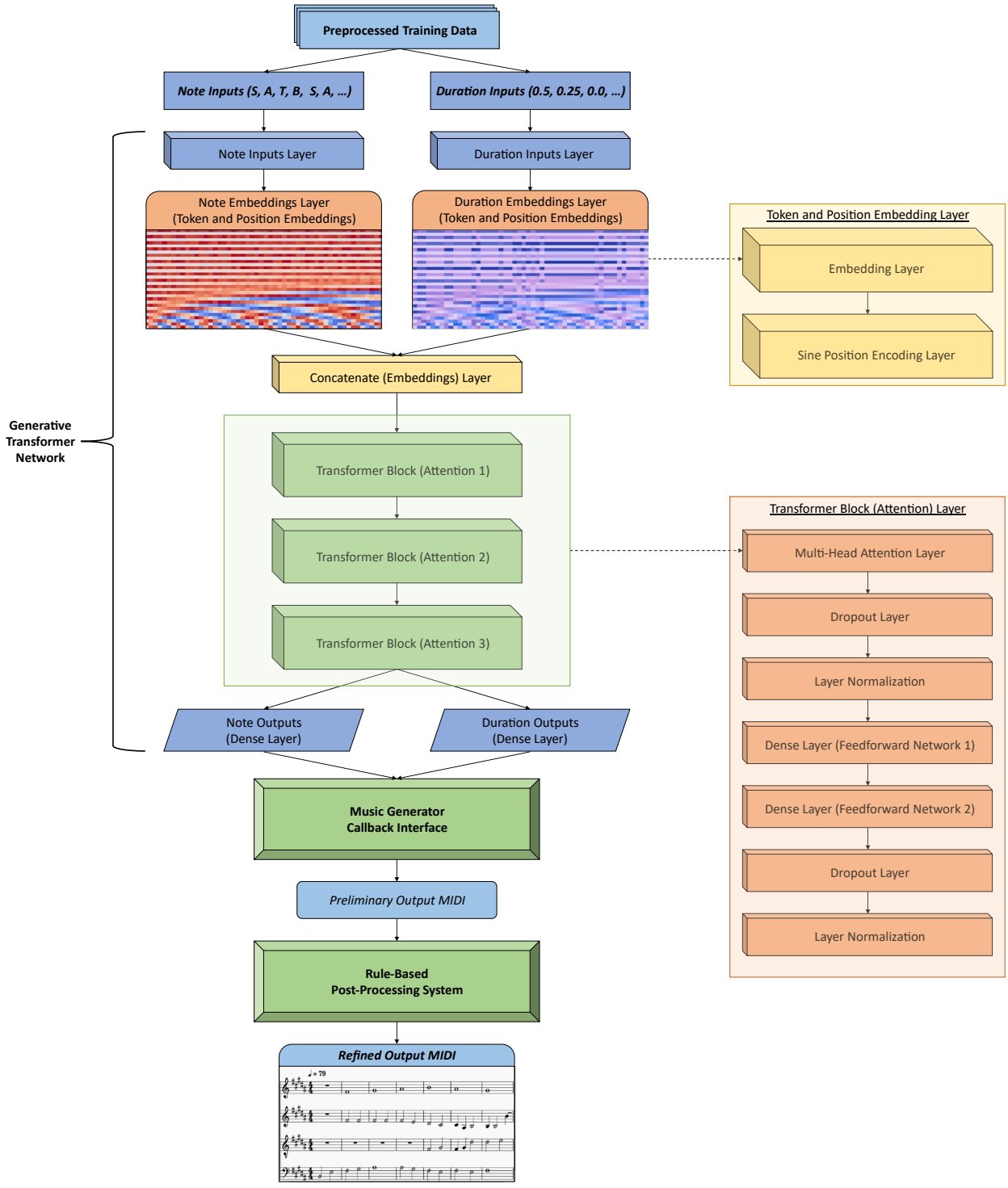


Figure 3.1: Choral-GTN System Architecture Diagram

3.4 Research Method and Design Appropriateness

3.4.1 Research Method

The research method employed in this study was a blend of **quantitative** and **qualitative** approaches, primarily focusing on the application of machine learning techniques to generate choral music and the subsequent evaluation of these compositions through a survey (Scribbr, 2017). This mixed-method approach was chosen for its ability to not only harness the computational power of AI in music generation but also to capture the nuanced human perception of these AI-created pieces. The quantitative aspect involved training a deep learning benchmark model on the CHORAL dataset and analyzing the output using statistical methods, while the qualitative component comprised a survey to gauge the public's ability to distinguish between AI-generated and human-composed music (see *Chapter 5.2*).

3.4.2 Research Design

The research design for this dissertation was carefully structured to address the study's objectives comprehensively. The first phase entailed the development and training of the Choral-GTN model using the CHORAL dataset, focusing on generating realistic four-part (SATB) choral music (see *Chapter 5.1*). This was followed by a critical evaluation phase, where the AI-generated compositions were presented to a diverse pool of respondents via a structured survey (see *Chapter 5.2*). The survey's design, including a "musical Turing test" with a 6-point Likert scale and demographic questions, was instrumental in capturing a wide range of perceptions and preferences across different listener backgrounds (see *Appendix D*). This two-pronged design approach – creating with AI and evaluating with human listeners – provided a holistic understanding of the capabilities and reception of AI in the field of choral music, reflecting upon the initial research questions (see *Chapter 1.2*).

The appropriateness of this research design lies in its alignment with the study's dual aims: to explore the potential of AI in creating realistic choral music and to understand how such music is perceived by various listener groups (i.e., groups divided by musical experience). The methodological choice of employing AI for creation and human respondents for evaluation ensured a balanced investigation into both the technical and perceptual aspects of AI-generated music (Steinberg, 1994). Additionally, the design's adaptability, evidenced by the inclusion of diverse musical elements in the dataset and the comprehensive survey structure, allowed for a thorough exploration of the research questions, making it well-suited for this innovative intersection of technology and musicology (Asenahabi, 2019).

3.5 Population and Sampling

This section discusses the population demographic of the model evaluation survey and the sampling strategy employed in obtaining respondents for this research.

3.5.1 Population

The population targeted for this study was broadly defined to include individuals from diverse backgrounds with varying levels of musical expertise. This encompassed classically trained **music professionals**, such as teachers and performers/composers, **students** with formal (classical) music education, **amateur musicians** (e.g., hobbyists), and general **music enthusiasts** with minimal or no formal training in music (see *Appendix D*). This wide demographic range was chosen to ensure a comprehensive assessment of the AI-generated music's perception across different levels of musical understanding and experience. The diversity within the population was critical in evaluating the AI model's effectiveness in creating music that resonates with or convincingly emulates the appeal to a broad audience, thereby providing insights into the model's success in mimicking human compositions (see *Chapter 5.2.1*).

3.5.2 Sampling

The sampling strategy employed in the study was a combination of convenience and purposive sampling methods (Shantikumar, 2018). Participants were selected on a voluntary basis from fellow teaching and administrative staff, university colleagues, and through the **Prolific.com** platform (see *Chapter 5.2*), facilitating access to a wider and more diverse respondent pool. This approach ensured the inclusion of individuals with a specific interest or background in music, as well as those from the general public, thereby capturing a broad spectrum of perceptions. While this sampling method may not provide a statistically representative sample of the entire population, it was deemed appropriate for the exploratory nature of this study, focusing on gathering varied perspectives and insights on AI-generated music. The chosen sampling approach balanced practicality with the need for diverse respondent experiences, making it well-suited for the study's objectives in understanding the range of reactions to AI-composed choral music.

3.6 Data Collection Procedure and Rationale

The data collection procedure for this study was designed to ensure a thorough and effective gathering of information pertinent to assessing my model's performance in generating choral music. This process involved two primary components: first, the collection of AI-generated music samples from the GTN model (after post-processing), and second, the administration of a structured online survey to gather participant responses (see *Chapter 5.3*). The AI-generated samples were systematically derived from the model trained on the CHORAL dataset, ensuring that the outputs were representative of the model's capabilities. Subsequently, the online survey, hosted through Pollfish.com and supplemented by responses from Prolific, was utilized to capture a wide array of perspectives from the target population. This survey method was selected for its efficiency, ability to reach a diverse audience, and its effectiveness in collecting qualitative and

quantitative data. The rationale behind this approach was to balance the technical evaluation of AI-generated music and the subjective interpretation of these compositions by listeners from varying musical backgrounds, thereby providing a comprehensive understanding of the model's success and areas for improvement (see *Chapter 6.2.1*).

In aligning with the study's objectives, the blended method of convenience and purposive sampling (from colleagues and Prolific.com) facilitated the inclusion of a broad spectrum of individuals across varying degrees of musical expertise. The deployment of Prolific proved essential in supplementing the sample to ensure reliability, statistical power, and a balanced representation across defined demographic segments. This approach allowed for the deliberate selection of participants to fill gaps potentially left by convenience sampling, particularly in achieving a diversified sample reflective of the study's target population. However, this method is not without its drawbacks. Convenience sampling, while efficient, carries the risk of sampling bias – potentially skewing the sample towards more accessible individuals within the researcher's network, which may not accurately represent the broader population (Shantikumar, 2018). Purposive sampling, aiming to rectify this through targeted selection, also faces challenges in ensuring the sample's comprehensive representativeness.

The survey's design was carefully considered to gather insights effectively, utilizing 60-second audio samples, rendered uniformly to control for production quality (using the "Muse Choir" sound font; see *Appendix D*), and presented in a randomized sequence to participants. Specifically, the sequence commenced with two AI-generated samples, followed by a human composition, succeeded by the most convincing AI-generated piece, and concluded with a sample unmistakably composed by a human. This setup aimed to neutralize potential sequence bias, aiming for a balanced representation of AI and human compositions throughout the survey and

ensuring that perceptions were shaped by the content rather than the presentation order. The use of Pollfish.com to host the survey (see *Chapter 5.2*) also ensured that participants were required to listen to each audio sample in its entirety before proceeding, as well as preventing partial or repeated submissions. However, the reliance on convenience and purposive sampling methods introduces inherent limitations in external validity, such as potential sampling bias and non-representativeness (e.g., the variation in sample size across demographics). While these methods enable focused recruitment and the attainment of a wide array of viewpoints, they may also lead to over- or under-representation of certain groups, impacting the generalizability of findings. The study thus employed strategies such as varied recruitment sources and controlled randomization in survey presentation to mitigate these effects.

3.7 Validity

This section highlights the internal and external validity of my research methods regarding the analysis of this dissertation's observational study.

3.7.1 Internal Validity

Internal validity in this study refers to the degree to which the research design and methods accurately capture and reflect the intended phenomena (Streefkerk, 2019) – specifically the ability of AI to generate realistic choral music and the perception of this music (as realistic) by various listener groups. To enhance internal validity, the research design incorporated controlled variables, such as the uniformity in the presentation of the music samples and the standardized format of the survey questions (see *Appendix D*). The use of a pre-processed and consistent dataset for training the AI model, along with a structured approach to generating music samples, helped in mitigating potential biases in the composition process. Moreover, the survey was designed to minimize subjective interpretation and leading questions that could influence respondents' answers.

However, it is important to acknowledge that the subjective nature of music perception and the diversity in respondents' musical backgrounds can introduce variability in the survey responses, which has the potential to affect the study's internal validity (hence this study's research question on music perception of AI compositions). Measures were taken to minimize these effects (e.g., concealing music origins, randomizing sample order, and subjective measures), but they cannot be entirely eliminated in a study of this nature, where subjective evaluation plays a significant role.

3.7.2 External Validity

External validity, or the extent to which the study's findings can be generalized to other contexts (Eubank, 2022), was a key consideration in my research design. The broad demographic range of the survey participants, including individuals with varying degrees of musical expertise and backgrounds, enhances the external validity of this study. This diverse sample allows for the generalization of the findings to a wider audience, providing insights into how different groups perceive AI-generated music (see *Chapter 6.1.3*). However, the specific focus on classical choral music and the use of a particular AI model or architecture places limitations on the applicability of the results to other musical genres or intelligent systems. Additionally, the use of online platforms for participant recruitment, while effective in reaching a wide audience, may not fully represent the global population's views. Future research could expand the external validity by exploring other genres, incorporating different AI models, and targeting a more geographically diverse participant pool (see *Chapter 6.2*).

CHAPTER 4
IMPLEMENTATION

In this chapter, I will discuss this study's approach to preprocessing and preparing the dataset for training the system, normalizing the token set, and the overall architecture and implementation of the generative and post-processing models.

4.1 Data Preparation

The process of preparing the dataset for training was both iterative and multifaceted, reflecting the necessity to accommodate a variety of ML and DL architectures in an attempt to identify the most efficient model for generating realistic choral music. The initial phase of data preparation involved converting all 1,000 MIDI files in the dataset into a tabular (CSV) format. Each file, representing one of the four voice parts (soprano, alto, tenor, bass), was meticulously processed to include columns such as **event**, **velocity**, **time**, **tempo**, **time_signature_count**, **time_signature_beat**, and **key_signature** (see *Appendix E.3*) – all of which were numerically encoded to achieve normalization. While this tabular approach provides valuable insights into the requirements for training preliminary ML models, tabular data is not optimally suited for the complexities inherent in most deep learning architectures, especially due to feature heterogeneity, dimensionality, and lacking temporal/spatial structure (Njagi, 2022). As such, the data preparation strategy was shifted towards tokenizing the musical data as strings. This approach significantly enhanced the data's compatibility with DL frameworks, enabling a more nuanced and effective training process (at the expense of data normalization, which was found to be unnecessary).

4.1.1 Dataset Preprocessing

The preprocessing stage of the "CHORAL" dataset was aimed at refining the raw MIDI files into a format conducive to deep learning applications (see *Appendix E.3*). This process initially involved employing the **music21** library to iterate through each MIDI file, applying the *chordify* method to facilitate chordal reduction across all voice parts (Cuthbert, 2023). This method aimed to simplify the musical information by combining similarly timed notes across different voices into chords. For each song, cumulative note and duration arrays are appended with starting tokens "START" and "0.0", respectively. Meta-tokens indicating tempo, key signature (including mode), and time signature, with respective suffixes for clarity (e.g., "65BPM", "F#:minor", and "3/4TS" respectively) are added to the note/duration lists, all coupled with a "0.0" duration to maintain consistency. The actual notes are then concatenated as strings containing the note and octave/pitch class, such as "C#5.A4.E3" for a chord where three voices sing a note at the same time; all notes not present within a chord were appended similarly (e.g., "Db4"), with "rest" for any individual rests, and durations appended in quarter length/fractional form (i.e., "1.0" for a quarter note, "4.0" for a whole note, "0.5" for an 8^{th} note, etc.).

This chordal reduction format is very inflexible in controlling the output – especially when generating new MIDI files in the same format. The outputs typically generated in the form of **piano reductions** and thus lacked clarity on which notes corresponded to which voice. These pieces were also uninteresting as the generated music was entirely chordal (i.e., **homorhythmic**), i.e., all voices moved at the exact same pace (typically all whole or half notes) with no counterpoint – although some of the harmonies were musically appealing. After iterating through all songs in the dataset, tokens were concatenated in groups of a specified sequence length (i.e., 50 tokens) separated by

spaces. Lastly, the complete note and duration lists were saved as Pickle files for simplicity of storage while retaining the serialized representation of the data (Selvaraj, 2023).

As a result of these generation issues (found through trial and error), the preprocessing method was modified to utilize a dictionary containing lists for each individual voice. This provided a much more fine-grained and concise representation of the data, as each note token was prefixed with the voice the note corresponded to (such as "S:Ab5" for a soprano note) – ensuring the notes generated for each voice respect the typical note range and tessitura of each voice part. Finally, I perform the same grouping of tokens into larger ones of the specified sequence length, and rather than saving a single Pickle file for notes across all voices and another for all durations, the notes and durations are instead saved in separate files for each voice (i.e., 8 Pickle files in total). It is also worth noting that due to limitations in the project's version control system (Git) and file size restrictions in GitHub, these large files (often over 200 MB) had to be split into multiple pieces – which I dubbed "**Pickle Slices**" – as a means of retaining structure in the project's git repository and ease of moving and loading the data after preprocessing. I also found that most models were not overfitting on the data (likely due to the concision of feature selection and thus preventing the "curse of dimensionality"; see Figure 4.1), implying the training data did not necessitate additional normalization (beyond token uniformity) or dimensionality reduction methods (Barla, 2022).

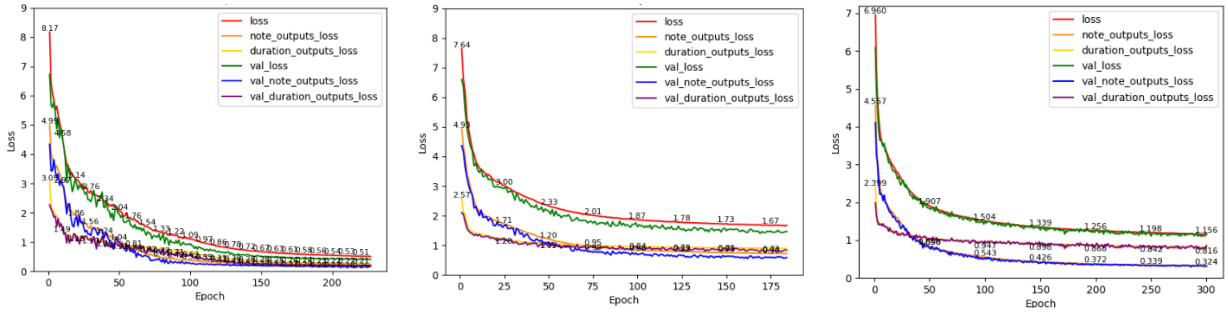


Figure 4.1: Old model training history demonstrating early convergence without overfitting

Following the preprocessing of the training data, I created another method for performing a meta-analysis on the data to store probabilistic metadata for each voice part collected from their respective set of MIDI files. This function performs a similar iterative preprocessing for each MIDI file like that of the training dataset, providing the following information (for each voice):

- Average duration (in seconds) before the first note (initial entrance)
- Probability distribution of time signatures (as both counts and beats) and key signatures
- Min, max, and average tempo

As such, this data provides us with the baseline for the sampling used in randomly picking a probable key and/or time signature during post-processing (see *Chapter 4.2.3*).

I also attempted processing other training sets, such as a set for voice introductions (e.g., the first 8 or so measures of each voice part), as well as generating four augmentations of the original training set (to form a total set of 5,000 MIDIs) using the following modifications:

1. Shift all notes down 6 half-steps (tritone), slow down all durations by 30%
2. Shift notes up 4 half-steps (major 3rd), speed up all BPMs by 40%
3. Shift notes up perfect 4th, slow down all BPMs by 50%
4. Shift notes down minor 6th, speed up all BPMs by 20%

While it seemed that this expansion of the training set should have positively influenced model training, I found that it instead either harmed model training and performance (most often causing a plateau very early on during training during training; see Figure 4.1) or provided similar or worse results during music generation (typically creating highly random or repetitive output). Thus, these datasets were deprecated as well. Although augmented datasets typically improve music classification-based models (Szelogowski, 2022), it may instead be more feasible to "rearrange" the compositions (either traditionally or by simply swapping similar voice parts) which could be

done programmatically also. This would allow for normalization such as transposition to a common key signature but at the possible expense of the "roles" of each voice contrapuntally (see *Chapter 2.1.4.2*).

4.1.2 Training Sequence Preprocessing

Upon creating the new choral-form dataset – i.e., where the data were saved in multiple files per voice part instead of one for all notes and another for durations – I modified the approach to loading the data for training to include a secondary preprocessing step to merge the note and duration tokens for all voices into uniform lists again (see *Appendix E.1*). While this solved some of the issues present in the non-choral models such as the lack of four-voice MIDI generation (since tokens were now tagged with their respective voice and could be split later; see *Appendix E.2*), it also led to new issues during generation. First, voices seemed to lack cohesion regarding voice leading and counterpoint between voices – leading to irregular, syncopated rhythms and durations. Second (and more importantly), some voice parts appeared to be "extinguishing" over time, in that the model appeared to be losing attention to certain voices as generation continued, causing them to drop out after the first minute or two of music (roughly 50-100 tokens). As well, the models would frequently generate MIDIs that were entirely empty.

Subsequent experiments revealed that the primary limitation in training benchmark DL models on the dataset was the initial method used to integrate the four voice parts, creating a significant bottleneck in model performance. Originally, these datasets were merged by interleaving token sequences in the order they appeared in the Pickle files – i.e., the first soprano sequence would be appended to the list, followed by the first alto, tenor, and bass sequences; repeating this process for all four voices. That is, tokens in the **notes** list would be merged as ["S:C5 S:D5 S:D#5 …", "A:rest A:F#4 …", "T:G3 T:rest …", "B:C3 B:A3 …", "S:A4 …"], for

example. This appeared to be a clear explanation of why the models were failing to retain attention to all voices over time or generating empty MIDI files, as some sequences may have fewer notes and more rests at numerous points in the training data. Likewise, some sequences were likely to only contain tokens of "rest" (and thus a duration of "0.0") given that choral pieces often feature long lengths of rests for voices – either as part solos/duets or simply a taceted section for a particular part (Lagbayi, 2022).

To solve these issues, I added additional preprocessing steps to perform two major changes before beginning training on the data. First, limiting the maximum number of sequential rests to prevent empty MIDIs/extinguishing voice streams (e.g., no more than 4 rests in a row), then reshaping the sequence tokens to be interleaved by voice part rather than uniformly – i.e., ["S:C5 A:rest T:G3 B:C3 S:D5 …", "S:D#5 A:B4 …"]. This ensured that all voices contained an equal number of tokens while also providing a means to modify sequence lengths during training.

4.2 System Components

The proposed "Choral-GTN" framework contains three key system components: the **Generative Transformer Network**, the **Music Generator**, and the **Post-Processing System**. The Generative Transformer model generates raw musical sequences, which are crafted into structured MIDI files by the Music Generator. Finally, the Post-Processing System meticulously refines these compositions, ensuring they adhere to traditional music theory principles. Together, these components form a comprehensive system for creating and enhancing AI-generated choral music.

4.2.1 Generative Transformer Network

The decision to employ a transformer-based architecture for my generative model – an architecture I name **"Generative Transformer Network" (GTN)** – stems from the transformer's unparalleled success in understanding and generating sequential data, especially within natural

language processing (see *Chapter 2.1.3.3*). This architecture's adaptability to music generation, particularly for complex choral compositions, offers a promising avenue for exploring deep learning's potential in artistic creation (Huang, 2019). The transformer model is uniquely suited for this task due to its ability to handle sequential dependencies and its flexibility in generating outputs, such as notes and durations, through separate layers tailored to each aspect of music (as demonstrated in the basis architecture by Foster & Friston (2023)). By treating music generation as a sequence prediction problem, the model learns to produce a coherent flow of musical elements, capturing the intricate relationships between notes, rhythms, and harmonies inherent in choral music. Hence, the rationale behind this architectural choice is to harness the transformer's capacity for capturing long-range dependencies (through its attention layers) and its effectiveness in modeling the probabilistic nature of music, aiming to generate compositions that are not only technically sound but also musically expressive (Briot et al., 2019).

The mathematical foundation of the **transformer block** layer within the GTN model involves several key components designed to process and generate musical sequences effectively (see *Appendix E.2*). At its core, the transformer employs a **causal attention** mechanism, ensuring that the prediction for a particular position is dependent only on known outputs at previous positions (see *Chapter 2.1.3.3*), thus preventing information leakage from future tokens (Foster & Friston, 2023). This is implemented through a causal attention mask, calculated (using Python's **TensorFlow** library) as follows:

$$\text{mask} = \text{range}(n\_dest)[:, \text{None}] \geq \text{range}(n\_src) - n\_src + n\_dest \tag{4.1}$$

The model also incorporates sine position encoding to inject information about the position of each token in the sequence, enhancing the model's awareness of the order of musical elements. This position encoding is given by:

$$\begin{aligned}\text{positional\_encodings} \\ &= \sin(position \cdot timescales) \cdot sin\_mask \\ &+ \cos(position \cdot timescales) \cdot cos\_mask\end{aligned} \tag{4.2}$$

where $timescales$ are calculated as a function of the sequence length and the dimensionality of the embeddings.

The **token and position embedding** layers play a crucial role in the transformer's architecture by providing a rich, contextual representation of each input token (Vaswani et al., 2017). Here, the token embeddings are vectors that represent the musical elements (notes and durations) in a high-dimensional space (where similar tokens are positioned closer together) which are combined with position embeddings to maintain the sequential order of the musical elements. The position embeddings are calculated using a sine and cosine function of different frequencies:

$$\begin{aligned}PE_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)\end{aligned} \tag{4.3}$$

where pos is the position, i is the dimension, and d_{model} is the dimensionality of the token embeddings (Vaswani et al., 2017). This approach allows the model to encode the position of each token within the sequence, providing the model with the ability to recognize patterns and structures over varying distances in the sequence.

The **multi-head attention** mechanism is another pivotal component of the transformer architecture, enabling the model to focus on different parts of the sequence for each prediction. This is performed by projecting the **queries** (Q), **keys** (K), and **values** (V) into multiple spaces, computed as follows:

$$\begin{aligned} \text{Attention}(Q, K, V) &= \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \\ \text{MultiHead}(Q, K, V) &= \text{Concat}(head_1, \ldots, head_h)W^O \\ \text{where } head_i &= \text{Attention}\big(QW_i^Q, KW_i^K, VW_i^V\big) \end{aligned} \tag{4.4}$$

where d_k is the dimension of the key vectors, W^* represents the parameter matrices (e.g., weights), and h is the number of heads defined by the model's hyperparameters (typically 8; Vaswani et al., 2017). This allows the model to capture different types of relationships across the sequence. In multi-head attention, this process is paralleled across multiple heads, allowing the model to attentively process information from different representational spaces. The outputs of these heads are then concatenated and linearly transformed into the expected dimensionality.

Layer normalization and the dense (i.e., feed-forward) network layers follow the attention mechanism, further processing the attended representations (see *Appendix E.2*). Layer normalization is applied to each input vector separately, normalizing the inputs across the features:

$$\text{LayerNorm}(x) = \gamma\left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta \tag{4.5}$$

where μ and σ^2 are the mean and variance of the features, ϵ is a small constant to prevent division by zero, and γ and β are learnable parameters for scaling and shifting (Ba et al., 2016). This normalization helps stabilize the learning process by creating better-behaving parameter gradients (Xiong et al., 2020). The network then passes through two dense layers, with the first acting as a feed-forward network to increase the model's capacity to capture complex relationships:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)\, W_2 + b_2 \tag{4.6}$$

where W_1, W_2, b_1, and b_2 are the weights and biases of the dense layers, and $\max(0, x)$ denotes the ReLU activation function (Vaswani et al., 2017). These components collectively form the

Transformer Block, multiple layers of which are stacked within the GTN architecture (see *Chapter 3.3*), each contributing to the model's ability to generate rich, nuanced musical sequences by learning from the structure and content of the training data.

Expanding on the transformer block's operation, the architecture resourcefully handles note and duration inputs through a dual-input system, which is needed for maintaining the distinction between melodic and rhythmic elements throughout the generation process (Foster & Friston, 2023). Initially, notes and durations are processed through separate embedding layers to capture their unique characteristics before being concatenated (see *Appendix E.1*). This concatenated vector then serves as the input to the transformer blocks, ensuring that the model has a comprehensive understanding of both the harmonic and rhythmic context at each step of the sequence. The embeddings not only provide a dense representation of the inputs but also, through the addition of sine position encodings, imbue the sequence with temporal context, essential for sequential prediction tasks like music generation (Huang et al., 2019).

These numerous transformer blocks operate in concert, with each block building on the outputs of the previous one, allowing for increasingly refined attention and representation of the musical sequence. The multi-head attention within each block enables the model to focus on different parts of the musical sequence, considering both the current state and the accumulated knowledge from previous blocks. This hierarchical processing ensures that the model can capture both local interactions (such as note-to-note or duration-to-duration relationships) and global structures (like musical phrases or motifs) within the composition. After passing through multiple transformer blocks, the concatenated embeddings, now richly informed by the sequence's context, are directed toward two separate dense layers – one for notes and another for durations (see *Appendix E.1*). These layers act as classifiers, outputting the probabilities of the next note and

duration in the sequence, effectively translating the abstract representation back into tangible musical elements (Foster & Friston, 2023).

Regarding the mathematical aspects of the training process, the **Noam scheduler** and **Adam optimizer** are critical for adjusting the learning rate throughout training, ensuring efficient and effective model optimization. The Noam scheduler adjusts the learning rate to the formula:

$$lr = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \tag{4.7}$$

This adaptive learning rate helps balance the trade-off between learning speed and stability (see *Appendix E.2*), enabling rapid progress in early training stages and fine-tuning as the model converges (Vaswani et al., 2017). The Adam optimizer, a staple in training deep learning models, further refines the model's parameters through its moment-based approach. It calculates adaptive learning rates for each parameter from estimates of the first and second moments of the gradients:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \widehat{m}_t \tag{4.8}$$

where θ represents the parameters, η the learning rate (from the Noam scheduler), $\widehat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the first and second moments of the gradients, and ϵ is a small scalar added for numerical stability (Kan, 2023). Finally, the model's output is fine-tuned through TensorFlow's **Sparse Categorical Cross-Entropy** loss function, which measures the difference between the predicted probabilities and the actual distribution of the next notes and durations (see *Chapter 5.1*). This loss function is particularly suited for classification tasks with multiple classes, such as predicting the next note or duration, guiding the model towards accurate predictions by penalizing deviations from the true distribution.

Despite the transformer architecture's prowess in capturing the complexities of music generation, certain limitations necessitate a **hybrid approach**, combining AI-driven creation with

rule-based post-processing (see *Chapter 4.2.3*). One primary limitation lies in the transformer's inherent challenge in adhering strictly to the nuanced rules of music theory, particularly those governing voice-leading and counterpoint in choral music. While the architecture excels at learning patterns and generating musically coherent sequences, its probabilistic nature sometimes leads to outputs that, though technically correct, may not always align with the stylistic and theoretical standards of classical composition (Hadjeres & Nielsen, 2020). This gap between AI-generated content and music theory compliance underscores the rationale for integrating a hybrid post-processing model. Such a system should meticulously apply music theory rules to refine and correct the AI's output, ensuring that the final compositions not only exhibit creativity and complexity but also adhere to the traditional standards of choral music. My hybrid model thus leverages the strengths of both AI and rule-based methodologies, harnessing the generative power of the transformer while ensuring the musical output respects the established conventions of choral writing, thus addressing the architectural limitations and fulfilling the artistic and theoretical criteria of the genre.

4.2.1.1 Hyperparameter and Architecture Search

The task of refining and optimizing the GTN's architecture and its hyperparameters was approached with a methodical and comprehensive strategy, employing a blend of random and grid search techniques to explore the vast hyperparameter space. This dual-faceted approach aimed to strike a balance between the breadth of random search, which covers a wide range of possible values with no predetermined order, and the depth of grid search, which methodically tests all combinations within a specified range. The implementation began with defining a set of potential values for key hyperparameters – such as embedding dimension, feed-forward dimension, key dimension, number of heads, gradient clipping value, dropout rate, ℓ_2 regularization strength, and

the number of Transformer blocks. These parameters were chosen based on their significant impact on model performance and training dynamics (Kamsetty, 2020).

During the hyperparameter optimization phase, a model tuner function (see *Appendix E.1*) dynamically constructed models based on the current set of hyperparameters being evaluated, allowing for efficient exploration of different configurations. The use of the Keras Tuner library's **RandomSearch** and **GridSearch** classes facilitates this exploration, with each class offering distinct advantages. Random search provides a stochastic sampling of the hyperparameter space, offering quicker insights into effective configurations without exhaustive testing (Selvaraj, 2022). In contrast, grid search allows for a thorough examination of all possible combinations of the predefined values, ensuring no potential configuration was overlooked. The optimization process was iteratively executed, with each trial running for a varying number of epochs on a subset of the dataset, thus balancing the need for comprehensive evaluation with computational feasibility.

The results from these searches were analyzed to identify the best-performing set of hyperparameters based on the objective criterion of minimizing validation loss. This systematic exploration not only informed the selection of the final model's hyperparameters but also provided valuable insights into the architecture's sensitivity to different parameter settings (see *Chapter 5.1*). I utilized **parallel coordinates plots** to visualize these searches – a powerful tool for multidimensional data visualization, where each hyperparameter is represented by a vertical line (or axis), and the range of values it can take is plotted along that line (see Figure 4.2). A set of hyperparameters tested during the search is represented as a polyline intersecting each axis at the point corresponding to the value of that hyperparameter, offering an intuitive understanding of the relationships between multiple hyperparameters and the model's performance – further guiding the refinement process.

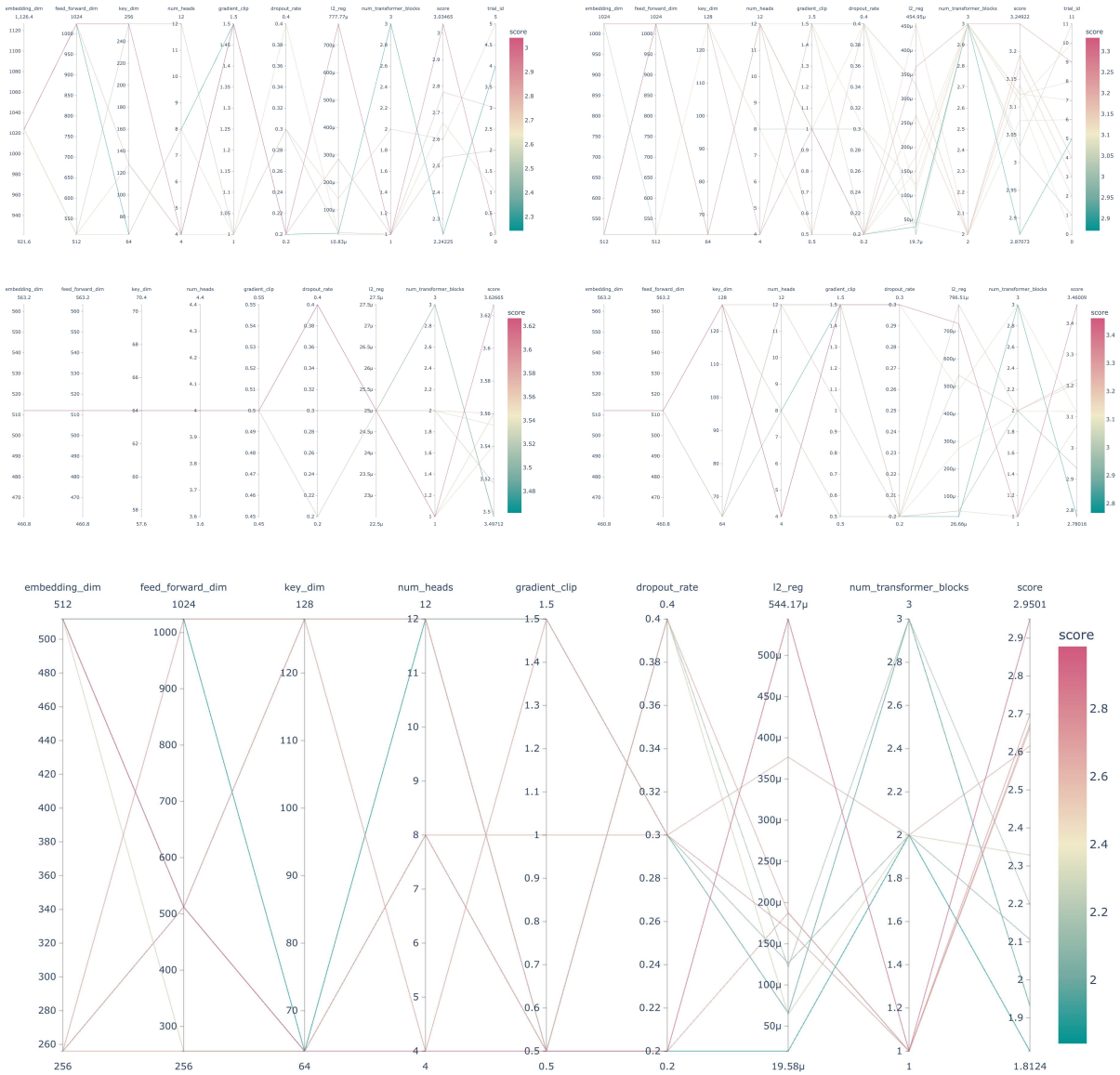


Figure 4.2: Sample parallel coordinates plots from the hyperparameter search trials

4.2.1.2 Deprecated Generative Model Architectures

In designing the "Choral-GTN" system, numerous generative model architectures were explored and subsequently set aside for future potential work due to various limitations and inefficacies in meeting the project's objectives. The exploration of these architectures provided valuable insights into the challenges of generating polyphonic choral music, leading to the eventual adoption of the multi-layer transformer architecture. The **Performer** (with Fast Attention) was one of the initial architectures attempted, inspired by its promise of scaling attention mechanisms efficiently (Choromanski et al., 2022). Despite its theoretical appeal, the Performer struggled with fitting the polyphonic choral music data effectively. The architecture's design, while innovative, was not easily adaptable to handling two separate input layers for notes and durations, a critical requirement for capturing the nuances of choral compositions. This limitation, coupled with difficulties in replicating the model's performance as described in its foundational paper, led to its omission in favor of more adaptable solutions. However, future work and adjustments based on the current final architecture may potentially find the Performer (or another advanced architecture such as the Nyströmformer; Xiong et al., 2021) to outperform the vanilla transformer block.

Music Transformer (Huang et al., 2019) and **Markov Chain** (e.g., Buys, 2011) models were also tested, each facing its unique set of challenges. The Music Transformer, despite its potential for handling sequential data, proved insufficiently sophisticated for polyphonic music, failing to integrate two input layers effectively (similar to the Performer). The Markov Chain lacked the sophistication needed to produce musically meaningful outputs, especially in capturing the complex interplay of voices in choral music (Yin et al., 2023). These models, while promising in certain contexts (such as using an MC in a hybrid system), could not fulfill the project's requirements for generating high-quality choral compositions given the current dataset design.

RNN, **LSTM**, and dual input **Bidirectional-LSTM** architectures were explored for their potential in sequence modeling (Ji et al., 2020). However, these models consistently underperformed with four-part polyphonic scores, only achieving moderate success with choral reductions. The inherent limitations of RNNs and LSTMs in capturing long-term dependencies were exacerbated in the context of complex choral music, leading to outputs that lacked coherent melodic and harmonic structures. The addition of attention mechanisms and bidirectional layers offered slight improvements but still fell short of this project's ambitions.

GAN, **HRED**, **GPT-2**, **VAE/CVAE**, and **Q-Learning** (RL) **Networks** represented further forays into leveraging advanced machine learning techniques for music generation. Each of these models, despite their successes in other domains, faced significant challenges in learning the sophisticated structures of polyphonic choral music. From the inability to fit well to the data (as was the case with the HRED, VAE, and Q-Learning models) to the production of musically incoherent outputs (GAN, including RNN-GAN), these architectures demonstrated the necessity of a more sophisticated architecture suited to the task of polyphonic music generation. Particularly, the MusicBPE Transformer (Liu et al., 2022) and GPT-2 (with transfer learning) models – while initially promising due to their success in language-based modeling – could not be effectively trained to produce satisfactory musical outputs with polyphony, highlighting the unique challenges of applying these architectures to music generation with long-term structures (Dai et al., 2022).

The exploration of these models underscored a crucial realization: generating choral music, with its rich polyphony and intricate voice leading, demands an architecture capable of understanding and replicating the deep musical relationships inherent in such compositions.[2] This journey through various generative models ultimately reinforced the decision to pursue a hybrid

[2] All deprecated models (and additional training figures) can be seen in the project's GitHub repository (Szelogowski, 2022/2023), including in commits to the main branch containing the various training attempts.

(multi-layer) transformer model, which, through its sophisticated attention mechanisms and adaptability, proved capable of overcoming the limitations encountered with other architectures (see *Appendix E.2*). Unlike the Performer, Music Transformer, or other RNN-based architectures, the transformer's sophisticated attention mechanism allows it to discern and replicate the nuanced relationships between notes across multiple voice parts, effectively modeling both the harmonic and rhythmic structures that define choral compositions (Liu et al., 2022). Nonetheless, the development process – marked by trial and error with these diverse models – was instrumental in identifying the unique requirements of choral music generation and selecting an architecture that could meet these complex demands.

4.2.2 Callback-Integrated MIDI Generator

The callback-integrated music generation system represents a secondary mechanism designed to harness the generative capabilities of the GTN model, enabling dynamic music creation and facilitating the evaluation of model performance throughout the training process (see *Appendix E.2*). This system's architecture is meticulously crafted to accommodate the generation of new polyphonic MIDI files, both as a means of testing the model's progress at the end of each training epoch and as the primary method for generating music with the fully trained model (see *Appendix E.1*). The implementation begins with parsing musical elements, utilizing the **music21** library to create MIDI streams for each voice, convert note, duration, and expression tokens (e.g., BPM and time/key signatures) from strings (text) into musical objects appended to each stream, and write the streams to a single, multi-track MIDI file.

The core functionality of this system lies in its ability to predict and assemble these tokens into coherent musical sequences using a provided GTN model. This process starts with the generation of initial "START" tokens and progresses through iterative predictions of subsequent

notes and durations, effectively weaving together a “tapestry” of musical elements. The system employs a probability sampling technique to select new notes and durations from the model’s output, ensuring a degree of variability and creativity in the generated music (Foster & Friston, 2023). This technique is mathematically formulated to weigh the model’s predictions, adjusting them based on a **temperature** parameter that controls the randomness of note/duration selection:

$$\mathrm{P}(n) = \frac{e^{\frac{\log(probs_n)}{T}}}{\sum_{i=1}^{N} e^{\frac{\log(probs_n)}{T}}} \tag{4.9}$$

where $\mathrm{P}(n)$ is the probability of selecting note (or duration) n, $probs_n$ is the predicted probability of note n, T is the temperature controlling the randomness of the selection, and N is the total number of possible notes (likewise for durations). This calculated probability distribution is then used to randomly select the next note and duration in the sequence, ensuring that the generation process remains both guided by the model’s learning and infused with an element of unpredictability (see *Appendix E.2*).

As the system assembles these musical sequences, it leverages the rich functionalities of **music21** to encode the generated notes and durations back into MIDI format, crafting complete musical compositions. The integration of this callback mechanism within the training loop serves as a critical bridge between the theoretical aspects of AI music generation and the practical application of these concepts in creating real-world musical compositions, while also providing a tangible metric for us to gauge the capability of the model over time. Through this approach, the system not only validates the model’s effectiveness in generating musically coherent pieces but also opens avenues for exploring the artistic potential of AI in the domain of choral music.

The complete music generation process is initiated by specifying the GTN model (i.e., weights) to employ (for token generation), note/duration vocabulary datasets, and configuring the

generation parameters – including the length of the generation, the number of compositions to generate, and the **temperature**, which influences the variability and creativity of the generated music. The system dynamically loads the model weights and vocabularies for notes and durations, preparing the GTN model for the generation task. Depending on the specified configuration – such as the generation vocabulary and any adjustments for transposed datasets – the model and MIDI generator are tailored to adjust to the generation needs, ensuring the output aligns with the desired musical characteristics.

The generation system utilizes the Music Generator, a callback component that acts as a consumer of the GTN model, to synthesize new compositions using the GTN's output tokens (see *Appendix E.1*). By feeding seed notes and durations into the model (or simply "START" and "0.0" tokens for each voice), the system initiates the generative process, with the Music Generator extending these seeds into full-fledged musical pieces. This iterative generation leverages the model's learned musical knowledge, applying the temperature-controlled probability sampling to decide on each subsequent note and duration, thereby crafting sequences that embody both structural coherence and creative flair. The resultant music is then converted into MIDI files, with the system performing checks to ensure the validity and completeness of the generated compositions, such as verifying the presence of multiple voice parts and non-empty MIDI output.

4.2.3 Rule-Based Post-Processing System

The rule-based post-processing system is designed to refine the raw output from the GTN model and Music Generator into compositions that adhere more closely to traditional music theory principles. This system operates on several levels, each addressing different aspects of musical theory and voice leading to ensure the generated music not only sounds pleasant but also respects the conventions of choral music (i.e., those of the Common Practice Period; see *Appendix C.1*).

Ideally, this post-processor should act as a sort of musicologist, correcting the model's work like that of a music theory student (thus forming the hybrid component of this study's model; Takyar, 2023). The implementation of this system is broken down into six main steps, each targeting specific musical elements that commonly require adjustment in AI-generated compositions (see *Appendix E.4*):

1. Diatonic Adjustment
2. Parallel Motion Correction
3. Doubling Correction
4. Melodic Interval Correction
5. Skips and Leaps Correction
6. Dissonance Handling

Each step is performed in sequence, with some repetition to assist the system in fixing any new errors caused by prior corrections (similar to a mathematical optimization model).

Step 1 involves transposing notes to ensure they are diatonic to the specified key signature – a crucial component for maintaining harmonic coherence within the piece, especially in very noisy generations where many notes may be overly "random" or non-diatonic. The process identifies notes that fall outside the scale defined by the key signature and transposes them to the nearest scale degree. Mathematically, this can be seen as minimizing the pitch distance d between a note n and the set of pitches S that belong to the diatonic scale, and applying a pitch shift s that satisfies:

$$s = \arg\min_{s}(|n + s - S|) \tag{4.10}$$

Contrastingly, Step 2 targets the elimination of parallel fifths and octaves between voice parts, a fundamental rule in counterpoint to prevent voices from moving too similarly and losing

independence (see *Appendix C.2*). The system identifies sequences of intervals between two parts that maintain the same intervallic distance over consecutive notes. When such sequences are detected, the offending note in the latter part is altered by a step in the opposite direction of the motion to break the parallelism, ensuring a more varied and texturally rich counterpoint. Step 3 thus focuses on adjusting chords to avoid doubling of critical tones such as the leading tone (i.e., scale degree $\hat{7}$), which can create unresolved tension or diminish the harmonic clarity. This involves algorithmically altering doubled notes within chords to distribute the tonal content more evenly across the voices, enhancing the overall harmonic balance (Gotham et al., 2021).

Steps 4 and 5 (correcting melodic intervals, skips, and leaps) are dedicated to smoothing individual voice lines to avoid overly large leaps that can make singing parts challenging and detract from the melodic flow. These steps employ rules to constrain melodic motion, encouraging stepwise movement or smaller intervals that are more singable and typical of choral music. When large intervals or awkward leaps are detected, the system algorithmically adjusts the pitch of subsequent notes to create a more gradual and cohesive melodic line. Finally, Step 6 aims to resolve dissonances appropriately, ensuring that sevenths resolve downwards and that dissonant intervals such as the tritone are resolved or avoided in the context of the chordal structure. This step is vital for maintaining the tonal integrity of the composition, ensuring that tension created by dissonances is resolved in a way that is satisfying and aligns with Western tonal harmony.

Utilizing a rule-based post-processing system within the framework is a sophisticated endeavor that goes beyond merely applying each step sequentially, however. The intricacy of music composition, especially in adhering to the conventions of choral music, necessitates a dynamic and recursive approach to post-processing. After the initial pass through the six steps of correction, the system engages in a secondary phase where Steps 1 (Diatonic Adjustment) and 2

(Parallel Motion Correction) are revisited. This iterative process is designed to address any new discrepancies or musical issues that may arise as a consequence of earlier corrections. For instance, altering pitches to correct parallel fifths or octaves might inadvertently introduce notes that are non-diatonic to the piece's key (Gotham et al., 2021). Reapplying the diatonic adjustment ensures that all notes remain within the correct scale, thereby maintaining harmonic coherence.

This recursive correction process underscores a critical aspect of employing AI in music generation: the balance between creativity and conformity to musical rules. By rerunning specific steps, the system meticulously refines the generated music, ensuring that any changes made to correct one aspect do not detrimentally impact another. This approach mimics the iterative refinement process inherent in human composition, where composers often revisit and revise their work to achieve the desired musical outcome. Furthermore, the final implementation of the post-processing system incorporates a versatile mechanism for altering the key and tempo of the generated compositions. This feature allows for adjustments based on either user input or a randomized selection informed by a probabilistic analysis of the dataset. The system's ability to change the key and tempo post-generation adds a layer of customization and experimentation, enabling users to explore various musical expressions and styles within the choral framework. The meta-analysis of the dataset plays a crucial role here, providing a statistical foundation for these adjustments. By analyzing the distribution of keys, tempos, and other musical elements across the dataset, the system can make informed decisions when selecting new keys or tempos randomly, ensuring that these choices are representative of common practices in choral music.

This sophisticated post-processing system, with its recursive corrections and customizable adjustments, highlights the Choral-GTN framework's innovative blend of AI-generated creativity and rule-based musical theory, as well as the necessity of a hybrid approach to classical music

generation. The ability to not only generate new compositions but also refine and adapt them according to both established musical rules and user preferences bridges the gap between traditional music composition and modern technological innovation. However, it is worth noting that my system is merely a simplified interpretation of some music theory conventions, and a more sophisticated approach (such as a decision tree) is necessary for more complex or larger works.

4.3 Limitations

Limitations in the CHORAL dataset (such as size, scope, and SATB-only voicing), coupled with its broad variation in musical elements (e.g., style, key, and tempo), without any normalization like key transposition or timing standardization, pose significant challenges to generating consistent outputs, thus leading to a degree of unpredictability in the model's performance still (see *Chapter 6.2*). Moreover, despite its ability to elaborate on user-provided inputs, the GTN model often struggles with generating cohesive compositions autonomously (i.e., with no input seed for notes/durations), necessitating multiple iterations to achieve a satisfactory outcome. This issue is compounded by difficulties in maintaining coherence in longer sequences and the absence of a nuanced, task-specific loss function that could potentially enhance learning efficiency. The post-processing system faces similar limitations due to the complex nature of MIDI structures and the intricacies of encoding sophisticated music theory rules into a programmable format. As such, these challenges highlight the necessity for expanded datasets, improved model design, and advanced post-processing methods, marking critical areas for future developments (see *Chapter 6.2.1*).

CHAPTER 5
EVALUATION

This chapter provides a comprehensive evaluation of the Generative Transformer Network model and the relevant metrics for the system, as well as a statistical analysis of the evaluation study encompassing the compositions generated by the system.

5.1 Model Evaluation

For the transformer model, the (transposed) dataset is split into **90% training** and **10% testing**/validation for cross-validation (Refaeilzadeh et al., 2009) with a **batch size of 128**. This ratio, along with the **training sequence length of 52** (13 tokens per voice for each sequence, or length 100/25 tokens per voice for one set of model weights; see *Chapter 5.1.1*), was found through trial and error as the various models were being tested for the optimal training conditions. The data were prepared for training using **text vectorization** layers – one for **notes** and another for **durations** – creating tokenized sequence sets, vectorization layers, and vocabularies (i.e., integer-encoded "lookup" tables used to obtain the original token strings; TensorFlow, 2023). During training, the model is optimized using the Adam optimizer (a common choice for sequence-based models; Jadhav et al., 2023), a **Noam scheduler**, and TensorFlow's **Sparse Categorical Cross-Entropy** loss for both note and duration outputs. Noam is a learning rate scheduler tailored to transformer models (commonly used for large-scale NLP tasks) that provides warm-up and decay phases to stabilize and gradually decrease learning rates as a model approaches convergence, often leading to better training stability and improved model performance (Vaswani et al., 2017).

Sparse [Categorical] Cross-Entropy (SCE) is a loss function commonly used in generative/sequence-based models (especially music) given its memory efficiency and gradient amplification for classes/musical tokens that have low predicted probabilities but are correct (Jagannathan et al., 2022). It is a type of Categorical Cross-Entropy where the target labels are (discrete) integer values, rather than one-hot encoded vectors (Garcia-Valencia, 2020) – for memory efficiency, this is particularly relevant for music generation given the large number of possible output tokens (i.e., the cardinalities of the note and duration vocabularies). SCE is calculated as follows:

$$L_{\text{SCE}} = -\frac{1}{N}\sum_{i=1}^{N} \log(\hat{y}_i[y_i]) \tag{5.1}$$

where:

- y_i is the true class label for the i-th sample, represented as an integer,
- $\hat{y}_i$ is the model predicted probability distribution for the i-th sample (**Softmax** activation),
- $\hat{y}_i[y_i]$ is the predicted probability of the true class y_i,
- $-\log(\hat{y}_i[y_i])$ is the loss for a single sample (the negative logarithm of the predicted probability corresponding to the ground truth),
- N is the batch size (the total number of samples), and
- L_{SCE} is the mean SCE loss for N samples.

During training, the model calculates three different loss values per epoch: **note_outputs_loss**, **duration_outputs_loss**, and the overall **training loss** (see Table 5.3). The overall loss is typically computed as the weighted sum of all training losses, calculated here as

$$L_{\text{overall}} = w_1 \times \text{notes\_loss} + w_2 \times \text{durations\_loss} \tag{5.2}$$

where w_1 and w_2 are the respective weights for each loss value (in this case, $w_1 = w_2 = 1$).

5.1.1 Model Results

Following the successful architecture design after hyperparameter searching (see *Chapter 4.2.1.1*) and optimal dataset preparation (see *Chapter 4.1.1*), I trained numerous different variant models in an attempt to narrow down the model with the lowest overall loss value. Primarily, this involved slightly adjusting the number of neurons in the transformer's Feedforward dimension, training sequence length (52 or 100), dataset range (25% and 100% of the full training set), and regularization parameters (ℓ_2 regularization, dropout, and **gradient clip** of 1.5 to prevent steep/exploding gradients; Bajaj, 2022). Over the course of training 15 different models of the same architecture with slight changes to hyperparameters, only three yielded promising results regarding the music generated by the model – irrespective of the training/validation loss values (see Table 5.1).

Model Weight Set	Embedding Dimension	Feedforward Dimension	Key Dimension	# of Heads	Transformer Blocks
Transposed_2 (2nd best)	512	**1024**	64	8	3
Transposed_3 (best)	512	512	64	8	3
Transposed_13 (worst)	512	512	64	8	3

Table 5.1: Choral-GTN hyperparameters

Model Weight Set	Sequence Length	Dropout Rate	ℓ_2 (Ridge) Regularization	Dataset Range	Truncate to Min Length
Transposed_2 (2nd best)	**100**	0.3	1e-4	100%	Before removing rests
Transposed_3 (best)	52	0.5	0.0005	**25%**	**After removing rests**
Transposed_13 (worst)	52	0.01→ 1e-7 (gradual)	0.005→1e-6 (gradual)	100%	Before removing rests

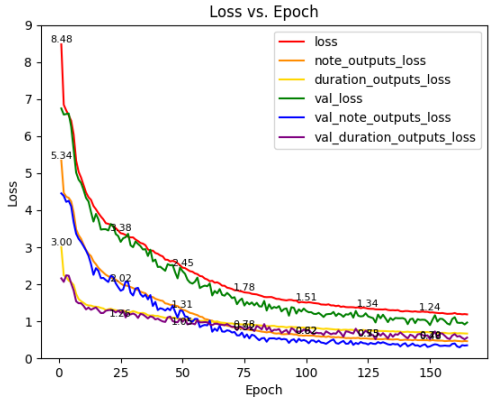
Table 5.2: Choral-GTN training parameters (i.e., truncate refers to dataset sequences)

Numerous unanticipated factors became apparent following these numerous training attempts, however. For example, during training data preprocessing (when the datasets for all four voice parts are merged), the order in which the sequences were truncated to the minimum length – either before or after removing long stretches of rests – appeared to make a significant difference in the generated output even though the training set size and typical training losses were nearly identical. The sequence length also appeared to have surprisingly little effect on the loss values, though its effect on the generated music was too difficult to discern and was thus typically left at 52 tokens per sequence (see Table 5.2). Similarly, the models appeared to respond significantly poorly to regularization – including dropout. While this is often expected for ℓ_2 regularization since it prioritizes validation loss over training loss (Tewari, 2021), a punishing effect is much less expected of dropout regularization (Melis et al., 2017) – although this can be a beneficial indication that the model architecture aptly suits the dataset.

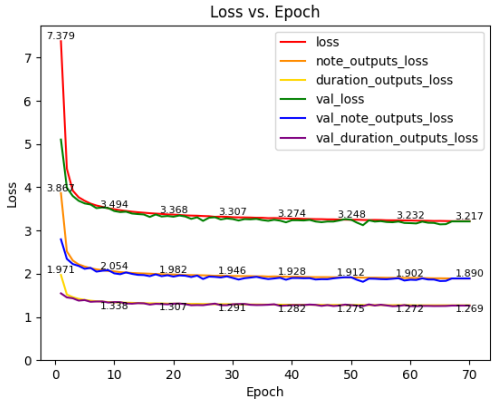
The most unexpected outcome of training the numerous models was that the model weights with the best training history generated the worst (musically unrealistic, i.e., "random") music, while the models with the seemingly worse training history generated the more realistic compositions (see Table 5.3 and Figure 5.1). As such, I retain the three best model weights (regarding music generation "realism") in the final system to allow users to choose which model to employ at runtime – each of which was utilized to cumulatively obtain enough AI-generated compositions for the evaluation survey (see *Appendix D*). However, when provided a **seed** input (i.e., starting notes and durations; see *Appendix E.1*), all three model weights appear to perform more similarly than when generating a completely new composition.

Model Weight Set	Epochs	Loss	Note Loss	Duration Loss
Transposed_2 (2nd best)	165	1.1875 (Train) 0.9676 (Val)	0.4634 (Train) 0.3524 (Val)	0.6697 (Train) 0.5608 (Val)
Transposed_3 (best)	67	3.2167 (Train) 3.2103 (Val)	1.8899 (Train) 1.8900 (Val)	1.2689 (Train) 1.2626 (Val)
Transposed_13 (worst)	300	0.8704 (Train) 0.8505 (Val)	0.2148 (Train) 0.2006 (Val)	0.6391 (Train) 0.6334 (Val)

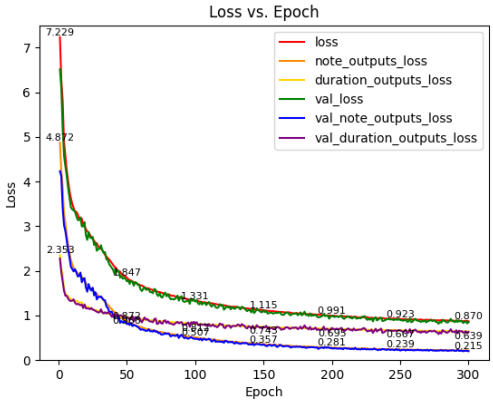
Table 5.3: Choral-GTN final training history



(Transposed_2)



(Transposed_3)



(Transposed_13)

Figure 5.1: Choral-GTN training history plots

This phenomenon, where a model with higher loss values generates subjectively better music than one with lower loss values, appears to not be uncommon in creative applications of machine learning like music generation – particularly where the loss function itself is not specially

tailored toward the application (Li & Linberg, 2023). In this project's case, however, it may be due to several potential reasons:

- **Loss Function Limitations** – The loss function used in training may not perfectly capture the qualities that make music sound good to human ears. A lower loss value means the model is better at minimizing the specific error defined by the loss function, but this does not necessarily translate to better music. Since this project utilizes a simple SCE loss, metrics for training the model do not account for the nuances of musicality, harmony, rhythm, and style that are important for pleasant music generation.
- **Overfitting vs. Generalization** – The model with lower loss might be overfitting to the training data. This means it may be learning the training data too well, including its noise and peculiarities, which may not generalize well to creating new, pleasing music. The model with higher loss might be more generalized and, therefore, better at creating diverse and creative outputs.
- **Exploration vs. Exploitation** – A model with higher loss might be more **exploratory** in its generation process, leading to more varied and interesting results. In contrast, a model with very low loss might **exploit** certain patterns it learned during training, leading to repetitive and less creative outputs; this became especially noticeable when varying the temperature value of the Music Generator system in an attempt to remove or reduce harsh repetition.
- **Complexity of Musical Structures** – Capturing the long-term structures and complexities of music is a challenging task for machine learning models. A model with lower loss might be focusing too much on getting the short-term notes and durations correct, at the expense of the overall structure and progression of the piece, which are crucial for musicality.

5.2 Observational Study

Following the completion of the generative model, an observational study was designed to evaluate the ability of diverse listeners to distinguish between AI-generated and human-composed choral music – i.e., a "Musical Turing Test." The study was structured as a survey, wherein participants were presented with five choral music samples rendered (from MIDI) using a realistic choral soundfont, Muse Choir. Three of these samples were generated by the top three performing AI model weights, while the remaining two were human-composed pieces. As such, the arrangement of these samples was randomized to prevent any order bias. The original survey design (including all sheet music for both AI and human scores) can be seen in *Appendix D*, and the untrimmed audio samples can be found on GitHub (Szelogowski, 2022/2023).

Respondents were asked to gauge the origin of each music sample using a nuanced 6-point Likert scale, with options ranging from "AI - Strongly believe" to "Human - Strongly believe." This scale was designed to capture a gradient of perception and belief regarding the source of the compositions, allowing for a more detailed analysis of respondents' ability to discern between AI-generated and human-created music. Following the assessment of each sample, participants were also asked to indicate their level of musicianship or musical ability, categorized into four distinct groups: **Music teacher/professional** (with formal training and qualifications), **Music student** (encompassing college-level or equivalent experience, including Advanced Placement), **Amateur musician** (hobbyists with some background in composition/theory or performance), and **Music listener/enjoyer** (those with little or no formal musical experience).

The survey respondents comprised a diverse group of 500 individuals, including fellow employees from the School District of Janesville, university colleagues (both professors and fellow students), and additional participants recruited via Prolific.com, a Human Computation platform

well-known for providing research participants similar to Amazon Mechanical Turk. This eclectic mix of participants was chosen to ensure a wide range of musical backgrounds and perceptual experiences, thereby enriching the study's findings with varied perspectives. Upon completing the survey, respondents were directed to a debriefing page. Here, they were given an opportunity to revisit the music samples, this time with access to an answer key revealing the origin of each piece. This aspect of the study was designed to encourage reflection on their initial responses and to gather feedback, providing deeper insights into their perception and decision-making process.

5.2.1 Power Analysis

Conducting a power analysis was an essential step to ensure the robustness and validity of this study's statistical findings. This analysis was employed to determine the minimum sample size required to effectively detect a true effect, should one exist, in the ability of different demographic groups to distinguish between AI-generated and human-composed music. This is crucial because an inadequately powered study could lead to two potential pitfalls: first, it might fail to detect a significant effect (Type II error), even if there is a real difference in the perception of AI and human compositions; and second, it could lead to overestimating the effect size if an effect is detected with a small sample (Statistics Solutions, 2021). By performing a power analysis, I aimed to balance the risk of these errors while ensuring efficient use of resources and determine the optimal number of respondents needed to achieve a high enough statistical power (typically set at 80% or higher), which in turn increased the likelihood of correctly rejecting the **Null Hypothesis** (H_0, which represents no effect) when it was false (i.e., accepting the **Alternative Hypothesis** H_a, which represents an effect with statistical significance). This methodical approach in determining the sample size was designed to fortify the study's statistical reliability, providing a solid foundation for drawing meaningful conclusions from the survey data.

A power analysis was performed using the following parameters (in Python, using the **statsmodels** library; see *Appendix E.5*):

- Hypothesis:
    - H_0: The AI-generated music is indistinguishable from human-composed music (e.g., 50% of people believe it is human-composed).
    - H_a: The AI-generated music is distinguishable from human-composed music (e.g., significantly more or less than 50% believe it is human-composed).
- Sample size (n): 500 respondents
- Significance Level (α): 0.05
- Beta Level (β): 0.2
- Anticipated Effect Size: 0.1 ($p_2 - p_1$)
    - Proportion under H_0 (p_1): 50%
    - Proportion under H_a (p_2): 60%
- Statistical Power ($1 - \beta$): 0.8 (i.e., 80%)

The analysis yielded a proportional effect size of -0.2, suggesting a small to moderate difference (by magnitude) between the proportions being compared – i.e., the difference between 50% of respondents identifying a piece as AI-generated and 60% doing so is modest. As such, the **NormalIndPower** Python class was employed to perform the power analysis,[3] which performs statistical power calculations for Z-tests with two independent samples (Perktold et al., 2023). This test yielded a required sample size of ~387 respondents, validating the adequacy of the final sample size from the evaluation survey.

[3] Although the survey data was initially analyzed using a χ^2 test, the "GofChisquarePower" class failed to converge (yielding a sample size of 10, even after correcting the degrees of freedom); I found that my initial test using the "NormalIndPower" test was thus more realistic and generalizable to the various testing methods applied.

Similarly, a sensitivity analysis was performed to validate the sufficiency of the power analysis given the proportional effect size.

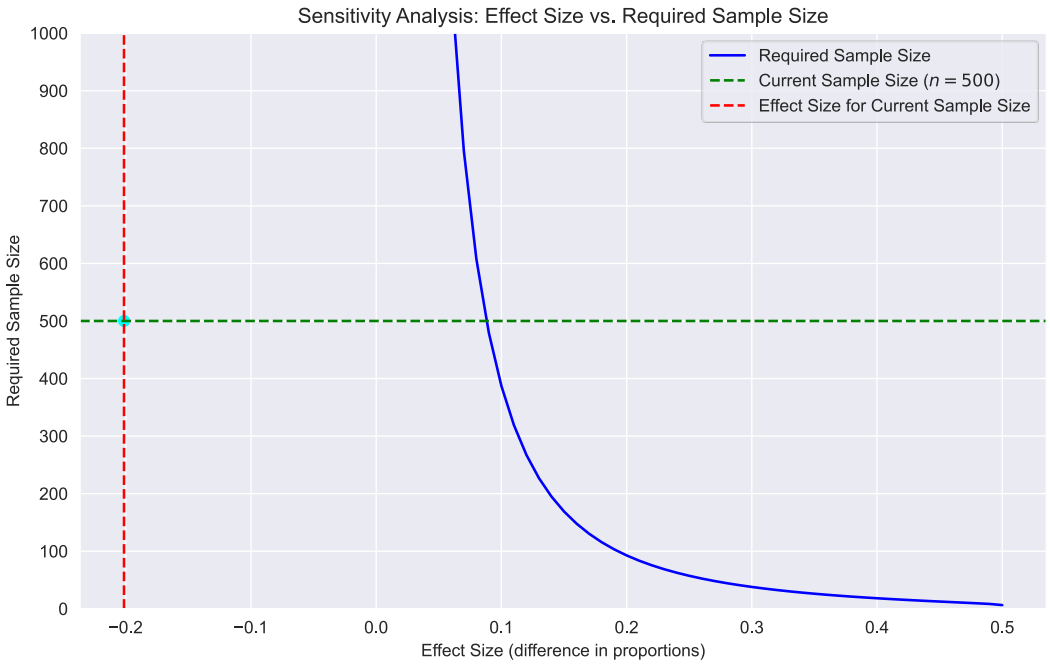


Figure 5.2: Sensitivity analysis plot

The sensitivity plot (see Figure 5.2) shows that for an effect size of -0.2, the current sample size of 500 is more than sufficient. This is evident given the line representing the current sample size is above the curve of required sample sizes for this effect size. Hence, the study has enough participants to reliably detect even this relatively small difference, reducing the likelihood of a Type II error (failing to detect a real effect). Regarding statistical power (the probability that the test correctly rejects H_0 when it is false), the present sample size shows that this study is well-powered to detect even small differences between the proportions, assuming other parameters like α remain constant.

The adequacy of the sample size lends confidence to this project's results, suggesting that if the study did not find significant differences in certain comparisons (e.g., some of the χ^2 Test results; see *Chapter 5.3.1*), it is less likely due to insufficient data and more likely reflective of the actual similarities in perceptions across different groups. As well, the proportional effect size (-0.2), in conjunction with the power analysis, implies that the study design is robust for detecting even small differences in perceptions of AI and human compositions, and assures that the sample size is adequate to give the statistical tests enough sensitivity to detect the desired effects.

5.3 Data Analysis and Survey Evaluation

This section delineates the comprehensive data analysis and evaluation of the survey, encompassing a systematic examination of the responses gathered from the survey participants rigorously assessed through various statistical methodologies. The following analysis is twofold: first, it involves a detailed assessment of the responses to the survey questions, which were aimed at gauging the participants' ability to distinguish between AI-generated and human-composed music. This component of the analysis can be seen in Table 5.4, offering a clear and quantifiable insight into the respondents' perceptions and judgments. Second, the section provides an evaluation of the demographic data, meticulously collated to elucidate the distribution of musical expertise and background among the survey participants (see Table 5.5). The demographic analysis serves as a critical underpinning for understanding the diversity and representativeness of the survey sample. Together, these analyses form the cornerstone of the subsequent statistical testing and interpretation, underpinning the empirical findings that drive the conclusions of this research. It is worth noting that while I attempted to obtain a balanced distribution of musical demographics, it was unfeasible to limit the number of non-musician respondents and I believe that a left-skewed distribution is more representative of the general population (see Figure 5.3).

Question	#1 (AI)	#2 (AI)	#3 (Human)	#4 (AI)	#5 (Human)
AI - Strongly believe	8.20	8.40	10.20	7.00	5.60
AI - Mostly believe	21.80	17.80	21.40	22.20	11.00
AI - Somewhat believe	19.40	19.20	19.60	22.60	13.00
Human - Somewhat believe	15.40	16.20	18.00	18.00	20.80
Human - Mostly believe	27.00	26.00	20.60	23.00	29.40
Human - Strongly believe	8.20	12.40	10.20	7.20	20.20

Table 5.4: Survey results per question (by percentage of respondents; *n*=500)

Demographic	Percentage
Music teacher/professional	8.20
Music student	12.40
Amateur musician	28.20
Music listener/enjoyer	51.20

Table 5.5: Survey demographics per group (by percentage of respondents)

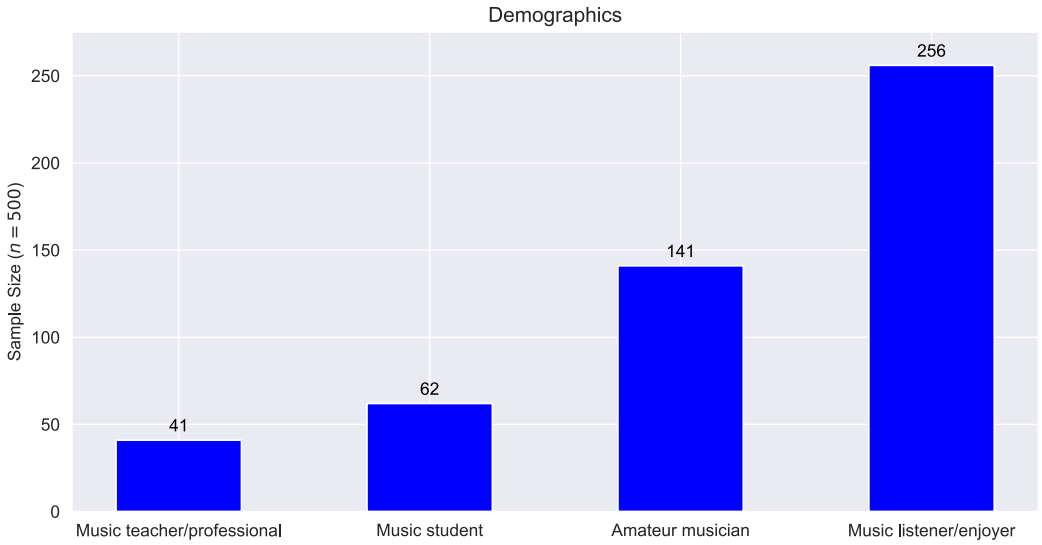


Figure 5.3: Survey demographics by count

5.3.1 Preliminary Chi-Square Analysis

This subsection presents the preliminary Chi-Square (χ^2) analysis conducted on the collected survey data, a pivotal component of the study's quantitative evaluation. The χ^2 Test – a fundamental statistical tool used to determine if two categorical variables have a significant association (Suresh, 2019) – is employed here to examine the independence of the categorical variables within the survey results. Specifically, the analysis aims to test hypotheses related to the frequency with which respondents identified music samples as either AI-generated or human-composed. This test is instrumental in determining whether there are statistically significant differences in the identification patterns across the various samples. As such, the following null/alternative hypotheses were tested:

- H_0: There is no statistically significant difference between the frequency of respondents identifying a music piece as AI-generated and the frequency of respondents identifying it as human-composed – i.e., any observed difference in the identification of AI and human compositions is due to random chance or sampling variability.
- H_a: There is a statistically significant difference between the frequency of respondents identifying a music piece as AI-generated and the frequency of respondents identifying it as human-composed – i.e., the pattern of responses is not what would be expected by random chance alone.

The results of this preliminary analysis (discussed in detail in the following section) are crucial in establishing the foundational understanding necessary for interpreting the broader implications of the survey findings and providing initial insights that inform the subsequent, more nuanced phases of statistical investigation.

5.3.1.1 Chi-Square Initial Results

These tests (including those in the following sections) were executed using the Python programming language (inside a **Jupyter Notebook** environment; see *Appendix E.5*), leveraging its powerful statistical capabilities and libraries. Python's **scipy.stats** module, renowned for its robust statistical functions, was utilized to perform the χ^2 Test, ensuring high accuracy and efficiency in the computation. This approach involved the aggregation of survey responses into contingency tables, where the frequencies of the responses for each music sample were tabulated into binary categories (see Figure 5.4) and compared against expected frequencies under the null hypothesis. By applying this method, I aimed to uncover any significant statistical differences in the participants' identification of the music as AI-generated or human-composed.

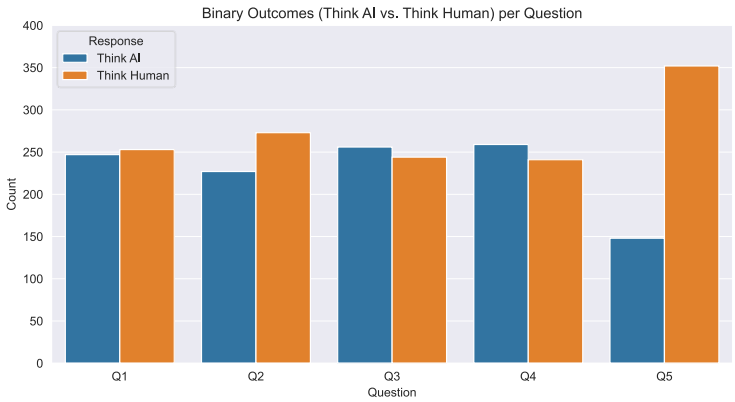


Figure 5.4: Contingency table of responses as binary outcomes

Question	χ^2 Statistic	p-value	Reject H_0?	Degrees of Freedom
Q1	0.036001	0.849513	False	1
Q2	2.120487	0.14534	False	1
Q3	0.144021	0.704316	False	1
Q4	0.324105	0.569151	False	1
Q5	43.423096	0.00000156	True	1

Table 5.6: χ^2 Test results per question

Given the results of the χ^2 Test (see Table 5.6), I find that for questions 1-4, the test fails to reject the null hypothesis. Here, failing to reject the null hypothesis implies that respondents could not reliably distinguish between AI and human-composed music (our intended result), indicating that there is no statistically significant difference in the respondents' ability to identify these music pieces as AI-generated or human-composed. These findings suggest that the level of musical ability does not significantly impact the respondents' perceptions of these particular music samples, pointing to a high level of realism in the AI-generated music for these samples. Likewise, only the test for question 5 rejects the null hypothesis; this significant result suggests that most respondents were able to distinguish this human-composed piece from the AI-generated compositions, indicating that certain qualities of this composition were more identifiable to the listeners. Hence, the following implications were noted:

- **Perceptual Uniformity in AI Compositions** – The lack of significant differences for Q1-Q4 underscores the effectiveness of the AI model in creating music that challenges listeners' ability to discern its origin, reflecting the AI's capability to mimic human compositions.
- **Distinguishing Human Compositions** – The significant result for Q5 highlights that certain human-composed music can possess distinct characteristics that make it more recognizable to listeners, contrasting with the AI-generated pieces.
- **Model Performance** – The general inability of respondents across various backgrounds to reliably identify the origin of most music samples (except for Q5) demonstrates the AI model's success in producing compositions that closely emulate human creations.

5.3.2 Kruskal-Wallis Test

The subsequent phase of this study's analysis involves the application of the Kruskal-Wallis Test to further investigate the survey data – a non-parametric statistical method used to determine if independent groups have the same mean (Lomuscio, 2021). Here, the test is specifically employed to discern whether the level of musical ability among respondents influences their capacity to accurately identify the origin of the music samples – whether AI-generated or human-composed. The Kruskal-Wallis Test is particularly suited for this analysis due to its efficacy in handling ordinal data and its ability to compare more than two independent groups. In this context, the groups are defined by the varying levels of musical expertise and ability reported by the survey participants; as such, the hypotheses formulated for this test are defined on the premise that musical training and experience may have a measurable impact on the perceptual differentiation between AI-generated and human-composed music (see Figure 5.5). Thus, the following hypotheses were tested:

- H_0: The distribution of responses (AI or Human) is the same across different levels of musical ability (the median response is equal across all groups) – i.e., the level of musical ability does not affect the ability to correctly identify AI-generated versus human-composed music.
- H_a: There is a difference in the distribution of responses among at least one of the levels of musical ability – i.e., at least one group's ability to identify AI-generated versus human-composed music is statistically different from the others.

This evaluation of the distribution of responses across different musical ability categories provides critical insights into the relationship between musical expertise and the recognition of AI involvement in music composition.

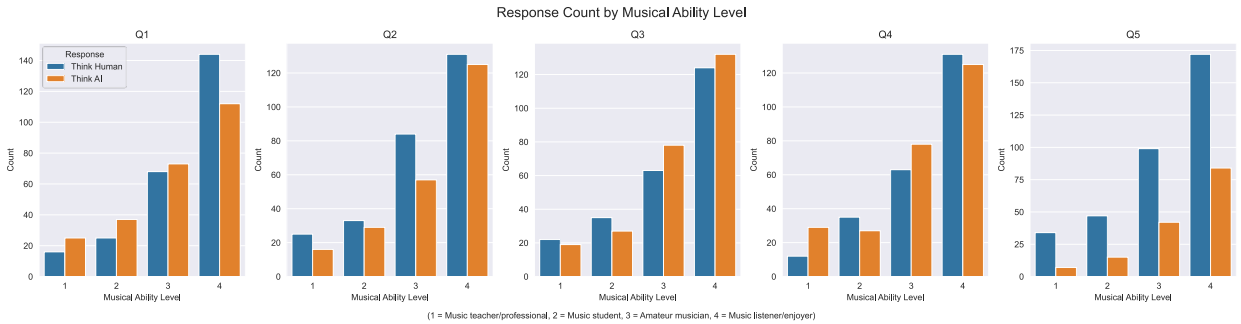


Figure 5.5: Count plot of survey responses by musical expertise

5.3.2.1 Kruskal-Wallis Initial Results

In implementing the test, my approach involved categorizing the survey responses based on the respondents' self-reported levels of musical ability and then analyzing these categories to determine if their perceptions of the music samples differed significantly. Each music sample was treated as a separate dataset, with the responses serving as independent observations. The Kruskal-Wallis Test was thus applied to compare the distribution of these responses across the various levels of musical expertise (see Figure 5.6). This methodology enabled a comprehensive evaluation of whether musical training and experience influenced the ability of respondents to distinguish between AI-generated and human-composed music.

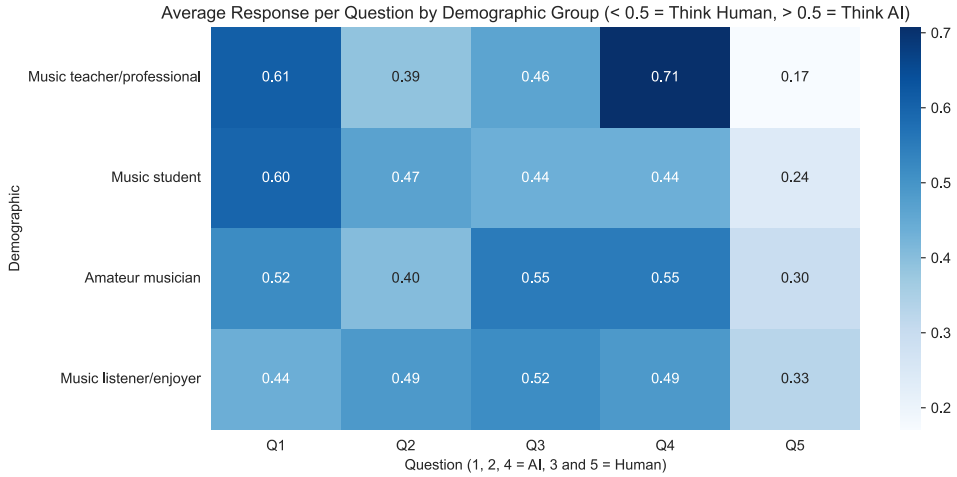


Figure 5.6: Heatmap of mean response distribution by demographic group

Question	Kruskal-Wallis Statistic	p-value	Reject H_0?
Q1	8.387896	0.03864	True
Q2	3.334107	0.342924	False
Q3	2.805512	0.422593	False
Q4	9.162945	0.027201	True
Q5	5.216878	0.156588	False

Table 5.7: Kruskal-Wallis Test results

Given the results of the Kruskal-Wallis Test (see Table 5.7), I find that for questions 2, 3, and 5, the test fails to reject the null hypothesis. This suggests that there is no significant difference across the demographic groups in terms of their perceptions of these music samples – i.e., the ability to identify these pieces as AI or human-composed does not seem to be significantly influenced by the respondents' level of musical ability. Likewise, the tests successfully reject the null hypothesis for **questions 1** and **4**; this implies that there is a statistically significant difference in how respondents from different demographic groups perceived the music. In other words, the level of musical ability appears to influence respondents' ability to distinguish between AI and human compositions for these particular music samples. Hence, the following implications were noted:

- **Variation in Perception by Demographics** – The results for Q1 and Q4 indicate that musical background may play a role in how certain pieces of music are perceived.
- **Consistency Across Groups** – The results for Q2, Q3, and Q5 suggest consistency in perception across different levels of musical experience, indicating that these pieces might have qualities that make them uniformly identifiable (or unidentifiable) as AI or human compositions.

5.3.3 Post-Hoc Analysis

Following the preliminary statistical evaluations conducted via the χ^2 and Kruskal-Wallis tests, this section delves into the post-hoc analyses – a crucial step in further elucidating the relationships uncovered in the initial findings and exploring the specific differences between groups identified in the earlier stages of this study. These analyses are particularly pertinent in cases where the initial tests indicate significant differences or trends that merit a more detailed investigation (McClenaghan, 2023). As such, they are instrumental in pinpointing the exact nature and locus of these differences, thereby providing a more granular understanding of the data.

Due to the limitations in variability and/or size of certain data subsets, the conventional approach of using Dunn's Test for post-hoc analysis following the Kruskal-Wallis Test was deemed unsuitable (see *Appendix E.5*). Instead, I adopted a tailored approach: employing the χ^2 Test for the larger sample groups and Fisher's Exact Test for the smaller sample groups. This decision was informed by the need to appropriately address the distinct characteristics of the data subsets, ensuring the accuracy and relevance of the post-hoc inquiries. The χ^2 Test was selected for its efficacy in analyzing larger samples where expected frequencies are sufficient, while Fisher's Exact Test was chosen for smaller samples due to its precision in handling cases with lower frequencies. The following subsections will detail the methodologies and findings of these post-hoc analyses, shedding light on the intricate patterns and associations within the research data.

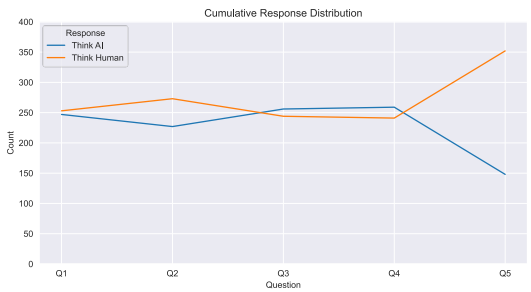


Figure 5.7: Cumulative response distribution of binary outcomes

5.3.3.1 Chi-Square Test of Independence

Building on the insights from the preliminary analyses, this post-hoc test seeks to address the potential relationships between the larger demographic categorizations of the survey respondents ("Amateur musician" and "Music listener/enjoyer") and their perceptions regarding the origins of the music samples – whether they were AI-generated or human-composed. The χ^2 Test of Independence is aptly suited for this exploration, as it is designed to analyze the frequency of occurrences within categorical data and assess whether the distribution of these frequencies is influenced by another categorical variable (Suresh, 2019). In applying this test, I aim to statistically ascertain if the differences in belief about the music's origin are merely coincidental or if they exhibit a significant association with the respondents' demographic groups. Thus, the following hypotheses were tested:

- H_0: The distribution of responses (AI or Human) is the same across the demographic groups being compared – i.e., there is no significant association between the demographic groups and the belief about the music's origin.
- H_a: The distribution of responses differs significantly across the demographic groups – i.e., there is a suggested association between the demographic groups and the belief about the music's origin.

This post-hoc analysis is pivotal in understanding the nuances of how various (less musically experienced) listener profiles might influence the discernment of AI and human elements in music, contributing to the depth and rigor of this dissertation's overall research findings.

5.3.3.2 Post-Hoc Chi-Square Test Results

My approach to performing the first post-hoc test involved meticulously structuring the survey data into cross-tabulated contingency tables (for the two larger groups) by respondents' beliefs about the music's origin against their demographic categorizations (see *Chapter 5.3.1*). The primary objective was to statistically assess the independence of these two variables using the χ^2 statistic and the corresponding p-values for each demographic group (see Table 5.8). This method allowed us to rigorously evaluate whether the observed frequencies of responses differed significantly from what would be expected under the null hypothesis of independence. By applying this analytical technique, I sought to illuminate any significant associations between the respondents' background in music and their perceptions of the music samples, thereby providing a nuanced understanding of the interplay between listener demographics and their ability to discern AI elements in music composition.

Question	χ^2 Statistic	p-value	Reject H_0?	Degrees of Freedom
Q1	2.040548	0.153154	False	1
Q4	1.284357	0.25709	False	1

Table 5.8: Post-Hoc χ^2 Test results

Following the post-hoc χ^2 Test (see Table 5.8), I find that for questions 1 and 4, the test fails to reject the null hypothesis. This suggests that there is no statistically significant difference in the way the larger demographic groups ("Music listener/enjoyer" and "Amateur musician") perceived the music as AI or human-composed for those two samples, also implying that the ability to identify the origin of those samples was similar across these groups. Likewise, the lack of significant differences indicates a consistency in perception across the largest demographic groups for these particular pieces of music. Hence, the following implications were noted:

- **AI Model Performance** – Since there was no significant difference in perception across the demographic groups for Q1 and Q4, this aligns with the overall finding that listeners, regardless of their musical background, found it challenging to distinguish between AI-generated and human-composed music. This supports the success of the AI models in creating music that is not easily distinguishable from human compositions.
- **General Audience Perception** – The results particularly highlight that the general (less musically experienced) audience did not perceive the music differently from other (more musical) groups, suggesting that the AI's output was uniformly convincing or ambiguous.

5.3.3.3 Fisher's Exact Test

For the two smaller demographic groups ("Music teacher/professional" and "Music student"), I employ Fisher's Exact Test – a statistical test used to determine if a significant (i.e., nonrandom) association exists between two categorical variables (Weisstein, 2023). Given the limited sample sizes within these subsets, Fisher's Exact Test provides a more appropriate methodological approach compared to the χ^2 Test, particularly when dealing with low-frequency counts in categorical data. The primary aim of this test is to rigorously determine if there are significant associations between the smaller (more musically experienced) demographic groups and their perceptions regarding the origins of the music samples (AI or human). As such, the following hypotheses were tested:

- H_0: There is no significant association between the demographic groups and the belief about whether the music is AI or human-composed – i.e., the proportions of responses are similar across the groups being compared.
- H_a: There is a significant association between the demographic groups and the belief about the music's origin – i.e., the proportions of responses differ between the groups.

By applying Fisher's Exact Test, I seek to uncover any significant relational patterns that may exist within these smaller groups, offering a deeper insight into how demographic groups with more musical experience discern between AI and human elements in music. This analysis is crucial for comprehensively understanding the nuanced differences in perception that might not be as pronounced or detectable in the larger, less experienced demographic cohorts.

5.3.3.4 Post-Hoc Fisher's Exact Test Results

Like the previous post-hoc test (see *Chapter 5.3.3.2*), I cross-tabulate 2x2 contingency tables from the initial χ^2 Test (see *Chapter 5.3.1*) to provide a precise arrangement of the observed frequencies – an essential step for the accurate application of Fisher's Exact Test. Here, the test is used to calculate the exact probability of obtaining the observed distribution of responses under the null hypothesis of no association between the demographic group and perceptions of the music's origin. This method was chosen for its suitability in handling smaller sample sizes, where the assumption of a normal distribution, as required in other tests, is not tenable.

Comparison	Odds Ratio	*p*-value	Reject H_0?
Q1 - 1 vs. 2	0.9472	1.0	False
Q1 - 2 vs. 1	0.9472	1.0	False
Q4 - 1 vs. 2	0.319212	0.008658	True
Q4 - 2 vs. 1	0.319212	0.008658	True

Table 5.9: Post-Hoc Fisher's Exact Test results

Following the post-hoc Fisher's Exact test (see Table 5.9), I find that for question 1, the test fails to reject the null hypothesis. This indicates that there is no statistically significant association between the smaller demographic groups ("Music teacher/professional" and "Music student") and their beliefs about whether the sample was AI-generated or human-composed – i.e., both groups responded similarly. In contrast, the test successfully rejects the null hypothesis for

question 4, suggesting that there is a statistically significant difference in how these two groups perceived the sample for that question. That is, the ability to identify Q4 as AI/human seems to differ significantly between these two groups. Hence, the following implications were noted:

- **Q1 (General Agreement)** – The lack of a significant difference in Q1 implies that both music teachers/professionals and music students had similar difficulty (or ease) in identifying the music's origin. This might suggest that Q1's composition was such that it equally challenged or fooled both groups.
- **Q4 (Significant Difference)** – The significant difference observed in Q4 suggests that one group was better able to distinguish the origin of the music than the other. This could be due to the specific characteristics of the music sample in Q4, which may have been more easily identifiable as AI or human-composed by one group over the other.

CHAPTER 6
DISCUSSION

In this chapter, I discuss the results of the model evaluation through the observational study analysis, the implications of this analysis as they pertain to this dissertation's research questions, and the limitations and methods of potential improvements of the generative system.

6.1 Findings

6.1.1 Observational Study Analysis Results

This section represents a comprehensive consolidation and examination of the findings derived from the various statistical tests performed on the survey data, synthesizing the outcomes of the χ^2, Kruskal-Wallis, and post-hoc analyses. This synthesis not only highlights the key patterns and trends observed across different demographic groups but also contextualizes these findings within the larger framework of this dissertation's research objectives. By amalgamating the results from the varied analytical approaches, I aim to provide a nuanced understanding of the respondents' abilities to differentiate between my AI-generated compositions and human-composed music, and how these abilities are influenced by factors such as musical expertise. The insights gleaned here are instrumental in advancing the academic discourse on the intersection of AI technology and **music perception**, forming the basis for the concluding arguments and recommendations of this dissertation.

6.1.1.1 Initial Chi-Square Analysis

The preliminary χ^2 tests were performed to assess whether there was a statistically significant difference in the frequency of respondents identifying music pieces as AI-generated versus human-composed (see *Chapter 5.3.1*). These tests indicated a general uniformity in perception across the larger demographic groups for most samples, suggesting that my AI model was effective in creating compositions indistinguishable from those composed by humans. Notably, the exception observed in one human-composed sample (Q5) – which was significantly identified as human-composed (see *Chapter 5.3.1.1*) – further underscores the discerning nature of the respondents and validates the reliability of the survey responses, thus providing a robust basis for the subsequent analyses.

6.1.1.2 Kruskal-Wallis Response Distribution Analysis

Subsequent Kruskal-Wallis tests were conducted to examine if there were differences in perceptions of AI-generated versus human-composed music across various demographic groups based on their musical ability (see *Chapter 5.3.2*). The tests revealed significant differences in perception for certain music samples (notably the samples in questions Q1 and Q4, produced by the two weaker models), suggesting that listeners' abilities to distinguish between AI and human compositions varied based on their musical background and supporting the hypothesis that musical training influences music perception. Interestingly, the test did not reveal a significant difference across groups for Q2 (generated by the best AI model weights). This suggests a uniformly high level of realism in the model's composition, as it successfully blurs the lines between AI-generated and human-composed music for listeners across all levels of musical expertise.

6.1.1.3 Post-Hoc Analyses

Post-hoc analyses were conducted to further explore the nuances in perception among different groups. Subsequent χ^2 tests, performed for the larger demographic groups (Amateur musician and Music listener/enjoyer), found no significant differences in the ability of these groups to differentiate between AI and human compositions for most of the samples – indicating a general uniformity in perception across these groups (see *Chapter 5.3.3.2*). These tests confirmed the initial findings of no significant differences in how these groups perceived most of the music samples, reinforcing my model's effectiveness in creating compositions that were broadly indistinguishable from human compositions. The Fisher's Exact tests, applied to the smaller demographic groups (Music teacher/professional and Music student), revealed significant differences in perception for Q4 (produced by the worst of the three models) – indicating distinct perceptual capabilities between more musically experienced groups (see *Chapter 5.3.3.4*). This highlighted that certain nuances in AI-generated music might be more discernible to those with specific types of musical training; however, this also contrasted with the findings for Q2 (where such differences were not observed), underscoring the exceptional performance of the best AI model weights in mimicking human composition.

6.1.2 Implications

The collective findings from this study's statistical analyses provide a comprehensive understanding of the perception of AI-generated music. While the model demonstrated a high level of proficiency in creating compositions that were broadly indistinguishable from human compositions, particularly in Q2, there were instances (e.g., Q4) where specific listener groups with higher levels of musical expertise could discern differences. These insights emphasize the

sophistication of the novel AI model and its potential in the field of music composition while also highlighting the subtle yet impactful role of musical training in the perception of music.

6.1.3 Answers to Research Questions

This section revisits and addresses the core research questions outlined in *Chapter 1.2*. These questions, fundamental to the scope and direction of this study, sought to explore the intersection of artificial intelligence and music composition, particularly focusing on the feasibility of AI in creating realistic choral music and the public's perception of such music. The subsequent paragraphs offer a detailed examination of the answers to each of these questions, drawing from the extensive analyses and findings presented in the preceding sections.

Research question one asked whether a current Machine Learning or Deep Learning architecture is feasible for generating musically realistic four-part (SATB) choral music, and if a suitable training dataset exists for this purpose. The findings from this study affirmatively indicate that not only is such an AI architecture feasible, but it also effectively generates choral compositions that are convincingly realistic. The deployment of a sophisticated hybrid AI model, trained on my carefully curated dataset, demonstrated a significant capability in mimicking the complexity and nuances of SATB choral music. This model's success is underpinned by the advancements in deep learning techniques and the availability of comprehensive musical datasets, which collectively facilitate the generation of high-quality, realistic choral music.

Research question two sought to determine whether a hybrid Artificial Intelligence model could generate choral music to a degree that the general public would struggle to differentiate from genuine human compositions. The empirical evidence gathered through the survey, analyzed using a series of statistical tests, suggests a positive response. The majority of the survey participants, encompassing a wide range of musical backgrounds, were not able to consistently distinguish

between the AI-generated and human-composed music samples. This outcome is indicative of the AI model's proficiency in producing choral music that not only resonates with the qualities of human composition but also convincingly blurs the lines between AI and human creativity in the realm of classical music.

Research question three delved into whether individuals of varying musical backgrounds perceive AI-generated classical music differently than non-musicians. The analysis of survey responses, particularly through the Kruskal-Wallis and post-hoc tests, revealed nuanced differences in perception based on the respondents' musical expertise. While the general trend suggested a uniform difficulty in distinguishing AI-generated music across all groups, certain subtleties in perception were observed, especially among those with more advanced musical training. This suggests that while some AI-generated classical music may be broadly perceived as realistic, the depth of musical background can influence the discernment of certain aspects of AI compositions, highlighting the complex interplay between musical expertise and the perception of AI-generated music.

In summary, the exploration of these pivotal research questions has yielded insightful revelations about the capabilities and impacts of AI in the domain of choral music composition. It is evident that modern ML/DL architectures, when coupled with appropriate datasets, are not only feasible but also proficient in creating musically realistic SATB choral compositions. The study highlights my model's success in emulating human compositional qualities and reveals that while the AI-generated music is broadly realistic to diverse audiences, nuances in musical training can subtly influence its perception, underscoring AI's significant yet complex role in the creative domain of music.

6.2 System Limitations

While revealing significant insights into AI-generated choral music, this study is subject to certain limitations that must be acknowledged for a comprehensive understanding of its scope and implications. First, the "CHORAL" dataset, pivotal in training the GTN model, presents constraints in terms of diversity and complexity. Consisting of only 1,000 MIDI files and limited to four-part (SATB) compositions, the dataset encompasses a wide variance in style, key, tempo, and time signatures. This heterogeneity, without standardization measures like transposing to a uniform key or restricting to common time, leads to challenges in the model's output, often resulting in randomness in note and time signature generation.

Second, the GTN model itself, while adept at completing songs given initial inputs, frequently struggles to produce "realistic" compositions from scratch. This necessitates multiple generation attempts to identify the most coherent output, a process underscored by retaining three sets of best model weights for user selection during generation. Additionally, the model's tendency to produce less coherent outputs over extended sequences, particularly beyond the initial one or two minutes of music, indicates potential limitations in capturing long-term structural relationships (a problem seen in many other state-of-the-art models; see *Chapter 2.2*). The absence of a more sophisticated, task-specific loss function may also be impeding the model's learning capabilities.

Lastly, the post-processing system, designed to refine the AI-generated compositions, faces its own set of limitations. While it aims to approximate common rules of voice leading in music theory, the effectiveness of these rules is not consistently ascertainable (and requires multiple passes through the composition), given the complexities of the MIDI structures and the challenges in translating intricate musical theory into programmable code (see *Appendix C*). Furthermore, evaluating the impact of these post-processing rules is complex without conducting a detailed

harmonic and formal analysis of each generated composition, both before and after processing. As such, these limitations highlight areas for potential improvement and future research, underscoring the need for more expansive datasets, enhanced model capabilities, and more sophisticated post-processing techniques. Acknowledging these constraints is crucial in contextualizing the study's findings and in guiding subsequent efforts to advance the field of AI-generated music.

6.2.1 Improvements

A critical improvement for this project's system lies in the expansion and diversification of the "CHORAL" dataset. Increasing the dataset size and incorporating a broader range of compositions, including those with more than four voices, can significantly enrich the model's learning base. Adding specialized corpora, such as a collection of **accompaniments** or **lyrics**, and categorizing songs by style, meter, tempo, and key could refine the model's output, ensuring a more nuanced generation of choral music (see *Chapter 7.2*). Grouping songs by these musical characteristics would allow the model to learn more specific patterns and styles, leading to outputs that are stylistically consistent and musically coherent (Gunasekar et al., 2023).

The current post-processing system, while functional, represents an approximation of complex voice leading rules in music theory. To advance this aspect, training an AI-based system such as a decision tree could be more effective in implicitly learning these rules than through explicit programming. This approach could help identify and rectify sophisticated voice leading and counterpoint errors, such as parallel fifths or tritone leaps. However, the challenge remains in algorithmically correcting these errors in a musically sensible manner and time complexity. Developing a system that combines error detection with intelligent correction mechanisms would mark a significant leap in automating the refinement of AI-generated compositions.

The music generator's current limitations, including its restrictions on generating tempo and key and time signature changes, could be addressed to allow more dynamic compositions. While these restrictions were initially imposed to maintain realism, gradually introducing controlled variability in these aspects could lead to more sophisticated and less predictable compositions. Furthermore, enhancing the generator's support for non-SATB (dictionary) structured MIDI generation would expand the system's applicability to a wider range of musical genres and formats, as well as enable support for other types of model architectures (e.g., MCs).

Improving the Choral-GTN model entails a multifaceted approach, starting with adjustments in the training process and exploring alternative architectures. Developing more sophisticated loss functions and auxiliary objectives that better quantify musical quality is key to enhancing the model's output. Integrating mechanisms like memory recall, multi-scale architectures, and cross-attention modules could enable the model to capture and replicate the complex structures inherent in music composition (Mangal et al., 2019). Implementing conditional blocks and attention biases based on music theory, embedding modulation over time, and experimenting with regularization techniques and attention windowing are additional strategies that could refine the model's understanding of musical structure and enhance its generative capabilities. These improvements aim not only to augment the technical proficiency of the model but also to elevate the artistic quality of the AI-generated music, aligning it more closely with human compositional standards.

While the transformer architecture has proven effective, there remains potential for exploring other, perhaps more advanced, architectures that could offer further improvements (see *Chapter 7.2*). Theoretical advancements in attention mechanisms, such as dynamic attention models or attention with adaptive scaling, might provide even finer control over the generative

process, allowing for more detailed and expressive musical outputs. Additionally, architectures that incorporate elements of reinforcement learning or unsupervised learning could offer new pathways for enhancing the model's ability to learn from and adapt to musical data without explicit supervision.

Among the architectures tested, hybrid models combining the transformer's attention mechanism with the generative capabilities of GANs (without the potential risk of mode collapse; Gainetdinov, 2023) or the sequential modeling strengths of LSTMs might warrant further exploration. Such hybrid models could leverage the strengths of each component architecture, potentially leading to innovations in AI-driven music generation that surpass the capabilities of this study's current system. Integrating a **human-in-the-loop**/RL approach could also significantly refine the training process by incorporating subjective feedback directly into model development (Justus, 2023). Additionally, employing a discriminator system (similar to a GAN) would allow the model to iteratively improve by distinguishing between higher and lower-quality compositions. This approach could enhance the model's generative accuracy through iterative/comparative learning.

CHAPTER 7
CONCLUSION

This chapter discusses the achievements and contributions of the dissertation with respect to its research goals and objectives and provides research questions for future work to expand on.

7.1 Conclusions

The primary objective of this dissertation was to assess the capability of a novel AI model to generate realistic four-part (SATB) choral music compositions that are indistinguishable from those composed by humans. The survey data, collected from a diverse range of respondents with varying levels of musical expertise (with the sample size validated through power and sensitivity analyses), provided a rich dataset for evaluating this objective. Likewise, the statistical analyses conducted – the χ^2, Kruskal-Wallis, and post-hoc tests – have yielded insightful results.

The findings from the Kruskal-Wallis tests, particularly for questions Q1 and Q4 in the survey, indicated a significant difference in the perception of music samples across the larger demographic groups. However, the subsequent χ^2 tests on these larger groups revealed no significant difference in the ability to distinguish between AI and human compositions. This suggests a general consistency in how the music was perceived, regardless of the listeners' musical background (i.e., a uniformity in how listeners of varying levels of musical experience were challenged). Hence, overall, my AI model successfully created music that was largely indistinguishable from human compositions, meeting the primary goal of the model's performance.

The Fisher's Exact tests conducted on the smaller demographic groups revealed a notable exception, however; particularly for Q4, a significant difference was observed between music teachers/professionals and music students. This suggests that certain nuances in the AI-generated music might be more discernible to those with specific types of musical training or experience. Nonetheless, the overall difficulty experienced by respondents across all levels of musical ability in distinguishing AI compositions from human ones is a testament to the sophistication and realism of the model-generated music. This finding is particularly significant in the context of the evolving role of AI in creative fields like music composition, underscoring the potential of AI to not only assist in but also autonomously create compositions that resonate with a wide audience.

7.1.1 Contributions

The work presented in this dissertation contributes significantly to the field of generative and AI-assisted music composition, particularly in the domain of classical choral music. A key contribution is the development and curation of the "CHORAL" dataset, a comprehensive and specialized collection of 1,000 SATB classical choral MIDIs. This dataset not only offers individual voice part splits for each piece but also includes an accompaniment-free version, four diverse augmentations, and a transposed set in C major/A minor (used to train the final model). Such a rich and varied dataset is unprecedented in this field and serves as a valuable resource for future research in AI-based music composition.

Furthermore, this dissertation introduces the benchmark "Choral-GTN" system, a novel hybrid AI system that leverages a transformer-based network with a callback-integrated music generation system. This model is intricately designed, combining advanced machine learning techniques with music theory and composition principles. The integration of a rule-based post-processing system enhances the system's ability to generate realistic and coherent choral

compositions. This system not only achieves the primary research goal of generating choral compositions from scratch but also demonstrates proficiency in completing partially started compositions based on provided note and duration seeds.

The observational study analysis revealed that my AI model was generally successful in creating music wherein listeners (irrespective of their musical background) found it challenging to distinguish from human compositions. The model successfully created pieces indistinguishable from human compositions, with the performance of the best model weights (in Q2) particularly noteworthy. However, the experiment also highlighted the intricacies of musical perception, particularly among those with specialized training, offering a deeper understanding of the interplay between AI-generated art and human expertise. As such, this research contributes significantly to the fields of AI in creative arts and musicology, showcasing both the capabilities and limitations of current AI technology in music composition and the nuanced dynamics of music perception.

Additionally, this dissertation makes a significant contribution through its highly comprehensive literature review, which meticulously details the evolution and state-of-the-art in classical music generation. It not only provides an extensive overview of various AI models and techniques used over time but also critically analyzes their strengths, limitations, and applicability to choral music. This thorough examination of existing research serves as a valuable guide for future studies and establishes a solid foundation for the development of advanced AI systems in music composition. Likewise, the methodology and findings from this research, in combination with the findings presented in the literature review, deepen the understanding of AI's capabilities in classical music generation and open pathways for more intuitive and sophisticated AI-assisted composition tools. The Choral-GTN system thus stands as a testament to the potential of AI in creative domains, bridging the gap between technological innovation and artistic expression.

7.2 Future Work

The research presented in this dissertation may be extended to numerous fields, including music education, compositional technologies, film and video game scoring, **Music Perception and Cognition (MPC)**, and **Music Information Retrieval (MIR)**. Although the Choral-GTN system and CHORAL dataset are designed specifically for the generation of four-part classical choral music, it could be expanded to include additional (i.e., 8-12+) voice parts, musical form constraint, piano/instrumental **accompaniment**, and a subsequent **lyric generation** model, which could be integrated into the system to provide a complete composition. Likewise, the system could be extended to create AI-generated arrangements of existing pieces given specific parameters (e.g., instruments, voices, form, etc.), or **transfer learning** could be utilized to retrain the model on instrumental music generation. Given the current state of the system, however, the model and post-processing system could be recombined into an extension for a music composition software such as MuseScore to create a "copilot" system for composers and arrangers, allowing for AI-assisted composition and educational tools (e.g., "autocompletion") in a manner similar to technologies like GitHub Copilot. Further, exploring the integration of emotional, dynamical, or thematic elements into the AI composition process could lead to more contextually aware and expressive music outputs, enhancing the application in storytelling mediums like cinema and theatre or providing methods of "phrasing" compositions in music performance. This study's evaluation survey and corresponding analysis may also serve as a baseline for future studies in both the ethics and detection of AI-generated music and the perception of AI compositions by the general public.

REFERENCES

Adler, S. (2016). *The Study of Orchestration* (Fourth edition). W. W. Norton & Company.

Agnew, S. (2019, July 18). *Working with MIDI data in Python using Mido*. Twilio Blog. https://www.twilio.com/blog/working-with-midi-data-in-python-using-mido

AIContentfy. (2023, March 5). *AI-generated music: A new form of art*. AIContentfy. https://aicontentfy.com/en/blog/ai-generated-music-new-form-of-art

Allan, M., & Williams, C. (2004). Harmonising Chorales by Probabilistic Inference. *Advances in Neural Information Processing Systems*, *17*. https://dl.acm.org/doi/10.5555/2976040.2976044

Alvarez, N. (2009, March). *Tomás Luis de Victoria*. https://www.uma.es/victoria/scores.html

Arya, A., Botelho, L., Cañete, F., Kapadia, D., & Salehi, Ö. (2022). *Music Composition Using Quantum Annealing* (arXiv:2201.10557). arXiv. https://doi.org/10.48550/arXiv.2201.10557

Asenahabi, B. (2019). *Basics of Research Design: A Guide to selecting appropriate research design*. *6*, 76–89.

Bajaj, A. (2022, July 21). *Understanding Gradient Clipping (and How It Can Fix Exploding Gradients Problem)*. Neptune.Ai. https://neptune.ai/blog/understanding-gradient-clipping-and-how-it-can-fix-exploding-gradients-problem

Band Pioneer. (2023, May 1). *Using AI to Create Music of the Future*. Band Pioneer. https://bandpioneer.com/reviews/exploring-the-new-frontier-of-ai-music-generators

Barla, N. (2022, July 22). *Dimensionality Reduction for Machine Learning*. Neptune.Ai. https://neptune.ai/blog/dimensionality-reduction

Benward, B., & Saker, M. (2020). *Music in Theory and Practice Volume 1* (10th edition). McGraw Hill.

Berto, F., & Tagliabue, J. (2017). Cellular Automata. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2022). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2022/entries/cellular-automata/

Bilotta, E., Pantano, P., & Talarico, V. (2000). *Music Generation through Cellular Automata: How to Give Life to Strange Creatures*. https://www.semanticscholar.org/paper/Music-Generation-through-Cellular-Automata%3A-How-to-Bilotta-Pantano/764f311fe082214c22e9def0947b010672e13ca3

Borisov, V., Leemann, T., Seßler, K., Haug, J., Pawelczyk, M., & Kasneci, G. (2024). Deep Neural Networks and Tabular Data: A Survey. *IEEE Transactions on Neural Networks and Learning Systems*, 1–21. https://doi.org/10.1109/TNNLS.2022.3229161

Briot, J.-P., Hadjeres, G., & Pachet, F.-D. (2019). *Deep Learning Techniques for Music Generation—A Survey* (arXiv:1709.01620). arXiv. https://doi.org/10.48550/arXiv.1709.01620

Burnham, K. (2020, May 6). *Artificial Intelligence vs. Machine Learning: What's the Difference?* Graduate Blog. https://graduate.northeastern.edu/resources/artificial-intelligence-vs-machine-learning-whats-the-difference/

Buys, J. (2011). *Generative Models of Music for Style Imitation and Composer Recognition*. University of Stellenbosch.

Cambridge. (2013). Reorchestration. In *Cambridge Dictionary* (4th Edition, p. 1856). Cambridge University Press. https://dictionary.cambridge.org/us/dictionary/english/reorchestration

Caren, M. (2020). TRoco: A generative algorithm using jazz music theory. *Proceedings of the 1st Joint Conference on AI Music Creativity*, 9. https://doi.org/10.5281/zenodo.4285336

Carnovalini, F., & Rodà, A. (2020). Computational Creativity and Music Generation Systems: An Introduction to the State of the Art. *Frontiers in Artificial Intelligence*, *3*. https://www.frontiersin.org/articles/10.3389/frai.2020.00014

Chase, S. (2020, June 17). *What Is Counterpoint In Music: A Complete Guide*. https://hellomusictheory.com/learn/counterpoint/

Chew, E. (2014). *Mathematical and Computational Modeling of Tonality: Theory and Applications* (Vol. 204). https://doi.org/10.1007/978-1-4614-9475-1

ChoralTech. (2023, February). *MIDI Inventory*. https://www.choraltech.us/inventory.htm

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., Belanger, D., Colwell, L., & Weller, A. (2022). *Rethinking Attention with Performers* (arXiv:2009.14794). arXiv. https://doi.org/10.48550/arXiv.2009.14794

Choubey, V. (2020, July 27). Understanding Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM). *Analytics Vidhya*. https://medium.com/analytics-vidhya/undestanding-recurrent-neural-network-rnn-and-long-short-term-memory-lstm-30bc1221e80d

Conklin, D. (2003). Music Generation from Statistical Models. *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*. http://dx.doi.org/10.1080/09298215.2016.1173708

Cravit, R. (2023, February 3). What is a Decision Tree & How to Make One. *Venngage*. https://venngage.com/blog/what-is-a-decision-tree/

Cristina, S. (2022, September 17). The Transformer Model. *MachineLearningMastery.Com*. https://machinelearningmastery.com/the-transformer-model/

Cuthbert, M. S. (2023, June 12). *User's Guide, Chapter 9: Chordify—Music21 Documentation*. Music21 Documentation. https://web.mit.edu/music21/doc/usersGuide/usersGuide_09_chordify.html

Cuthbert, M. S., & Ariza, C. (2010). Music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data. *11th International Society for Music Information Retrieval Conference*, 637–642. https://doi.org/10.5281/zenodo.1416114

Dai, S., Yu, H., & Dannenberg, R. B. (2022, December 4). What is missing in deep music generation? A study of repetition and structure in popular music. *Proceedings of the 23rd International Society for Music Information Retrieval Conference*. ISMIR 2022, Bengaluru, India. https://ismir2022program.ismir.net/poster_136.html

DeepAI. (2019a, May 17). *Artificial Intelligence*. DeepAI. https://deepai.org/machine-learning-glossary-and-terms/artificial-intelligence

DeepAI. (2019b, May 17). *Deep Learning*. DeepAI. https://deepai.org/machine-learning-glossary-and-terms/deep-learning

DeepAI. (2019c, May 17). *Machine Learning*. DeepAI. https://deepai.org/machine-learning-glossary-and-terms/machine-learning

DeepAI. (2019d, May 17). *Neural Network*. DeepAI. https://deepai.org/machine-learning-glossary-and-terms/neural-network

Dey, R. (2023, April 1). The Rise of Generative AI: Creating Art, Music, and Writing with Machine Learning. *Medium*. https://rajatendu.medium.com/the-rise-of-generative-ai-creating-art-music-and-writing-with-machine-learning-7e3ee96feb66

Douek, J. (2013). Music and emotion—A composer's perspective. *Frontiers in Systems Neuroscience*, *7*, 82. https://doi.org/10.3389/fnsys.2013.00082

Droit-Volet, S., Ramos, D., Bueno, L., & Bigand, E. (2013). Music, emotion, and time perception: The influence of subjective emotional valence and arousal? *Frontiers in Psychology*, *4*. https://www.frontiersin.org/articles/10.3389/fpsyg.2013.00417

D-Wave. (2020). *What is Quantum Annealing? D-Wave System Documentation*. https://docs.dwavesys.com/docs/latest/c_gs_2.html

E R, S. (2021, June 17). Understand Random Forest Algorithms With Examples. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

Ellenwood, C. (2018). *Music Theory III Course Packet*.

Ens, J., & Pasquier, P. (2020). *MMM: Exploring Conditional Multi-Track Music Generation with the Transformer* (arXiv:2008.06048). arXiv. https://doi.org/10.48550/arXiv.2008.06048

Eubank, N. (2022). *Internal and External Validity—Unifying Data Science*. https://www.unifyingdatascience.org/html/internal_v_external_validity.html

Fang, C., Li, Z., & Ye, Z. (2021). Automatic Music Creation Based on Bayesian Networks. *Proceedings of the 2020 4th International Conference on Vision, Image and Signal Processing*, 1–6. https://doi.org/10.1145/3448823.3448871

Fernando, B., Fromont, E., Muselet, D., & Sebban, M. (2012). Supervised learning of Gaussian mixture models for visual vocabulary generation. *Pattern Recognition*, *45*(2), 897–907. https://doi.org/10.1016/j.patcog.2011.07.021

Forsgren, S., & Martiros, H. (2022). *Riffusion—Stable diffusion for real-time music generation*. https://github.com/riffusion/riffusion

Foster, D., & Friston, K. (2023). *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play* (2nd edition). O'Reilly Media.

Freedline, A. (2021, February 4). Algorhythms: Generating Music with D-Wave's Quantum Annealer. *MIT 6.S089—Intro to Quantum Computing*. https://medium.com/mit-6-s089-intro-to-quantum-computing/algorhythms-generating-music-with-d-waves-quantum-annealer-95697ec23ccd

French, R. (2013). *Cellular Automata and Music: A New Representation* [Honors Thesis, Union College]. https://digitalworks.union.edu/theses/665

Gainetdinov, A. (2023, March 7). *GAN Mode Collapse explanation*. Medium. https://pub.towardsai.net/gan-mode-collapse-explanation-fa5f9124ee73

Garcia-Valencia, S. (2020). *Cross entropy as objective function for music generative models* (arXiv:2006.02217). arXiv. https://doi.org/10.48550/arXiv.2006.02217

Goetschalckx, L., Andonian, A., & Wagemans, J. (2021). Generative adversarial networks unlock new methods for cognitive science. *Trends in Cognitive Sciences*, *25*(9), 788–801. https://doi.org/10.1016/j.tics.2021.06.006

Gotham, M., Gullings, K., Hamm, C., Hughes, B., Jarvis, B., Lavengood, M., & Peterson, J. (2021). *Open Music Theory* (Version 2). Oklahoma State University. https://viva.pressbooks.pub/openmusictheory/

Green, D. M. (1979). *Form in Tonal Music: An Introduction to Analysis, Second Edition* (2nd edition). Holt, Rinehart and Winston.

Groeneveld, N. (2023, March 9). *Scikit-Learn: A Comprehensive Machine Learning Library for Python | LinkedIn*. https://www.linkedin.com/pulse/scikit-learn-comprehensive-machine-learning-library-niels-groeneveld/

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., & Li, Y. (2023). *Textbooks Are All You Need* (arXiv:2306.11644). arXiv. https://doi.org/10.48550/arXiv.2306.11644

Hadjeres, G., & Nielsen, F. (2020). Anticipation-RNN: Enforcing unary constraints in sequence generation, with application to interactive music generation. *Neural Computing and Applications*, *32*. https://doi.org/10.1007/s00521-018-3868-4

Hadjeres, G., Pachet, F., & Nielsen, F. (2017). *DeepBach: A Steerable Model for Bach Chorales Generation* (arXiv:1612.01010). arXiv. https://doi.org/10.48550/arXiv.1612.01010

Haefele, H. (2022, September 26). Composing vs. Arranging: What Musicians Should Know. *Hannah B Flute*. https://hannahbflute.com/2022/09/26/composing-vs-arranging-what-musicians-should-know/

Herremans, D., & Chew, E. (2016). *Tension ribbons: Quantifying and visualising tonal tension*.

Herremans, D., & Chew, E. (2019). MorpheuS: Generating Structured Music with Constrained Patterns and Tension. *IEEE Transactions on Affective Computing*, *10*(4), 510–523. https://doi.org/10.1109/TAFFC.2017.2737984

Ho, G. (2019, March 9). *Autoregressive Models in Deep Learning—A Brief Survey*. GeorgeHo.Com. https://www.georgeho.org/deep-autoregressive-models/

Hodgson, J. (2005, March 31). *Midi Choral Music*. http://www.learnchoralmusic.co.uk/

Högberg, J. (2005). *Wind in the Willows – Generating Music by Means of Tree Transducers*. *3845*, 153–162. https://doi.org/10.1007/11605157_13

Holtzman, S. R. (1981). Using Generative Grammars for Music Composition. *Computer Music Journal*, *5*(1), 51–64. https://doi.org/10.2307/3679694

Hsiao, W.-Y., Liu, J.-Y., Yeh, Y.-C., & Yang, Y.-H. (2021, January 7). *Compound Word Transformer: Learning to Compose Full-Song Music over Dynamic Directed Hypergraphs*. arXiv.Org. https://arxiv.org/abs/2101.02402v1

Huang, C.-Z. A. (2019). *Deep Learning for Music Composition: Generation, Recommendation and Control* [Harvard University]. https://dash.harvard.edu/handle/1/42029468

Huang, C.-Z. A., Cooijmans, T., Roberts, A., Courville, A., & Eck, D. (2017). Counterpoint by Convolution. *Proceedings of the 18th International Society for Music Information Retrieval Conference*, 211–218. https://doi.org/10.48550/arXiv.1903.07227

Huang, C.-Z. A., Duvenaud, D., Arnold, K. C., Partridge, B., Oberholtzer, J. W., & Gajos, K. Z. (2014). Active learning of intuitive control knobs for synthesizers using gaussian processes. *Proceedings of the 19th International Conference on Intelligent User Interfaces*, 115–124. https://doi.org/10.1145/2557500.2557544

Huang, C.-Z. A., Duvenaud, D., & Gajos, K. Z. (2016). ChordRipple: Recommending Chords to Help Novice Composers Go Beyond the Ordinary. *Proceedings of the 21st International Conference on Intelligent User Interfaces*, 241–250. https://doi.org/10.1145/2856767.2856792

Huang, C.-Z. A., Vaswani, A., Uszkoreit, J., Simon, I., Hawthorne, C., Shazeer, N. M., Dai, A. M., Hoffman, M., Dinculescu, M., & Eck, D. (2019). *Music Transformer: Generating*

Music with Long-Term Structure. International Conference on Learning Representations. https://doi.org/10.48550/arXiv.1809.04281

Huang, Q., Park, D. S., Wang, T., Denk, T. I., Ly, A., Chen, N., Zhang, Z., Zhang, Z., Yu, J., Frank, C., Engel, J., Le, Q. V., Chan, W., Chen, Z., & Han, W. (2023). *Noise2Music: Text-conditioned Music Generation with Diffusion Models* (arXiv:2302.03917). arXiv. https://doi.org/10.48550/arXiv.2302.03917

Hung, H.-T., Ching, J., Doh, S., Kim, N., Nam, J., & Yang, Y.-H. (2021). *EMOPIA: A Multi-Modal Pop Piano Dataset For Emotion Recognition and Emotion-based Music Generation* (arXiv:2108.01374). arXiv. https://doi.org/10.48550/arXiv.2108.01374

Hutchinson, R. (2023). *Music Theory for the 21st-Century Classroom*. University of Puget Sound. https://musictheory.pugetsound.edu/mt21c/MusicTheory.html

Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: Principles and practice* (2nd edition). OTexts.

IBM Cloud Education. (2020). *What is Machine Learning? | IBM*. https://www.ibm.com/topics/machine-learning

IBM Cloud Education. (2023). *What is Artificial Intelligence (AI) ? | IBM*. https://www.ibm.com/topics/artificial-intelligence

Jackson, E. (2020). Counterpoint | Music Theory, Composition & Polyphon. In *Encyclopedia Britannica*. Britannica. https://www.britannica.com/art/counterpoint-music

Jadhav, R., Mohite, A., Chakravarty, D., & Nalbalwar, S. (2023). *Automatic Music Generation Using Deep Learning*. 674–685. https://doi.org/10.2991/978-94-6463-196-8_51

Jagannathan, A., Chandrasekaran, B., Dutta, S., Patil, U. R., & Eirinaki, M. (2022). Original Music Generation using Recurrent Neural Networks with Self-Attention. *2022 IEEE*

International Conference On Artificial Intelligence Testing (AITest), 56–63. https://doi.org/10.1109/AITest55621.2022.00017

Jean, J. (2021, August 10). *Music Theory and Practice – Introduction to Counterpoint*. Perennial Music and Arts. https://www.perennialmusicandarts.com/post/counterpoint

Jha, A. (2023, October 20). *Power of NumPy: A Fundamental Python Library for Numerical Computing | LinkedIn*. LinkedIn. https://www.linkedin.com/pulse/power-numpy-fundamental-python-library-numerical-computing-akash-jha-qwgyf/

Ji, S., Luo, J., & Yang, X. (2020). *A Comprehensive Survey on Deep Music Generation: Multi-level Representations, Algorithms, Evaluations, and Future Directions* (arXiv:2011.06801). arXiv. https://doi.org/10.48550/arXiv.2011.06801

Jordanous, A., & Keller, B. (2016). Modelling Creativity: Identifying Key Components through a Corpus-Based Approach. *PLOS ONE*, *11*(10), e0162959. https://doi.org/10.1371/journal.pone.0162959

Justus, A. A. (2023, September 11). *Music Generation using Human-In-The-Loop Reinforcement Learning*. LinkedIn. https://www.linkedin.com/pulse/sneak-peak-music-generation-using-human-in-the-loop-aju-ani-justus/?utm_source=rss&utm_campaign=articles_sitemaps&utm_medium=google_news

Kamsetty, A. (2020, October 6). Hyperparameter Optimization for

Karakus, E., & Kose, H. (2020). Conditional restricted Boltzmann machine as a generative model for body-worn sensor signals. *IET Signal Processing*, *14*(10), 725–736. https://doi.org/10.1049/iet-spr.2020.0154

Keller, R., & Morrison, D. (2007). A grammatical approach to automatic improvisation. *Proceedings of the 4th Sound and Music Computing Conference, SMC 2007*.

Kitahara, T. (2017). Music Generation Using Bayesian Networks. *Machine Learning and Knowledge Discovery in Databases*, 368–372. https://doi.org/10.1007/978-3-319-71273-4_33

Kostka, S., Payne, D., & Almén, B. (2017). *Tonal Harmony* (8th edition). McGraw Hill.

Kumar, A. (2022, November 6). Differences: Decision Tree & Random Forest. *Data Analytics*. https://vitalflux.com/differences-between-decision-tree-random-forest/

Kumar, K. (2023, January 26). *Probabilistic Neural Networks: An Introduction to Probability Theory in Machine Learning*. Medium. https://ai.plainenglish.io/probabilistic-neural-networks-an-introduction-to-probability-theory-in-machine-learning-844091af3500

Lagbayi, K. (2022, November 17). What Is Tacet in Music- A Quick Guide. *Phamox Music*. https://phamoxmusic.com/tacet-in-music/

Lavrenko, V., & Pickens, J. (2003). Polyphonic music modeling with random fields. *Proceedings of the Eleventh ACM International Conference on Multimedia*, 120–129. https://doi.org/10.1145/957013.957041

Lei, L., Sun, Y., Liu, Y., Roxas, R., & Raga, R. (2022). Research and Implementation of Text Generation Based on Text Augmentation and Knowledge Understanding. *Computational Intelligence and Neuroscience*, *2022*, 1–10. https://doi.org/10.1155/2022/2988639

Lerdahl, F., & Jackendoff, R. S. (1983). *A Generative Theory of Tonal Music*. MIT Press. https://www.amazon.com/Generative-Theory-cognitive-theory-representation/dp/0262120941/

Lewis, G. E. (2000). Too Many Notes: Computers, Complexity and Culture in "Voyager." *Leonardo Music Journal*, *10*, 33–39.

Li, C. (2019, September 24). *A Retrospective of AI + Music*. Medium. https://blog.prototypr.io/a-retrospective-of-ai-music-95bfa9b38531

Li, Y., & Linberg, J. (2023). *Music Generation with Generative Adversarial Networks* [Jönköping University]. https://urn.kb.se/resolve?urn=urn:nbn:se:hj:diva-62106

Liu, J., Dong, Y., Cheng, Z., Zhang, X., Li, X., Yu, F., & Sun, M. (2022). Symphony Generation with Permutation Invariant Language Model. *Proceedings of the 23rd International Society for Music Information Retrieval Conference*, 551–558. https://doi.org/10.5281/zenodo.7316721

Lomuscio, S. (2021, December 7). *Getting Started with the Kruskal-Wallis Test | UVA Library*. https://library.virginia.edu/data/articles/getting-started-with-the-kruskal-wallis-test

Lu, P., Tan, X., Yu, B., Qin, T., Zhao, S., & Liu, T.-Y. (2022). MeloForm: Generating Melody with Musical Form based on Expert Systems and Neural Networks. *Proceedings of the 23rd International Society for Music Information Retrieval Conference*, 567–574. https://doi.org/10.5281/zenodo.7316725

Ma, D., Liu, B., Qiao, X., Cao, D., & Yin, G. (2020). Coarse-To-Fine Framework For Music Generation via Generative Adversarial Networks. *Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference & 2020 3rd International*

Conference on Big Data and Artificial Intelligence, 192–198. https://doi.org/10.1145/3409501.3409534

Mangal, S., Modak, R., & Joshi, P. (2019). LSTM Based Music Generation System. *IARJSET*, *6*(5), 47–54. https://doi.org/10.17148/IARJSET.2019.6508

Mao, H. H., Shin, T., & Cottrell, G. W. (2018). DeepJ: Style-Specific Music Generation. *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, 377–382. https://doi.org/10.1109/ICSC.2018.00077

Mariani, G., Tallini, I., Postolache, E., Mancusi, M., Cosmo, L., & Rodolà, E. (2023). *Multi-Source Diffusion Models for Simultaneous Music Generation and Separation* (arXiv:2302.02257). arXiv. https://arxiv.org/abs/2302.02257v3

McClenaghan, E. (2023, March 20). *Post-Hoc Tests in Statistical Analysis*. Neuroscience from Technology Networks. http://www.technologynetworks.com/neuroscience/articles/post-hoc-tests-in-statistical-analysis-371174

Mehra, S., & Hasanuzzaman, M. (2020). *Detection of Offensive Language in Social Media Posts.* [Master Thesis, Cork Institute of Technology]. https://doi.org/10.13140/RG.2.2.23097.80485

Melis, G., Dyer, C., & Blunsom, P. (2017, July 18). *On the State of the Art of Evaluation in Neural Language Models*. arXiv.Org. https://arxiv.org/abs/1707.05589v2

Merritt, R. (2022, March 25). *What Is a Transformer Model?* NVIDIA Blog. https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/

Micchi, G., Bigo, L., Giraud, M., Groult, R., & Levé, F. (2021). I Keep Counting: An Experiment in Human/AI Co-creative Songwriting. *Transactions of the International Society for Music Information Retrieval*, *4*(1), Article 1. https://doi.org/10.5334/tismir.93

Miranda, E. R. (Ed.). (2022). *Quantum Computer Music: Foundations, Methods and Advanced Concepts* (1st ed. 2022 edition). Springer.

Miranda, E. R., & Basak, S. T. (2021). *Quantum Computer Music: Foundations and Initial Experiments* (arXiv:2110.12408). arXiv. https://doi.org/10.48550/arXiv.2110.12408

Miranda, E. R., & Shaji, H. (2023). Generative Music with Partitioned Quantum Cellular Automata. *Applied Sciences*, *13*(4), Article 4. https://doi.org/10.3390/app13042401

Miranda, E. R., Yeung, R., Pearson, A., Meichanetzidis, K., & Coecke, B. (2021). *A Quantum Natural Language Processing Approach to Musical Intelligence* (arXiv:2111.06741). arXiv. https://doi.org/10.48550/arXiv.2111.06741

Mittal, G., Engel, J., Hawthorne, C., & Simon, I. (2021). *Symbolic Music Generation with Diffusion Models* (arXiv:2103.16091). arXiv. https://doi.org/10.48550/arXiv.2103.16091

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning* (2nd Edition). The MIT Press.

Moses, K. (2021, July 18). Encoder-Decoder Seq2Seq Models, Clearly Explained. *Analytics Vidhya*. https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b

Murchison, A. (2003). *Cellular Automata in Music*. http://www.science.smith.edu/~jfrankli/354s03/students/Anna/present/ca.html

MuseScore. (2022, December 6). *Muse Hub*. Muse Hub. https://www.musehub.com

Naruse, D., Takahata, T., Mukuta, Y., & Harada, T. (2022). Pop Music Generation with Controllable Phrase Lengths. *Proceedings of the 23rd International Society for Music Information Retrieval Conference*, 125–131. https://doi.org/10.5281/zenodo.7316611

Nassif, A. B., Shahin, I., Attili, I., Azzeh, M., & Shaalan, K. (2019). Speech Recognition Using Deep Neural Networks: A Systematic Review. *IEEE Access*, *7*, 19143–19165. https://doi.org/10.1109/ACCESS.2019.2896880

Neves, P. L. T., Fornari, J., & Florindo, J. B. (2022). Generating music with sentiment using Transformer-GANs. *Proceedings of the 23rd International Society for Music Information Retrieval Conference*, 717–725. https://doi.org/10.5281/zenodo.7316763

Nevill-Manning, C. G., & Witten, I. H. (1997). *Identifying Hierarchical Structure in Sequences: A linear-time algorithm* (arXiv:cs/9709102). arXiv. https://doi.org/10.48550/arXiv.cs/9709102

Njagi, M. (2022, October 31). *Why Deep Learning Underperforms with Tabular Data*. Medium. https://heartbeat.comet.ml/why-deep-learning-underperforms-with-tabular-data-11435a0aba86

ODSC Community. (2020, December 15). Understanding the Mechanism and Types of Recurrent Neural Networks. *Open Data Science - Your News Source for AI, Machine Learning & More*. https://opendatascience.com/understanding-the-mechanism-and-types-of-recurring-neural-networks/

Oshiro, S. (2022). *QuiKo: A Quantum Beat Generation Application* (arXiv:2204.04370). arXiv. https://doi.org/10.48550/arXiv.2204.04370

Pai, A. (2020, February 17). CNN vs. RNN vs. ANN - Analyzing 3 Types of Neural Networks in Deep Learning. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/

Pandata. (2023, April 28). Mastering Statistical Analysis with Statsmodel Library (with Code). *Medium*. https://medium.com/@panData/mastering-statistical-analysis-with-statsmodel-library-with-code-d59e84c5b371

Parent, E. (2023, January 21). *Composing music with cellular automata*. Deepnote. https://deepnote.com/@essia/Composing-music-with-cellular-automata-fcc5111a-b546-4c3a-b633-8ce733ced853

Perktold, J., Seabold, S., & Taylor, J. (2023, December 14). *statsmodels.stats.power.NormalIndPower—Statsmodels 0.14.1*. Statsmodels Documentation. https://www.statsmodels.org/stable/generated/statsmodels.stats.power.NormalIndPower.html

Raffel, C. (2023, March 3). *pretty_midi—Pretty_midi 0.2.10 documentation*. https://craffel.github.io/pretty-midi/

Ray, S. (2017, September 8). Top 10 Machine Learning Algorithms (with Python and R Codes). *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/

Refaeilzadeh, P., Tang, L., & Liu, H. (2009). Cross-Validation. In L. LIU & M. T. ÖZSU (Eds.), *Encyclopedia of Database Systems* (pp. 532–538). Springer US. https://doi.org/10.1007/978-0-387-39940-9_565

Rocca, J. (2021, March 21). *Understanding Variational Autoencoders (VAEs)*. Medium. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

Scribbr. (2017). *Research Methods | Definitions, Types, Examples*. Scribbr. https://www.scribbr.com/category/methodology/

Selvaraj, N. (2022, October 5). *Hyperparameter Tuning Using Grid Search and Random Search in Python*. KDnuggets. https://www.kdnuggets.com/hyperparameter-tuning-using-grid-search-and-random-search-in-python

Selvaraj, N. (2023, February). *Python Pickle Tutorial: Object Serialization*. https://www.datacamp.com/tutorial/pickle-python-tutorial

Sethi, V. (2019, December 29). *Types of Neural Networks (and what each one does!) Explained*. Medium. https://towardsdatascience.com/types-of-neural-network-and-what-each-one-does-explained-d9b4c0ed63a1

Shantikumar, S. (2018). *Methods of sampling from a population | Health Knowledge*. HealthKnowledge. https://www.healthknowledge.org.uk/public-health-textbook/research-methods/1a-epidemiology/methods-of-sampling-population

Singh, M., & Mehr, S. A. (2023). Universality, domain-specificity and development of psychological responses to music. *Nature Reviews Psychology*, *2*(6), Article 6. https://doi.org/10.1038/s44159-023-00182-z

Singh, R. (2020, June 19). *Understanding neural networks through visualization | Druva*. Druva. https://www.druva.com/blog/understanding-neural-networks-through-visualization

Stanek, M. (2017, September 15). Moravec's paradox. *Medium*. https://medium.com/@froger_mcs/moravecs-paradox-c79bf638103f

Statistics Solutions. (2021, August 10). *Statistical Power Analysis*. Statistics Solutions. https://www.statisticssolutions.com/dissertation-resources/sample-size-calculation-and-sample-size-justification/statistical-power-analysis/

Steinberg, L. (1994). Research methodology for AI and design. *AI EDAM*, *8*(4), 283–287. https://doi.org/10.1017/S0890060400000962

Streefkerk, R. (2019, May 15). *Internal vs. External Validity | Understanding Differences & Threats*. Scribbr. https://www.scribbr.com/methodology/internal-vs-external-validity/

Suha, S. (2018, December 15). *A Comprehensive Guide to Convolutional Neural Networks—The ELI5 way | Saturn Cloud Blog*. SaturnCloud. https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

Suresh, A. (2019, November 27). What is a Chi-Square Test? Formula, Examples & Application. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2019/11/what-is-chi-square-test-how-it-works/

Szelogowski, D. (2021). *Generative Deep Learning for Virtuosic Classical Music: Generative Adversarial Networks as Renowned Composers* (arXiv:2101.00169). arXiv. https://doi.org/10.48550/arXiv.2101.00169

Szelogowski, D. (2022). *Deep Learning for Musical Form: Recognition and Analysis* [University of Wisconsin - Whitewater]. https://doi.org/10.13140/RG.2.2.33554.12481

Szelogowski, D. (2023). *Danielathome19/Choral-GTN* [GitHub Repository]. https://github.com/danielathome19/Choral-GTN (Original work published 2022)

Szelogowski, D. (2023, December 1). *Pollfish—Dissertation Evaluation*. https://wss.pollfish.com/link/b4f0e52b-1702-4dc5-b9ac-c24ac6f6eac2

Takyar, A. (2023, October 30). *Hybrid AI: A comprehensive overview*. LeewayHertz - AI Development Company. https://www.leewayhertz.com/hybrid-ai/

TensorFlow. (2023, October 4). *Tf.keras.layers.TextVectorization | TensorFlow v2.14.0*. TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization

Tewari, U. (2021, November 10). Regularization—Understanding L1 and L2 regularization for Deep Learning. *Analytics Vidhya*. https://medium.com/analytics-vidhya/regularization-understanding-l1-and-l2-regularization-for-deep-learning-a7b9e4a409bf

Tham, I. (2021, August 30). *Generating Music using Deep Learning*. Medium. https://towardsdatascience.com/generating-music-using-deep-learning-cb5843a9d55e

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, *LIX*(236), 433–460. https://doi.org/10.1093/mind/LIX.236.433

Van Der Merwe, B., & Schulze, W. (2011). Music Generation with Markov Models. *IEEE MultiMedia*, *18*(3), 78–85. https://doi.org/10.1109/MMUL.2010.44

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need* (arXiv:1706.03762). arXiv. https://doi.org/10.48550/arXiv.1706.03762

Voita, L. (2023, April 9). *Seq2seq and Attention*. Lena-Voita. https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Waldrop, M. M. (2019). What are the limits of deep learning? *Proceedings of the National Academy of Sciences*, *116*(4), 1074–1077. https://doi.org/10.1073/pnas.1821594116

Weisstein, E. W. (2023). *Fisher's Exact Test* [Text]. Wolfram Research, Inc. https://mathworld.wolfram.com/

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T.-Y. (2020). *On Layer Normalization in the Transformer Architecture* (arXiv:2002.04745). arXiv. https://doi.org/10.48550/arXiv.2002.04745

Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., & Singh, V. (2021). *Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention* (arXiv:2102.03902). arXiv. https://doi.org/10.48550/arXiv.2102.03902

Yanchenko, A. K., & Mukherjee, S. (2018). *Classical Music Composition Using State Space Models* (arXiv:1708.03822). arXiv. https://doi.org/10.48550/arXiv.1708.03822

Yang, L., Zhang, Z., Song, Y., Hong, S., Xu, R., Zhao, Y., Zhang, W., Cui, B., & Yang, M.-H. (2023). *Diffusion Models: A Comprehensive Survey of Methods and Applications* (arXiv:2209.00796). arXiv. https://doi.org/10.48550/arXiv.2209.00796

Yegulalp, S. (2024, January 5). *What is TensorFlow? The machine learning library explained*. InfoWorld. https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html

Yin, Z., Reuben, F., Stepney, S., & Collins, T. (2023). Deep learning's shallow gains: A comparative evaluation of algorithms for automatic music generation. *Machine Learning*, *112*(5), 1785–1822. https://doi.org/10.1007/s10994-023-06309-w

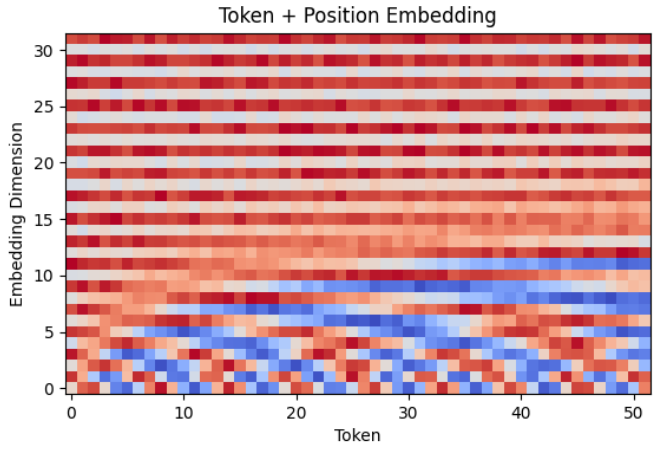
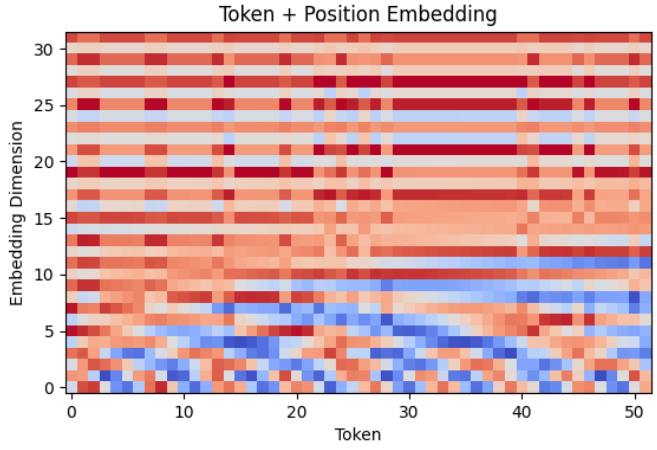
Zhou, J., Zhu, H., & Wang, X. (2023). *Choir Transformer: Generating Polyphonic Music with Relative Attention on Transformer* (arXiv:2308.02531). arXiv. https://doi.org/10.48550/arXiv.2308.02531

APPENDICES

APPENDIX A
EXAMPLE TRAINING DATA AND ATTENTION PLOTS

This appendix contains examples of the data used in training the model architecture plotted as embedding graphs, including audio and text input, as well as sample attention plots during model generation. The embedding plots represent the original dataset, i.e., the "Combined Transposed" dataset after interweaving voice parts.

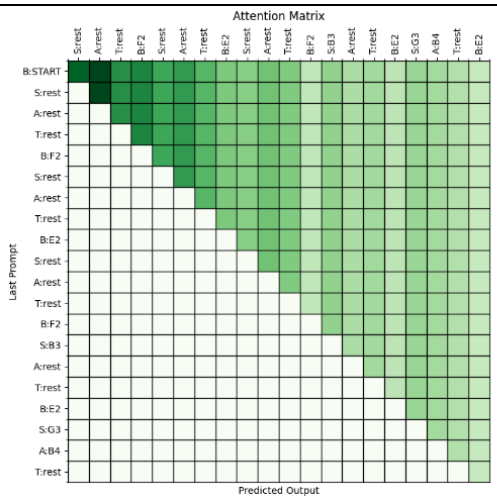
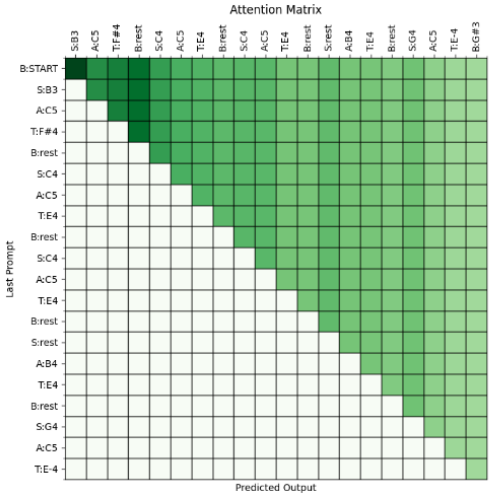
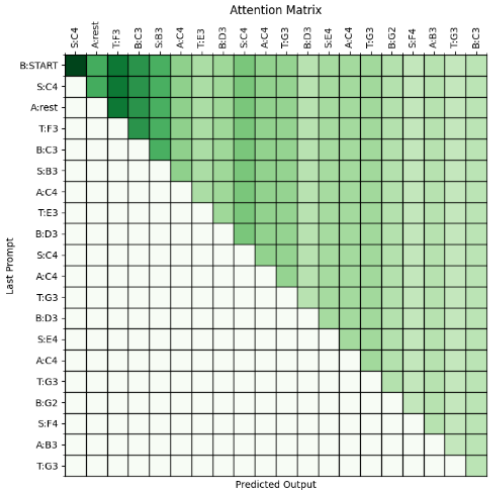
Model Input Data

	Combined Note Token and Position Embeddings
	Combined Duration Token and Position Embeddings

Notes: S:C5 A:E4 T:G3 B:C3 S:B4 A:E4 T:G3 B:C3 S:A4 A:E4 T:C4 B:C3 S:G4 A:E4 T:C4 B:C3 S:rest A:rest T:rest B:C3 S:rest A:rest T:rest B:C3 S:G4 A:E4 T:C4 B:C3 S:A4 A:E4 T:C4 B:D3 S:E4 A:E4 T:C4 B:D3 S:G4 A:D4 T:A3 B:D3 S:G4 A:D4 T:A3 **Durations:** 3.0 3.0 3.0 3.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 3.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.5 1.0 1.0 1.0 0.5 1.5 1.5 1.5 1.0 0.5 0.5 0.5	Sample model input tokens (for seeded generation); produced by **data_utils.py** (see *Appendix E.3*) from Bruckner's Locus Iste (see *Appendix B*)
S:rest 4 4.0 6 A:A5 173 3.0 7 T:B-3 72 4.0 6 B:E4 63 1.0 2 S:rest 4 4.0 6 A:rest 3 3.0 7 T:A3 15 4.0 6 B:E4 63 1.0 2 S:rest 4 4.0 6 A:E5 57 3.0 7 T:F#3 79 4.0 6 B:rest 2 0.5 3 S:rest 4 4.0 6 A:D5 54 3.0 7 T:E3 29 4.0 6 B:E4 63 0.5 3 S:rest 4 4.0 6 A:E5 57 3.0 7 T:F3 45 4.0 6 B:E4 63 1.0 2 S:rest 4 4.0 6 A:D5 54 3.0 7 T:F3 45 4.0 6 B:E4 63 1.0 2 S:rest 4 4.0 6 A:E5 57 3.0 7 T:G#3 78 4.0 6 B:E4 63 1.0 2	Sample model output tokens (used by the Music Generator callback; see *Appendix E.2*), including sample note, note vocab index, sample duration, and duration vocab index

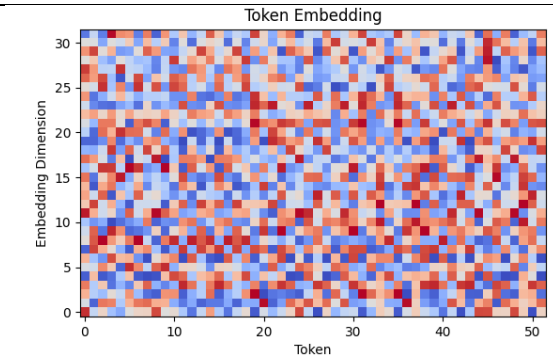
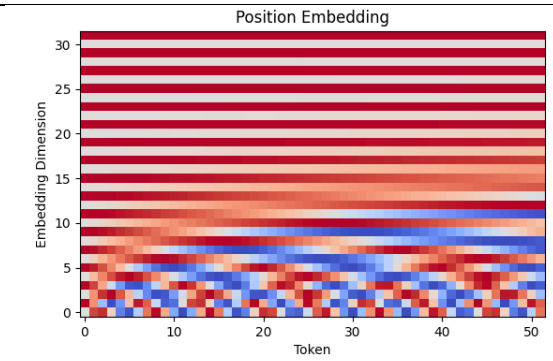
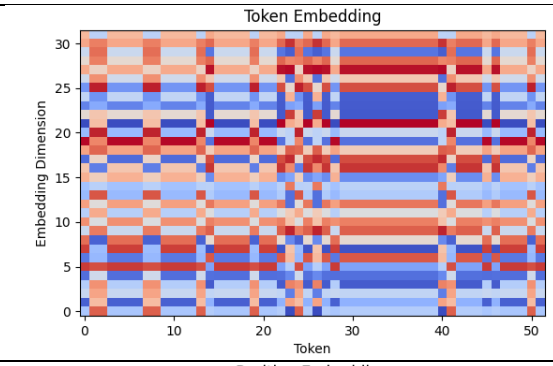
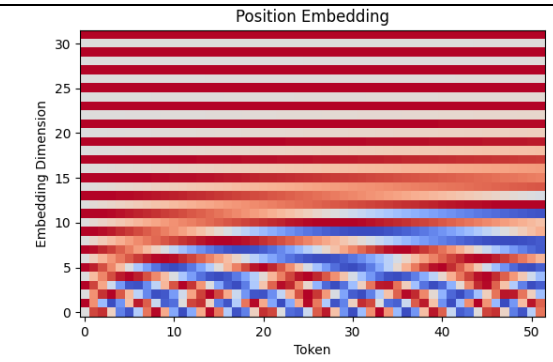
A.1 Model Attention

The following figures display sample attention plots for the three sets of model weights (from best to worst):

	Transposed_3
	Transposed_2
	Transposed_13

A.2 Individual Embeddings

The following figures display each of the dataset embeddings before being combined for the model input:

	Note token embeddings
	Note position embeddings
	Duration token embeddings
	Duration position embeddings

APPENDIX B
COMPOSITIONS USED FOR TRAINING

This appendix contains an alphabetical listing of each piece of music used as training data for the model architecture (grouped by their respective composer) and the list of extracted features.

Aaron Copland

At the river
Ching-a-ring chaw
I bought me a cat
Long time ago
Stomp your foot
Zions walls

Abraham Kaplan

The eight days of Chanukah - 1. Haneirot halalu
The eight days of Chanukah - 3. Maoz tzur
The eight days of Chanukah - 4. Mamlakha
The eight days of Chanukah - 5. Am Ne'emaney
The eight days of Chanukah - 6. Heikhalo
The eight days of Chanukah - 7. Malki
The eight days of Chanukah - 8. Ikhlu
The Song of Songs, which is Solomon's - 10. Look not upon me, because I am black
The Song of Songs, which is Solomon's - 11. Thou hast ravished my heart
The Song of Songs, which is Solomon's - 2. The Song of Songs
The Song of Songs, which is Solomon's - 3. Let him kiss me
The Song of Songs, which is Solomon's - 5. The voice of my beloved
The Song of Songs, which is Solomon's - 7. Who is this that cometh out of the wilderness
The Song of Songs, which is Solomon's - 8. I am black and comely

Adrian Batten

O Sing Joyfully

Adrian Willaert

Dulces exuviae

Alan Bergman
Nice 'n' easy
Still, still, still

Alessandro Scarlatti
Ad Te Domine levavi

Alexander Borodin
Polovtsian dances (m. 547-end)
Polovtsian dances (m. 95-210)

Alfred Burt
All on a Christmas morning
Bright, bright the holly berries
Caroling, caroling
Jesu parvule
O hearken ye
We'll dress the house

Alice Parker
Behold! the grace appears
Joy to the world! The Lord is come
While shepherds watched their flocks by night

Alicia S Carpenter
Zigeunerleben (Gypsy life)

Andrew Jacobson
A lovely rose is sprung

Andrew Lloyd Webber
Pie Jesu

Anonymous
A los maitines era
Agneau de Dieu
Amen, gloire et louange
Aquella fuerza grande
Con qué la lavaré
Dadme albricias
Devenez ce que vous recevez
Dindirindin
Estas noches atan largas
God is love
Hanacpachap
Je vous ai choisis

Laudate Dominum
Llaman a Teresica
Muchos van de amor heridos
No la debemos dormir
Por ese mar de Helesponto
Rey a quien reyes adoran
Ríu, Ríu, Chíu
Saint le seigneur
Serrana dónde dormisteis
Verbum caro factum est
Vi los barcos madre

Anton Bruckner

In te, Domine speravi
Locus Iste *(including 2 variant MIDIs from unnamed arrangers)*
Salvum fac
Te deum
Te ergo

Anton Diabelli

Puer natus est nobis

Antonio Lotti

Agnus Dei
Credo
Gloria
Kyrie
Sanctus et Benedictus

Antonio Vivaldi

Credo in unum Deum
Crucifixus
Cum Sancto Spiritu
Domine fili unigenite
Et in terra pax hominibus
Et incarnatus est
Et resurrexit
Gloria (RV 589) - Cum Sancto Spiritu
Gloria (RV 589) - Domine Deus, Agnus Dei
Gloria (RV 589) - Domine fili unigenite
Gloria in excelsis Deo
Gratias agimus tibi
Propter magnam gloriam tuam
Qui tollis peccata mundi
Quoniam tu solos Sanctus

Antonio de Cabezón
De la Virgen

Arnold Freed
Dance alleluia

Arthur Honegger
Christmas Cantata
Une cantate de noel

Arthur Seymour Sullivan
Angel voices, ever singing
Finale

Arthur Warrell
A merry Christmas

Barrington Brooks
Betelehemu

Bates G Burt
All on a Christmas morning
Bright, bright the holly berries
Caroling, caroling
Jesu parvule
O hearken ye
We'll dress the house

Benjamin Britten
A new year carol
As dew in Aprille
Procession
Recession
Welcome Ode - Canon

Bob Thiele
What a wonderful world

Boris Ord
Adam lay ybounden

Bronwyn Edwards
No bridges - 10. No bridges
No bridges - 11. Ferry
No bridges - 5. Secret destinations
No bridges - 6. Strawberry kisses

No bridges - 7. Seashore
No bridges - 9. Wings over Colvos Passage *(including an additional arrangement)*
No bridges [2022] - No bridges
Secret destinations
Strawberry kisses

Bruce Trinkley
Villancico

Camille Saint-Saens
Christmas oratorio - Gloria in altissimis
Christmas oratorio - Tollite hostias
Des pas dans l'allée
Mass for four voices (op. 4) - Agnus Dei
Mass for four voices (op. 4) - Gloria
Mass for four voices (op. 4) - Kyrie
Mass for four voices (op. 4) - Sanctus
Quam dilecta
Quare freemuerunt gentes
Trinquons

Carl Orff
Carmina Burana - 10. Were diu werlt alle min
Carmina Burana - 2. Fortune plango vulnera
Carmina Burana - 3. Veris leta facies

Carl Strommen
They can't take that away from me

Carlyle Sharpe
Laudate nomen

Caroline Alice Elgar
False love

Carolus Hacquart
Domine, Deus meus

Cecil Spring-Rice
I vow to thee, my country

Cecilia McDowall
A winter's night - 1. In dulci jubilo
A winter's night - 2. O little one sweet
A winter's night - 3. Noel nouvelet

Cesar Franck
Panis Angelicus

Charles B Lovekin
Ave Maria
The Third Joyous Mystery, the Birth of the Christ, Jesus

Charles Hubert Hastings Parry
I know my soul hath power
My soul, there is a country
O pray (m. 82-107)
The Pied Piper of Hamelin
Vivat (m. 62-74)

Charles Steggall
Remember now thy creator

Charles Villiers Stanford
O for a closer walk with God

Charles Wesley
Hark! The herald angels sing

Charles Wood
Ding dong, merrily on high

Christina Rosetti
In the bleak midwinter

Christopher Beck
Heimr Arnadalr *(including an additional arrangement)*
The great thaw *(including an additional arrangement)*
Vuelie *(including an additional arrangement)*

Christopher Hussey
What a wonderful world

Christopher Tye
O come, ye servants of the Lord

Claude Goudimel
Gloria in Excelsis
Or sus serviteurs du Seigneur

Claudin de Sermisy
Au joli bois

Colin Hand
Behold a simple, tender babe

Conrad Susa
A Christmas garland
God rest ye merry (m. 27-111)
I saw three ships (m. 120-139)
Lully lullay (m. 206-224)
Noel (m. 330-343)
Noel (m. 4-9)
O come all ye faithful (m. 224-260)
The First Noel (m. 272-326)
We three kings (m. 141-194)

Craig Hanson
Elegy

Craig Phillips
Hodie Christus natus est

Dale Warland
What child is this?

Daniel Pinkham
O magnum mysterium

David Hahn
Tirlee! Tirlo!

David Mooney
Don Oiche Ud I mBeithil
Wexford carol

Dick Smith
Winter wonderland

Dietrich Buxtehude
Das Neugeborne Kindelein
Der Herr ist mit mir
The little newborn Jesus child (Das Neugeborne Kindelein)

Diu Bose Heyward
Summertime

Edgar L Bainton
And I saw a new heaven

Edith Hugo Bosman
Inkosi Jesu

Eduardo Ocón
Cor mundum crea
Libera me
Tunc imponent

Edward C Bairstow
I sat down

Edward Elgar
False love
Psalm 93
The snow

Edwin Fissinger
Silent night

Elizabeth Bartlett
Country wedding

Emanuel Geibel
Zigeunerleben (Gypsy life)

Emilio Murillo
La Cabana

Emily Dickenson
Let down the bars of death

Eric Lane Barnes
Lambscapes - V. Orff
Lambscapes - VI. A la Sons of the Pioneers
Lambscapes - VII. Gospel

Ernani Aguiar
Salmo 150 (Psalm 150)

Esteban de Brito
Aleph. Ego vir videns
Assumpsit Jesus
Circundederunt
De lamentatione (Jueves Santo)
Ductus est Jesus
Esto mihi

In illo tempore dicebat Jesus
Incipit lamentatio
Inter vestibulum
Jod. Manum suam
Regina caeli

Fanny Mendelssohn Hensel
Gartenlieder (op. 3) - 1. Horst du nicht die Baume rauschen
Gartenlieder (op. 3) - 2. Schone Fremde
Gartenlieder (op. 3) - 3. Im Herbste
Gartenlieder (op. 3) - 4. Morgengruss
Gartenlieder (op. 3) - 5. Abendlich schon rauscht der Wald
Gartenlieder (op. 3) - 6. Im Wald

Felice Anerio
Christus factus est

Felix Bernard
Winter wonderland

Felix Mendelssohn
Elijah - 29. He, watching over Israel
Grant us Thy peace (Verleih uns frieden)
Hark! The herald angels sing
Herr, nun lässest
Mein Herz erhebet Gott
Psalm 100 - Jauchzet dem Herrn
Verleih uns frieden

Francis Pott
Angel voices, ever singing

Francis Poulenc
Salve-Regina

Francisco Guerrero
Ad caenam agni providi
Antes que comáis a Dios
Aurea luce
Canite tuba (Blow ye the trumpet)
Conditor alme siderum
Deus tuorum militum
Dios inmortal
Hoy, Joseph
Huid, huid
Hymnum canamus gloriae

Iesu Corona virginum
Lauda mater ecclesia
Los reyes siguen la estrella
Magnificat octavi toni
Magnificat quarti toni
Magnificat secundi toni
Magnificat sexti toni
Magnificat tertii toni
Niño Dios de amor herido
O celestial medicina
O lux beata trinitas
O qué mesa y qué manjar
Pasión según San Marcos
Quicumque Christum quaeritis
Sanctorum meritis
Tibi Christi splendor patris
Todo cuanto pudo dar

Francisco Sanz
Tradiderunt

Francisco de la Torre
Adoramoste, Señor

Frank Speller
Mass of St. Louis - I. Kyrie eleison
Mass of St. Louis - II. Gloria
Mass of St. Louis - III. Sanctus
Mass of St. Louis - IV. Agnus Dei
Psalm 23

Franz Gruber
Silent night

Franz Joseph Haydn
Mass in Time of War - Agnus Dei
Mass in Time of War - Benedictus
Mass in Time of War - Kyrie
Mass in Time of War - Sanctus
Missa in Angustiis (Nelson Mass) - 3a. Credo
Missa in Angustiis (Nelson Mass) - 4. Sanctus
Missa in Angustiis (Nelson Mass) - 6b. Dona nobis
Missa Sancti Bernardi de Offida - Dona nobis pacem
Te deum laudamus

Franz Joseph Mohr
Silent night

Franz Schubert
Ave Maria
Ingrediente Domino
Mass In G - Agnus Dei
Mass In G - Benedictus
Mass In G - Credo
Mass In G - Gloria
Mass In G - Kyrie
Mass In G - Sanctus
Tantum Ergo
Zum Sanctus

Fray Jerónimo González
Serenísima una noche

Fred Bock
Peace, peace

Frederick Delius
Sea Drift

Friedrich Filitz
Lead us, heavenly Father, lead us

Friedrich Schiller
Nänie (op. 82)

Frode Fjellheim
Heimr Arnadalr
The great thaw
Vuelie

Gabriel Fauré
Cantique de Jean Racine
Libera me
Offertorium
Pavana
Sanctus

Gary D Cannon
If music

Gary Fry
El cant dels ocells (Catalonian)
La jornada (Venezuelan)

Gaspar Fernández
Desnudito parece mi niño

George Frideric Handel
Benedicat Vobis
Canticorum Jubilo
Hallelujah
Messiah - 12. For unto us a child is born
Messiah - 17. Glory To God
Messiah - 21. His yoke is easy, and his burthen is light
Messiah - 22. Behold the Lamb of God
Messiah - 24. Surely he hath borne our griefs
Messiah - 25. And with his stripes we are healed
Messiah - 26. All we like sheep have gone astray
Messiah - 28. He trusted in God
Messiah - 35. Let all the angels
Messiah - 39. Their sound is gone out
Messiah - 4. And the glory of the Lord
Messiah - 41. Let us break their bonds asunder
Messiah - 44. Hallelujah
Messiah - 46. Since by man came death
Messiah - 51. But thanks be to God
Messiah - 53. Worthy is the lamb that was slain
Messiah - 54. Amen
Messiah - 7. And he shall purify
Messiah - 9. O thou that tellest good tidings to Zion

George Gershwin
Embraceable you
Fidgety feet
Love is here to stay
Summertime
They can't take that away from me

George R Woodward
Ding dong, merrily on high

George Weiss
What a wonderful world

Gerald Finzi
In terra pax (rehearsal 16)

Gioacchino Rossini
I gondolieri

Gioachino Rossini
O salutaris Hostia

Giovanni Battista Pergolesi
Deposuit potentes
Et misericordia
Magnificat
Sicut erat in principio
Sicut locutus est

Giovanni Pierluigi da Palestrina
Benedictus
Credo
Gloria
Kyrie
O Domine
Salvator Mundi
Sanctus
Sicut cervus desiderat
Surge propera

Giselle Wyers
And all shall be well - Encounter
And all shall be well - It is in song
And all shall be well - Sometimes I choose a cloud
And all shall be well - Song is the infinite time
And all shall be well - We shall not cease
And all shall be well - Whatever happens

Giuseppe Verdi
Anvil chorus (Coro di zingari e canzoni)
Ave Maria
Dies irae
Tuba mirum

Glenn Koponen
Zions walls

Greg Bartholomew
Moon man

Guillaume de Machaut
Christe qui lux es

Kyrie

Gustav Holst
Bring us in good ale
Christmas day
I vow to thee, my country

Gustav Mahler
Ich bin der Welt abhanden gekommen

Hal Johnson
Aint got time to die (Purifoy)

Hans Leo Hassler
Cantate Domino
Dixit Maria

Harold Edwin Darke
In the bleak midwinter

Harold Heiberg
Cantique de Jean Racine

Hawley Ades
All on a Christmas morning
Bright, bright the holly berries
Caroling, caroling
Jesu parvule
O hearken ye
We'll dress the house

Hector Berlioz
L'Adieu des bergers
The shepherds farewell

Heinrich Isaac
Innsbruch, ich muss dich lassen *(including an additional unnamed arrangement)*

Henry F Lyte
Praise, my soul, the king of heaven

Henry J Gauntlett
Once in royal David's city

Henry Loosemore
O Lord, increase my faith

Henry Purcell
- Alleluya
- Come, ye sons of art
- If music be the food of love
- O God Thou Art My God
- See nature, rejoicing
- The day that such a blessing gave
- With drooping wings

Henry T Smart
- Angels, from the realms of glory

Herbert Howells
- Requiem - II

Hildor Lundvik
- Early spring

Horace Everett
- Stomp your foot

Hubert Waelrant
- Als ick u vinde

Hugo Distler
- Lo how a rose e'er blooming

Ira Gershwin
- Embraceable you
- Fidgety feet
- Love is here to stay
- Summertime
- They can't take that away from me

Irvin L Brusletten
- Country wedding

Irving Fine
- Ching-a-ring chaw
- I bought me a cat
- Long time ago

Isaac Watts
- Joy to the world
- My shepherd will supply my need

Jacob Arcadelt
Il bianco e dolce cigno
Vitam que faciunt

Jacob Handl
Ecce concipies (Behold, thou shalt conceive)

Jacques Berthier
Allez par toute la terre

Jacquet de Mantua
O Jesu Christe
O vos omnes

James Montgomery
Angels, from the realms of glory *(including an additional unnamed arrangement)*

James Stephens
Anthony O Daly
Mary Hynes

Jean Mouton
Benedicta es caelorum regina
Gaude Barbara
Quaeramus cum pastoribus
Vulnerasti cor meum

Jean Planson
Chambriere, chambriere

Jean Richafort
Quem dicunt homines

Jean Sibelius
Finlandia Hymn

Jean-Joseph Roux
Alleluia de Saint-Augustin

Jean-Paul Lecot
Eglise du Seigneur

Jeffrey Van
O be joyful

Jeremiah Clarke
Praise the Lord, O Jerusalem

Jeremy Birchall
What a wonderful world

Jessie Seymour Irvine
The Lord's my shepherd - Crimond

Johann Christian Bach
Magnificat

Johann Sebastian Bach
Ah dearest Jesus
All darkness flies
An dir, du Vorbild grosser Frauen
Aria de la suite en Re
Beside Thy cradle here I stand
Bless the Lord, my soul
Bless the Lord, my soul - I. Choral vs.1
Bless the Lord, my soul - VII. Choral vs.4
Bless the Lord, my soul - XI. Choral vs.6
Break forth, O beauteous heavenly light
Cantata 118 - O Jesu Christ, Mein's Lebens Licht
Cantata No. 4 Christ lag in todesbanden - Versus 1
Cantata No. 4 Christ lag in todesbanden - Versus 4
Cantata No. 4 Christ lag in todesbanden - Versus 7
Christ lag in todesbanden
Christians be joyful
Come and thank Him
Crucifixus
Dein Will gescheh
Der Leib zwar in der Erden
Die Kunst der Fuge - Contrapunctus 2
Die Kunst der Fuge - Contrapunctus 9
Doch Königin, du stirbest nicht
Dona nobis pacem
Ein Sohn ist uns gegeben
Gloria sei dir gesungen
Glory be to God
Glory to God in the highest
Gratias agimus tibi
Hear, King of angels
Herr Christ, der einge Gottessohn
Herr, unser Herrscher
Herzliebster Jesu

How shall I fitly meet Thee
Ich will den Namen Gottes loben
Jesu, joy of Man's desiring
Jesu meine Freude - 1. Choral
Jesus bleibet meine Freude
Jesus, who didst ever guide me
Komm, tod, du schlafes bruder
Kyrie II
Lass, Fürstin, lass noch einen Strahl
Lasset uns den nicht zertailen
Let us even now go to Bethlehem
Liebster Gott, wenn werd ich sterben
Lobet den Herrn alle Heiden
Lord, when our haughty foes assail us
Now vengeance hath been taken
O Jesu Christ, meins Lebens Licht
O Mensch
O praise the Lord, all ye Nations
O'er us no more the fears of hell
Qui tollis
Rejoice and sing
Sei Lob und Preis mit Ehren
Sheep may safely graze (BWV 208)
The Lord hath all these wonders wrought
Thee with tender care
This proud heart
Wachet auf
Wahrlich
Was Gott tut, das ist wohlgetan
Wenn ich einmal soll Scheiden
Wir Christenleut
With all Thy hosts
Within yon gloomy manger
Wohl mir, daß ich Jesum habe

Johannes Brahms

Abendlied
Der Abend
Der Gang zum Liebchen (op. 31 no.3)
Ein Deutsches Requiem - I. Selig sind, die da Leid tragen
Ein Deutsches Requiem - II. Denn alles Fleisch es ist wie Gras
Ein Deutsches Requiem - III. Fugue only
Ein Deutsches Requiem - VII. Selig sind die Toten
In Stiller Nacht
IV. Wie lieblich sind deine Wohnungen
Nänie (op. 82)

Neue Liebeslieder - 1
Neue Liebeslieder - 12
Neue Liebeslieder - 14
Neue Liebeslieder - 15
Neue Liebeslieder - 2
Neue Liebeslieder - 7
Neue Liebeslieder - 8
O schoene Nacht
Warum ist das Licht

John Bennet

Come shepherds follow me
I languish to complain me
Oh sleep fond fancy
Oh sweet grief *(including an additional unnamed arrangement)*
Rest now Amphion
Since neither tunes of joy
Thirsis sleepest thou
Weep, O Mine Eyes *(including an additional unnamed arrangement)*
Ye restless thoughts

John Dowland

Come again

John Farmer

Fair Phyllis
Lord's Prayer

John Francis Wade

O come, all ye faithful

John Gardner

Tomorrow shall be my dancing day

John Goss

Praise, my soul, the king of heaven
These are they which follow the Lamb

John Henry Fowler

I vow to thee, my country
The shepherds farewell

John IV of Portugal

Crux fidelis

John Ireland
Greater love hath no man

John Jacob Niles
I wonder as I wander

John Leavitt
Ose shalom

John Mason Neale
Good king Wenceslas

John Redford
Rejoice in the Lord alway

John Rutter
A mighty fortress is our God
Agnus Dei
Amazing grace
Be thou my vision
Christ is made the sure foundation
Finale (Dona nobis pacem)
Icicles
Joy to the world
Lo He comes with clouds descending
Lux aeterna
Oh God our help in ages past
Out of the deep
Pie Jesu
Requiem aeternam
The king of love my shepherd is
The Lord is my shepherd

John Sanders
The collects
The Lords prayer

John Stainer
God So Loved The World
Good king Wenceslas
How beautiful upon the mountain
The First Noel

John Wilbye
Adieu, Sweet Amaryllis

Jonathan Willcocks
- Fecit potentiam
- Gloria patri
- Magnificat

Joseph Brackett
- Simple gifts

Josquin des Prez
- Agnus Dei
- Credo
- Dulces exuviae
- El grillo
- Fama malum
- Gloria
- Kyrie
- Mille regretz
- Sanctus

Juan Arañés
- Un sarao de la chacona

Juan Domingo Vidal
- Clamabat Autem Mulier

Juan Francés de Iribarren
- Ave Maria
- Missus est angelus
- O quam suavis
- O quot undis lacrimarum
- O sacrum convivium
- Omnes gentes
- Pange lingua
- Petite et accipietis
- Quién nos dirá de una flor
- Regina Caeli I
- Regina Caeli II
- Regina Caeli III
- Sancta et Inmaculata
- Spiritus Domini
- Stabat Mater a 4
- Stabat Mater a 5
- Surrexit Pastor
- Te invocamus
- Vere languores
- Vexilla regis

Juan Gutiérrez de Padilla
Stabat Mater

Juan Páez
Canite tuba
Diligam te Domine
Sciant gentes
Tribulationes Cordis
Tu es Deus
Veni ad liberandum nos
Veni Domine

Juan Vásquez
Absolve Domine
Agnus Dei
Benedictus Dominus
Complaceat tibi
Convertere Domine
Credo videre
De los álamos vengo
Delicta
Dicen a mí que los amores he
Dirige Domine
Domine Jesu Christe
In loco pascuae
Kyrie
Lindos ojos habéis
Manus tuae
Nequando rapiat
Parce mihi Domine
Por amores lo maldijo
Quien amores tiene
Requiem aeternam
Requiescant in pace
Responde mihi
Sana Domine

Juan de Anchieta
Libera me
O bone Jesu

Juan de Torres Rocha
Lucinda, tus cabellos

Juan de Triana
Dinos Madre del donsel

Juan del Encina
Caldero y llave, madona
Gasajemonos de hucia
Hoy comamos y bebamos
Más vale trocar
Mi libertad en sosiego
Ojos garzos ha la niÃ±a
Si habrá en este baldrés
Todos los bienes del mundo

Katharine Emily Roberts
Nos galan

Kenneth Jennings
I gondolieri

Kevin Siegfried
Benediction
Heavenly display
Peace

Kirby Shaw
Fidgety feet
Love is here to stay
Nice 'n' easy

Kirke Mechem
Blow ye the trumpet

Knut Nystedt
Gloria

Larry Shackley
Angels, from the realms of glory

Lee Mendelson
Christmas time is here

Leonard Bernstein
Chichester Psalms - I
Chichester Psalms - II
Chichester Psalms - III

Lew Spence
Nice 'n' easy

Lewis Henry Horton
I wonder as I wander

Lili Boulanger
Soleils de Septembre
Sous bois

Llibre Vermell de Montserrat
Stella Splendens

Lloyd Larson
O living bread of heaven

Lodovico Viadana
Exsultate justi

Louise Bogan
To be sung on the water

Lowell Mason
Joy to the world

Ludovico da Viadana
Exultate Justi

Ludwig Senfl
Non usitata nec tenui ferar

Ludwig van Beethoven
Choral Fantasy
Dona nobis pacem
Mass In C Major (op. 86) - I. Kyrie
Mass In C Major (op. 86) - II. Gloria
Mass In C Major (op. 86) - III. Credo
Mass In C Major (op. 86) - V. Agnus Dei
Opferlied (Op 121b)
Symphony No. 9 - Finale

Luis de Narváez
De profundis

Marbrianus de Orto
Dulces exuviae

Marc-Antoine Charpentier
In nativitatem Domini canticum

Marilyn Bergman
Nice 'n' easy

Marilyn Keith
Still, still, still

Mark Riese
Noel nouvelet

Martin Luther
Verleih uns frieden

Martin Shaw
Coventry carol

Mateo Flecha
El fuego
El jubilate
La bomba
La guerra
La justa
La negrina
Què farem del pobre Joan
Teresica hermana

Matthew Culloton
Silent night

Maurice Durufle
Requiem - Agnus Dei
Requiem - Kyrie
Requiem - Lux aeterna
Tu es Petrus

Maurice Greene
Lord, let me know mine end

Melchior Franck
Da pacem Domine
Viens chanter avec nous

Melchor Robledo
Domine Jesu Christe

Michael Praetorius
Es ist ein ros entsprungen

Michael Wise
The Ways of Zion do mourn

Morten Lauridsen
Dirait-on
In te, Domine, speravi
Luci serene e chiare

Moses George Hogan
My God is so high

Mykola Leontovich
Carol of the bells

Mzilikazi Khumalo
Bawo thixo somandla

Narcis Casanovas
Amicus Meus
Judas Mercator Pessimus
Unus Ex Discipulis Meis

Nicolás Gombert
Aspice Domine

Norman DelloJoio
Bit (103-107)
To Saint Cecelia - D
To Saint Cecelia - E
To Saint Cecelia - G
To Saint Cecelia - H
To Saint Cecelia - M
To Saint Cecelia - O
To Saint Cecelia - P
To Saint Cecelia - S
To Saint Cecelia - V

Norman Luboff
Still, still, still

Orlande de Lassus
Libera me, Domine

Orlando di Lasso
Matona Mia Cara

PDQ Bach
Good King Kong looked out
O little town of Hackensack
Throw the yule log on, Uncle John

Paul Crabtree
Fix me, Jesus
I want to be ready
My Lord, what a morning

Paul Hofhaimer
Vides ut alta stet

Pedro Ruimonte
Has visto

Pedro de Cristo
Es nascido

Peter Schickele
Good King Kong looked out
O little town of Hackensack
Throw the yule log on, Uncle John

Peter Wilhousky
Carol of the bells

Philip Ledger
Hark! The herald angels sing
O come, all ye faithful

Pierre Attaignant
Tourdion

Pierre Certon
La, la, la, je ne l'ose dire

Pierre Passereau
Il est bel et bon

Pyotr Ilyich Tchaikovsky
The crown of roses

Quirinio Gasparini
Adoramus te Christe

Ralph Vaughan Williams
- Dona nobis pacem - II
- Dona nobis pacem - IV Dirge for two veterans
- Fantasia on Christmas carols
- Festival te Deum
- For all the saints
- The Old Hundredth psalm tune

Randall Thompson
- Alleluia
- Choose something like a star
- Glory to God in the highest
- Have ye not known
- Say ye to the righteous
- The noise of a multitude
- The paper reeds by the brooks
- The road not taken
- Woe unto them

Raymond Wilding White
- At the river

Richard Farrant
- Call to remembrance

Richard Nance
- Set me as a seal
- Songs of Celebration - I
- Songs of Celebration - II

Rick Powell
- Peace, peace

Robert Frost
- Choose something like a star
- The road not taken

Robert Lucas de Pearsall
- In dulci jubilo (1-31)
- Sing we and chaunt it

Robert Russell Bennett
- The battle hymn of the republic

Robert Schumann
- Zigeunerleben (Gypsy life)

Robert Shaw
For all the saints
Mary had a baby

Robert Southwell
Behold a simple, tender babe

Robert Stone
Lord's Prayer

Roger Emerson
Winter wonderland

Roupen Shakarian
O be joyful

Samuel Barber
Anthony O Daly
Let down the bars of death
Mary Hynes
Sure on this shining night (op. 13, no. 3)
To be sung on the water

Samuel Sebastian Wesley
Blessed be the God and Father
Praise the Lord, O my soul
Wash me thoroughly

Sebastián de Vivanco
O quam suavis

Stanley Vann
My Love's An Arbutus

Steve Zegree
Christmas time is here

Sylvia Powell
Peace, peace

Teena Chinn
Embraceable you

Thoinot Arbeau
Belle qui tiens ma vie

Thomas Morley

April is in my mistress face
Nolo mortem peccatoris

Thomas Tallis

O Lord, give thy Holy Spirit

Tom Clarke

Moon man

Tomás Luis de Victoria

Ad caenam agni providi
Ad preces nostras
Amicus meus
Animam meam dilectam
Ascendens Christus - Benedictus
Asperges me
Astiterunt reges
Aurea luce
Ave Maria
Ave maris stella
Ave maris stella - Credo
Ave maris stella - Gloria
Ave maris stella - Sanctus
Benedictus Dominus
Caligaverunt oculi mei
Christe redemptor omnium
Communio
Conditor alme siderum
Date ei de fructu
Deus tuorum militum
Dixit Dominus I
Dixit Dominus VII
Doctor bonus amicus Dei
Doctor egregie
Domine ad adjuvardum
Ecce quomodo moritur
Ecce sacerdos magnus
Ego sum panis vivus
Eram quasi agnus
Estote fortes in bello
Exultet coelum laudibus
Gaudent in caelis
Graduale
Hic vir despiciens mundum
Hostis Herodes impie

Hujus obtentu Deus
Introitus
Iste confessor
Iste sanctus
Jesu corona virginum
Jesu dulcis memoria
Jesu nostra redemptio
Lauda mater Ecclesia
Laudate pueri VII
Libera me
Lucis creator optime
Magi viderunt stellam
Miserere mei Deus
Ne timeas Maria
Nunc dimittis
O decus apostolicum
O doctor optime
O lux beata trinitas
O magnum mysterium
O quam gloriosum est regnum
O quam metuendus est
O vos omnes
Offertorium
Pange lingua
Pasión según San Juan
Pasión según San Mateo
Peccantem me quotidie
Petrus beatus
Popule Meus
Pueri Hebraeorum
Quam pulchri sunt gressus tui
Quicumque Christum quaeritis
Quodcumque vinclis
Recessit pastor noster
Requiem a 4 voces - Agnus Dei
Requiem a 4 voces - Kyrie
Requiem a 4 voces - Sanctus et Benedictus
Rex gloriose martyrum
Sancta Maria succurre miseris
Sanctorum meritis
Senex Puerum portabat
Seniores populi
Sepulto Domino
Taedet animam meam
Tamquam ad latronem
Te Deum

Te lucis ante terminum
Tibi Christe
Tradiderunt me
Tristes erant Apostoli
Unus ex discipulis
Urbs beata Jerusalem
Ut queant laxis
Veni Creator Spiritus
Veni sponsa Christi
Vere languores nostros
Vexilla regis
Vidi aquam

Traditional

A merry Christmas
Adam lay ybounden
Angels, from the realms
Benediction
Betelehemu
Chanticleer (Nativity)
Coventry carol
De tierra lejana venimos (Puerto Rican)
Eia, eia
El cant dels ocells (Catalonian)
Falan-tiding
Good king Wenceslas
Hark! The herald angels sing
Heavenly display
Highlands cathedral
In the bleak midwinter
Infant holy, infant lowly
Jasmin-flower (Moo-Lee-Hwa)
La jornada (Venezuelan)
My God is so high
Nos galan
Nukapianguaq (p. 3)
Nukapianguaq (p. 9)
O come, all ye faithful
Ose shalom
Peace
Sauntrai Na Maighdine
Siyahamba
Soon-ah will be done
The First Noel
The holly and the ivy
The twelve days of Christmas

Welsh carol
Wexford carol
What child is this?

Valeri Kikta
La luce della tacite stelle - I. Sereno
La luce della tacite stelle - III. Fulmine
La luce della tacite stelle - IV. Inno

Via Olatunji
Betelehemu

Vince Guaraldi
Christmas time is here

Virgil Thomson
My shepherd will supply my need

Wendell Whalum
Betelehemu

Wihla Hutson
All on a Christmas morning
Bright, bright the holly berries
Caroling, caroling
Jesu parvule
O hearken ye
We'll dress the house

William Austin
Chanticleer (Nativity)

William Averitt
The lands that long in darkness (m. 19-55)
When marshalled on the mighty plain (m. 175-235)

William Byrd
Ave Verum Corpus
Misa a cuatro voces - 1. Kyrie
Misa a cuatro voces - 2. Gloria
Misa a cuatro voces - 3. Credo
Misa a cuatro voces - 4. Sanctus
Misa a cuatro voces - 5. Benedictus
Misa a cuatro voces - 6. Agnus Dei

William C Dix
What child is this?

William Dawson
Soon-ah will be done

William Haubrich
Highlands cathedral

William Kirkpatrick
Away in a manger

William Schwenk Gilbert
Finale

William Shakespeare
If music

William Steffe
The battle hymn of the republic

William Walsham How
For all the saints

William Walton
Jubilate Deo (m. 99-141)

Wolfgang Amadeus Mozart
Ave Verum Corpus
Davidde penitente (K469) - 1. Coro: Alzai le flebili
Davidde penitente (K469) - 10. Coro: Chi in Dio sol spera
Davidde penitente (K469) - 2. Coro: Cantiam le glorie e le lodi
Great Mass in C minor (K427) - Cum Sancto Spiritu
Great Mass in C minor (K427) - Gloria in excelsis
Great Mass in C minor (K427) - Jesu Christe
Kyrie (K341)
Mass in C (K317) - Sanctus
Missa Brevis in D Major (K194) - 1. Kyrie
Missa Brevis in D Major (K194) - 2. Gloria
Missa Brevis in D Major (K194) - 3. Credo
Missa Brevis in D Major (K194) - 5. Benedictus
Missa Brevis in D Major (K194) - 6. Agnus Dei
Regina Coeli (K276)
Requiem (K626) - Agnus Dei
Requiem (K626) - Agnus Dei - fugue
Requiem (K626) - Benedictus

Requiem (K626) - Benedictus - fugue
Requiem (K626) - Confutatis
Requiem (K626) - Dies irae
Requiem (K626) - Domine Jesu
Requiem (K626) - Hostias
Requiem (K626) - Kyrie
Requiem (K626) - Lacrymosa
Requiem (K626) - Lux aeterna
Requiem (K626) - Recordare
Requiem (K626) - Requiem
Requiem (K626) - Requiem - fugue
Requiem (K626) - Rex tremendae
Requiem (K626) - Sanctus
Requiem (K626) - Sanctus - fugue
Requiem (K626) - Tuba mirum

Zoltán Kodály

Missa brevis - Credo
Missa brevis - Gloria

B.1 Dataset Features

The final pre-processed token dataset contains the following MIDI features (for all four voices [S, A, T, B]) stored in Pickle files:

- Notes (transposed to diatonic C major/A minor) including pitch class (e.g., C5, A-4, D#6)
- Durations (quarter length – e.g., 1.0 for quarter note, 2.0 for half note, 0.5 for 8th note)
- Rests (as "rest")
- Metronome marks (in Beats Per Minute [int])
- Time signature (as a ratio string, e.g., "3/4")
- Key signature (including key and mode)

The GitHub repository (Szelogowski, 2022/2023) also contains the following variations of the dataset (as well as the "meta-analysis" data; see *Chapter 4.1.1*):

- **Pickle**
    - Soprano, alto, tenor, and bass notes and durations (all separately)
    - Transposed soprano, alto, tenor, and bass notes and durations (separately, sliced into multiple parts for Git)
- **MIDI**
    - All soprano, alto, tenor, and bass MIDIs (separately)
    - Combined MIDIs (with any accompaniment removed)
    - Combined transposed MIDIs
    - Combined augmented MIDIs (four variations)
- **CSV** (deprecated)
    - Soprano, alto, tenor, and bass files (separately) containing each event, event velocities and times, tempi, time signatures (count and beat), and key signatures

APPENDIX C
RELATED DEFINITIONS

This appendix presents additional terminology and compositional techniques that may aid in familiarizing the reader with related subject vocabulary.

C.1 Voice Leading

Voice leading pertains to how individual melodic lines (voices) are combined to create harmonies (Ellenwood, 2018). The following rules were used as the baseline for my post-processing system as well, allowing for more musically sophisticated AI-generated compositions:

- **Parallel Motions and Unisons**
    - Avoid parallel unisons and octaves as they diminish the independence of voices.
    - Parallel fifths are typically forbidden but can appear under specific conditions, such as in a Neapolitan Sixth to Tonic progression.
    - Parallel fourths are acceptable and often used.
- **Doubling Rules**
    - The leading tone (seventh degree of the scale) should never be doubled due to its tendency to resolve upwards.
    - Avoid doubling altered tones (sharpened or flattened) since they can lead to ambiguous tonality.
    - Never double the seventh in a seventh chord, nor the respective tones in extended chords like ninths, elevenths, etc.

- **Melodic Intervals**
    - The augmented second is usually avoided in melodies because it can sound disjunct and can lead to tonal ambiguity.
    - Conversely, the diminished seventh can be used if followed by a motion in the opposite direction.
    - The augmented fourth (tritone) is generally avoided melodically, while its inversion is accepted.
- **Skips and Leaps**
    - Skips larger than an octave or a sixth should be followed by stepwise motion in the opposite direction to maintain melodic contour.
- **Harmonic Intervals and Dissonances**
    - A perfect fourth against the bass is only permitted when it occurs as part of the third inversion of a seventh chord.
    - Dissonances like sevenths should be resolved down by step. Notable exceptions exist in later Romantic music where dissonances resolve in unconventional ways.
- **Triad Doubling and Dissonance Treatment**
    - It is standard to double the root of a root position triad to solidify the harmony.
    - While certain guidelines suggest always doubling the root, musical context can necessitate otherwise to favor counterpoint or to avoid parallelisms.
- **Exceptions and Evolution**
    - Many of the rules are relaxed or reinterpreted in post-common practice periods, such as the Romantic era, to achieve greater expressiveness and harmonic complexity.

C.1.1 Triad Inversions

- **Triads**
    - Chord inversions are used to create melodic bass lines and to add variety. Each triad position has important characteristics which must be taken into account.
- **Root Position Triads**
    - Root position triads are the most stable. Root position triads reflect and reinforce the natural structure of the overtone series. In most cases, double the root when triads are in root position.
- **First Inversion Triads**
    - First inversion triads are less stable than root position triads. They are used to create melodic and stepwise bass lines (Example: I–vii°6–I6; bass line is 1–2–3). Double the root (or sometimes the fifth) in I6 and V6 and double the third in vii°6 and ii°6. In IV6 and ii6 the third is often doubled. USE FIRST INVERSION FOR DIMINISHED TRIADS (ii°6 and vii°6) AND DOUBLE THE THIRD FACTOR.
- **Second Inversion Triads**
    - Second inversion triads are the least stable. This instability is created by the interval of a fourth above the bass note. In traditional tonal practice, the fourth is considered to be a “dissonant” interval; the upper voice tends to resolve downward, creating thirds. IT IS NOT DESIRABLE TO CONCLUDE PROGRESSIONS WITH SECOND INVERSION TRIADS. Second inversion triads are employed in FOUR WAYS:
        1. In an arpeggiating bass line (Example: I–I6–I64–I)
        2. As a neighboring chord over a bass pedal tone (Example: I–IV64–I)

3. As a passing chord (Example: I–V64–I6; V–ii64–V65)
4. At cadences – this always involves the Tonic triad (Example: I–IV–I64–V–I)

- In most second inversion triads, double the bass note, or fifth factor

C.1.2 Connecting Chords in Root Position

I. ROOTS A FIFTH OR FOURTH APART

- A. The Direction Method:
    - If the root motion is DOWN BY FIFTH (or up by fourth), move the three upper voices DOWN to the nearest chord tones.
    - If the root motion is UP BY FIFTH (or down by fourth), move the three upper voices UP to the nearest chord tones.
- B. The Common Tone Method:
    - Keep the common tone in the same upper voice.
    - Move the two remaining upper voices in stepwise motion, in the same direction to the nearest chord tones.

II. ROOTS A SECOND APART

- A. Move all upper voices in contrary motion to the bass. If you do not do this, you will have many parallel fifths, octaves, and unisons.
- B. EXCEPTION: V–VI. MOVE SCALE DEGREE 7 UP TO SCALE DEGREE 1.

III. ROOTS A THIRD APART

- A. Keep the two common tones in the same two upper voices.
- B. Move the remaining voice in stepwise motion to the nearest chord tone.

C.2 Species Counterpoint

Counterpoint involves the weaving together of relatively independent melodies to form a musical composition (Gotham et al., 2021).

- **Definition and Purpose**
    - Species counterpoint is a pedagogical tool designed to teach the principles of counterpoint as practiced in the 16th century.
    - It was codified by Johann Joseph Fux in the 18th century in his treatise "Gradus ad Parnassum" and has been a foundational study for classical composers.
- **The Five Species**
    1. First Species (Note Against Note)
        - Each note in the counterpoint has the same duration as its corresponding note in the cantus firmus (the fixed, pre-existing "basis" melody).
    2. Second Species (Two Notes Against One)
        - For every note in the cantus firmus, there are two notes in the counterpoint.
    3. Third Species (Four Notes Against One)
        - The counterpoint has four notes for each note of the cantus firmus.
    4. Fourth Species (Syncopation or Ligature)
        - The counterpoint creates a suspension with the cantus firmus, resolving down by step.
    5. Fifth Species (Florid Counterpoint)
        - A combination of the first four species, with freedom of rhythm and melody to create a more complex counterpoint.

- **Rules and Techniques**
    - Begin and end on a perfect consonance.
    - Employ only consonances (perfect and imperfect) on strong beats.
    - Use dissonances (passing tones, neighbor tones, suspensions) carefully and resolve them correctly.
    - Maintain the independence of each line through motion and rhythm.
    - Avoid parallel fifths and octaves to keep the voices independent.
- **Practical Application**
    - Start with simple melodies and progress to more complex ones.
    - Practice writing counterpoint against a given cantus firmus.
    - Apply the rules of counterpoint to two-part, three-part, and four-part writing.
- **Historical Context**
    - Counterpoint was central to the music of the Renaissance and Baroque periods.
    - Understanding species counterpoint is essential for analyzing and appreciating the works of composers from these periods, such as Palestrina, Bach, and Handel.
- **Modern Relevance**
    - While modern music has evolved in harmony and rhythm, the principles of voice independence and melodic interplay continue to be relevant.
    - Contemporary composers still employ counterpoint to achieve textural and harmonic depth in their compositions.

APPENDIX D
SURVEY DESIGN

This appendix contains the survey structure and sheet music for the questionnaire samples (rendered in MuseScore from the original MIDI files). The original survey was hosted on Pollfish.com[4] (Szelogowski, 2023) and provided users with a debriefing page (upon completion) discussing which samples were AI-generated or human-composed, as well as providing the opportunity to listen to any of the audio samples again as desired. All five MIDI samples were rendered using the "Muse Choir" soundfont from the Muse Sounds library (MuseScore, 2022) and assessed using a 6-point Likert scale to gauge how strongly respondents believed each sample to be either AI or human (see *Chapter 5.2*).

[4] Pollfish was chosen over other online survey hosts like Qualtrics due to its customizability options and free inclusion of audio sample-based questions, which also allow only the initial sample playback and require respondents to listen to the entire sample before proceeding.

Survey Welcome:

This survey will present a musical Turing test for a new (W.I.P.) generative AI model for 4-part (SATB) choral music as part of a Ph.D. dissertation evaluation. You will be presented with 5 audio samples rendered from MIDI using a choir soundfont; some of them will be AI-generated, the others will be human-composed (i.e., compositions from real composers), and you simply need to respond with which one you believe the piece was composed by. The samples will be ~30-60 seconds long and may be from any part of the original MIDI, not necessarily the beginning. To prevent potential bias, please listen to this brief audio sample (of a real piece of choral music) rendered with the soundfont so you know what to expect all 5 samples to sound like. Then, choose "Yes" to proceed if you have read and understand these terms.

[13 SECOND AUDIO SAMPLE FROM GABRIEL FAURÉ'S "Requiem - Introitus"]

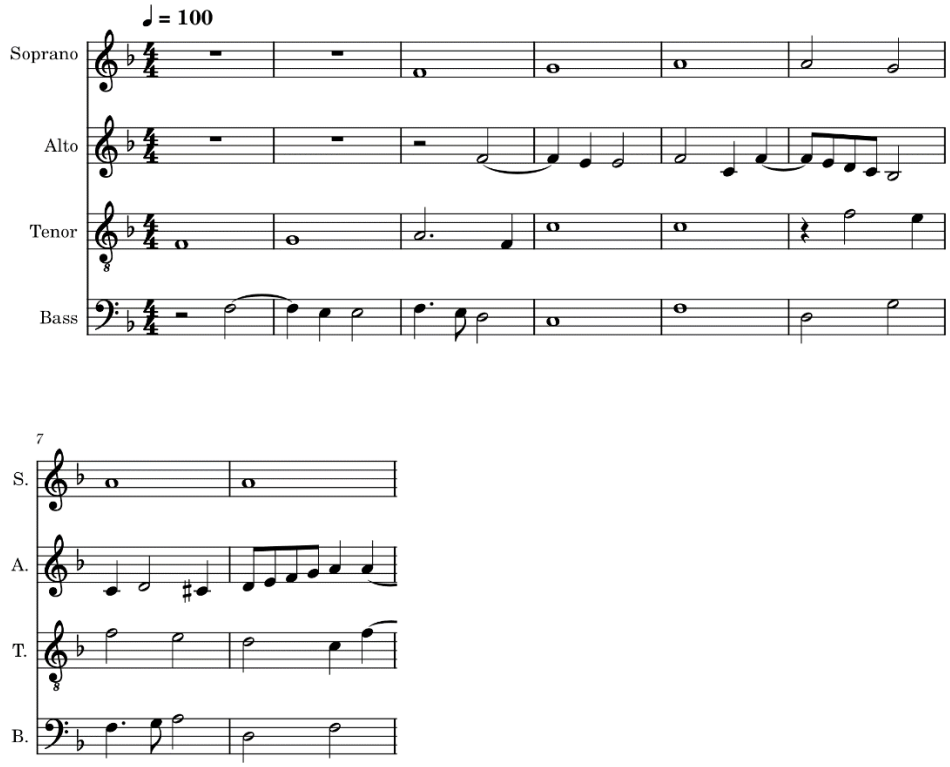


Figure D.1: Sheet music for survey "welcome" audio sample

- Yes

Question 1:

Who do you believe composed the following audio sample?

[35 SECOND AUDIO SAMPLE FROM MODEL WEIGHT SET “transposed_2”]

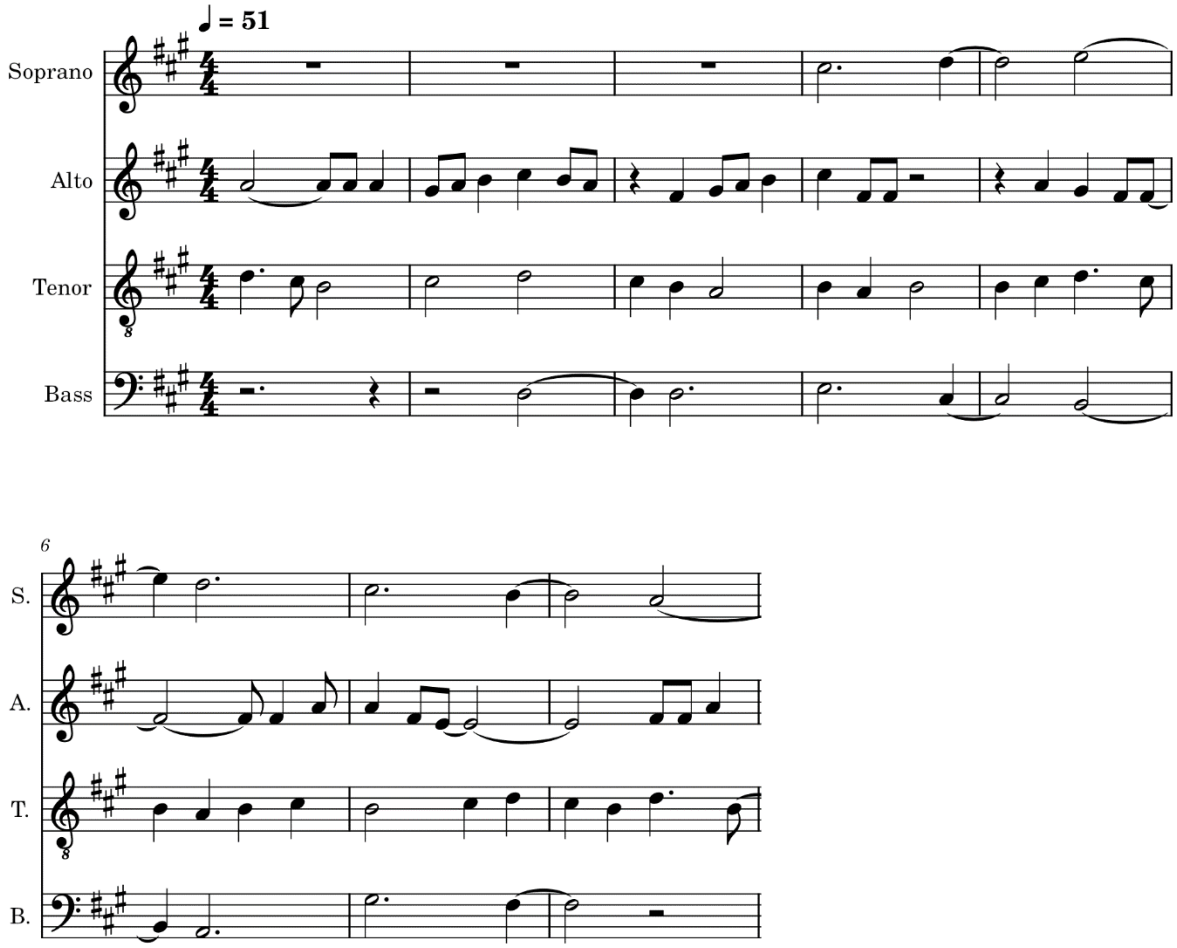


Figure D.2: Sheet music for survey “Q1” audio sample

- AI - Strongly believe
- AI - Mostly believe
- AI - Somewhat believe
- Human - Somewhat believe
- Human - Mostly believe
- Human - Strongly believe

Question 2:

Who do you believe composed the following audio sample?

[57 SECOND AUDIO SAMPLE FROM MODEL WEIGHT SET “transposed_3”]

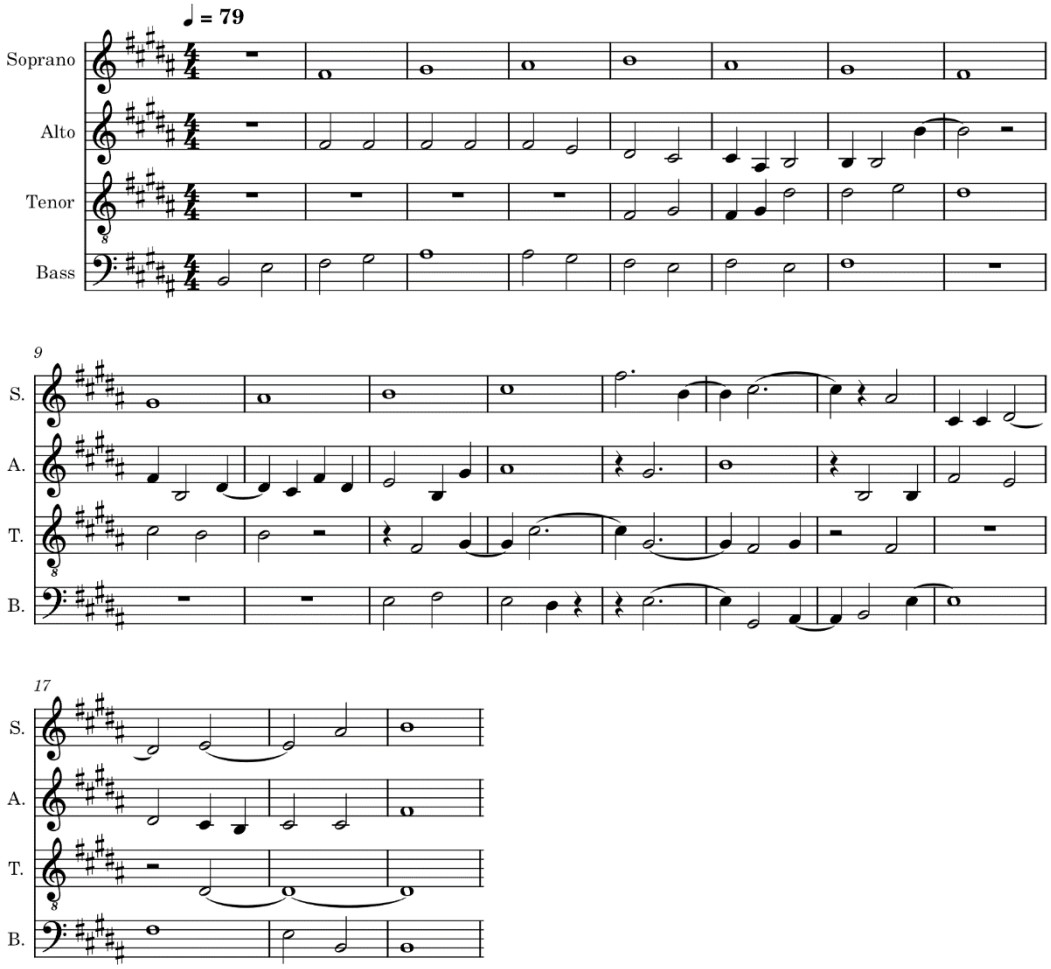


Figure D.3: Sheet music for survey “Q2” audio sample

- AI - Strongly believe
- AI - Mostly believe
- AI - Somewhat believe
- Human - Somewhat believe
- Human - Mostly believe
- Human - Strongly believe

Question 3:

Who do you believe composed the following audio sample?

[30 SECOND AUDIO SAMPLE FROM JOHN BENNET'S "Oh Sweet Grief"]

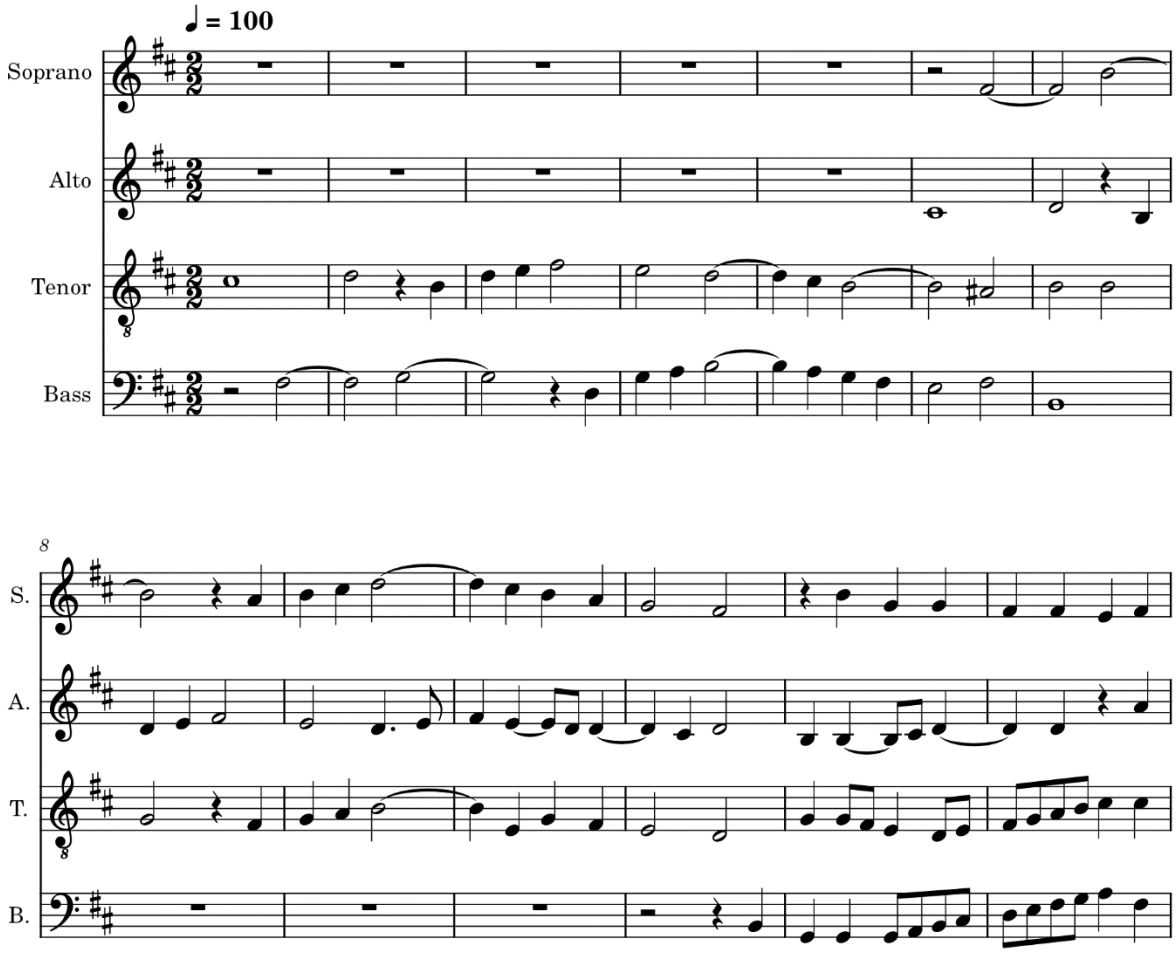


Figure D.4: Sheet music for survey "Q3" audio sample

- AI - Strongly believe
- AI - Mostly believe
- AI - Somewhat believe
- Human - Somewhat believe
- Human - Mostly believe
- Human - Strongly believe

Question 4:

Who do you believe composed the following audio sample?

[67 SECOND AUDIO SAMPLE FROM MODEL WEIGHT SET “transposed_13”]

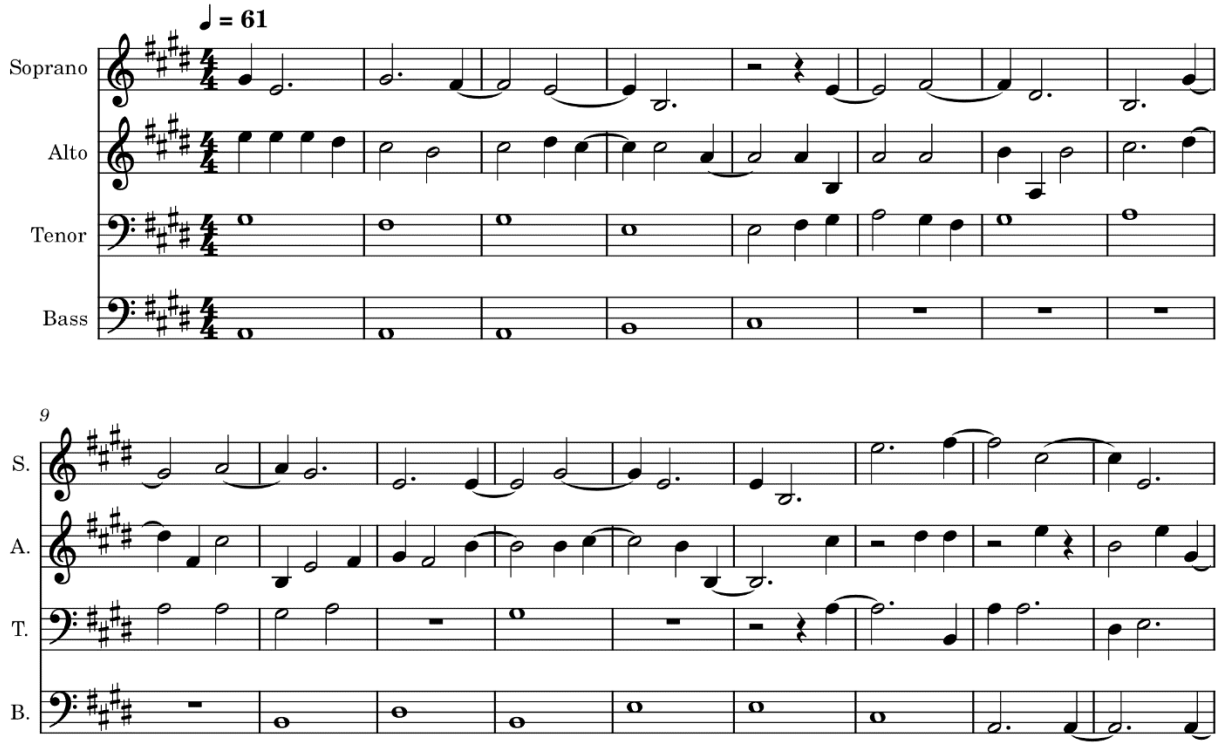


Figure D.5: Sheet music for survey “Q4” audio sample

- AI - Strongly believe
- AI - Mostly believe
- AI - Somewhat believe
- Human - Somewhat believe
- Human - Mostly believe
- Human - Strongly believe

Question 5:

Who do you believe composed the following audio sample?

[46 SECOND AUDIO SAMPLE FROM TOMÁS LUIS DE VICTORIA'S "Benedictus Dominus"]

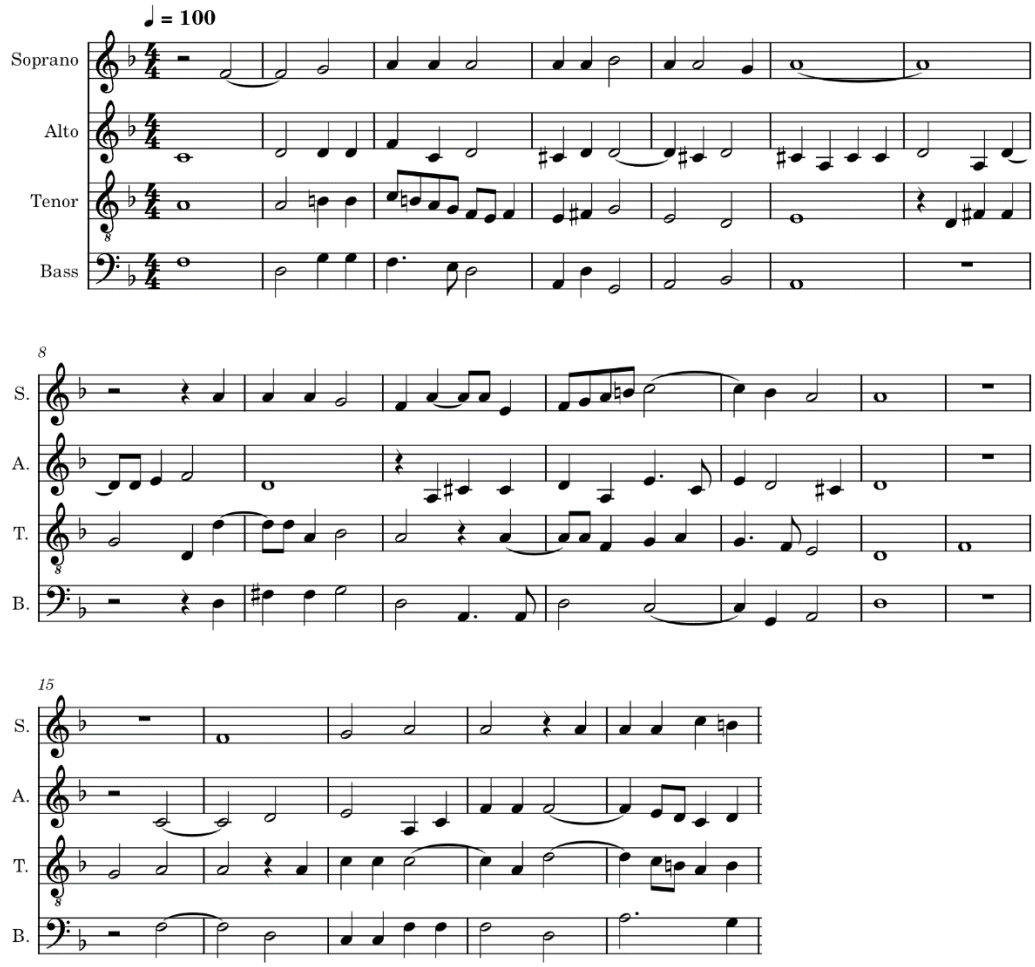


Figure D.6: Sheet music for survey "Q5" audio sample

- AI - Strongly believe
- AI - Mostly believe
- AI - Somewhat believe
- Human - Somewhat believe
- Human - Mostly believe
- Human - Strongly believe

Question 6:

Please describe your current level of musicianship/musical ability.

- Music teacher/professional (classically trained, undergraduate degree or higher)
- Music student (college-level or equivalent experience, including AP)
- Amateur musician (hobbyist, some background in composition/theory)
- Music listener/enjoyer (little or no experience)

APPENDIX E
IMPLEMENTATION OF CHORAL-GTN SYSTEM

This appendix presents an overview of the code used to implement the final system, including the Generative Transformer Network, Music Generator, Rule-based post-processing system, and assisting functions for data preparation, model training, and generation. Some code has been deprecated due to poor performance or lack of present purpose for the final model and has been removed from this dissertation but can be found on GitHub (alongside additional tool scripts used in preparing the dataset; Szelogowski, 2022/2023).

E.1 Main.py

```
import gc
import sys
import ast
import time
import random
import logging
import warnings
import numpy as np
import plotly.express as px
import matplotlib.pyplot as plt
from Transformer import *
from keras import layers
from keras import losses
from functools import partial
from keras import backend as k
from keras.regularizers import l2
from keras.optimizers import Adam
from keras.optimizers import AdamW
from keras.models import Sequential
from keras_tuner import RandomSearch
from keras.src.utils import plot_model
from keras.callbacks import EarlyStopping
from data_utils import key_signature_to_number
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from keras_tuner import HyperParameters, Objective, tuners


tf.get_logger().setLevel(logging.ERROR)
k.set_image_data_format('channels_last')
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
os.environ['TORCH_USE_CUDA_DSA'] = "1"
# os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)

if not sys.warnoptions:
    warnings.simplefilter("ignore")
warnings.filterwarnings("ignore", category=DeprecationWarning)


def generate_composition(dataset="Combined_choral", generate_len=50, num_to_generate=3,
seed_notes=[], seed_durs=[],
                         choral=False, suffix="", temperature=0.5, verify_voices=False):
    DATAPATH = f"Weights/Composition/{dataset}" if not choral else
f"Weights/Composition_Choral{suffix}"
    with open(f"{DATAPATH}/{dataset}_notes_vocab.pkl", "rb") as f:
        notes_vocab = pkl.load(f)
    with open(f"{DATAPATH}/{dataset}_durations_vocab.pkl", "rb") as f:
        durations_vocab = pkl.load(f)

    if suffix == "_Transposed2":
        model = build_model(len(notes_vocab), len(durations_vocab), embedding_dim=512,
feed_forward_dim=1024,
```

```
 54.                             key_dim=64, dropout_rate=0.3, l2_reg=1e-4,
num_transformer_blocks=3, num_heads=8)
 55.     else:
 56.         model = build_model(len(notes_vocab), len(durations_vocab), embedding_dim=512,
feed_forward_dim=512, key_dim=64,
 57.                             num_heads=8, dropout_rate=0.5, l2_reg=0.0005,
num_transformer_blocks=3, gradient_clip=1.5)
 58.     model.load_weights(f"Weights/Composition_Choral{suffix}/checkpoint.ckpt")
 59.     music_gen = MusicGenerator(notes_vocab, durations_vocab, generate_len=generate_len,
 60.                                choral=choral, verbose=True, top_k=30)
 61.
 62.     def fail(filename=None):
 63.         os.remove(filename)
 64.         print("Failed to generate piece; retrying...")
 65.
 66.     output_filenames = []
 67.     for i in range(num_to_generate):
 68.         while True:
 69.             if not choral:
 70.                 info = music_gen.generate(["START"], ["0.0"], max_tokens=generate_len,
 71.                                           temperature=temperature, model=model)
 72.                 midi_stream = info[-1]["midi"]  # .chordify()
 73.             else:
 74.                 start_notes = ["S:START", "A:START", "T:START", "B:START"]
 75.                 start_durations = ["0.0", "0.0", "0.0", "0.0"]
 76.                 if len(seed_notes) > 0 and len(seed_durs) > 0:
 77.                     start_notes += seed_notes
 78.                     start_durations += seed_durs
 79.                 info, midi_stream = music_gen.generate(start_notes, start_durations,
max_tokens=generate_len,
 80.                                                        temperature=temperature, model=model,
intro=True)
 81.             timestr = time.strftime("%Y%m%d-%H%M%S")
 82.             if not os.path.exists(f"Data/Generated/{dataset}{suffix}"):
 83.                 os.makedirs(f"Data/Generated/{dataset}{suffix}")
 84.             filename = os.path.join(f"Data/Generated/{dataset}{suffix}", "output-" + timestr +
".mid")
 85.             midi_stream.write("midi", fp=filename)
 86.             gc.collect()
 87.             # Check the output MIDI file -- if it's less than 0.25 kB, it's probably empty;
retry
 88.             if os.path.getsize(filename) < 250:
 89.                 fail(filename)
 90.             else:
 91.                 if verify_voices:
 92.                     # Load the generated MIDI file and check if it's valid (i.e., has more than
3 tracks)
 93.                     try:
 94.                         mini_gen = music21.converter.parse(filename)
 95.                         if len(mini_gen.parts) < 4:
 96.                             fail(filename)
 97.                             continue
 98.                     except Exception as _:
 99.                         fail(filename)
100.                         continue
101.                 output_filenames.append(filename)
102.                 break
103.         gc.collect()
104.         print(f"Generated piece {i+1}/{num_to_generate}")
105.
```

```
    return output_filenames


def build_model(notes_vocab_size, durations_vocab_size, gradient_clip=None,
                embedding_dim=256, feed_forward_dim=256, num_heads=5, key_dim=256,
dropout_rate=0.3, l2_reg=1e-4,
                num_transformer_blocks=2, verbose=True):
    note_inputs = layers.Input(shape=(None,), dtype=tf.int32)
    duration_inputs = layers.Input(shape=(None,), dtype=tf.int32)
    note_embeddings = TokenAndPositionEmbedding(notes_vocab_size, embedding_dim // 2,
l2_reg=l2_reg)(note_inputs)
    duration_embeddings = TokenAndPositionEmbedding(durations_vocab_size, embedding_dim // 2,
                                                    l2_reg=l2_reg)(duration_inputs)
    embeddings = layers.Concatenate()([note_embeddings, duration_embeddings])
    x = embeddings
    for i in range(num_transformer_blocks):
        x, _ = TransformerBlock(name=f"attention_{i+1}", embed_dim=embedding_dim,
ff_dim=feed_forward_dim,
                                num_heads=num_heads, key_dim=key_dim,
dropout_rate=dropout_rate, l2_reg=l2_reg)(x)
    note_outputs = layers.Dense(notes_vocab_size, activation="softmax", name="note_outputs",
                                kernel_regularizer=l2(l2_reg))(x)
    duration_outputs = layers.Dense(durations_vocab_size, activation="softmax",
name="duration_outputs",
                                    kernel_regularizer=l2(l2_reg))(x)
    model = models.Model(inputs=[note_inputs, duration_inputs], outputs=[note_outputs,
duration_outputs])
    lr_schedule = NoamSchedule(embedding_dim)
    optimizer = Adam(learning_rate=lr_schedule, clipnorm=gradient_clip)
    model.compile(optimizer, loss=[losses.SparseCategoricalCrossentropy(),
losses.SparseCategoricalCrossentropy()])
    if verbose:
        model.summary()
    return model


def build_model_tuner(hp, notes_vocab_size, durations_vocab_size):
    embedding_dim = hp.Choice('embedding_dim', values=[256, 512, 1024])
    feed_forward_dim = hp.Choice('feed_forward_dim', values=[256, 512, 1024])
    key_dim = hp.Choice('key_dim', values=[64, 128])
    num_heads = hp.Choice('num_heads', values=[4, 8, 12])
    gradient_clip = hp.Choice('gradient_clip', values=[0.5, 1.0, 1.5])
    dropout_rate = hp.Float('dropout_rate', min_value=0.2, max_value=0.5, step=0.1)
    l2_reg = hp.Float('l2_reg', min_value=1e-6, max_value=1e-3, sampling='LOG')
    num_transformer_blocks = hp.Choice('num_transformer_blocks', values=[1, 2, 3])
    model = build_model(notes_vocab_size, durations_vocab_size, gradient_clip=gradient_clip,
                        embedding_dim=embedding_dim, feed_forward_dim=feed_forward_dim,
num_heads=num_heads,
                        key_dim=key_dim, dropout_rate=dropout_rate, l2_reg=l2_reg,
                        num_transformer_blocks=num_transformer_blocks, verbose=False)
    return model


def train_choral_composition_model(epochs=100, suffix="", transposed=False):
    """Trains a choral Transformer model to generate notes and durations."""
    BATCH_SIZE = 128
    GENERATE_LEN = 25
    INCLUDE_AUGMENTED = False
    DATAPATH = "Data/Glob/Combined_choral" if not transposed else
"Data/Glob/Combined_transposed"
```

```
157.     VALIDATION_SPLIT = 0.1
158.
159.     def merge_voice_parts(voice_parts_notes, voice_parts_durations, seq_len=50,
max_rest_len=4):
160.         merged_notes = []
161.         merged_durations = []
162.         # Old design: truncate all voice parts to the length of the shortest one (working)
163.         # min_length = min([len(voice_parts_notes[voice]) for voice in voice_parts_notes])
164.         # for voice in voice_parts_notes:
165.         #     voice_parts_notes[voice] = voice_parts_notes[voice][:min_length]
166.         #     voice_parts_durations[voice] = voice_parts_durations[voice][:min_length]
167.         # New design attempt 1 -- put notes in cross-voice order ([S, A, T, B, S, A, T, B],
...)
168.         notes_sequences = {"S": [], "A": [], "T": [], "B": []}
169.         durations_sequences = {"S": [], "A": [], "T": [], "B": []}
170.         # for i in range(min_length):
171.         #     for voice in voice_parts_notes:
172.         for voice in voice_parts_notes:
173.             for i in range(len(voice_parts_notes[voice])):
174.                 if max_rest_len is None:
175.                     notes_sequences[voice] += voice_parts_notes[voice][i].split(" ")
176.                     durations_sequences[voice] += voice_parts_durations[voice][i].split(" ")
177.                 else:  # Attempt 1.5 -- limit the number of sequential rests
178.                     split_notes = voice_parts_notes[voice][i].split(" ")
179.                     split_durations = voice_parts_durations[voice][i].split(" ")
180.                     rest_cnt = 0
181.                     for j in range(len(split_notes)):
182.                         if "rest" in split_notes[j]:
183.                             rest_cnt += 1
184.                         else:
185.                             rest_cnt = 0
186.                         if rest_cnt <= max_rest_len:
187.                             notes_sequences[voice].append(split_notes[j])
188.                             durations_sequences[voice].append(split_durations[j])
189.                 pass
190.         # Attempt 1.75 -- truncate to the minimum length after removing rests
191.         min_length = min([len(notes_sequences[voice]) for voice in notes_sequences])
192.         for voice in notes_sequences:
193.             notes_sequences[voice] = notes_sequences[voice][:min_length]
194.             durations_sequences[voice] = durations_sequences[voice][:min_length]
195.         note_parts_combined = []
196.         duration_parts_combined = []
197.         for i in range(0, min_length * 4, 4):  # each iteration processes one SATB set
198.             if i + 4 > min_length * 4:  # if we're at the end and there's no full SATB set
199.                 break
200.             for part in ['S', 'A', 'T', 'B']:
201.                 note_parts_combined.extend(notes_sequences[part][i // 4:i // 4 + 1])
202.                 duration_parts_combined.extend(durations_sequences[part][i // 4:i // 4 + 1])
203.         # Split the combined sequences into chunks of seq_len
204.         for i in range(0, len(note_parts_combined), seq_len):
205.             merged_notes.append(' '.join(note_parts_combined[i:i + seq_len]))
206.             merged_durations.append(' '.join(duration_parts_combined[i:i + seq_len]))
207.         return merged_notes, merged_durations
208.
209.     voices = ["S", "A", "T", "B"]
210.     voice_parts_notes = {}
211.     voice_parts_durations = {}
212.     for voice in voices:
213.         print(f"Loading {voice} voice parts from {DATAPATH}...")
```

```
214.         voice_parts_notes[voice] =
load_pickle_from_slices(f"{DATAPATH}/Combined_{voice}_choral_notes", False)
215.         voice_parts_durations[voice] =
load_pickle_from_slices(f"{DATAPATH}/Combined_{voice}_choral_durations", False)
216.         if INCLUDE_AUGMENTED:
217.             for i in range(1, 5):
218.                 aug_notes =
load_pickle_from_slices(f"{DATAPATH}/Combined_aug{i}_{voice}_choral_notes", False)
219.                 aug_dur =
load_pickle_from_slices(f"{DATAPATH}/Combined_aug{i}_{voice}_choral_durations", False)
220.                 voice_parts_notes[voice] += aug_notes
221.                 voice_parts_durations[voice] += aug_dur
222.
223.     notes, durations = merge_voice_parts(voice_parts_notes, voice_parts_durations, seq_len=52)
# seq_len=32, 52, 100
224.     DATARANGE = .25  # May be better to shrink the dataset here rather than after tokenizing
225.     notes = notes[:int(DATARANGE * len(notes))]
226.     durations = durations[:int(DATARANGE * len(durations))]
227.     notes_seq_ds, notes_vectorize_layer, notes_vocab = create_transformer_dataset(notes,
BATCH_SIZE)
228.     durations_seq_ds, durations_vectorize_layer, durations_vocab =
create_transformer_dataset(durations, BATCH_SIZE)
229.     seq_ds = tf.data.Dataset.zip((notes_seq_ds, durations_seq_ds))
230.
231.     notes_vocab_size = len(notes_vocab)
232.     durations_vocab_size = len(durations_vocab)
233.
234.     # Save vocabularies if they don't exist
235.     if not os.path.exists(f"Weights/Composition_Choral{suffix}"):
236.         os.makedirs(f"Weights/Composition_Choral{suffix}")
237.     if not
os.path.exists(f"Weights/Composition_Choral{suffix}/Combined_choral_notes_vocab.pkl"):
238.         with open(f"Weights/Composition_Choral{suffix}/Combined_choral_notes_vocab.pkl", "wb")
as f:
239.             pkl.dump(notes_vocab, f)
240.     if not
os.path.exists(f"Weights/Composition_Choral{suffix}/Combined_choral_durations_vocab.pkl"):
241.         with open(f"Weights/Composition_Choral{suffix}/Combined_choral_durations_vocab.pkl",
"wb") as f:
242.             pkl.dump(durations_vocab, f)
243.
244.     # Create the training set of sequences and the same sequences shifted by one note
245.     def prepare_inputs(notes, durations):
246.         notes = tf.expand_dims(notes, -1)
247.         durations = tf.expand_dims(durations, -1)
248.         tokenized_notes = notes_vectorize_layer(notes)
249.         tokenized_durations = durations_vectorize_layer(durations)
250.         x = (tokenized_notes[:, :-1], tokenized_durations[:, :-1])
251.         y = (tokenized_notes[:, 1:], tokenized_durations[:, 1:])
252.         return x, y
253.
254.     ds = seq_ds.map(prepare_inputs)  # .shuffle(1024, seed=0) shuffle may be a hindrance #
.batch(BATCH_SIZE)
255.
256.     # Splitting dataset into training and validation
257.     ds_size = ds.cardinality().numpy()
258.     train_size = int((1 - VALIDATION_SPLIT) * ds_size)
259.     train_ds = ds.take(train_size)
260.     val_ds = ds.skip(train_size)
261.
```

```
262.     def hyperparameter_search(tuner_trials=15, t_epochs=10, dataset_size=1.0,
263.                               plot=True, grid_search=False, resume=False, t_suffix=""):
264.         ptune = partial(build_model_tuner, notes_vocab_size=notes_vocab_size,
durations_vocab_size=durations_vocab_size)
265.         tuner_dir = 'Weights/Hyperparameter_search'
266.         project_name = 'choral_composition'
267.         should_overwrite = not resume or not os.path.exists(os.path.join(tuner_dir,
project_name))
268.
269.         if not grid_search:
270.             tuner = RandomSearch(
271.                 ptune,
272.                 objective=Objective("val_loss", direction="min"),
273.                 max_trials=tuner_trials,
274.                 executions_per_trial=1,
275.                 directory=tuner_dir,
276.                 project_name=project_name,
277.                 overwrite=should_overwrite)
278.         else:
279.             tuner = tuners.GridSearch(
280.                 ptune,
281.                 objective=Objective("val_loss", direction="min"),
282.                 directory=tuner_dir,
283.                 project_name=project_name,
284.                 overwrite=should_overwrite)
285.         train_ds_sm = train_ds.take(int(dataset_size * train_size))
286.         val_ds_sm = val_ds.take(int(dataset_size * (ds_size - train_size)))
287.         tuner.search(train_ds_sm, validation_data=val_ds_sm, epochs=t_epochs)
288.         best_hp = tuner.get_best_hyperparameters(num_trials=1)[0]
289.         print("Best Hyperparameters: ", best_hp.get_config())
290.         t_model = tuner.get_best_models(num_models=1)[0]
291.         t_model.summary()
292.
293.         if plot:
294.             trials = tuner.oracle.get_best_trials(num_trials=tuner_trials)
295.             results = [trial.hyperparameters.values for trial in trials]
296.             results_df = pd.DataFrame(results)
297.             results_df['score'] = [trial.score for trial in trials]
298.             results_df['trial_id'] = [trial.trial_id for trial in trials]
299.             if not os.path.exists("Logs/HyperparameterSearches"):
300.                 os.makedirs("Logs/HyperparameterSearches")
301.
results_df.to_csv(f'Logs/HyperparameterSearches/hyperparameter_results{t_suffix}.csv', index=False)
302.
303.             fig = px.parallel_coordinates(
304.                 results_df,
305.                 color="score",
306.                 labels={col: col for col in results_df.columns},
307.                 color_continuous_scale=px.colors.diverging.Tealrose,
308.                 color_continuous_midpoint=results_df['score'].mean()
309.             )
310.             fig.show()
311.             fig.write_image(f"Images/Hyperparameter_results{t_suffix}.svg", width=1200,
height=600)
312.         return t_model
313.
314.     # model = hyperparameter_search(grid_search=False, tuner_trials=15, t_epochs=50,
315.     #                               resume=False, t_suffix="_9", dataset_size=1.0)
316.     gc.collect()
317.     # Best Transposed model (.125 [models 5, 10] and .25 [9] datasets);  original key_dim=128
```

```
318.     model = build_model(notes_vocab_size, durations_vocab_size, embedding_dim=512,
feed_forward_dim=512, num_heads=8,
319.                         key_dim=64, dropout_rate=0.000001, l2_reg=1e-6,
num_transformer_blocks=3, gradient_clip=1.5)
320.     # Transposed model (original has 2 transformer blocks, #2 has 3)
321.     # model = build_model(notes_vocab_size, durations_vocab_size, embedding_dim=512,
feed_forward_dim=1024, num_heads=8,
322.     #                     key_dim=64, dropout_rate=0.3, l2_reg=WEIGHT_DECAY,
num_transformer_blocks=3, gradient_clip=1.5)
323.     plot_model(model, to_file=f'Images/Combined_choral_composition{suffix.lower()}_model.png',
324.                show_shapes=True, show_layer_names=True, expand_nested=True)
325.
326.     LOAD_MODEL = False
327.     if LOAD_MODEL and os.path.exists(f"Weights/Composition_Choral{suffix}"):
328.         model.load_weights(f"Weights/Composition_Choral{suffix}/checkpoint.ckpt")
329.         print("Loaded model weights")
330.
331.     checkpoint_callback =
callbacks.ModelCheckpoint(filepath=f"Weights/Composition_Choral{suffix}/checkpoint.ckpt",
332.                                                       save_weights_only=True, save_freq="epoch",
verbose=0)
333.     tensorboard_callback = callbacks.TensorBoard(log_dir=f"Logs/Combined_Choral")
334.     early_stopping = EarlyStopping(monitor='val_loss', patience=25, restore_best_weights=True)
# patience=5
335.
336.     # Tokenize starting prompt
337.     music_generator = MusicGenerator(notes_vocab, durations_vocab, generate_len=GENERATE_LEN,
choral=True)
338.
339.     # Train the model
340.     model.fit(train_ds, validation_data=val_ds, epochs=epochs, verbose=1,
341.               callbacks=[checkpoint_callback, early_stopping, tensorboard_callback])  # ,
music_generator
342.
343.     model.save(f"Weights/Composition_Choral{suffix}/Combined_choral.keras")
344.
345.     # Test the model
346.     TEST_MODEL = False
347.     if TEST_MODEL:
348.         start_notes = ["S:START", "A:START", "T:START", "B:START"]
349.         start_durations = ["0.0", "0.0", "0.0", "0.0"]
350.         info, midi_stream = music_generator.generate(start_notes, start_durations,
max_tokens=50, temperature=0.5)
351.         timestr = time.strftime("%Y%m%d-%H%M%S")
352.         midi_stream.write("midi", fp=os.path.join(f"Data/Generated/Combined_choral", "output-"
+ timestr + ".mid"))
353.
354.     pass
355.
356.
357. # Deprecated (for now)
358. def train_composition_model(dataset="Soprano", epochs=100, load_augmented_dataset=False):
359.     """Trains a Transformer model to generate notes and durations."""
360.     PARSE_MIDI_FILES = not os.path.exists(f"Data/Glob/{dataset}_notes.pkl")
361.     PARSED_DATA_PATH = f"Data/Glob/{dataset}_"
362.     POLYPHONIC = True
363.     PLOT_TEST = False
364.     INCLUDE_AUGMENTED = load_augmented_dataset
365.     SEQ_LEN = 50
366.     BATCH_SIZE = 256
```

```
367.     GENERATE_LEN = 50
368.     WEIGHT_DECAY = 1e-4
369.
370.     if dataset != "Combined":
371.         file_list = glob.glob(f"Data/MIDI/VoiceParts/{dataset}/Isolated/*.mid")
372.     else:
373.         file_list = glob.glob(f"Data/MIDI/VoiceParts/{dataset}/*.mid")
374.     parser = music21.converter
375.
376.     if PARSE_MIDI_FILES and dataset != "Combined":
377.         print(f"Parsing {len(file_list)} {dataset} midi files...")
378.         notes, durations = parse_midi_files(file_list, parser, SEQ_LEN + 1, PARSED_DATA_PATH,
379.                                             verbose=True, enable_chords=POLYPHONIC, limit=None)
380.     else:
381.         if dataset != "Combined":
382.             notes, durations = load_parsed_files(PARSED_DATA_PATH)
383.         else:
384.             notes = load_pickle_from_slices(f"Data/Glob/Combined/Combined_notes",
INCLUDE_AUGMENTED)
385.             durations = load_pickle_from_slices(f"Data/Glob/Combined/Combined_durations",
INCLUDE_AUGMENTED)
386.             if INCLUDE_AUGMENTED:
387.                 dataset += "_augmented"
388.
389.     example_notes = notes[658]
390.     # example_durations = durations[658]
391.     # print("\nNotes string\n", example_notes, "...")
392.     # print("\nDuration string\n", example_durations, "...")
393.
394.     notes_seq_ds, notes_vectorize_layer, notes_vocab = create_transformer_dataset(notes,
BATCH_SIZE)
395.     durations_seq_ds, durations_vectorize_layer, durations_vocab =
create_transformer_dataset(durations, BATCH_SIZE)
396.     seq_ds = tf.data.Dataset.zip((notes_seq_ds, durations_seq_ds))
397.
398.     # Display the same example notes and durations converted to ints
399.     example_tokenised_notes = notes_vectorize_layer(example_notes)
400.     # example_tokenised_durations = durations_vectorize_layer(example_durations)
401.     # print("{:10} {:10}".format("note token", "duration token"))
402.     # for i, (note_int, duration_int) in \
403.     #         enumerate(zip(example_tokenised_notes.numpy()[:11],
example_tokenised_durations.numpy()[:11],)):
404.     #     print(f"{note_int:10}{duration_int:10}")
405.
406.     notes_vocab_size = len(notes_vocab)
407.     durations_vocab_size = len(durations_vocab)
408.
409.     # Save vocabularies
410.     with open(f"Weights/Composition/{dataset}/{dataset}_notes_vocab.pkl", "wb") as f:
411.         pkl.dump(notes_vocab, f)
412.     with open(f"Weights/Composition/{dataset}/{dataset}_durations_vocab.pkl", "wb") as f:
413.         pkl.dump(durations_vocab, f)
414.
415.     # # Display some token:note mappings
416.     # print(f"\nNOTES_VOCAB: length = {len(notes_vocab)}")
417.     # for i, note in enumerate(notes_vocab[:10]):
418.     #     print(f"{i}: {note}")
419.     #
420.     # print(f"\nDURATIONS_VOCAB: length = {len(durations_vocab)}")
421.     # # Display some token:duration mappings
```

```
    # for i, note in enumerate(durations_vocab[:10]):
    #     print(f"{i}: {note}")

    # Create the training set of sequences and the same sequences shifted by one note
    def prepare_inputs(notes, durations):
        notes = tf.expand_dims(notes, -1)
        durations = tf.expand_dims(durations, -1)
        tokenized_notes = notes_vectorize_layer(notes)
        tokenized_durations = durations_vectorize_layer(durations)
        x = (tokenized_notes[:, :-1], tokenized_durations[:, :-1])
        y = (tokenized_notes[:, 1:], tokenized_durations[:, 1:])
        return x, y

    ds = seq_ds.map(prepare_inputs)  # .repeat(DATASET_REPETITIONS)

    # example_input_output = ds.take(1).get_single_element()
    # print(example_input_output)

    tpe = TokenAndPositionEmbedding(notes_vocab_size, 32)
    token_embedding = tpe.token_emb(example_tokenised_notes)
    position_embedding = tpe.pos_emb(token_embedding)
    embedding = tpe(example_tokenised_notes)

    def plot_embeddings(in_embedding, title):
        plt.imshow(np.transpose(in_embedding), cmap="coolwarm", interpolation="nearest",
origin="lower")
        plt.title(title)
        plt.xlabel("Token")
        plt.ylabel("Embedding Dimension")
        plt.show()

    plot_embeddings(token_embedding, "Token Embedding")
    plot_embeddings(position_embedding, "Position Embedding")
    plot_embeddings(embedding, "Token + Position Embedding")

    # model = build_model(notes_vocab_size, durations_vocab_size, feed_forward_dim=512,
num_heads=8)
    model = build_model(notes_vocab_size, durations_vocab_size, embedding_dim=512,
feed_forward_dim=1024, num_heads=8,
                        key_dim=64, dropout_rate=0.3, l2_reg=WEIGHT_DECAY,
num_transformer_blocks=2, gradient_clip=1.0)
    plot_model(model, to_file=f'Images/{dataset}_composition_model.png',
               show_shapes=True, show_layer_names=True, expand_nested=True)

    LOAD_MODEL = True
    if LOAD_MODEL:
        model.load_weights(f"Weights/Composition/{dataset}/checkpoint.ckpt")
        print("Loaded model weights")

    train_size = int(0.8 * len(notes))
    val_size = len(notes) - train_size
    train_ds = ds.take(train_size)
    val_ds = ds.skip(train_size).take(val_size)

    early_stopping = callbacks.EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True, verbose=1)
    checkpoint_callback =
callbacks.ModelCheckpoint(filepath=f"Weights/Composition/{dataset}/checkpoint.ckpt",
                                                    save_weights_only=True, save_freq="epoch",
verbose=0)
```

```
475.     tensorboard_callback = callbacks.TensorBoard(log_dir=f"Logs/{dataset}")
476.
477.     # Tokenize starting prompt
478.     music_generator = MusicGenerator(notes_vocab, durations_vocab, generate_len=GENERATE_LEN)
479.     # model.fit(ds, epochs=epochs, callbacks=[checkpoint_callback, tensorboard_callback,
music_generator])
480.     model.fit(train_ds, validation_data=val_ds, epochs=epochs, verbose=1,
481.               callbacks=[early_stopping, checkpoint_callback, tensorboard_callback,
music_generator])
482.     model.save(f"Weights/Composition/{dataset}.keras")
483.
484.     # Test the model
485.     info = music_generator.generate(["START"], ["0.0"], max_tokens=50, temperature=0.5)
486.     midi_stream = info[-1]["midi"].chordify()
487.     timestr = time.strftime("%Y%m%d-%H%M%S")
488.     midi_stream.write("midi", fp=os.path.join(f"Data/Generated/{dataset}", "output-" + timestr
+ ".mid"))
489.
490.     if PLOT_TEST:
491.         max_pitch = 127  # 70
492.         seq_len = len(info)
493.         grid = np.zeros((max_pitch, seq_len), dtype=np.float32)
494.
495.         for j in range(seq_len):
496.             for i, prob in enumerate(info[j]["note_probs"]):
497.                 try:
498.                     pitch = music21.note.Note(notes_vocab[i]).pitch.midi
499.                     grid[pitch, j] = prob
500.                 except:
501.                     pass
502.
503.         fig, ax = plt.subplots(figsize=(8, 8))
504.         ax.set_yticks([int(j) for j in range(35, 70)])
505.         plt.imshow(grid[35:70, :], origin="lower", cmap="coolwarm", vmin=-0.5, vmax=0.5,
extent=[0, seq_len, 35, 70])
506.         plt.title("Note Probabilities")
507.         plt.xlabel("Timestep")
508.         plt.ylabel("Pitch")
509.         plt.show()
510.
511.         plot_size = 20
512.         att_matrix = np.zeros((plot_size, plot_size))
513.         prediction_output = []
514.         last_prompt = []
515.
516.         for j in range(plot_size):
517.             atts = info[j]["atts"].max(axis=0)
518.             att_matrix[: (j + 1), j] = atts
519.             prediction_output.append(info[j]["chosen_note"][0])
520.             last_prompt.append(info[j]["prompt"][0][-1])
521.
522.         fig, ax = plt.subplots(figsize=(8, 8))
523.         _ = ax.imshow(att_matrix, cmap="Greens", interpolation="nearest")
524.         ax.set_xticks(np.arange(-0.5, plot_size, 1), minor=True)
525.         ax.set_yticks(np.arange(-0.5, plot_size, 1), minor=True)
526.         ax.grid(which="minor", color="black", linestyle="-", linewidth=1)
527.         ax.set_xticks(np.arange(plot_size))
528.         ax.set_yticks(np.arange(plot_size))
529.         ax.set_xticklabels(prediction_output[:plot_size])
530.         ax.set_yticklabels(last_prompt[:plot_size])
```

```
531.         ax.xaxis.tick_top()
532.         plt.setp(ax.get_xticklabels(), rotation=90, ha="left", va="center",
rotation_mode="anchor")
533.         plt.title("Attention Matrix")
534.         plt.xlabel("Predicted Output")
535.         plt.ylabel("Last Prompt")
536.         plt.show()
537.
538.         pass
539.
540.
541. def generate_main():
542.     DATASET = "Combined_choral"
543.     num_to_gen = int(input("Enter the number of pieces to generate: ") or 5)
544.     generate_len = int(input("Enter the length of each piece [around 100-200 works best; 200 by
default]: ") or 200)
545.     temperature = float(input("Enter the temperature to use [0.5-1.0; 0.65 by default]: ") or
0.65)
546.     suffix = input("Enter the model suffix [_Transposed2, _Transposed13; _Transposed3 by
default]: ") or "_Transposed3"
547.     do_seed = input("Do you want to seed the generation with a specific sequence? [y/n; n by
default]: ") or "n"
548.     if do_seed == "y":  # See the data_utils script to convert a MIDI file to a seed
549.         seed_notes = input("\tEnter notes, alternating SATB (e.g., \"S:C5 A:B-3 T:E4 B:rest
S:B4 A:G3 T:F#4 B:F3\"): ")
550.         seed_durs = input("\tEnter durations (as float, where 1.0 = quarter note; e.g., 4.0 2.0
1/3 1.0 1.0 0.5 ...): ")
551.         seed_notes = seed_notes.split(" ")
552.         seed_durs = seed_durs.split(" ")
553.         if len(seed_notes) != len(seed_durs):
554.             raise ValueError("Seed notes and durations must be the same length!")
555.     else:
556.         seed_notes = []
557.         seed_durs = []
558.     output_files = generate_composition(DATASET, generate_len=generate_len,
num_to_generate=num_to_gen,
559.                                         choral=True, suffix=suffix, temperature=temperature,
560.                                         seed_notes=seed_notes, seed_durs=seed_durs)
561.     print("Generated the following files:", output_files, "\nPlease post-process the ones you
like best.")
562.
563.
564. if __name__ == '__main__':
565.     print("Hello, world!")
566.     generate_main()
567.     # train_composition_model("Combined", epochs=100, load_augmented_dataset=True)
568.     # generate_composition("Combined_augmented", num_to_generate=5, generate_len=200,
temperature=2.75)
569.     # train_choral_composition_model(epochs=300, suffix="_Transposed15", transposed=True)
570.     # generate_composition("Combined_choral", num_to_generate=10, generate_len=200,
choral=True,
571.     #                      temperature=.65, suffix="_Transposed3")
572.     # for tempr in [0.65, 0.55, 0.45]:  # 0.55, ... , 1.0
573.     #     generate_composition("Combined_choral", num_to_generate=5, generate_len=200,
574.     #                          choral=True, temperature=tempr, suffix="_Transposed2")
575.
```

E.2 Transformer.py

```
import music21.clef
from abc import ABC
from data_utils import *
from keras.regularizers import l2
from keras import layers, callbacks, models
from keras.optimizers.schedules import LearningRateSchedule


def causal_attention_mask(batch_size, n_dest, n_src, dtype):
    i = tf.range(n_dest)[:, None]
    j = tf.range(n_src)
    m = i >= j - n_src + n_dest
    mask = tf.cast(m, dtype)
    mask = tf.reshape(mask, [1, n_dest, n_src])
    mult = tf.concat([tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32)], 0)
    return tf.tile(mask, mult)


class SinePositionEncoding(layers.Layer):
    def __init__(self, max_wavelength=10000, **kwargs):
        super().__init__(**kwargs)
        self.max_wavelength = max_wavelength

    def call(self, inputs):
        input_shape = tf.shape(inputs)
        seq_length = input_shape[-2]
        hidden_dim = input_shape[-1]
        position = tf.cast(tf.range(seq_length), self.compute_dtype)
        min_freq = tf.cast(1 / self.max_wavelength, dtype=self.compute_dtype)
        timescales = tf.pow(min_freq, tf.cast(2 * (tf.range(hidden_dim) // 2),
                                              self.compute_dtype) / tf.cast(hidden_dim,
self.compute_dtype))
        angles = tf.expand_dims(position, 1) * tf.expand_dims(timescales, 0)
        # Even indices are sine, odd are cosine
        cos_mask = tf.cast(tf.range(hidden_dim) % 2, self.compute_dtype)
        sin_mask = 1 - cos_mask
        # Embedding shape is [seq_length, hidden_size]
        positional_encodings = (tf.sin(angles) * sin_mask + tf.cos(angles) * cos_mask)
        return tf.broadcast_to(positional_encodings, input_shape)

    def get_config(self):
        config = super().get_config()
        config.update({"max_wavelength": self.max_wavelength,})
        return config


class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, vocab_size, embed_dim, l2_reg=1e-4):
        super(TokenAndPositionEmbedding, self).__init__()
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.l2_reg = l2_reg
        self.token_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim,
                                          embeddings_initializer="he_uniform",
                                          embeddings_regularizer=l2(self.l2_reg))
        self.pos_emb = SinePositionEncoding()

```

```
    def call(self, x):
        embedding = self.token_emb(x)
        positions = self.pos_emb(embedding)
        return embedding + positions

    def get_config(self):
        config = super().get_config()
        config.update({"vocab_size": self.vocab_size, "embed_dim": self.embed_dim, "l2_reg": self.l2_reg})
        return config


class TransformerBlock(layers.Layer):
    def __init__(self, name, num_heads=5, key_dim=256, embed_dim=256, ff_dim=256, dropout_rate=0.3, l2_reg=None):
        super(TransformerBlock, self).__init__(name=name)
        self.num_heads = num_heads
        self.key_dim = key_dim
        self.embed_dim = embed_dim
        self.ff_dim = ff_dim
        self.dropout_rate = dropout_rate
        self.l2_reg = l2_reg
        self.attn = layers.MultiHeadAttention(num_heads=num_heads, key_dim=self.key_dim, output_shape=self.embed_dim)
        self.dropout_1 = layers.Dropout(self.dropout_rate)
        self.ln_1 = layers.LayerNormalization(epsilon=1e-6)
        self.ffn_1 = layers.Dense(self.ff_dim, activation="relu", kernel_regularizer=l2(l2_reg))
        self.ffn_2 = layers.Dense(self.embed_dim, kernel_regularizer=l2(l2_reg))
        self.dropout_2 = layers.Dropout(self.dropout_rate)
        self.ln_2 = layers.LayerNormalization(epsilon=1e-6)

    def call(self, inputs):
        input_shape = tf.shape(inputs)
        batch_size = input_shape[0]
        seq_len = input_shape[1]
        causal_mask = causal_attention_mask(batch_size, seq_len, seq_len, tf.bool)
        causal_mask = tf.expand_dims(causal_mask, 1)
        attention_output, attention_scores = self.attn(inputs, inputs, attention_mask=causal_mask,
                                                       return_attention_scores=True)
        attention_output = self.dropout_1(attention_output)
        out1 = self.ln_1(inputs + attention_output)
        ffn_1 = self.ffn_1(out1)
        ffn_2 = self.ffn_2(ffn_1)
        ffn_output = self.dropout_2(ffn_2)
        return self.ln_2(out1 + ffn_output), attention_scores

    def get_config(self):
        config = super().get_config()
        config.update({
            "key_dim": self.key_dim,
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "ff_dim": self.ff_dim,
            "dropout_rate": self.dropout_rate,
            "l2_reg": self.l2_reg,
        })
        return config


```

```
113. class MusicGenerator(callbacks.Callback):
114.     def __init__(self, index_to_note, index_to_duration, top_k=10, generate_len=50,
115.                  output_path="Data/Generated/Training", choral=False, verbose=False):
116.         super().__init__()
117.         self.index_to_note = index_to_note
118.         self.note_to_index = {note: index for index, note in enumerate(index_to_note)}
119.         self.index_to_duration = index_to_duration
120.         self.duration_to_index = {duration: index for index, duration in
enumerate(index_to_duration)}
121.         self.top_k = top_k
122.         self.generate_len = generate_len
123.         self.output_path = output_path
124.         self.verbose = verbose
125.         self.choral = choral
126.
127.     def sample_from(self, probs, temperature):
128.         if self.top_k == 0:
129.             probs = probs ** (1 / temperature)
130.             probs = probs / np.sum(probs)
131.             return np.random.choice(len(probs), p=probs), probs
132.         sorted_indices = np.argsort(probs)[::-1]
133.         top_indices = sorted_indices[:self.top_k]
134.         top_probs = probs[top_indices]
135.         top_probs = top_probs ** (1 / temperature)
136.         top_probs = top_probs / np.sum(top_probs)
137.         sampled_index = np.random.choice(top_indices, p=top_probs)
138.         return sampled_index, probs
139.
140.     def get_note(self, notes, durations, temperature, instrument=None):
141.         sample_note_idx = 1
142.         # while sample_note_idx == 1:
143.         while sample_note_idx == 1 or self.index_to_note[sample_note_idx] == "START":
144.             sample_note_idx, note_probs = self.sample_from(notes[0][-1], temperature)
145.             sample_note = self.index_to_note[sample_note_idx]
146.         sample_duration_idx = 1
147.         while sample_duration_idx == 1:
148.             sample_duration_idx, duration_probs = self.sample_from(durations[0][-1],
temperature)
149.             sample_duration = self.index_to_duration[sample_duration_idx]
150.
151.         new_note = get_midi_note(sample_note, sample_duration, instrument) if not self.choral
else \
152.             get_choral_midi_note(sample_note, sample_duration)
153.
154.         return (
155.             new_note,
156.             sample_note_idx,
157.             sample_note,
158.             note_probs,
159.             sample_duration_idx,
160.             sample_duration,
161.             duration_probs,
162.         )
163.
164.     @staticmethod
165.     def get_last_attention_layer(model):
166.         for layer in reversed(model.layers):
167.             if layer.name.startswith("attention"):
168.                 return layer
169.         raise ValueError("No attention layer found in the model.")
170.
```

```
171.     def generate(self, start_notes, start_durations, max_tokens, temperature,
172.                  clef="choral", model=None, intro=False, instrument=None):
173.         if model is not None:
174.             self.model = model
175.         last_attention_layer = self.get_last_attention_layer(self.model)
176.         attention_model = models.Model(inputs=self.model.input,
outputs=last_attention_layer.output)
177.         # attention_model = models.Model(inputs=self.model.input,
outputs=self.model.get_layer("attention").output)
178.         start_note_tokens = [self.note_to_index.get(x, 1) for x in start_notes]
179.         start_duration_tokens = [self.duration_to_index.get(x, 1) for x in start_durations]
180.         sample_note = None
181.         sample_duration = None
182.         info = []
183.
184.         if not self.choral:
185.             midi_stream = music21.stream.Stream()
186.
187.             if clef == "treble":
188.                 midi_stream.append(music21.clef.TrebleClef())
189.             elif clef == "bass":
190.                 midi_stream.append(music21.clef.BassClef())
191.             elif clef == "tenor":
192.                 midi_stream.append(music21.clef.Treble8vbClef())
193.             elif clef == "choral":
194.                 midi_stream.append(music21.clef.TrebleClef())
195.                 midi_stream.append(music21.clef.BassClef())
196.
197.             if instrument is not None:
198.                 instruments = {"Soprano": music21.instrument.Soprano(), "Alto":
music21.instrument.Alto(),
199.                                "Tenor": music21.instrument.Tenor(), "Bass":
music21.instrument.Bass()}
200.                 midi_stream.append(instruments[instrument])
201.
202.             for sample_note, sample_duration in zip(start_notes, start_durations):
203.                 new_note = get_midi_note(sample_note, sample_duration, instrument)
204.                 if new_note is not None:
205.                     midi_stream.append(new_note)
206.
207.             if intro:
208.                 info.append({
209.                     "prompt": [start_notes.copy(), start_durations.copy()],
210.                     "midi": midi_stream,
211.                     "chosen_note": (sample_note, sample_duration),
212.                     "note_probs": 1,
213.                     "duration_probs": 1,
214.                     "atts": [],
215.                 })
216.
217.             while len(start_note_tokens) < max_tokens:
218.                 x1 = np.array([start_note_tokens])
219.                 x2 = np.array([start_duration_tokens])
220.                 notes, durations = self.model.predict([x1, x2], verbose=0)
221.
222.                 repeat = True
223.                 while repeat:
224.                     (
225.                         new_note,
226.                         sample_note_idx,
227.                         sample_note,
```

```
228.                         note_probs,
229.                         sample_duration_idx,
230.                         sample_duration,
231.                         duration_probs,
232.                     ) = self.get_note(notes, durations, temperature, instrument)
233.
234.                     if (isinstance(new_note, music21.chord.Chord) or isinstance(new_note,
music21.note.Note) or
235.                         isinstance(new_note, music21.note.Rest)) and sample_duration == "0.0":
236.                         repeat = True
237.                         continue
238.                     elif intro and (isinstance(new_note, music21.tempo.MetronomeMark) or
239.                                     isinstance(new_note, music21.key.Key) or
240.                                     isinstance(new_note, music21.meter.TimeSignature)):
241.                         repeat = True
242.                         continue
243.                     else:
244.                         repeat = False
245.
246.                 if new_note is not None:
247.                     midi_stream.append(new_note)
248.
249.                 _, att = attention_model.predict([x1, x2], verbose=0)
250.
251.                 info.append({
252.                     "prompt": [start_notes.copy(), start_durations.copy()],
253.                     "midi": midi_stream,
254.                     "chosen_note": (sample_note, sample_duration),
255.                     "note_probs": note_probs,
256.                     "duration_probs": duration_probs,
257.                     "atts": att[0, :, -1, :],
258.                 })
259.                 start_note_tokens.append(sample_note_idx)
260.                 start_duration_tokens.append(sample_duration_idx)
261.                 start_notes.append(sample_note)
262.                 start_durations.append(sample_duration)
263.
264.                 if sample_note == "START":
265.                     break
266.
267.             return info
268.         else:
269.             voice_streams = {
270.                 'S': music21.stream.Part(),
271.                 'A': music21.stream.Part(),
272.                 'T': music21.stream.Part(),
273.                 'B': music21.stream.Part()
274.             }
275.
276.             clefs = {
277.                 'S': music21.clef.TrebleClef(),
278.                 'A': music21.clef.TrebleClef(),
279.                 'T': music21.clef.Treble8vbClef(),
280.                 'B': music21.clef.BassClef()
281.             }
282.
283.             for voice, stream in voice_streams.items():
284.                 stream.append(clefs[voice])
285.
286.             for sample_token, sample_duration in zip(start_notes, start_durations):
287.                 voice_type = sample_token.split(":")[0]
```

```
288.                 new_note = get_choral_midi_note(sample_token, sample_duration)
289.                 if new_note is not None:
290.                     if voice_type not in ["S", "A", "T", "B"]:
291.                         voice_streams["S"].append(new_note)
292.                     else:
293.                         voice_streams[voice_type].append(new_note)
294.             pass
295.
296.             if intro:
297.                 info.append({
298.                     "prompt": [start_notes.copy(), start_durations.copy()],
299.                     "midi": voice_streams,
300.                     "chosen_note": (sample_note, sample_duration),
301.                     "note_probs": 1,
302.                     "duration_probs": 1,
303.                     "atts": [],
304.                 })
305.
306.             while len(start_note_tokens) < max_tokens * 4:
307.                 x1 = np.array([start_note_tokens])
308.                 x2 = np.array([start_duration_tokens])
309.                 notes, durations = self.model.predict([x1, x2], verbose=0)
310.
311.                 repeat = True
312.                 while repeat:
313.                     (
314.                         new_note,
315.                         sample_note_idx,
316.                         sample_note,
317.                         note_probs,
318.                         sample_duration_idx,
319.                         sample_duration,
320.                         duration_probs,
321.                     ) = self.get_note(notes, durations, temperature)
322.
323.                     voice_type = sample_note.split(":")[0]
324.
325.                     if (isinstance(new_note, music21.chord.Chord) or isinstance(new_note,
music21.note.Note) or
326.                         isinstance(new_note, music21.note.Rest)) and sample_duration == "0.0":
327.                         repeat = True
328.                         continue
329.                     elif intro and (isinstance(new_note, music21.tempo.MetronomeMark) or
330.                                     isinstance(new_note, music21.key.Key) or
331.                                     isinstance(new_note, music21.meter.TimeSignature)):
332.                         repeat = True
333.                         continue
334.                     else:
335.                         repeat = False
336.
337.                     if new_note is not None:
338.                         if voice_type not in ["S", "A", "T", "B"]:
339.                             voice_streams["S"].append(new_note)
340.                         else:
341.                             voice_streams[voice_type].append(new_note)
342.                 pass
343.
344.                 _, att = attention_model.predict([x1, x2], verbose=0)
345.
346.                 info.append({
347.                     "prompt": [start_notes.copy(), start_durations.copy()],
```

```
348.                     "midi": voice_streams,
349.                     "chosen_note": (sample_note, sample_duration),
350.                     "note_probs": note_probs,
351.                     "duration_probs": duration_probs,
352.                     "atts": att[0, :, -1, :],
353.                 })
354.
355.                 start_note_tokens.append(sample_note_idx)
356.                 start_duration_tokens.append(sample_duration_idx)
357.                 start_notes.append(sample_note)
358.                 start_durations.append(sample_duration)
359.
360.                 if sample_note == "START":
361.                     break
362.
363.             midi_stream = music21.stream.Score()
364.             for voice, stream in voice_streams.items():
365.                 midi_stream.insert(0, stream)
366.
367.             return info, midi_stream
368.
369.     def on_epoch_end(self, epoch, logs=None):
370.         if not self.choral:
371.             info = self.generate(["START"], ["0.0"], max_tokens=self.generate_len,
temperature=0.5)
372.             midi_stream = info[-1]["midi"].chordify()
373.         else:
374.             start_notes = ["S:START", "A:START", "T:START", "B:START"]
375.             start_durations = ["0.0", "0.0", "0.0", "0.0"]
376.             info, midi_stream = self.generate(start_notes, start_durations,
377.                                               max_tokens=self.generate_len*4, temperature=0.5)
378.         if self.verbose:
379.             print(info[-1]["prompt"])
380.         midi_stream.write("midi", fp=os.path.join(self.output_path, "output-" +
str(epoch+1).zfill(4) + ".mid"))
381.
382.
383. class NoamSchedule(LearningRateSchedule, ABC):
384.     def __init__(self, d_model, warmup_steps=4000):
385.         super(NoamSchedule, self).__init__()
386.         self.d_model = tf.constant(d_model, dtype=tf.float32)
387.         self.warmup_steps = tf.constant(warmup_steps, dtype=tf.float32)
388.
389.     def __call__(self, step):
390.         step = tf.cast(step, tf.float32)
391.         arg1 = tf.math.rsqrt(step)
392.         arg2 = step * tf.pow(self.warmup_steps, -1.5)
393.         return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
394.
395.     def get_config(self):
396.         return {
397.             'd_model': self.d_model,
398.             'warmup_steps': self.warmup_steps
399.         }
400.
```

E.3 Data_utils.py

```python
import os
import glob
import mido
import random
import music21
import pretty_midi
import numpy as np
import pandas as pd
import pickle as pkl
import tensorflow as tf
from keras import layers
from fractions import Fraction


# region Dataframes
def note_number_to_name(note_number):
    """Converts a MIDI note number to a note name with pitch class."""
    note_names = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']
    octave = note_number // 12 - 1
    note = note_names[note_number % 12]
    return note + str(octave)


def key_signature_to_number(key_signature):
    mapping = ['A', 'B', 'C', 'D', 'E', 'F', 'G',
               'A#', 'B#', 'C#', 'D#', 'E#', 'F#', 'G#',
               'Ab', 'Bb', 'Cb', 'Db', 'Eb', 'Fb', 'Gb',
               'Am', 'Bm', 'Cm', 'Dm', 'Em', 'Fm', 'Gm',
               'A#m', 'B#m', 'C#m', 'D#m', 'E#m', 'F#m', 'G#m',
               'Abm', 'Bbm', 'Cbm', 'Dbm', 'Ebm', 'Fbm', 'Gbm']
    if str(key_signature).isnumeric():
        return mapping[int(key_signature)]
    return mapping.index(key_signature)


def midi_to_dataframe(midi_file):
    """Converts a MIDI file to a pandas dataframe.
    The dataframe has the following columns:
    - event: The name of the note or rest
    - velocity: The velocity of the note
    - time: The time in seconds of the event
    - tempo: The tempo in beats per minute at the time of the event
    - time_signature_count: The time signature numerator at the time of the event
    - time_signature_beat: The time signature denominator at the time of the event
    - key_signature: The key signature at the time of the event
    """
    mid = mido.MidiFile(midi_file)

    events = []
    velocities = []
    times = []
    tempi = []
    time_signatures = []
    key_signatures = []

    current_tempo = 500000  # MIDI default tempo (microseconds per beat)
```

```
    current_time_signature = '4/4'  # Default time signature
    current_key_signature = 'C'  # Default key signature
    current_time = 0  # Current time in seconds
    last_event_time = 0  # Time of the last event

    for _, track in enumerate(mid.tracks):
        for i, msg in enumerate(track):
            time_delta = mido.tick2second(msg.time, mid.ticks_per_beat, current_tempo)
            current_time += time_delta

            if msg.type == 'note_on':
                if msg.velocity > 0:
                    if current_time > last_event_time:
                        # There is a gap between the last event and this one, insert a rest
                        # events.append('rest')
                        events.append(-1)
                        velocities.append(0)
                        times.append(last_event_time)
                        tempi.append(mido.tempo2bpm(current_tempo))
                        time_signatures.append(current_time_signature)
                        # key_signatures.append(current_key_signature)
                        key_signatures.append(key_signature_to_number(current_key_signature))
                    # events.append(note_number_to_name(msg.note))
                    events.append(msg.note)
                    velocities.append(msg.velocity)
                    times.append(current_time)
                    tempi.append(mido.tempo2bpm(current_tempo))
                    time_signatures.append(current_time_signature)
                    # key_signatures.append(current_key_signature)
                    key_signatures.append(key_signature_to_number(current_key_signature))
                    last_event_time = current_time
            elif msg.type == 'set_tempo':
                current_tempo = msg.tempo  # May need to record in a meta_events list also
            elif msg.type == 'time_signature':
                current_time_signature = f"{msg.numerator}/{msg.denominator}"
            elif msg.type == 'key_signature':
                current_key_signature = msg.key
        current_time = 0

    # Split the time signature into two arrays, one for the numerator and one for the
denominator
    time_signature_counts = []
    time_signature_beats = []

    for time_signature in time_signatures:
        time_signature_counts.append(time_signature.split('/')[0])
        time_signature_beats.append(time_signature.split('/')[1])

    df = pd.DataFrame({'event': events, 'velocity': velocities, 'time': times, 'tempo': tempi,
                       'time_signature_count': time_signature_counts, 'time_signature_beat':
time_signature_beats,
                       'key_signature': key_signatures})
    return df


def transpose_df_to_row(dataframe):
    """Transpose a dataframe to a single row where each column is a 1D array from the original
dataframe."""
    np.set_printoptions(threshold=np.inf)
    df = pd.DataFrame()
```

```
114.     for column in dataframe.columns:
115.         # Turn each column into a 1D array, then turn the array into a string in the form "[1,
2, 3, ...]"
116.         df[column] = [np.array2string(dataframe[column].to_numpy(), separator=',')]
117.     return df
118.
119.
120. def build_dataset(data_dir):
121.     """Builds a dataset from a directory of MIDI files."""
122.     df = pd.DataFrame()
123.     for root, dirs, files in os.walk(data_dir):
124.         for file in files:
125.             if file.lower().endswith('.mid') or file.lower().endswith('.midi'):
126.                 df = pd.concat([df, transpose_df_to_row(midi_to_dataframe(os.path.join(root,
file)))])
127.     return df
128.
129.
130. def save_dataset(dataframe, output_file):
131.     dataframe.to_csv(output_file, index=False, sep=';')
132.
133.
134. def create_all_datasets():
135.     SOPRANO_PATH = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Soprano\Isolated")
136.     ALTO_PATH = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Alto\Isolated")
137.     TENOR_PATH = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Tenor\Isolated")
138.     BASS_PATH = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Bass\Isolated")
139.     print("Building datasets...")
140.     df_soprano = build_dataset(SOPRANO_PATH)
141.     df_alto = build_dataset(ALTO_PATH)
142.     df_tenor = build_dataset(TENOR_PATH)
143.     df_bass = build_dataset(BASS_PATH)
144.     print("Saving datasets...")
145.     save_dataset(df_soprano, os.path.join(os.getcwd(), r"Data\Tabular\Soprano.csv"))
146.     save_dataset(df_alto, os.path.join(os.getcwd(), r"Data\Tabular\Alto.csv"))
147.     save_dataset(df_tenor, os.path.join(os.getcwd(), r"Data\Tabular\Tenor.csv"))
148.     save_dataset(df_bass, os.path.join(os.getcwd(), r"Data\Tabular\Bass.csv"))
149.     print("Complete!")
150.
151. # endregion Dataframes
152.
153.
154. # region ChoralTransformer
155. def parse_single_midi_to_notedur_output():
156.     filepath = input("Enter the path to the MIDI file: ").replace('"', '')
157.     datalen = float(input("What percentage of the file would you like to return? (0-100; 100%
by default): ") or 100.0)
158.     datalen = datalen / 100.0 if datalen >= 1 else datalen
159.     max_rests = int(input("What is the max number of consecutive rests allowed? (12 by default;
-1 for all): ") or 12)
160.     max_rests = None if max_rests == -1 else max_rests
161.
162.     parser = music21.converter
163.     seq_len = 50
164.     all_voices_data = {'S': [], 'A': [], 'T': [], 'B': []}
165.     score = parser.parse(filepath)
166.     for part, voice in zip(score.parts, all_voices_data.keys()):
167.         notes = []
168.         durations = []
169.         for element in part.flat:
```

```
            note_name = None
            duration_name = None
            if isinstance(element, music21.note.Rest):
                note_name = voice + ":" + str(element.name)
                duration_name = str(element.duration.quarterLength)
            elif isinstance(element, music21.note.Note):
                note_name = voice + ":" + str(element.nameWithOctave)
                duration_name = str(element.duration.quarterLength)
            if note_name and duration_name:
                notes.append(note_name)
                durations.append(duration_name)
        for j in range(len(notes) - seq_len):
            all_voices_data[voice].append({
                'notes': " ".join(notes[j: (j + seq_len)]),
                'durations': " ".join(durations[j: (j + seq_len)])
            })

    def merge_voice_parts(voice_parts_notes, voice_parts_durations, max_rest_len=12):
        merged_notes = []
        merged_durations = []
        notes_sequences = {"S": [], "A": [], "T": [], "B": []}
        durations_sequences = {"S": [], "A": [], "T": [], "B": []}
        for voice in voice_parts_notes:
            for i in range(len(voice_parts_notes[voice])):
                if max_rest_len is None:
                    notes_sequences[voice] += voice_parts_notes[voice][i].split(" ")
                    durations_sequences[voice] += voice_parts_durations[voice][i].split(" ")
                else:
                    split_notes = voice_parts_notes[voice][i].split(" ")
                    split_durations = voice_parts_durations[voice][i].split(" ")
                    rest_cnt = 0
                    for j in range(len(split_notes)):
                        if "rest" in split_notes[j]:
                            rest_cnt += 1
                        else:
                            rest_cnt = 0
                        if rest_cnt <= max_rest_len:
                            notes_sequences[voice].append(split_notes[j])
                            durations_sequences[voice].append(split_durations[j])
                    pass
        min_length = min([len(notes_sequences[voice]) for voice in notes_sequences])
        for voice in notes_sequences:
            notes_sequences[voice] = notes_sequences[voice][:min_length]
            durations_sequences[voice] = durations_sequences[voice][:min_length]
        note_parts_combined = []
        duration_parts_combined = []
        for i in range(0, min_length * 4, 4):
            if i + 4 > min_length * 4:
                break
            for part in ['S', 'A', 'T', 'B']:
                note_parts_combined.extend(notes_sequences[part][i // 4:i // 4 + 1])
                duration_parts_combined.extend(durations_sequences[part][i // 4:i // 4 + 1])
        merged_notes.append(' '.join(note_parts_combined))
        merged_durations.append(' '.join(duration_parts_combined))
        return merged_notes, merged_durations

    all_notes = {"S": [], "A": [], "T": [], "B": []}
    all_durations = {"S": [], "A": [], "T": [], "B": []}
    for voice, data in all_voices_data.items():
        for entry in data:
```

```
230.             all_notes[voice].append(entry['notes'])
231.             all_durations[voice].append(entry['durations'])
232.     notes, durations = merge_voice_parts(all_notes, all_durations, max_rest_len=max_rests)
233.     str_notes = ' '.join(notes).split(" ")
234.     str_notes = ' '.join(str_notes[:int(len(str_notes) * datalen)])
235.     str_durations = ' '.join(durations).split(" ")
236.     str_durations = ' '.join(str_durations[:int(len(str_durations) * datalen)])
237.     print(f"\nNotes:\n{str_notes}\n\nDurations:\n{str_durations}")
238.
239.
240. def parse_choral_midi_files(file_list, parser, seq_len, parsed_data_path=None, verbose=False,
241.                             limit=None, mm_limit=0, include_key=True):
242.     all_voices_data = {'S': [], 'A': [], 'T': [], 'B': []}
243.
244.     if limit is not None:
245.         file_list = file_list[:limit]
246.
247.     for i, file in enumerate(file_list):
248.         if verbose:
249.             print(i + 1, "Parsing %s" % file)
250.         try:
251.             score = parser.parse(file)
252.         except Exception as e:
253.             print(f"\tError parsing file {file}: {e}")
254.             continue
255.         for part, voice in zip(score.parts, all_voices_data.keys()):
256.             notes = ["START"]
257.             durations = ["0.0"]
258.             if mm_limit != 0:
259.                 part = part.measures(0, mm_limit)
260.             for element in part.flat:
261.                 note_name = None
262.                 duration_name = None
263.                 if isinstance(element, music21.tempo.MetronomeMark):
264.                     note_name = str(element.number) + "BPM"
265.                     duration_name = "0.0"
266.                 elif isinstance(element, music21.key.Key) and include_key:
267.                     note_name = str(element.tonic.name) + ":" + str(element.mode)
268.                     duration_name = "0.0"
269.                 elif isinstance(element, music21.meter.TimeSignature):
270.                     note_name = str(element.ratioString) + "TS"
271.                     duration_name = "0.0"
272.                 elif isinstance(element, music21.note.Rest):
273.                     note_name = voice + ":" + str(element.name)
274.                     duration_name = str(element.duration.quarterLength)
275.                 elif isinstance(element, music21.note.Note):
276.                     note_name = voice + ":" + str(element.nameWithOctave)
277.                     duration_name = str(element.duration.quarterLength)
278.                 if note_name and duration_name:
279.                     notes.append(note_name)
280.                     durations.append(duration_name)
281.             notes.append("END")
282.             durations.append("0.0")
283.             for j in range(len(notes) - seq_len):
284.                 all_voices_data[voice].append({
285.                     'notes': " ".join(notes[j: (j + seq_len)]),
286.                     'durations': " ".join(durations[j: (j + seq_len)])
287.                 })
288.     if parsed_data_path:
289.         for voice, data in all_voices_data.items():
```

```
290.             with open((parsed_data_path + f"{voice}_choral_notes.pkl"), "wb") as f:
291.                 pkl.dump([entry['notes'] for entry in data], f)
292.             with open((parsed_data_path + f"{voice}_choral_durations.pkl"), "wb") as f:
293.                 pkl.dump([entry['durations'] for entry in data], f)
294.     return all_voices_data
295.
296.
297. def get_choral_midi_note(sample_token, sample_duration):
298.     new_note = None
299.     try:
300.         voice_type, sample_note = sample_token.split(":")[0],
":".join(sample_token.split(":")[1:])
301.         if "BPM" in sample_token:
302.             new_note =
music21.tempo.MetronomeMark(number=int(round(float(sample_token.split("BPM")[0]))))
303.         elif "TS" in sample_token:
304.             new_note = music21.meter.TimeSignature(sample_token.split("TS")[0])
305.         elif "major" in sample_note or "minor" in sample_note:
306.             tonic, mode = sample_token.split(":")
307.             new_note = music21.key.Key(tonic, mode)
308.         elif sample_note == "rest":
309.             new_note = music21.note.Rest()
310.             new_note.duration = music21.duration.Duration(float(Fraction(sample_duration)))
311.             new_note.storedInstrument = get_voice_instrument(voice_type)
312.         elif sample_note != "START" and sample_note != "END":
313.             new_note = music21.note.Note(sample_note)
314.             new_note.duration = music21.duration.Duration(float(Fraction(sample_duration)))
315.             new_note.storedInstrument = get_voice_instrument(voice_type)
316.     except Exception:
317.         return None
318.     return new_note
319.
320.
321. def get_voice_instrument(voice_type):
322.     if voice_type == "Soprano":
323.         return music21.instrument.Soprano()
324.     elif voice_type == "Alto":
325.         return music21.instrument.Alto()
326.     elif voice_type == "Tenor":
327.         return music21.instrument.Tenor()
328.     elif voice_type == "Bass":
329.         return music21.instrument.Bass()
330.     else:
331.         return music21.instrument.Vocalist()
332.
333.
334. def parse_midi_files(file_list, parser, seq_len, parsed_data_path=None, verbose=False,
enable_chords=False, limit=None):
335.     notes = []
336.     durations = []
337.
338.     if limit is not None:
339.         file_list = file_list[:limit]
340.     for i, file in enumerate(file_list):
341.         if verbose:
342.             print(i + 1, "Parsing %s" % file)
343.         score = parser.parse(file).chordify()
344.         notes.append("START")
345.         durations.append("0.0")
346.         for element in score.flat:
```

```
347.             note_name = None
348.             duration_name = None
349.             # if isinstance(element, music21.clef.Clef):
350.             #     note_name = f"{element.sign}:{element.line}:{element.octaveChange}CLEF"
351.             #     duration_name = "0.0"
352.             if isinstance(element, music21.tempo.MetronomeMark):
353.                 note_name = str(element.number) + "BPM"
354.                 duration_name = "0.0"
355.             elif isinstance(element, music21.key.Key):
356.                 note_name = str(element.tonic.name) + ":" + str(element.mode)
357.                 duration_name = "0.0"
358.             elif isinstance(element, music21.meter.TimeSignature):
359.                 note_name = str(element.ratioString) + "TS"
360.                 duration_name = "0.0"
361.             elif isinstance(element, music21.chord.Chord):
362.                 note_name = '.'.join(n.nameWithOctave for n in element.pitches) if
enable_chords \
363.                             else element.pitches[-1].nameWithOctave
364.                 duration_name = str(element.duration.quarterLength)
365.             elif isinstance(element, music21.note.Rest):
366.                 note_name = str(element.name)
367.                 duration_name = str(element.duration.quarterLength)
368.             elif isinstance(element, music21.note.Note):
369.                 note_name = str(element.nameWithOctave)
370.                 duration_name = str(element.duration.quarterLength)
371.             if note_name and duration_name:
372.                 notes.append(note_name)
373.                 durations.append(duration_name)
374.         if verbose:
375.             print(f"{len(notes)} notes parsed")
376.
377.     notes_list = []
378.     duration_list = []
379.     if verbose:
380.         print(f"Building sequences of length {seq_len}")
381.     for i in range(len(notes) - seq_len):
382.         notes_list.append(" ".join(notes[i: (i + seq_len)]))
383.         duration_list.append(" ".join(durations[i: (i + seq_len)]))
384.     if parsed_data_path:
385.         with open((parsed_data_path + "notes.pkl"), "wb") as f:
386.             pkl.dump(notes_list, f)
387.         with open((parsed_data_path + "durations.pkl"), "wb") as f:
388.             pkl.dump(duration_list, f)
389.
390.     return notes_list, duration_list
391.
392.
393. def get_midi_note(sample_note, sample_duration, instrument=None):
394.     new_note = None
395.     # if "CLEF" in sample_note:
396.     #     sign, line, octave_change = sample_note.split("CLEF")[0].split(":")
397.     #     new_note = music21.clef.Clef(sign=sign, line=int(line),
octaveChange=int(octave_change))
398.     instruments = {"Soprano": music21.instrument.Soprano(), "Alto": music21.instrument.Alto(),
399.                    "Tenor": music21.instrument.Tenor(), "Bass": music21.instrument.Bass()}
400.     instrument = instruments[instrument] if instrument else music21.instrument.Vocalist()
401.     if "BPM" in sample_note:
402.         new_note =
music21.tempo.MetronomeMark(number=round(float(sample_note.split("BPM")[0])))
403.     elif "TS" in sample_note:
```

```
404.         if int(sample_note.split("TS")[0].split("/")[0]) > 16:
405.             sample_note = "12/" + sample_note.split("/")[1]
406.         new_note = music21.meter.TimeSignature(sample_note.split("TS")[0])
407.     elif "major" in sample_note or "minor" in sample_note:
408.         tonic, mode = sample_note.split(":")
409.         new_note = music21.key.Key(tonic, mode)
410.     elif sample_note == "rest":
411.         new_note = music21.note.Rest()
412.         new_note.duration = music21.duration.Duration(float(Fraction(sample_duration)))
413.         new_note.storedInstrument = instrument
414.     elif "." in sample_note:
415.         notes_in_chord = sample_note.split(".")
416.         chord_notes = []
417.         for current_note in notes_in_chord:
418.             n = music21.note.Note(current_note)
419.             n.duration = music21.duration.Duration(float(Fraction(sample_duration)))
420.             n.storedInstrument = instrument
421.             chord_notes.append(n)
422.         new_note = music21.chord.Chord(chord_notes)
423.     elif sample_note == "rest":
424.         new_note = music21.note.Rest()
425.         new_note.duration = music21.duration.Duration(float(Fraction(sample_duration)))
426.         new_note.storedInstrument = instrument
427.     elif sample_note != "START":
428.         new_note = music21.note.Note(sample_note)
429.         new_note.duration = music21.duration.Duration(float(Fraction(sample_duration)))
430.         new_note.storedInstrument = instrument
431.     return new_note
432.
433.
434. def meta_analysis(dataset="Soprano"):
435.     """Analyzes all MIDIs in the dataset to find the following:
436.     - Average duration (in seconds) before the first note (initial entrance)
437.     - Probability distribution of time signature counts
438.     - Probability distribution of time signature beats
439.     - Probability distribution of key signatures
440.     - Min, max, and average tempo
441.     """
442.     path = os.path.join(os.getcwd(), f"Data/MIDI/VoiceParts/{dataset}/Isolated")
443.     files = sorted([f for f in os.listdir(path) if f.lower().endswith('.mid') and f !=
'desktop.ini'])
444.     first_note_entrances = []
445.     time_signature_counts = []
446.     time_signature_beats = []
447.     key_signatures = []
448.     tempi = []
449.     for index, file in enumerate(files):
450.         try:
451.             print(f"Analyzing file {index + 1}/{len(files)}")
452.             score = music21.converter.parse(os.path.join(path, file))
453.             first_note_entrances.append(score.parts[0].flat.notes[0].offset)
454.             for element in score.flat:
455.                 if isinstance(element, music21.tempo.MetronomeMark):
456.                     tempi.append(element.number)
457.                 elif isinstance(element, music21.key.Key):
458.                     key_signatures.append(element.tonic.name + ":" + element.mode)
459.                 elif isinstance(element, music21.meter.TimeSignature):
460.                     time_signature_counts.append(element.numerator)
461.                     time_signature_beats.append(element.denominator)
462.         except Exception as e:
```

```
463.             print(f"\tError parsing file {file}: {e}")
464.     results = {
465.         "first_entrance": np.mean(first_note_entrances),
466.         "time_signature_counts": np.unique(time_signature_counts, return_counts=True),
467.         "time_signature_beats": np.unique(time_signature_beats, return_counts=True),
468.         "key_signatures": np.unique(key_signatures, return_counts=True),
469.         "tempi": {"min": np.min(tempi), "max": np.max(tempi), "mean": np.mean(tempi)}
470.     }
471.     with open(os.path.join(os.getcwd(), f"Weights/VoiceMetadata/{dataset}_meta_analysis.pkl"),
"wb") as f:
472.         pkl.dump(results, f)
473.     print(results)
474.
475.
476. def view_meta_analysis(dataset="Soprano"):
477.     with open(f"Weights/VoiceMetadata/{dataset}_meta_analysis.pkl", "rb") as f:
478.         results = pkl.load(f)
479.     print(results)
480.
481.
482. def combine_intros():
483.     path = os.path.join(os.getcwd(), "Data/Generated/")
484.     latest_files = []
485.     voices = ["Soprano", "Alto", "Tenor", "Bass"]
486.     for voice in voices:
487.         c_path = os.path.join(path, f"Intro_{voice}/")
488.         files = sorted([f for f in os.listdir(c_path) if f.lower().endswith('.mid') and f !=
'desktop.ini'])
489.         latest_files.append(os.path.join(path, f"Intro_{voice}/{files[-1]}"))
490.     print(f"Combining files: {['/'.join(f.split('/')[-2:]) for f in latest_files]}")
491.     new_midi = mido.MidiFile()
492.     for i, file in enumerate(latest_files):
493.         midi = mido.MidiFile(file)
494.         if i == 0:
495.             new_midi.ticks_per_beat = midi.ticks_per_beat
496.             new_midi.tracks.append(midi.tracks[0])
497.         for j, track in enumerate(midi.tracks):
498.             if j != 0:
499.                 track.name = voices[i]
500.                 new_midi.tracks.append(track)
501.     new_midi.save(os.path.join(path, "Intro_All/" + latest_files[0].split("/")[-1]))
502.
503.
504. def validate_and_generate_metatrack(dataset="Soprano", key=None, time_sig=None, tempo=None,
entrance=None):
505.     with open(f"Weights/VoiceMetadata/{dataset}_meta_analysis.pkl", "rb") as f:
506.         metadata = pkl.load(f)
507.
508.     if key is None:
509.         # Choose a random key signature from the "key_signatures" list in the metadata using
the
510.         # probabilities from the counts (i.e., the more common keys are more likely to be
chosen)
511.         all_keys = ['C:major', 'G:major', 'D:major', 'A:major', 'E:major', 'B:major',
'F#:major', 'C#:major',
512.                     'F:major', 'B-:major', 'E-:major', 'A-:major', 'D-:major', 'G-:major', 'C-
:major',
513.                     'A:minor', 'E:minor', 'B:minor', 'F#:minor', 'C#:minor', 'G#:minor',
514.                     'D:minor', 'G:minor', 'C:minor', 'F:minor', 'B-:minor', 'E-:minor']
515.         key_signatures = metadata["key_signatures"][0]
```

```
516.         key_probs = metadata["key_signatures"][1]
517.         for c_key in all_keys:
518.             if c_key not in key_signatures:
519.                 key_signatures = np.append(key_signatures, c_key)
520.                 key_probs = np.append(key_probs, 1)
521.         key = np.random.choice(key_signatures, p=key_probs / np.sum(key_probs))
522.
523.     if time_sig is None:
524.         # Use time_signature_counts and time_signature_beats from metadata to choose a random
(probable) time signature
525.         time_sigs = metadata["time_signature_counts"][0]
526.         time_sig_probs = metadata["time_signature_counts"][1]
527.         time_sig_beats = metadata["time_signature_beats"][0]
528.         time_sig_beats_probs = metadata["time_signature_beats"][1]
529.         time_sig_count = np.random.choice(time_sigs, p=time_sig_probs / np.sum(time_sig_probs))
530.         time_sig_beats = np.random.choice(time_sig_beats, p=time_sig_beats_probs /
np.sum(time_sig_beats_probs))
531.         time_sig = f"{time_sig_count}/{time_sig_beats}TS"
532.     else:
533.         if "TS" not in time_sig:
534.             time_sig = f"{time_sig}TS"
535.
536.     if tempo is None:
537.         # Pick a random tempo based on the min, max, and mean tempos from
538.         # the metadata (with a weighted probability closer to the mean)
539.         tempo_min = metadata["tempi"]['min']
540.         tempo_max = metadata["tempi"]['max']
541.         tempo_mean = metadata["tempi"]['mean']
542.         tempo_min = tempo_min if tempo_min >= 45 else 45 + random.randint(0,
int(tempo_mean)//2)
543.         tempo_max = tempo_max if tempo_max <= 180 else 180 - random.randint(0,
int(tempo_mean)//2)
544.         tempo_probs = np.array([1 / (tempo_mean - tempo_min), 1 / (tempo_max - tempo_mean)])
545.         tempo_probs = tempo_probs / np.sum(tempo_probs)
546.         try:
547.             tempo = f"{int(np.random.choice([tempo_min, tempo_max], p=abs(tempo_probs)))}BPM"
548.         except Exception as _:
549.             tempo = "120BPM"
550.     else:
551.         tempo = f"{tempo}BPM"
552.
553.     if entrance is None:
554.         # Pick a random entrance between 0 and 1.5*first_entrance in the metadata
555.         first_entrance = metadata["first_entrance"]
556.         entrance = np.random.uniform(0, 1.5 * first_entrance)
557.         entrance = round(entrance * 4) / 4  # Round to nearest 0.25
558.
559.     return key, time_sig, tempo, entrance
560.
561.
562. def create_transformer_dataset(elements, batch_size=256):
563.     ds = (tf.data.Dataset.from_tensor_slices(elements).batch(batch_size,
drop_remainder=True).shuffle(1000))
564.     vectorize_layer = layers.TextVectorization(standardize=None, output_mode="int")
565.     vectorize_layer.adapt(ds)
566.     vocab = vectorize_layer.get_vocabulary()
567.     return ds, vectorize_layer, vocab
568.
569.
570. def load_parsed_files(parsed_data_path, from_slices=False):
```

```
571.     if from_slices:
572.         notes = load_pickle_from_slices(parsed_data_path + "notes")
573.         durations = load_pickle_from_slices(parsed_data_path + "durations")
574.         return notes, durations
575.     with open((parsed_data_path + "notes.pkl"), "rb") as f:
576.         notes = pkl.load(f)
577.     with open((parsed_data_path + "durations.pkl"), "rb") as f:
578.         durations = pkl.load(f)
579.     return notes, durations
580.
581.
582. def compile_midi_from_voices():
583.     soprano_path = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Soprano\Isolated")
584.     alto_path = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Alto\Isolated")
585.     tenor_path = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Tenor\Isolated")
586.     bass_path = os.path.join(os.getcwd(), r"Data\MIDI\VoiceParts\Bass\Isolated")
587.     soprano_files = sorted([f for f in os.listdir(soprano_path) if f.lower().endswith('.mid')
and f != 'desktop.ini'])
588.     alto_files = sorted([f for f in os.listdir(alto_path) if f.lower().endswith('.mid') and f
!= 'desktop.ini'])
589.     tenor_files = sorted([f for f in os.listdir(tenor_path) if f.lower().endswith('.mid') and f
!= 'desktop.ini'])
590.     bass_files = sorted([f for f in os.listdir(bass_path) if f.lower().endswith('.mid') and f
!= 'desktop.ini'])
591.     for soprano_file, alto_file, tenor_file, bass_file in zip(soprano_files, alto_files,
tenor_files, bass_files):
592.         new_midi = mido.MidiFile()
593.         soprano_midi = mido.MidiFile(os.path.join(soprano_path, soprano_file))
594.         new_midi.tracks.append(soprano_midi.tracks[0])
595.         new_midi.ticks_per_beat = soprano_midi.ticks_per_beat
596.         for voice_path, voice_file in zip([soprano_path, alto_path, tenor_path, bass_path],
597.                                           [soprano_file, alto_file, tenor_file, bass_file]):
598.             voice_midi = mido.MidiFile(os.path.join(voice_path, voice_file))
599.             for i, track in enumerate(voice_midi.tracks):
600.                 if i != 0:
601.                     new_midi.tracks.append(track)
602.                 elif i == 0 and voice_file != soprano_file:
603.                     for msg in track:
604.                         new_midi.tracks[0].append(msg)
605.         filename = soprano_file.split(".")[0] + "_all.mid"
606.         new_midi.save(os.path.join(os.getcwd(), "Data/MIDI/VoiceParts/Combined", filename))
607.         print("Saved file: " + filename)
608.     pass
609.
610.
611. def augment_midi_files(path):
612.     """Augments MIDI file dataset using the following methods:
613.     - Transpose by # semitones (with adjusted key signature to match)
614.     - Speed up/slow down tempo (adjust bpm)
615.     """
616.     MAJOR_COF = ['C', 'G', 'D', 'A', 'E', 'B', 'F#', 'Db', 'Ab', 'Eb', 'Bb', 'F']
617.     MINOR_COF = ['A', 'E', 'B', 'F#', 'C#', 'G#', 'D#', 'Bb', 'F', 'C', 'G', 'D']
618.
619.     def adjust_key_signature(midi_file, semitones):
620.         for key_change in midi_file.key_signature_changes:
621.             current_key = key_change.key_number
622.             is_major = current_key < 12
623.             current_key_name = MAJOR_COF[current_key] if is_major else MINOR_COF[current_key -
12]
624.             if is_major:
```

```
625.                 new_key_index = (MAJOR_COF.index(current_key_name) + semitones) % 12
626.                 new_key_name = MAJOR_COF[new_key_index]
627.                 new_key_number = MAJOR_COF.index(new_key_name)
628.             else:
629.                 new_key_index = (MINOR_COF.index(current_key_name) + semitones) % 12
630.                 new_key_name = MINOR_COF[new_key_index]
631.                 new_key_number = MINOR_COF.index(new_key_name) + 12
632.             key_change.key_number = new_key_number
633.         return midi_file
634. 
635.     def adjust_pitch(midi_file, semitones):
636.         for instrument in midi_file.instruments:
637.             for note in instrument.notes:
638.                 note.pitch += semitones
639.                 # Ensure the pitch remains within MIDI bounds (0-127)
640.                 note.pitch = min(max(note.pitch, 0), 127)
641.         return midi_file
642. 
643.     def adjust_tempo(midi_file_path, factor):
644.         midi_file = mido.MidiFile(midi_file_path)
645.         for track in midi_file.tracks:
646.             for msg in track:
647.                 if msg.type == 'set_tempo':
648.                     msg.tempo = int(msg.tempo * factor)
649.         midi_file.save(midi_file_path)
650. 
651.     files = sorted([f for f in os.listdir(path) if f.lower().endswith('.mid')
652.                     or f.lower().endswith('.midi') and f != 'desktop.ini'])
653.     tempo_adjustments = [1.3, 0.6, 2.0, 0.9]
654. 
655.     for file in files:
656.         print("Augmenting file: " + file)
657.         midi = pretty_midi.PrettyMIDI(os.path.join(path, file))
658.         for i in range(1, 5):
659.             new_midi = midi
660.             if i == 1:
661.                 # Shift down 6 half-steps (tritone), slow down all durations by 30%
662.                 new_midi = adjust_pitch(new_midi, -6)
663.                 new_midi = adjust_key_signature(new_midi, -6)
664.             elif i == 2:
665.                 # Shift up 4 half-steps (major 3rd), speed up all BPMs by 40%
666.                 new_midi = adjust_pitch(new_midi, 4)
667.                 new_midi = adjust_key_signature(new_midi, 4)
668.             elif i == 3:
669.                 # Shift up perfect 4th, slow down all BPMs by 50%
670.                 new_midi = adjust_pitch(new_midi, 5)
671.                 new_midi = adjust_key_signature(new_midi, 5)
672.             elif i == 4:
673.                 # Shift down minor 6th, speed up all BPMs by 20%
674.                 new_midi = adjust_pitch(new_midi, -8)
675.                 new_midi = adjust_key_signature(new_midi, -8)
676.             # elif i == 5:
677.             # Shift up minor 2nd, speed up all BPMs by 60% (0.4)
678.             if not os.path.exists(os.path.join(path, f"Augment_{i}")):
679.                 os.makedirs(os.path.join(path, f"Augment_{i}"))
680.             output_path = os.path.join(path, f"Augment_{i}",
f"{os.path.splitext(file)[0]}_aug{i}.mid")
681.             new_midi.write(output_path)
682.             tempo_factor = tempo_adjustments[i - 1]
683.             adjust_tempo(output_path, tempo_factor)
```

```
684.     pass
685.
686.
687. def glob_midis(path, output_path="Data/Glob/Combined/Combined_", suffix="", choral=False,
688.                measure_limit=0, seq_len=50, include_key=True):
689.     POLYPHONIC = True
690.     file_list = glob.glob(path + "/*.mid")
691.     parser = music21.converter
692.     print(f"Parsing {len(file_list)} midi files...")
693.     if not choral:
694.         _, _ = parse_midi_files(file_list, parser, seq_len + 1, output_path + suffix,
695.                                 verbose=True, enable_chords=POLYPHONIC, limit=None)
696.     else:
697.         _ = parse_choral_midi_files(file_list, parser, seq_len + 1, output_path + suffix,
verbose=True,
698.                                     limit=None, mm_limit=measure_limit,
include_key=include_key)
699.     print("Complete!")
700.
701.
702. def slice_pickle(path, slices=4):
703.     """Slices a pickle file into smaller pieces for easier uploading to GitHub."""
704.     with open(path, "rb") as f:
705.         data = pkl.load(f)
706.         print("Found data of length:", len(data))
707.     slice_size = len(data) // slices
708.     for i in range(slices):
709.         start_index = i * slice_size
710.         end_index = (i + 1) * slice_size if i != slices - 1 else len(data)
711.         slice_data = data[start_index:end_index]
712.         base_name = os.path.basename(path)
713.         name, ext = os.path.splitext(base_name)
714.         output_path = os.path.join(os.path.dirname(path), f"{name}_{i + 1}{ext}")
715.         with open(output_path, 'wb') as f:
716.             pkl.dump(slice_data, f)
717.         print(f"Saved slice {i + 1} to {output_path}")
718.
719.
720. def load_pickle_from_slices(filename, include_augmented=False):
721.     """Loads a pickle file that has been sliced into smaller pieces for easier uploading to
GitHub."""
722.     dir_name = os.path.dirname(filename)
723.     base_name = os.path.basename(filename)
724.     # name, ext = os.path.splitext(base_name)
725.     slice_files = sorted(glob.glob(os.path.join(dir_name, f"{base_name}_*.pkl")))  #
{name}_*{ext}
726.     if include_augmented:
727.         base, dset = base_name.split("_")[:2]
728.         slice_files.extend(sorted(glob.glob(os.path.join(dir_name,
f"{base}_aug*{dset}*.pkl"))))
729.         slice_files = sorted(slice_files)
730.     if not slice_files:
731.         raise ValueError(f"No sliced pickle files found for {filename}")
732.     combined_data = []
733.     for slice_file in slice_files:
734.         with open(slice_file, 'rb') as f:
735.             slice_data = pkl.load(f)
736.             combined_data.extend(slice_data)
737.     print("Loaded data of length:", len(combined_data))
738.     return combined_data
```

```
739.
740.
741. def unify_transpose_midi_files(path):
742.     transposed_dir = os.path.join(path, "Transposed")
743.     if not os.path.exists(transposed_dir):
744.         os.makedirs(transposed_dir)
745.     file_list = glob.glob(path + "/*.mid")
746.     for i, file in enumerate(file_list):
747.         try:
748.             print(i + 1, "Parsing %s" % file)
749.             if os.path.exists(os.path.join(transposed_dir, os.path.basename(file))):
750.                 continue
751.             # Load MIDI file and transpose to C major/A minor
752.             midi = music21.converter.parse(file)
753.             key = midi.analyze('key')
754.             if key.mode == "major":
755.                 interval = music21.interval.Interval(key.tonic, music21.pitch.Pitch('C'))
756.             else:
757.                 interval = music21.interval.Interval(key.tonic, music21.pitch.Pitch('A'))
758.             transposed_midi = midi.transpose(interval)
759.             transposed_file_path = os.path.join(transposed_dir, os.path.basename(file))
760.             transposed_midi.write('midi', fp=transposed_file_path)
761.         except Exception as e:
762.             print(f"\tError parsing file {file}: {e}")
763.
764. # endregion VoiceTransformer
765.
766.
767. if __name__ == "__main__":
768.     print("Hello, world!")
769.     parse_single_midi_to_notedur_output()
770.     # data_path = os.path.join(os.getcwd(),
r"Data\MIDI\VoiceParts\Tenor\Isolated\534_001393_tenT.mid")
771.     # df_mid = midi_to_dataframe(data_path)
772.     # pd.set_option('display.max_rows', None)
773.     # pd.set_option('display.max_columns', None)
774.     # print(df_mid)
775.     # print(transpose_df_to_row(df_mid))
776.     # create_all_datasets()
777.     # compile_midi_from_voices()
778.     # slice_pickle("Data/Glob/Combined/Combined_notes.pkl")
779.     # slice_pickle("Data/Glob/Combined/Combined_durations.pkl")
780.     # load_pickle_from_slices("Data/Glob/Combined/Combined_notes")
781.     # load_pickle_from_slices("Data/Glob/Combined/Combined_durations")
782.     # augment_midi_files("Data/MIDI/VoiceParts/Combined")
783.     # glob_midis("Data/MIDI/VoiceParts/Combined", "Data/Glob/Combined/Combined_")
784.     # slice_pickle("Data/Glob/Combined/Combined_notes.pkl", slices=5)
785.     # slice_pickle("Data/Glob/Combined/Combined_durations.pkl", slices=5)
786.     """
787.     for i in range(1, 5):
788.         glob_midis(f"Data/MIDI/VoiceParts/Combined/Augment_{i}",
"Data/Glob/Combined/Combined_aug{i}_")
789.     for i in range(1, 5):
790.         slice_pickle(f"Data/Glob/Combined/Combined_aug{i}_notes.pkl")
791.         slice_pickle(f"Data/Glob/Combined/Combined_aug{i}_durations.pkl")
792.     """
793.     # load_pickle_from_slices("Data/Glob/Combined/Combined_notes", True)
794.     # load_pickle_from_slices("Data/Glob/Combined/Combined_durations", True)
795.     # glob_midis("Data/MIDI/VoiceParts/Combined", "Data/Glob/Combined_choral/Combined_",
choral=True)
```

```
    # for i in range(1, 5):
    #     print("Parsing augmented dataset", i)
    #     glob_midis(f"Data/MIDI/VoiceParts/Combined/Augment_{i}",
    #                f"Data/Glob/Combined_choral/Combined_aug{i}_", choral=True)
    # for voice in ["S", "A", "T", "B"]:
    #     slice_pickle(f"Data/Glob/Combined_choral/Combined_{voice}_choral_notes.pkl",
slices=3)
    #     slice_pickle(f"Data/Glob/Combined_choral/Combined_{voice}_choral_durations.pkl",
slices=2)
    #     for i in range(1, 5):
    #
slice_pickle(f"Data/Glob/Combined_choral/Combined_aug{i}_{voice}_choral_notes.pkl", slices=3)
    #
slice_pickle(f"Data/Glob/Combined_choral/Combined_aug{i}_{voice}_choral_durations.pkl", slices=2)
    """
    glob_midis("Data/MIDI/VoiceParts/Combined", "Data/Glob/Combined_mm1-8/Combined_",
choral=True, measure_limit=8)
    for i in range(1, 5):
        glob_midis("Data/MIDI/VoiceParts/Combined/Augment_1", "Data/Glob/Combined_mm1-
8/Combined_aug1_", "", True, 8)
        glob_midis("Data/MIDI/VoiceParts/Combined/Augment_2", "Data/Glob/Combined_mm1-
8/Combined_aug2_", "", True, 8)
        glob_midis("Data/MIDI/VoiceParts/Combined/Augment_3", "Data/Glob/Combined_mm1-
8/Combined_aug3_", "", True, 8)
        glob_midis("Data/MIDI/VoiceParts/Combined/Augment_4", "Data/Glob/Combined_mm1-
8/Combined_aug4_", "", True, 8)
    """
    # unify_transpose_midi_files("Data/MIDI/VoiceParts/Combined")
    # glob_midis("Data/MIDI/VoiceParts/Combined/Transposed",
"Data/Glob/Combined_transposed/Combined_",
    #            choral=True, include_key=False)
    # for voice in ["S", "A", "T", "B"]:
    #     slice_pickle(f"Data/Glob/Combined_transposed/Combined_{voice}_choral_notes.pkl",
slices=4)
    #     slice_pickle(f"Data/Glob/Combined_transposed/Combined_{voice}_choral_durations.pkl",
slices=2)
    # for voice in ["Soprano", "Alto", "Tenor", "Bass"]:
    #     meta_analysis(voice)
    # view_meta_analysis("Soprano")
    # combine_intros()
    pass

```

E.4 Post_process.py

```
import os
from music21 import *
from data_utils import validate_and_generate_metatrack


def load_midi(midi_path):
    return converter.parse(midi_path)


# Step 1
def make_notes_diatonic(score, key_signature):
    """
    Transposes notes to be diatonic to the given key signature.
    """
    dt_scale = scale.MajorScale(key_signature) if key_signature.isupper() else scale.MinorScale(key_signature.lower())
    scale_pitches = [p for p in dt_scale.getPitches('A0', 'A9')]
    for part in score.parts:
        for note in part.flat.notes:
            if note.pitch not in scale_pitches:
                # Find the closest diatonic pitch
                closest_pitch = sorted(scale_pitches, key=lambda p: abs(p.midi - note.pitch.midi))[0]
                # Calculate the interval between the current note pitch and the closest diatonic pitch
                interval_to_closest_pitch = interval.Interval(noteStart=note.pitch, noteEnd=closest_pitch)
                # Transpose the note by this interval
                note.transpose(interval_to_closest_pitch, inPlace=True)
    return score


# Step 2
def has_parallel_motion(interval1, interval2):
    """Checks if two intervals are both perfect and in parallel motion (fifths, octaves, unisons)."""
    return interval1.isPerfectConsonance() and interval2.isPerfectConsonance() \
        and interval1.directedName == interval2.directedName


def find_parallel_intervals(parts, interval_type, num_to_compare):
    """
    Find parallel intervals of a given type (like fifths, octaves) between two parts.
    :param      parts: Tuple of two parts to compare (part1, part2)
    :param      interval_type: The type of interval to check for parallels (P5, P8)
    :param      num_to_compare: Number of intervals to check in sequence for parallel motion
    :return:    List of indices where parallel intervals start
    """
    part1, part2 = parts
    parallels = []

    # Flatten the parts to get a single stream of notes
    part1_notes = part1.flat.notes
    part2_notes = part2.flat.notes

    # Get all start times (offsets) for both parts where notes are present
```

```
    part1_offsets = [n.offset for n in part1_notes]
    part2_offsets = [n.offset for n in part2_notes]

    # Iterate over all note pairs and compare intervals at offsets present in both parts
    for offset in set(part1_offsets) & set(part2_offsets):
        notes1_at_offset = part1_notes.getElementsByOffset(offset, mustBeginInSpan=False,
mustFinishInSpan=False)
        notes2_at_offset = part2_notes.getElementsByOffset(offset, mustBeginInSpan=False,
mustFinishInSpan=False)

        # Compare intervals at this offset
        for i in range(len(notes1_at_offset) - num_to_compare + 1):
            current_intervals = [interval.Interval(n1, n2) for n1, n2 in
zip(notes1_at_offset[i:i + num_to_compare],
notes2_at_offset[i:i + num_to_compare])]

            # Check if all compared intervals are of the specific type and in parallel motion
            if all(ivl.simpleName == interval_type for ivl in current_intervals):
                parallels.append((offset, i))

    return parallels


def correct_notes(part1, part2, index, interval_type):
    """
    A simple correction by moving the second note in the part2 down a step if it's a fifth or
octave,
    or up if it's a unison, to break the parallel movement.
    """
    n1 = part1.notes[index]
    n2 = part2.notes[index]

    if interval_type in ['P5', 'P8']:
        # Move n2 down by step to avoid parallel motion, ensure diatonicism.
        new_pitch = n2.pitch.getLowerEnharmonic()
    elif interval_type == 'P1':
        # Move n2 up by step
        new_pitch = n2.pitch.getHigherEnharmonic()
    else:
        return  # If it's neither, no action is taken

    # Ensure the new pitch is diatonic to C major/A minor by simplifying to natural notes
    # TODO: make this work with any key (e.g., same as step 1)
    n2.pitch = new_pitch.getNatural()


def apply_voice_leading(score):
    """
    Applies voice leading rules to the score, correcting parallel fifths, octaves, and unisons.
    """
    soprano, alto, tenor, bass = score.parts
    parts_to_check = [(soprano, alto), (alto, tenor), (tenor, bass)]

    for parts in parts_to_check:
        # Check for parallels in fifths and octaves (can use "P1" for unisons as well)
        for interval_type in ['P5', 'P8', 'P1']:
            parallels = find_parallel_intervals(parts, interval_type, 2)  # Find parallels for
each type
            for parallel in parallels:
```

```
106.                 offset, index = parallel
107.                 # Correct the parallel notes; however, the simplicity of correction may
108.                 # introduce new voice-leading issues, such as direct fifths or octaves
109.                 correct_notes(parts[0], parts[1], index, interval_type)
110.     return score
111.
112.
113. # Step 3
114. def double_correct_tones(chord):
115.     """
116.     Corrects doubled leading tones, altered tones, and tones in seventh chords within a given
chord
117.     (assuming the chord is within a SATB setting and diatonic to C major/A minor).
118.     """
119.     # Assuming the chord is in root position for simplicity; if not, more checks are needed
120.     root, third, fifth, seventh = None, None, None, None  # Placeholder for chord tones
121.
122.     # If there's a seventh, it's in the last position in SATB settings for a seventh chord
123.     if len(chord.pitches) == 4:
124.         root, third, fifth, seventh = chord.pitches
125.
126.     # Check for leading tone, which is B in C major/A minor (only check if there's no seventh)
127.     # TODO: make this work with any key (e.g., same as step 1)
128.     if not seventh and any(p.name == 'B' for p in chord.pitches):
129.         # Find all instances of the leading tone
130.         leading_tones = [p for p in chord.pitches if p.name == 'B']
131.         if len(leading_tones) > 1:  # If there's a doubled leading tone
132.             # Move one of them to a different chord tone that's not already in the chord and
diatonic
133.             for p in leading_tones[1:]:  # Skip the first occurrence
134.                 p = move_to_available_pitch(chord, p)
135.
136.     # Check for altered tones and move them if doubled
137.     for p in chord.pitches:
138.         if p.accidental not in (None, 'natural'):  # If the tone is altered
139.             # We find another note that is not altered and not yet present in the chord to move
to
140.             p = move_to_available_pitch(chord, p)
141.
142.     # If there is a seventh, ensure it's not doubled
143.     if seventh and chord.pitches.count(seventh) > 1:
144.         # Move the doubled seventh to another chord tone not already in the chord and diatonic
145.         seventh = move_to_available_pitch(chord, seventh)
146.
147.     return chord
148.
149.
150. def move_to_available_pitch(chord, pitch_to_move):
151.     """
152.     Moves a doubled pitch to the nearest available diatonic pitch that isn't already in the
chord.
153.     """
154.     diatonic_pitches = ['C', 'D', 'E', 'F', 'G', 'A', 'B']  # TODO: make this work with any key
(e.g., same as step 1)
155.     current_chord_tones = [p.nameWithOctave for p in chord.pitches]
156.     for diatonic_pitch in diatonic_pitches:
157.         new_pitch_name_with_octave = diatonic_pitch + str(pitch_to_move.octave)  # Convert
octave to string
158.         if new_pitch_name_with_octave not in current_chord_tones:
159.             # Return the new pitch with the same octave as the pitch to move
```

```
160.             return note.Note(new_pitch_name_with_octave)
161.
162.     return pitch_to_move  # If all diatonic pitches are taken, return the original pitch
(unlikely in four-part harmony)
163.
164.
165. # Step 4
166. def correct_melodic_intervals(score):
167.     """
168.     Corrects melodic intervals within each voice part to adhere to voice-leading rules.
169.     Checks for augmented seconds, augmented fourths, and large leaps.
170.     """
171.     for part in score.parts:
172.         notes_to_check = []
173.         for elem in part.flat:
174.             if isinstance(elem, note.Note):
175.                 notes_to_check.append(elem)
176.
177.         for i in range(len(notes_to_check) - 1):
178.             current_note = notes_to_check[i]
179.             next_note = notes_to_check[i + 1]
180.             interv = interval.Interval(current_note, next_note)
181.
182.             # Check for augmented second and tritone and correct them
183.             if interv.name == 'A2' or interv.name == 'A4' or interv.semitones == 6:  #
Augmented second or tritone
184.                 # To correct, change the next note to either a step above or below the current
note
185.                 direction = -1 if interv.direction == 'ascending' else 1
186.                 new_next_pitch = pitch.Pitch(current_note.pitch.ps + direction)
187.                 new_next_pitch.octave = next_note.pitch.octave  # Keep the same octave as the
next note
188.                 next_note.pitch = new_next_pitch
189.
190.             # Check for skips larger than an octave or a sixth
191.             if interv.name in ['m7', 'M7', 'P8'] and (i < len(notes_to_check) - 2):  # Check
for room for correction
192.                 # The following note should move in stepwise motion in the opposite direction
193.                 following_note = notes_to_check[i + 2]
194.                 stepwise_direction = 1 if interv.direction == 'descending' else -1  # Reverse
the direction
195.                 corrected_pitch = pitch.Pitch(next_note.pitch.ps + stepwise_direction)
196.                 corrected_pitch.octave = following_note.pitch.octave
197.                 following_note.pitch = corrected_pitch  # Apply the corrected pitch to the
following note
198.
199.     return score
200.
201.
202. # Step 5: Correct skips and leaps
203. def correct_skips_and_leaps(score):
204.     """
205.     Corrects skips and leaps within each voice part to adhere to voice-leading rules.
206.     Any skip larger than a sixth must be followed by stepwise motion in the opposite direction.
207.     """
208.     for part in score.parts:
209.         notes_to_check = []
210.         for elem in part.flat.notesAndRests:
211.             if isinstance(elem, note.Note):
212.                 notes_to_check.append(elem)
```

```
213.
214.         for i in range(len(notes_to_check) - 2):  # Iterate with enough lookahead to correct
the following note
215.             current_note = notes_to_check[i]
216.             next_note = notes_to_check[i + 1]
217.             following_note = notes_to_check[i + 2]
218.
219.             # Calculate intervals between current and next, and next and following notes
220.             skip_interval = interval.Interval(current_note, next_note)
221.             leap_interval = interval.Interval(next_note, following_note)
222.
223.             # Correct skips and leaps that are too large
224.             if skip_interval.isSkip and skip_interval.semitones > 9:  # If skip is greater than
a major sixth
225.                 # Determine direction for correction: if skip is ascending, next step should be
descending, vice versa
226.                 if skip_interval.direction == "ascending":
227.                     # Ensuring next interval is a step down
228.                     if not leap_interval.isStep or leap_interval.direction == "ascending":
229.                         corrected_pitch = pitch.Pitch(next_note.pitch.ps - 2)  # Step down in
pitch space (MIDI number)
230.                         following_note.pitch = corrected_pitch
231.                 else:
232.                     # Ensuring next interval is a step up
233.                     if not leap_interval.isStep or leap_interval.direction == "descending":
234.                         corrected_pitch = pitch.Pitch(next_note.pitch.ps + 2)  # Step up in
pitch space
235.                         following_note.pitch = corrected_pitch
236.
237.     return score
238.
239.
240. # Step 6: Handle dissonances correctly
241. def handle_dissonances(score):
242.     """
243.     Resolves dissonances by ensuring that:
244.     - Sevenths resolve down by step.
245.     - A perfect fourth against the bass only occurs as part of the third inversion of a seventh
chord.
246.     """
247.     for part in score.parts:
248.         for measure in part.getElementsByClass('Measure'):
249.             for i, chord in enumerate(measure.getElementsByClass('Chord')):
250.                 # Process sevenths resolving down by step
251.                 if i > 0:  # If not the first chord, check the previous chord for sevenths to
resolve
252.                     previous_chord = measure.getElementsByClass('Chord')[i-1]
253.                     if previous_chord.seventh:  # Check if the previous chord has a seventh
254.                         seventh_note = previous_chord.seventh
255.                         seventh_index = previous_chord.pitches.index(seventh_note)
256.                         # Resolve the seventh down by step
257.                         resolved_pitch = pitch.Pitch(seventh_note.midi - 1)
258.                         chord.pitches[seventh_index] = resolved_pitch
259.
260.                 # Process perfect fourths
261.                 if chord.isTriad() and interval.Interval(chord.bass(),
chord.pitches[1]).simpleName == 'P4':
262.                     # Check if the fourth is allowed, if not, alter it
263.                     if chord.inversion() != 3:  # Ensure the fourth is allowed only in third
inversion chords
```

```
264.                         tenor_pitch = chord.pitches[1]  # The second pitch in SATB ordering
265.                         # Alter the tenor pitch to make a consonant interval with the bass
266.                         new_tenor_interval = interval.Interval(chord.bass(),
tenor_pitch).transpose('P5')
267.                         chord.pitches[1] = new_tenor_interval.noteStart
268.
269.     return score
270.
271.
272. def get_new_key_and_bpm():
273.     t_key, t_time_sig, t_tempo, entrance = validate_and_generate_metatrack('Soprano')
274.     return t_key, t_tempo
275.
276.
277. def change_bpm_and_key(midi_file_path, output_file_path, new_bpm=None, new_key_tonic=None):
278.     mf = midi.MidiFile()
279.     mf.open(midi_file_path)
280.     mf.read()
281.     mf.close()
282.     midi_stream = midi.translate.midiFileToStream(mf)
283.
284.     if new_bpm:
285.         # Remove existing tempo changes
286.         for el in midi_stream.flat.getElementsByClass(tempo.MetronomeMark):
287.             midi_stream.remove(el, recurse=True)
288.
289.         new_metronome_mark = tempo.MetronomeMark(number=new_bpm)
290.         midi_stream.insert(0, new_metronome_mark)
291.
292.     original_key = midi_stream.analyze('key')
293.     original_mode = original_key.mode
294.
295.     if new_key_tonic:
296.         # Construct the new key with the same mode as the original
297.         if original_mode == 'major':
298.             new_key = key.Key(new_key_tonic)
299.         elif original_mode == 'minor':
300.             new_key = key.Key(new_key_tonic, 'minor')
301.
302.         i = interval.Interval(original_key.tonic, new_key.tonic)
303.         midi_stream.transpose(i, inPlace=True)
304.         midi_stream.replace(original_key, new_key, allDerived=True)
305.
306.     # Save the modified MIDI stream to a file
307.     midi_stream.write('midi', fp=output_file_path)
308.     return original_mode
309.
310.
311. def save_midi(score, adjusted_midi_path):
312.     score.write('midi', fp=adjusted_midi_path)
313.
314.
315. def process_midi(midi_path, adjusted_midi_path='Data/Postprocessed', verbose=True):
316.     score = load_midi(midi_path)
317.     print("Loaded MIDI file; beginning post-processing...")
318.     score = make_notes_diatonic(score, 'C')
319.     print("Completed step 1\n" if verbose else "", end="")
320.     score = apply_voice_leading(score)
321.     print("Completed step 2\n" if verbose else "", end="")
322.     score = double_correct_tones(score)
```

```
323.     print("Completed step 3\n" if verbose else "", end="")
324.     score = correct_melodic_intervals(score)
325.     print("Completed step 4\n" if verbose else "", end="")
326.     score = correct_skips_and_leaps(score)
327.     print("Completed step 5\n" if verbose else "", end="")
328.     score = handle_dissonances(score)
329.     print("Completed step 6\n" if verbose else "", end="")
330.
331.     print("Rerunning steps 1 and 2 to adjust any new voice leading issues...\n" if verbose else
"", end="")
332.     score = apply_voice_leading(score)
333.     score = make_notes_diatonic(score, 'C')
334.     print("Completed rerun of steps 1 and 2\n" if verbose else "", end="")
335.
336.     if not os.path.exists(adjusted_midi_path):
337.         os.makedirs(adjusted_midi_path)
338.     output_path = os.path.join(adjusted_midi_path, os.path.basename(midi_path))
339.     save_midi(score, output_path)
340.     print(f"Saved post-processed MIDI file to {output_path}")
341.     return output_path
342.
343.
344. if __name__ == "__main__":
345.     # TODO: change MIDI instruments to SATB voices and validate pitch ranges + tessituras
346.     print("Hello, world!")
347.     path = input("Enter the path to the MIDI file: ").replace('"', '')
348.     output = process_midi(path)
349.
350.     bpm_option = input("Would you like to change the BPM of the MIDI file? "
351.                        "(\033[4mn\033[0mo/\033[4mr\033[0mandom/enter number [e.g., 160]): ")
352.     if bpm_option.lower() in "no":
353.         n_bpm = None
354.     elif bpm_option.lower() in "random":
355.         n_bpm = int(get_new_key_and_bpm()[1].split('BPM')[0])
356.         print(f"Randomly selected BPM: {n_bpm}")
357.     else:
358.         n_bpm = int(bpm_option)
359.
360.     key_option = input("Would you like to change the key of the MIDI file? "
361.                        "(\033[4my\033[0mes/\033[4mn\033[0mo/\033[4mr\033[0mandom): ")
362.     if key_option.lower() in "yes":
363.         n_key = input("Enter the key to change to (not including mode; e.g., C, D, Ab, F#): ")
364.     elif key_option.lower() != "no" and key_option.lower() in "random":
365.         n_key = get_new_key_and_bpm()[0].split(':')[0]
366.         print(f"Randomly selected key: {n_key}")
367.     else:
368.         n_key = None
369.
370.     if n_bpm is not None or n_key is not None:
371.         mode = change_bpm_and_key(output, output, n_bpm, n_key)
372.         keymsg = "" if n_key is None else f" and key ({n_key} {mode})"
373.         print(f"Finished changing BPM{keymsg}.")
374.     pass
375.
```

E.5 Survey_analysis.py

```
#!/usr/bin/env python
# coding: utf-8

# # Dissertation Survey Analysis

# ## Power Analysis
#
# 1. Define the Null Hypothesis ($H_0$) and Alternative Hypothesis ($H_a$):
#     * $H_0$: The AI-generated music is indistinguishable from human-composed music (e.g., 50% of people believe it's human-composed).
#     * $H_a$: The AI-generated music is distinguishable from human-composed music (e.g., significantly more or less than 50% believe it's human-composed).
# 2. Effect Size:
#    * The effect size here would be the difference in proportions (by the 6-point Likert scale).
#    * For example, I consider the AI successful if, instead of 50% (chance level), 60% or more respondents believe the music is human-composed.
#    * The effect size in this context is 0.1 (60% - 50%).
# 3. Significance Level ($\alpha$):
#     * Commonly set at 0.05. This is the probability of rejecting the null hypothesis when it is actually true (Type I error).
# 4. Power (1 - $\beta$):
#     * Typically set at 0.8 or 80%. This is the probability of correctly rejecting the null hypothesis when it's false (thus detecting an effect if there is one).
# 5. Sample Size:
#     * 500 respondents.

# In[22]:


get_ipython().system('pip install -q -U statsmodels')
from statsmodels.stats.power import NormalIndPower
from statsmodels.stats.proportion import proportion_effectsize

alpha = 0.05  # Significance level
power = 0.8   # Desired power
p1 = 0.5      # Proportion under null hypothesis (50%)
p2 = 0.6      # Proportion under alternative hypothesis (60%)
effect_size = proportion_effectsize(p1, p2)  # Effect size
current_sample_size = 500

# Calculate required sample size
analysis = NormalIndPower()
sample_size = analysis.solve_power(effect_size=effect_size, power=power, alpha=alpha, ratio=1.0)
print(f"Required sample size: {sample_size}")

# Assess if current sample size is adequate
is_adequate = current_sample_size >= sample_size
print(f"Is current sample size of 500 adequate? {'Yes' if is_adequate else 'No'}")


# In[23]:


import numpy as np
```

```
import matplotlib.pyplot as plt

# Range of effect sizes to consider
effect_sizes = np.linspace(0.01, 0.5, 50)  # Varying from a small to moderate effect size

# Calculate required sample sizes for each effect size
sample_sizes = [analysis.solve_power(effect_size=proportion_effectsize(p1, p1 + es),
                                     power=power, alpha=alpha, ratio=1.0)
                for es in effect_sizes]

plt.figure(figsize=(10, 6))
plt.plot(effect_sizes, sample_sizes, color='blue', label='Required Sample Size')
plt.axhline(y=current_sample_size, color='green', linestyle='--', label='Current Sample Size ($n=500$)')
plt.axvline(x=effect_size, color='red', linestyle='--', label='Effect Size for Current Sample Size')
plt.scatter(effect_size, current_sample_size, color='cyan')
plt.title('Sensitivity Analysis: Effect Size vs. Required Sample Size')
plt.xlabel('Effect Size (difference in proportions)')
plt.ylabel('Required Sample Size')
plt.ylim(0, 1000)
plt.yticks(np.arange(0, 1100, 100))
plt.legend()
plt.grid(True)
plt.savefig("Images/Analysis/SensitivityAnalysis.svg")
plt.show()


# ## Implications of Power Analysis
#
# The effect size of -0.2, calculated from the proportions under the null hypothesis ($p_1 = 0.5$) and the alternative hypothesis ($p_2 = 0.6$), is a measure of the magnitude of the difference we are testing.
#
# 1. **Magnitude of Difference**: An effect size of -0.2 suggests a small to moderate difference between the proportions we are comparing. I.e., the difference between 50% of respondents identifying a piece as AI-generated and 60% doing so is considered to be modest.
#
# 2. **Implications for Sample Size**: The sensitivity plot shows that for an effect size of -0.2, my current sample size of 500 is more than sufficient. This is evident given the line representing the current sample size is above the curve of required sample sizes for this effect size. Hence, we have enough participants to reliably detect even this relatively small difference, reducing the likelihood of a Type II error (failing to detect a real effect).
#
# 3. **Statistical Power**: A larger sample size increases the statistical power of the test, which is the probability that the test correctly rejects the null hypothesis when it is false. With 500 respondents, this study is well-powered to detect even small differences between the proportions, assuming other parameters like the significance level ($\alpha$) remain constant.
#
# 4. **Confidence in Results**: The adequacy of my sample size lends confidence to my results, suggesting that if the study did not find significant differences in certain comparisons (e.g., some of the Chi-Square test results), it is less likely due to insufficient data and more likely reflective of the actual similarities in perceptions across different groups.
#
# This effect size, in conjunction with the power analysis, implies that my study design is robust for detecting even small differences in perceptions of AI and human compositions, and assures that the sample size is adequate to give the statistical tests enough sensitivity to detect the desired effects.
```

```
 89.
 90. # <hr>
 91.
 92. # ## Survey Results ($n = 500$)
 93. #
 94. # **Think human:**
 95. #    * Q1. AI Sample #1 (model 2)  : 50.60%
 96. #    * Q2. AI Sample #2 (model 3)  : 54.60%
 97. #    * Q4. AI Sample #3 (model 13) : 48.20%
 98. #
 99. # **Think AI:**
100. #    * Q3. Human Sample #1.        : 51.20%
101. #    * Q5. Human Sample #2.        : 29.60%
102. #
103. # **Demographics:**
104. #    * Music teacher/professional  : 08.20%
105. #    * Music student               : 12.40%
106. #    * Amateur musician            : 28.20%
107. #    * Music listener/enjoyer      : 51.20%
108.
109. # ###  Results per Question
110. #
111. # | Question                | #1 (AI) | #2 (AI) | #3 (Human) | #4 (AI) | #5 (Human) |
112. # |-------------------------|---------|---------|------------|---------|------------|
113. # | AI - Strongly believe   | 8.20    | 8.40    | 10.20      | 7.00    | 5.60       |
114. # | AI - Mostly believe     | 21.80   | 17.80   | 21.40      | 22.20   | 11.00      |
115. # | AI - Somewhat believe   | 19.40   | 19.20   | 19.60      | 22.60   | 13.00      |
116. # | Human - Somewhat believe | 15.40   | 16.20   | 18.00      | 18.00   | 20.80      |
117. # | Human - Mostly believe  | 27.00   | 26.00   | 20.60      | 23.00   | 29.40      |
118. # | Human - Strongly believe | 8.20    | 12.40   | 10.20      | 7.20    | 20.20      |
119.
120. # In[24]:
121.
122.
123. import warnings
124. import numpy as np
125. import pandas as pd
126. import seaborn as sns
127. import matplotlib.pyplot as plt
128. warnings.filterwarnings('ignore')
129.
130. population = 500
131. dfs = pd.read_excel("Data/Survey/SurveyResults.xlsx", sheet_name=[f"Q{i}" for i in range(2,
8)])
132.
133. results = pd.concat([dfs[f"Q{i}"].iloc[:, 1] for i in range(2, 7)], axis=1)
134. results.columns = [f"Q{i}" for i in range(1, 6)]
135. results = pd.concat([results, dfs["Q2"].iloc[:, 0]], axis=1)
136. results.rename(columns={results.columns[-1]: results.columns[-1][3:]}, inplace=True)
137. results.index = results.iloc[:, -1]
138. results.drop(columns=results.columns[-1], inplace=True)
139. results = results.iloc[1:, :]
140. results
141.
142.
143. # In[25]:
144.
145.
```

```
146. demographics = dfs["Q7"].iloc[:, 1]
147. demographics.index = dfs["Q7"].iloc[:, 0]
148. demographics = demographics.iloc[1:]
149. demographics = pd.concat([demographics, demographics.apply(lambda x: int(x * population))],
axis=1)
150. demographics.columns = ["Percentage", "Count"]
151. demographics.index.name = "Musical Ability Level"
152. demographics
153.
154.
155. # In[26]:
156.
157.
158. # Plot the demographic data
159. x_labels = ['Music teacher/professional', 'Music student', 'Amateur musician', 'Music
listener/enjoyer']
160. plt.figure(figsize=(10, 5))
161. plt.bar(x_labels, demographics['Count'], color='blue', width=0.5)
162. for i, v in enumerate(demographics['Count']):
163.     plt.text(i, v + 5, str(int(v)), color='black', ha='center')
164. plt.title("Demographics")
165. plt.ylabel("Sample Size ($n=500$)")
166. plt.ylim(0, 275)
167. plt.savefig("Images/Analysis/Demographics.svg")
168. plt.show()
169.
170.
171. # ## Chi-Square Analysis
172. #
173. # Hypothesis:
174. # * $H_0$: There is no statistically significant difference between the frequency of
respondents identifying a music piece as AI-generated and the frequency of respondents identifying
it as human-composed.
175. # * $H_a$: There is a statistically significant difference between the frequency of respondents
identifying a music piece as AI-generated and the frequency of respondents identifying it as human-
composed.
176. #
177. # In other words,
178. # * $H_0$: Any observed difference in the identification of AI and human compositions is due to
random chance or sampling variability.
179. # * $H_a$: The pattern of responses is not what would be expected by random chance alone.
180.
181. # In[27]:
182.
183.
184. from scipy.stats import chi2_contingency
185.
186. # Convert percentages to counts for the binary outcome
187. results_counts = results.apply(lambda x: x * population)
188.
189. # For "Think AI",    sum the counts of "AI - Strongly believe", "Mostly believe", and "Somewhat
believe"
190. # For "Think human", sum the counts of "Human - Somewhat believe", "Mostly believe", and
"Strongly believe"
191. binary_outcomes = pd.DataFrame({
192.     'Think AI': results_counts.iloc[0:3].sum(),     # Sum the first three rows for AI beliefs
193.     'Think Human': results_counts.iloc[3:6].sum()   # Sum the next three rows for Human beliefs
194. })
```

```
195.
196. # Calculate the 'Think human' counts for each question
197. think_human_counts = binary_outcomes['Think Human']
198.
199. # Expected count for "Think human" and "Think AI" if there was a 50-50 chance (null hypothesis)
200. expected_count = population / 2
201. chi2_results = {}
202. for q in [f"Q{i}" for i in range(1, 6)]:
203.     think_human_count = think_human_counts[q]
204.     think_ai_count = population - think_human_count
205.
206.     observed_values = np.array([
207.         [think_human_count, think_ai_count],  # Observed counts
208.         [expected_count, expected_count]      # Expected counts if guesses were random
209.     ])
210.
211.     chi2_stat, p_val, dof, ex = chi2_contingency(observed_values, correction=False)
212.     reject_H0 = p_val < 0.05
213.
214.     chi2_results[q] = {
215.         'Chi-Square statistic': chi2_stat,
216.         'p-value': p_val,
217.         'Reject null hypothesis': reject_H0,
218.         'Degrees of freedom': dof,
219.         'Expected values': ex
220.     }
221.
222. # Here, failing to reject the null hypothesis implies that respondents could not reliably
distinguish between AI and human-composed music (our intended result).
223. chi2_results_df = pd.DataFrame(chi2_results).T
224. chi2_results_df.index.name = "Question"
225. chi2_results_df
226.
227.
228. # In[28]:
229.
230.
231. # Reformat the data for plotting as sns.countplot() requires
232. binary_outcomes_df = binary_outcomes.reset_index()
233. binary_outcomes_df = pd.melt(binary_outcomes_df, id_vars='index', value_vars=['Think AI',
'Think Human'])
234. binary_outcomes_df.columns = ['Question', 'Response', 'Count']
235.
236. plt.figure(figsize=(10, 5))
237. sns.barplot(x='Question', y='Count', hue='Response', data=binary_outcomes_df)
238. plt.title("Binary Outcomes (Think AI vs. Think Human) per Question")
239. plt.ylabel("Count")
240. plt.ylim(0, 400)
241. plt.savefig("Images/Analysis/BinaryOutcomes.svg")
242. plt.show()
243.
244.
245. # In[29]:
246.
247.
248. plt.figure(figsize=(10, 5))
249. plt.bar(chi2_results_df.index, chi2_results_df['p-value'], color='green', width=0.5)
250. for i, v in enumerate(chi2_results_df['p-value']):
```

```
251.     plt.text(i, v + 0.01, str(round(v, 3)), color='black', ha='center')
252. plt.title("Initial Chi-Square Test Result per Question")
253. plt.ylabel("p-value")
254. plt.ylim(0, 1)
255. plt.savefig("Images/Analysis/ChiSquareInitial.svg")
256. plt.show()
257.
258.
259. # ### Chi-Square Initial Results
260. #
261. # **Q1, Q2, Q3, and Q4: Fail to Reject Null Hypothesis**
262. #  * This indicates that there is no statistically significant difference in the respondents' ability to identify these music pieces as AI-generated or human-composed.
263. #  * These findings suggest that the level of musical ability does not significantly impact the respondents' perceptions of these particular music samples, pointing to a high level of realism in the AI-generated music for these samples.
264. #
265. # **Q5: Reject Null Hypothesis**
266. #  * This significant result suggests that respondents were able to distinguish this human-composed piece from the AI-generated compositions, indicating that certain qualities of this composition were more identifiable to the listeners.
267. #
268. # #### Implications
269. # * **Perceptual Uniformity in AI Compositions**: The lack of significant differences for Q1, Q2, Q3, and Q4 underscores the effectiveness of the AI model in creating music that challenges listeners' ability to discern its origin, reflecting the AI's capability in mimicking human compositions.
270. # * **Distinguishing Human Compositions**: The significant result for Q5 highlights that certain human-composed music can possess distinct characteristics that make it more recognizable to listeners, contrasting with the AI-generated pieces.
271. # * **Overall AI Model Performance**: The general inability of respondents across various backgrounds to reliably identify the origin of most music samples (except for Q5) demonstrates the AI model's success in producing compositions that closely emulate human creations.
272.
273. # ## Kruskal-Wallis Test
274. #
275. # Hypothesis:
276. # * $H_0$: The distribution of responses (AI or Human) is the same across different levels of musical ability. In other words, the median response is equal across all groups.
277. # * $H_a$: There is a difference in the distribution of responses among at least one of the levels of musical ability.
278. #
279. # In other words,
280. # * $H_0$: The level of musical ability does not affect the ability to correctly identify AI-generated versus human-composed music.
281. # * $H_a$: At least one group’s ability to identify AI-generated versus human-composed music is statistically different from the others.
282.
283. # In[30]:
284.
285.
286. # Pull the individual questions from the "Individuals Coded" sheet
287. individuals_coded = pd.read_excel("Data/Survey/SurveyResults.xlsx", sheet_name="Individuals Coded", index_col=0)
288. individuals_coded.drop(columns=individuals_coded.columns[0:2], inplace=True)
289. individuals_coded.drop(columns=individuals_coded.columns[-7:], inplace=True)
290. individuals_coded.columns = [f"Q{i}" for i in range(1, len(individuals_coded.columns) + 1)]
291. individuals_coded.rename(columns={individuals_coded.columns[-1]: "Demographic"}, inplace=True)
```

```
292.
293. sample_codes = {
294.     "1": "AI - Strongly believe",
295.     "2": "AI - Mostly believe",
296.     "3": "AI - Somewhat believe",
297.     "4": "Human - Somewhat believe",
298.     "5": "Human - Mostly believe",
299.     "6": "Human - Strongly believe"
300. }
301.
302. demographic_codes = {
303.     "1": "Music teacher/professional",
304.     "2": "Music student",
305.     "3": "Amateur musician",
306.     "4": "Music listener/enjoyer"
307. }
308.
309. individuals_coded
310.
311.
312. # In[31]:
313.
314.
315. # Convert the dataframe to be binary coded for the sample questions
316. # 1 = Think AI, 0 = Think Human
317. binary_coded = individuals_coded.copy()
318. for i in range(0, len(binary_coded.columns) - 1):
319.     binary_coded.iloc[:, i] = binary_coded.iloc[:, i].apply(lambda x: 1 if str(x) in ["1", "2",
"3"] else 0)
320. binary_coded = binary_coded.apply(pd.to_numeric)
321. binary_coded
322.
323.
324. # In[32]:
325.
326.
327. plt.figure(figsize=(10, 5))
328. sns.heatmap(binary_coded.groupby('Demographic').mean(), annot=True, cmap='Blues', fmt='.2f')
329. plt.title("Average Response per Question by Demographic Group (< 0.5 = Think Human, > 0.5 =
Think AI)")
330. plt.yticks(np.arange(0.5, len(demographic_codes.keys()), 1), demographic_codes.values(),
rotation=0)
331. plt.xlabel("Question (1, 2, 4 = AI, 3 and 5 = Human)")
332. plt.savefig("Images/Analysis/ResponseHeatmap.svg")
333. plt.show()
334.
335.
336. # In[33]:
337.
338.
339. fig, axes = plt.subplots(1, 5, figsize=(20, 5))
340. fig.suptitle("Response Count by Musical Ability Level", fontsize=16)
341. for i, q in enumerate([f"Q{i}" for i in range(1, 6)]):
342.     sns.countplot(ax=axes[i], x='Demographic', hue=q, data=binary_coded)
343.     axes[i].set_title(f"{q}")
344.     axes[i].set_xticks(np.arange(0, len(demographic_codes.keys()), 1))
345.     axes[i].set_xticklabels(demographic_codes.keys(), rotation=0)
346.     axes[i].set_xlabel("Musical Ability Level")
```

```
347.     axes[i].set_ylabel("Count")
348.     if i != 0:
349.         axes[i].get_legend().remove()
350.     else:
351.         axes[i].legend(title="Response")
352.         handles, labels = axes[i].get_legend_handles_labels()
353.         axes[i].legend(handles=handles, labels=['Think Human', 'Think AI'], title="Response")
354. fig.tight_layout(pad=1.0)
355. fig.text(0.5, -0.02, "(1 = Music teacher/professional, 2 = Music student, 3 = Amateur musician,
4 = Music listener/enjoyer)", ha='center')
356. plt.savefig("Images/Analysis/ResponseCountAbility.svg")
357. plt.show()
358.
359.
360. # In[34]:
361.
362.
363. from scipy.stats import kruskal
364.
365. results_kw = {}
366.
367. for q in [f"Q{i}" for i in range(1, 6)]:
368.     # Split the data into groups by demographic
369.     groups = [binary_coded[binary_coded['Demographic'] == int(code)][q] for code in
demographic_codes.keys()]
370.     stat, p = kruskal(*groups)
371.     reject_H0 = p < 0.05
372.     results_kw[q] = {
373.         'Kruskal-Wallis Statistic': stat,
374.         'p-value': p,
375.         'Reject null hypothesis': reject_H0,
376.     }
377.
378. results_kw_df = pd.DataFrame(results_kw).T
379. results_kw_df.index.name = "Question"
380. results_kw_df
381.
382.
383. # ### Kruskal-Wallis Initial Results
384. #
385. # **Q1 and Q4: Reject Null Hypothesis**
386. #  * This implies there is a statistically significant difference in how respondents from
different demographic groups perceived the music.
387. #  * In other words, the level of musical ability appears to influence respondents' ability to
distinguish between AI and human compositions for these particular music samples.
388. #
389. # **Q2, Q3, and Q5: Fail to Reject Null Hypothesis**
390. #  * This suggests that there is no significant difference across the demographic groups in
terms of their perceptions of these music samples.
391. #  * The ability to identify these pieces as AI or human-composed does not seem to be
significantly influenced by the respondents' level of musical ability.
392. #
393. # #### Implications
394. #  * **Variation in Perception by Demographics**: The results for Q1 and Q4 indicate that
musical background may play a role in how certain pieces of music are perceived.
395. #  * **Consistency Across Groups**: The results for Q2, Q3, and Q5 suggest a consistency in
perception across different levels of musical experience, indicating that these pieces might have
qualities that make them uniformly identifiable (or unidentifiable) as AI or human compositions.
```

```
396.
397. # ## Post-Hoc Analysis
398. #
399. # Since the data has insufficient variability and/or size to perform Dunn's test, we will
instead use a Chi-Square Test on the larger samples and a Fisher's Exact Test on the smaller
samples.
400. #
401. #
402. # ### Chi-Square Test of Independence (Larger Groups)
403. # Hypothesis:
404. # * $H_0$: The distribution of responses (AI or Human) is the same across the demographic
groups being compared.
405. # * $H_a$: The distribution of responses differs significantly across the demographic groups.
406. #
407. # In other words,
408. # * $H_0$: There is no significant association between the demographic groups and the belief
about the music's origin (AI or Human).
409. # * $H_a$: There is a suggested association between the demographic groups and the belief about
the music's origin.
410. #
411. #
412. # ### Fisher's Exact Test (Smaller Groups)
413. # Hypothesis:
414. # * $H_0$: There is no significant association between the demographic groups and the belief
about whether the music is AI or human-composed.
415. # * $H_a$: There is a significant association between the demographic groups and the belief
about the music's origin.
416. #
417. # In other words,
418. # * $H_0$: The proportions of responses are similar across the groups being compared.
419. # * $H_a$: The proportions of responses differ between the groups.
420.
421. # In[35]:
422.
423.
424. get_ipython().system('pip install -q scikit-posthocs')
425. import scikit_posthocs as sp
426.
427. for question in ['Q1', 'Q4']:  # Questions where Kruskal-Wallis test was significant
428.     data_for_test = binary_coded[[question, 'Demographic']].copy()
429.     data_for_test['Demographic'] = data_for_test['Demographic'].map(demographic_codes)
430.
431.     # Check group sizes and variability
432.     valid_groups = []
433.     for group_name, group_data in data_for_test.groupby('Demographic'):
434.         if len(group_data) > 1 and group_data[question].nunique() > 1:
435.             valid_groups.append(group_name)
436.
437.     # Perform Dunn's test if valid groups are found
438.     if len(valid_groups) > 1:
439.         filtered_data = data_for_test[data_for_test['Demographic'].isin(valid_groups)]
440.         dunn_test_result = sp.posthoc_dunn(filtered_data, val_col=question,
group_col='Demographic', p_adjust='bonferroni')
441.         print(f"Post-hoc comparisons for {question}:\n{dunn_test_result}\n")
442.     else:
443.         print(f"Insufficient data or variability for Dunn's test on {question}.")
444.
445.
```

```
# In[36]:


from scipy.stats import chi2_contingency

# Assuming the larger groups are 'Amateur musician' (code 3) and 'Music listener/enjoyer' (code
4)
chi2_posthoc_results = {}
for question in ['Q1', 'Q4']:
    contingency_table = pd.crosstab(
        binary_coded[binary_coded['Demographic'].isin([3, 4])]['Demographic'],
        binary_coded[question]
    )
    chi2, p, dof, ex = chi2_contingency(contingency_table)
    reject_H0 = p < 0.05
    chi2_posthoc_results[question] = {
        'Chi-Square statistic': chi2,
        'p-value': p,
        'Reject null hypothesis': reject_H0,
        'Degrees of freedom': dof,
        'Expected values': ex
    }

chi2_posthoc_results_df = pd.DataFrame(chi2_posthoc_results).T
chi2_posthoc_results_df.index.name = "Question"
chi2_posthoc_results_df


# In[37]:


from scipy.stats import fisher_exact

# Assuming the smaller groups are 'Music teacher/professional' (code 1) and 'Music student'
(code 2)
fisher_exact_results = {}
for question in ['Q1', 'Q4']:
    for group1 in [1, 2]:
        for group2 in [1, 2]:
            if group1 != group2:
                table = pd.crosstab(
                    binary_coded[binary_coded['Demographic'].isin([group1,
group2])]['Demographic'],
                    binary_coded[question]
                )
                oddsratio, p_value = fisher_exact(table)
                reject_H0 = p_value < 0.05
                fisher_exact_results[f"{question} - {group1} vs. {group2}"] = {
                    'Odds ratio': oddsratio,
                    'p-value': p_value,
                    'Reject null hypothesis': reject_H0,
                }

fisher_exact_results_df = pd.DataFrame(fisher_exact_results).T
fisher_exact_results_df.index.name = "Comparison"
fisher_exact_results_df


```

```
# In[38]:


chi2_posthoc_results_df['p-value'] = pd.to_numeric(chi2_posthoc_results_df['p-value'],
errors='coerce')
fisher_exact_results_df['p-value'] = pd.to_numeric(fisher_exact_results_df['p-value'],
errors='coerce')

fig, axes = plt.subplots(1, 2, figsize=(20, 5))
fig.suptitle("Post-Hoc Test Results", fontsize=16)
sns.heatmap(chi2_posthoc_results_df[['p-value']].T, annot=True, cmap='Blues', fmt='.2f',
ax=axes[0])
axes[0].set_title("Chi-Square Test")
sns.heatmap(fisher_exact_results_df[['p-value']].T, annot=True, cmap='Blues', fmt='.2f',
ax=axes[1])
axes[1].set_title("Fisher's Exact Test")
fig.tight_layout(pad=1.0)
plt.savefig("Images/Analysis/PostHocTestResults.svg")
plt.show()


# In[39]:


plt.figure(figsize=(10, 5))
sns.lineplot(data=binary_outcomes_df, x='Question', y='Count', hue='Response')
plt.title("Cumulative Response Distribution")
plt.ylabel("Count")
plt.ylim(0, 400)
plt.savefig("Images/Analysis/CumulativeResponse.svg")
plt.show()


# In[40]:


from statsmodels.stats.proportion import proportion_confint

# Confidence intervals for the proportions of responses in each demographic group
confidence_intervals = {}
for q in [f"Q{i}" for i in range(1, 6)]:
    for group in demographic_codes.keys():
        group_count = binary_coded[binary_coded['Demographic'] == int(group)][q].sum()
        group_size = len(binary_coded[binary_coded['Demographic'] == int(group)][q])
        ci = proportion_confint(group_count, group_size, alpha=0.05, method='normal')
        confidence_intervals[f"{q} - {group}"] = {
            'Lower': ci[0],
            'Upper': ci[1]
        }

confidence_intervals_df = pd.DataFrame(confidence_intervals).T
confidence_intervals_df.index.name = "Comparison"
confidence_intervals_df


# In[41]:


```

```
555. fig, axes = plt.subplots(1, 5, figsize=(20, 5))
556. fig.suptitle("Confidence Intervals for Proportions of Responses", fontsize=16)
557. for i, q in enumerate([f"Q{i}" for i in range(1, 6)]):
558.     sns.lineplot(ax=axes[i],
data=confidence_intervals_df[confidence_intervals_df.index.str.contains(q)], x='Comparison',
y='Lower')
559.     sns.lineplot(ax=axes[i],
data=confidence_intervals_df[confidence_intervals_df.index.str.contains(q)], x='Comparison',
y='Upper')
560.     axes[i].set_title(f"{q}")
561.     axes[i].set_xticks(np.arange(0, len(demographic_codes.keys()), 1))
562.     axes[i].set_xticklabels(demographic_codes.keys(), rotation=0)
563.     axes[i].set_xlabel("Musical Ability Level")
564.     axes[i].set_ylabel("Proportion")
565.     axes[i].set_ylim(0, 1)
566.     for j, v in
enumerate(confidence_intervals_df[confidence_intervals_df.index.str.contains(q)]['Lower']):
567.         axes[i].text(j, v - 0.075, str(round(v, 2)), color='black', ha='center')
568.     for j, v in
enumerate(confidence_intervals_df[confidence_intervals_df.index.str.contains(q)]['Upper']):
569.         axes[i].text(j, v + 0.025, str(round(v, 2)), color='black', ha='center')
570. fig.tight_layout(pad=1.5)
571. fig.text(0.5, -0.02, "(1 = Music teacher/professional, 2 = Music student, 3 = Amateur musician,
4 = Music listener/enjoyer)", ha='center')
572. plt.savefig("Images/Analysis/ConfidenceIntervals.svg")
573. plt.show()
574.
575.
576. # ### Post-Hoc Results
577. #
578. # #### Chi-Square Test
579. #
580. # **Interpretation**:
581. #  * **No Significant Difference in Perception**: The results suggest that there is no
statistically significant difference in the way the larger demographic groups (e.g., "Music
listener/enjoyer" and "Amateur musician") perceived the music as AI or human-composed for Q1 and Q4.
582. #     * This implies that their ability to identify whether the music was AI-generated or
human-composed was similar across these groups.
583. #  * **Consistency Across Groups**: The lack of significant differences indicates a consistency
in perception across the largest demographic groups for these particular pieces of music.
584. #
585. # **Implications**:
586. #  * **AI Model Performance**: Since there was no significant difference in perception across
the demographic groups for Q1 and Q4, this aligns with the overall finding that listeners,
regardless of their musical background, found it challenging to distinguish between AI-generated and
human-composed music.
587. #     * This supports the success of the AI models in creating music that is not easily
distinguishable from human compositions.
588. #
589. #  * **General Audience Perception**: The results particularly highlight that the general
audience (music listeners/enjoyers) did not perceive the music differently from other more musically
trained groups, suggesting that the AI's output was uniformly convincing or ambiguous.
590. #
591. #
592. # #### Fisher's Exact Test
593. #
594. # **Q1 Results**:
595. #  * Groups 1 and 2 (Music teacher/professional and Music student):
```

```
596. #     * The p-values are 1.0 for both comparisons.
597. #     * This indicates that there is no statistically significant association between these
demographic groups and their beliefs about whether Q1 was AI or human-composed.
598. #     * Essentially, both groups responded similarly to Q1.
599. #
600. # **Q4 Results**:
601. #  * Groups 1 and 2 (Music teacher/professional and Music student):
602. #     * The p-values are approximately 0.00866 for both comparisons.
603. #     * This is below the typical alpha level of 0.05, suggesting that there is a statistically
significant difference in how these two groups perceived the music sample in Q4.
604. #     * In other words, the ability to identify Q4 as AI-generated or human-composed seems to
differ significantly between these two groups.
605. #
606. # **Interpretation and Implications**:
607. #  * **Q1 (General Agreement)**: The lack of a significant difference in Q1 implies that both
music teachers/professionals and music students had similar difficulty (or ease) in identifying the
music's origin.
608. #    * This might suggest that Q1's composition was such that it equally challenged or fooled
both groups.
609. #
610. #  * **Q4 (Significant Difference)**: The significant difference observed in Q4 suggests that
one group was better able to distinguish the origin of the music than the other.
611. #    * This could be due to the specific characteristics of the music sample in Q4, which may
have been more easily identifiable as AI or human-composed by one group over the other.
612.
613.
614. ## Summary of Statistical Analyses and Findings (truncated; included in dissertation)
615.
```